The Combining of C++ and OpenGL

An Object-Oriented Framework for Graphics

Decai Deng

A Major Report
in
The Department
of
Computer Science

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

October 2001

0-612-68462-8

ABSTRACT

The Combining of C++ and OpenGL

An Object-Oriented Framework for Graphics

Decai Deng

Today, computer software systems are becoming fairly large and more and more complex than ever before. As a result of this, the development environment of systems is also changing. The designers and programmers have the feeling of anxiety and frustration for developing the maintainable, reusable, and high efficiency software system.

This major report focuses on the design of the graphics framework for combining object-oriented development with OpenGL software system on the Microsoft Windows platform.

OpenGL is a widely used standard for graphics programming. Although OpenGL programs vary widely, they have a number of elements in common. "Frameworks" are collections of abstract base classes that are intended to capture the common elements of a domain of applications. An OpenGL Framework would incorporate the common features of OpenGL programs, leaving the graphics programmer to fill in the details of the particular application. In this report, two OpenGL animations will be developed. They give an example of the OpenGL framework, which can be used to design the maintainable and reusable applications.

ACKNOWLEDGMENT

I would like to thank sincerely my supervisor Dr. Peter Grogono for his encouragement, for his invaluable suggestions and comments, and for his generous help and support on the subject of this major report. This work would not have been possible without his consistent guidance. I wish to take this opportunity to say, "Thank you, Dr. Grogono!"

I am grateful to my wife Jie Shi and my son Haiyu for their patience and moral support.

And I wish to express my thanks to all those people who have helped me in the production of this report.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Since the mid-eighties, graphics systems have become powerful tools for scientists and engineers. With the development of computer science and the improvement of level of people lives, the graphics systems, especially three-dimensional (3D), are assuming more and more important roles. For portability reasons, all of the graphics systems are based on one or more 3D graphics software including *OpenGL*, *SGI*, *SUN XGL*, or *HP Starbase*. Some of these software packages provide a graphics programming environment, while others provide APIs for graphics programming. Because these graphics software were developed earlier, they were usually limited by machine, data dependencies, and the approach of design and implementation. So they were typically not flexible or extensible. For example, OpenGL was developed in C programming language plus itself several libraries and statements. The programming style of OpenGL is the procedure approach. Moreover, OpenGL uses callback functions to handle windows and input events for graphics objects. There are always many global variables in an application system. Today, the software industries are becoming more and more complex and large. Programmers require they can develop some software systems that are reliable, extensible, readable, reusable, and maintainable. However, it is obviously difficult for OpenGL to meet these requirements with old programming methods.

The art of programming has changed dramatically over the years. Object-oriented methodology was developed after the procedure programming, and it has become more and more popular in the software industry. In fact, the object-oriented design and development is a tool for helping the programmers in the development of software systems. It provides an organizational method for developing the large and complex 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. Frameworks can also improve the documentation and maintenance of existing systems.

1.1 Goals

This Major Report will discuss the design and implement action of two simple three-dimensional animations. They are developed in C++ programming language and OpenGL graphics software system. In another word, the animations are developed by combining object-oriented programming with graphics. Through the design, three following goals can be reached.

- **Learn the idea of object-oriented frameworks**. Through the development of the application systems, master the basic idea frameworks in object-oriented programming, know what is a framework and how to design and develop an application software system using the framework technology.
- **Become familiar with basic OpenGL programming**. Through the design and implementation of the simple animations, study and understand the basic structure and the performance processes of an OpenGL program.
- **Know how to combine the OO framework design with OpenGL**. Applying the OO frameworks in OpenGL software, implement readable, reusable, and maintainable programs. And reduce the number of global variables in an application as more as possible, instead of using objects.

1.2 Scope

In this report, two simple animations will be designed and implemented in C++ and OpenGL. That is a combination object-oriented frameworks and graphics programming. Thus the importance of this report is OO frameworks and OpenGL.

Chapter two gives an overview of the object-oriented development. It defines the meaning of the object-oriented paradigm, and describes the main features of the object-oriented design and implementation such as objects, classes, inheritance, polymorphism, and so on.

Chapter three presents the frameworks in object-oriented programming. Here we state the definition of frameworks, some characteristics of a framework, and how to apply the frameworks. An example of frameworks, MVC, is introduced here, too.

Chapter four provides an overview for OpenGL software. It introduces the history of the OpenGL, the related libraries, the structure of an OpenGL application, and the syntax of OpenGL commands and some statements. And the advantage of combining the OO frameworks with OpenGL will be also stated in here.

Chapter five simply reviews the C++ programming language. It mainly introduced the basic type and the five cornerstones of C++, which are also basic features of object-oriented programming language.

Chapter six and chapter seven will illustrate the design and implementation of two three-dimensional animations.

Chapter eight discusses the results, problems encountered and solutions found in developing the framework for graphics systems.

Appendix A gives out the requirements of system for this application and an operation guide for the users. And appendix B lists all the source codes of the two systems.

Chapter 2: Object-Oriented Development

This Chapter provides an overview of Object-Oriented software development. It starts with a general discussion of software development and desirable qualities of software products. Then it gives an introduction of the basic concepts, principal features and techniques of the object-oriented design and development approach.

2.1 Software Development

The software industry was one of the most successful industries during the 1980s and 1990s. Not only was its growth in market value exponential, but also it was able to carry technologically advanced and renewed products at an unrelenting pace. Today, the computer software has become very popular in every aspect of our life. People in the World are becoming more and more dependent on the computer software systems, no matter from the very complex nuclear power plants to computerized trading systems of stock markets, or to personal organizers on palm-top computers. It is impossible to imagine the life of the cities without computer and computer software. However, computer software is expensive. The cost of purchasing, developing, maintaining, and upgrading software systems has become the largest single expenditure for many associations. The Objected-Oriented software development approach and framework can significantly improve current software development practice, and the software industry has widely accepted it in recent years.

2.1.1 Software Qualities Attributes

Software systems are the products of software development. According to George Wilkie [1], the most basic desirable qualities of software systems are stated as following.

- Usefulness: Software systems should sufficiently address the requirements of their intended users in solving the real problems and providing needed services.
- Reliability: Any software systems should perform as expected by users in terms of the correctness of the functions being performed, the availability of services, and an acceptable level of failures.
- Timeliness: Software systems should be completed and shipped in a timely manner. Otherwise, they may be less useful or even useless because of the changes in users' needs and operating environments.
- Maintainability: Software systems should be easily maintained, that is, making corrections, adaptations, and extensions without undue costs.
- Reusability: Some components of software systems should not be designed as ad hoc solutions to specific problems in specific contexts; rather they should be designed as general solutions to a class of problems in different contexts. Such general components can be adapted and reused many times.
- Efficiency: Software systems should not make wasteful use of system resources, including processing time, memory, and disk space.
- User Friendliness: Software systems should provide user-friendly interfaces. They should be appropriate for the capabilities and the background of the intended users to facilitate easy use and access to the full extent of the systems' functions.

Not all of the above desirable qualities can be achieved at the same time, nor are they of equal importance. Recent several decades, the technologies and approaches of computer hardware and software have been greatly improved; the memory and disk space are becoming less importance in software systems. But the software systems are also becoming more and more complex and expensive. So the maintainability and reusability of software systems have become more important features in software development. The Object-Oriented software development approach and framework that will be studied in this major report can improve these qualities.

2.2 Object-Oriented Programming Models

The Object-Oriented programming (OOP) model is a software development process that provides an organizational framework for developing the large and complex computer systems. And it is a tool designed to help the programmers in the development of software systems. In contrast with the functional approach that focuses on the algorithm, the Object-Oriented programming model uses a *data-oriented* approach for software development. In this approach, the data is encapsulated into objects and can be manipulated through methods or operations.

2.2.1 Evaluation of Object-Oriented Programming

Object-Oriented models represent a balanced view of the data and computation aspects of software systems. Object-Oriented models are composed of objects, which contain data and make computations on that data. The decomposition of complex systems is based on the structure of objects, classes, and the relationships among them. For example, an automobile is an object that consists of objects such as *engine*, *accelerator*, *gear*, *brake* etc. and implements operations such as *start car*, *depress accelerator*, *shift gear*, *depress brake*, and so on.

The Object-Oriented approach views the program as a *model* of the system being studied consisting of a set of interacting *objects*. Thus the process of the system development resembles almost exactly as real world situations. Today, the software systems are becoming larger and more complex; and they are hoped gear towards reusability, maintainability, and extendibility. However, the software system can be easily modified, extended and maintained if it is produced according to the object-oriented principles. Compare the traditional programming approach, the Object-Oriented programming model offers some advantages.

Object-Oriented programming provides a way of building programs through incremental modification.

In OOP, programs can often be extended by adding new code rather than altering existing code.

The same identifier can be used in various contexts to name related functions.

Object-Oriented programming language support encapsulation, which separates the interfaces from their implementation.

An important idea in OOP is that the call graph is created and modified during execution of programs; however in the procedural languages, the call graph can be statically inferred from the program text.

It is these characters, making OOP become a popular tool for developing systems in the software industry. For many people today, the word "programming" just means "object-oriented programming".

2.2.2 Requirements for OOP Language

The origin of object-oriented software development dates back to 1960s. From the issue of the first object-oriented programming language - Simula designed in the Norwegian Computing Centre in 1962, to the wide application of the pure OOP language - Java introduced by Sun Microsystems in 1998, there are already so many object-oriented programming languages. Although there are many differences between them, they must fulfill all or parts of the basic requirements. That is these OOP languages must support for the creation and functionality of objects, the object management by classes, and the management of classes through inheritance. And they should provide the support for the modularity, concurrency, some of message passing between modules, binding, and the polymorphism characteristics. Figure 2.1 illustrates some of the basic requirements for object-oriented programming languages.

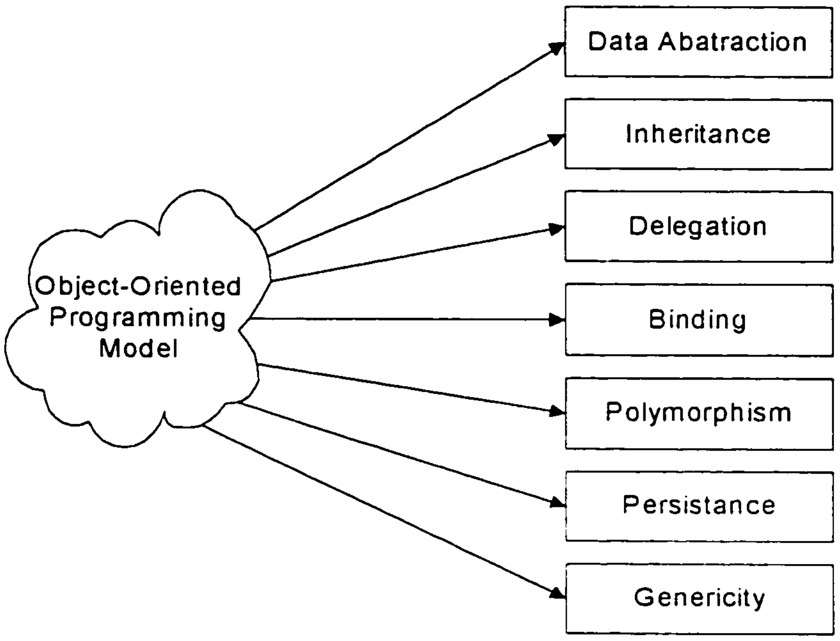


Figure 2.1 Requirements for OO programming language

2.3 Principal Features of OOP

This section will discuss the basic principles of object-oriented programming.

2.3.1 Objects and Classes

Objects and *classes* are two of the fundamental concepts in Object-Oriented development. There are two different aspects about objects and classes. One is their *interpretation* in the real world; and another one is the *representation* in the object-oriented model. The representation of objects and classes dealt with in the object-oriented model is only an approximation of the objects and classes in the real world. Table 2.1 illustrates the terms which objects and classes are interpreted in the real world and their representation in the model.

	Interpretation in the real world	**Representation in the model**
Object	Represents anything in the real world that can be distinctly identified.	Has a unique identity, a state, and *methods*.
Class	Be a set of objects with similar characteristics and methods. The objects are called the *instances* of the class.	Describes the structure of states and methods that can be shared by all its instances.

Table 2.1: The terms objects and classes present

The state of an object is composed of a set of *fields* or *attributes*, and their current values. The methods may access or manipulate the state. An object is denoted by a specific name that provides a unique identity for the object and allows the user to distinguish it from other objects. The features of an object refer to the combination of the state and the methods of the object.

A class serves as a template for creating its instances. It is to define the features of the classes to which the individual objects belong instead of defining the features of these objects. The terms *object* and *instance* are often interchangeable. And *methods* are sometimes called *operations*.

Objects communicate with one another through message passing. A message represents a command sent to an object – known as the receiver or the user of the message – to perform a certain action by invoking one of the methods of the receiver. Usually, the methods performed to pass message are called class interface. Figure 2.2 illustrates the message passing between an object and its user.

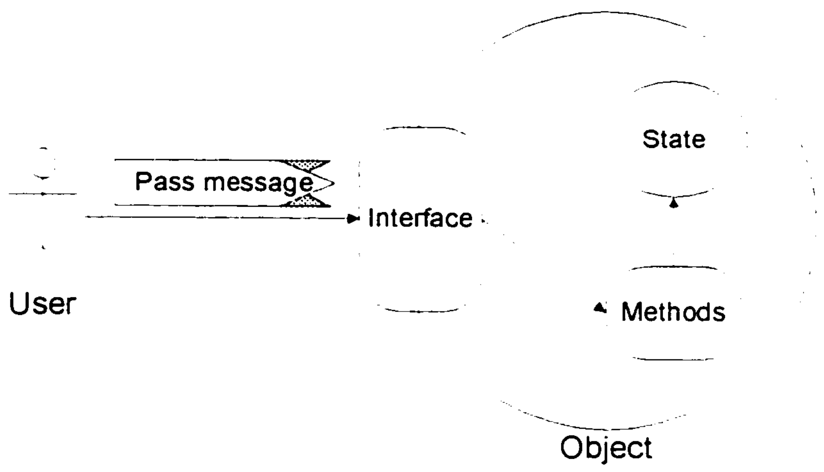


Figure 2.2: Message passing between an object and its user

The concepts of objects and classes are supported as two of the basic constructs in an object-oriented programming language. The purpose of class creation in an OOP environment is to reduce the complexity of a large system through the use of different techniques.

2.3.2 Modularity

The principle of modularity is another of the fundamental principles of the object-oriented approach. It is intended to control the complexity of large-scale systems through the use of the divide-and-conquer technique. The modularity principle ensures that a complex system should be decomposed into a set of highly cohesive but loosely coupled modules.

Decomposition of complex software systems into modules is one of the most intriguing tasks in software development. A system may be extremely complex in its totality, but a modular decomposition of the system is to break it down into modules so that each module is relatively small and simple, and the interactions among modules are relatively simple.

Typically, modular decompositions are hierarchical. And in the object-oriented approach, modules take the form of classes and packages.

2.3.3 Abstraction and Encapsulation

Abstraction and encapsulation are powerful tools for deriving modular decompositions of systems. Abstraction means to separate the essential from the nonessential characteristics of an entity. The result is a simpler but sufficiently accurate approximation of the original entity, obtained by removing or ignoring the nonessential characteristics.

Encapsulation is a closely related and complementary principle. It makes an agreement that the clients need know nothing more than the service contract while using the service. That is, the implementation of a module should be separated from its contractual interface and hidden from the clients of the module. Encapsulation is also known as information hiding. It is intended to reduce coupling among modules. The less the clients know about the implementation of the module, the looser the coupling between the module and its clients can be. An important benefit of encapsulation is that, 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.

2.3.4 Inheritance

Inheritance defines a relationship among classes. It is a key aspect of object-oriented programming, permitting the definition of new classes from previously defined ones. A class that inherits from another has all the features – including attributes and operations – defined in that class, plus its own. When class C2 inherits from class C1, class C2 is know as a *subclass* of class C1. And class C1 is known as a *superclass* of C2. Figure 2.3 illustrates an example of the relationship.

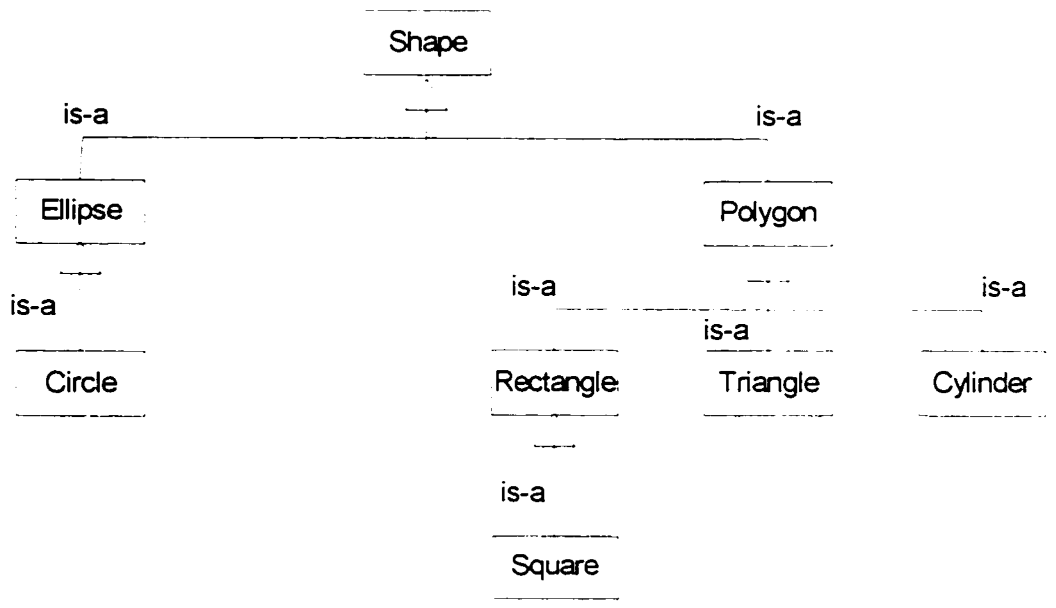


Figure 2.3: Inheritance relationship

Conceptually, inheritance models the *is-a* relationship in the real world; that is, if *Square* is a subclass of *Rectangle*, then every instance of *Square* is an instance of *Rectangle*, and everything that applies to the instances of *Square* also applies to instances of *Rectangle*.

An important technique associated with inheritance is **redefinition**. When inheriting from a class, it is necessary to provide new implementations of some features. In another word, inheritance allows the implementation of a superclass to be *shared* or *reused* by its subclasses.

There are two kinds of inheritance. One is known as single inheritance, in which each class may inherit from only one superclass; the inheritance relationship in figure 2.3 is just single inheritance. Another one referred to as *multiple inheritances*, which a class can inherit from multiple super classes. Figure 2.4 shows a simple example of this relationship. The class *Sphere* inherits from class *Shape* and *Colour*.

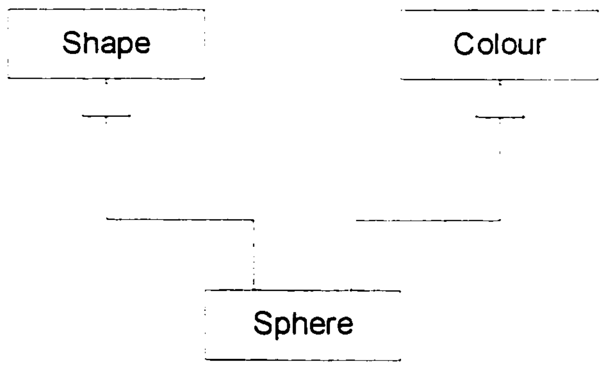


Figure 2.4: Multiple inheritance relationship

2.3.5 Polymorphism and Binding

The concept of sending different messages to different types of objects in object-oriented programming is referred to as *polymorphism*. In other words, polymorphism is the ability by which a method can be executed in more than one way, depending on the arguments and the return value. This means that the client class that sends a message does not need to know the class of the receiving instance. The client class provides only a request for a specified event, 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. According to Rao [1993], there are three different kinds of polymorphism.

- **Parametric polymorphism**: An implicit or explicit type parameter is used to determine the actual type of argument required for each of the polymorphism applications. So the same operation can be applied to different types of argument(s).
- **Inclusion polymorphism**: An object can belong to many different types that need not be disjoint. The object type may include one or more related types, as found in subtyping. In the class hierarchy, objects belonging to a class in the

hierarchy can be manipulated as belonging not only to that type, but also to its super types. Thus, certain operations on objects can work not only on objects of the subclasses but also on objects of the super classes.

- **Ad hoc polymorphism**: When a procedure works or appears to work on several types, it is called ad hoc polymorphism. It is similar to overloading and not considered to be a true polymorphism.

According to the processing time, polymorphism can be either static – resolved at compile time, or dynamic – resolved at run time.

Another fundamental principle of the object-oriented programming is binding. Binding refers to the time at which a decision is made concerning what function to execute. There are two types of binding: dynamic and static.

Dynamic binding or *runtime binding* is a way of implementing the polymorphism characteristic. Dynamic binding may also be referred to as "late binding" because the method is not bound to specific code until as late as the method is invoked at run-time. This is dependent upon the class of the object itself.

Compare with the dynamic binding, in *static binding* or *early binding*, the compiler makes the decision that which function will be called; and this depends upon the scope rules of the language.

2.3.6 Association and Aggregation

Associations represent general binary relationships among classes. If using graphical notation to represent the association relationships, each association may have an optional label that describes it. Figure 2.5 shows several associations among the *Student*, *Faculty*, and *Course* classes. In this figure, *Teach* and *Enrol* are the labels of the association between *Faculty* and *Course*. and between *Student* and *Course*, respectively.

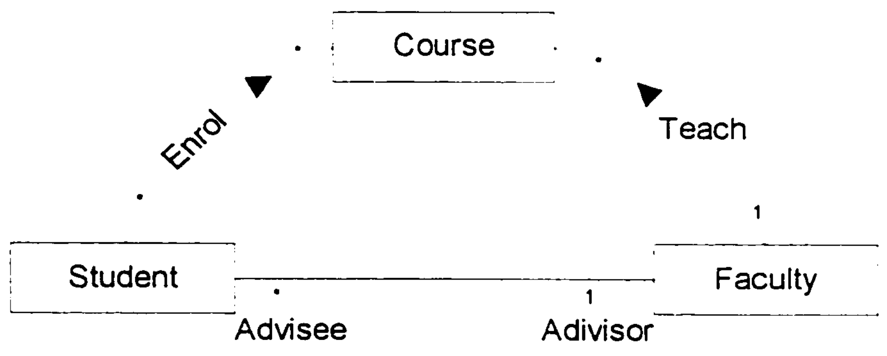


Figure 2.5: Association relationship

Aggregation is a special form of association. It represents the *has-a* relationship. A stronger form of aggregation is called *composition*, which implies exclusive ownership of the component class by the aggregate class.

Association and aggregation provide another way to reuse or share existed implementation of classes.

Chapter 3: Frameworks in OOP

With the development of software industry, the application software system is becoming bigger and more complex; designing object-oriented software is harder. However, designing reusable object-oriented software is even harder still. The designer must first find appropriate objects, then factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationship among them. Any experienced object-oriented designers will agree that a reusable and flexible design is difficult.

Frameworks can make it easier to design reusable application system. Frameworks help choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. It can also improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. In a word, frameworks help a designer get a design right faster.

This chapter will make a argument for the framework, what the definition is, what characteristics there are, and how to develop an application framework. An example of framework is given here, too.

3.1 Definition of a Framework

A framework is a collection of concrete and abstract classes that defines a software architecture intended to be reused for a specific purpose by different users and applications. The design of the framework fixes certain roles and responsibilities among the classes, as well as standard protocols for their collaboration. The framework provides simple mechanisms to customize. Customizing is typically done by inheritance an existed class of the framework and overriding a small number of methods.

According to Peter Deutsch [15], the definition of framework is as following.

"a collection of abstract classes, and their associated algorithms, constitute a kind of framework into which particular applications can insert their own specialized code by constructing concrete subclasses that work together. The framework consists of the abstract classes, the operations they implement, and the expectations placed on the concrete subclasses"

A framework supports the development of a family of applications. Typically a framework is developed by expert designers who have a deep knowledge of the application domain and rich experience of software design. On the other hand, a typical application developer who reuses the framework is less experienced and less knowledgeable of the domain.

3.2 Characteristics of Framework Application

The framework dictates the architecture of the application. It will define the extensive structure, its partitioning into classes and objects, the key responsibilities, how the classes and objects collaborate, and the thread of control. A framework defines beforehand these design parameters so that the application designer and/or implementer can focus on the specifics of the application. And the framework captures the design decisions that are common to its application domain.

A framework allows the user to reuse abstract designs, and pre-fabricated components in order to develop a system in the domain. A user may also customize existing components by subclassing. The design of the framework incorporates decisions about the distribution of control and responsibility, the protocols followed by components when communicating, and implementations for each of the major algorithms. Often the implementations are template methods that embody the overall structure of a computation and that call user's classes to perform sub-steps of the algorithm. Default implementations of each user class may be provided, and the user will subclass in order to override or specialize the operation which implements the sub-steps.

Not only can the user of a framework build applications faster as a result, but also the applications have similar structures. This makes them easier to maintain. And they are more consistent to their users.

Furthermore, since applications are so dependent on the framework for their design, they are particularly sensitive to changes in framework interfaces. As a framework evolves, applications have to evolve with it. This makes loose coupling all the more important. Otherwise even a minor change to the framework will lead to major repercussions.

However, a framework is not an easy thing to understand when one first uses it. The design is very abstract, to factor out commonality; the design is incomplete, requiring additional subclasses to create an application; the design provides flexibility for several hotspots, not all of which are needed in the application at hand; and the collaborations and the resulting dependencies between classes can be indirect and obscure. On the other hand, because many design decisions have been made for them, the users of a framework may lose some creative freedom.

3.3 Developing an Application Framework

An application framework is developed in response to feedback from reusers. An initial framework is usually based on past experience or by careful construction of one or two applications, remembering the need for flexibility, reusability and clarity of concepts. Each consequent reuse points out shortfalls in these qualities in the existing framework as one stretches the architecture to accommodate the new application. By addressing the issues raised, the framework evolves coverage of domain concepts, and clarity of the concepts and the dimensions along which they vary.

The design and implementation of an application framework depends heavily on abstract classes, polymorphism, and inheritance. The major steps in developing an application framework can be summarized as follows.

- **Identify and analyze the application domain, find appropriate objects, and identify the framework.** Object-oriented programme are made up of objects. The hard part about object-oriented design is decomposing an application system into objects. In fact, many objects in a design come form the analysis model. If the application domain is large, it should be decomposed into a set of possible frameworks that can be used to build a solution. Existing software solutions must be analyzed to identify their commonalities and the differences. The frameworks help identify less-obvious abstractions and the objects.
- **Determine object granularity and identify the primary abstractions.** Objects of an application domain can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. Framework can address the issues: decide what should be an object, clarify the role and responsibility of each abstraction, design the main communication protocols between the primary abstractions, and document them clearly and precisely.
- **Specify object interface and design how a user interacts with the framework.** An object's interface characterizes the complete set of requests that can be sent to the object. The requests may include the operation's name, the objects it takes as parameters, and the operation's return value. Object interfaces are fundamental in object-oriented systems. Frameworks, here, help define the interfaces by identifying their key elements and the kinds of data that get sent across an interface. And frameworks also specify relationships between these interfaces, provide concrete examples of the user interaction, and provide a main program illustrating how the abstract classes are related to each other and to the classes for user interaction.
- **Specify object implementations in the framework.** An object's class defines how the object is implemented. The class specifies the object's internal data and representation, and defines the operations the object can perform. New classes can be defined in terms of existing classes using class inheritance. And an abstract class is one whose main purpose is to define a common interface

for its subclasses. The developers of the framework commit to an interface defined by an abstract class, and instantiate concrete classes somewhere in the application systems.

- **Build reuse mechanisms in the framework.** The challenge of frameworks is to build flexible, reusable software design. There are several techniques for reusing functionality in object-oriented development. They are: class inheritance defines the implementation of one class in terms of another's; object composition obtains new functionality by assembling or composing objects to get more complex functionality; parameterized types, which are also known as generics and templates, let the developer can change the types that a class can use. It is just these reuse techniques make the OO design and framework more and more popular and important.
- **Maintain and reuse the framework.** Each framework lets some aspect of system structure vary independently of other aspects. So maintaining a framework is comparatively simple. However, the maintainer of a framework can reuse it in the same or even other application domains.

When analyzing existing applications to determine reusable components and abstractions, one might restructure the classes in order to separate what is common across applications from what is unique to one application.

3.4 Kinds of Framework Reuse

Frameworks emphasize design reuse over code reuse. But different users of a framework usually focus on the different aspects of the framework. And they need the different requirements document for the framework. Here are the different kinds of people who reuse a framework.

3.4.1 Framework Maintainer

A developer who responds for the maintenance and evolution of a framework must understand the design of the framework. The developer should be both domain experts and software experts. He must clearly know the internal framework design, the design rationale, the application domain, and the required flexibility of the framework.

There are many aspects of the framework that need to be understood, including the application domain, the extensive architecture and its arguments, the reasoning behind the selection of the hotspots, which design pattern provides flexibility, and why each design pattern was selected. Additionally, other aspects required to be grasped are the responsibility of each class, their interface contracts, and the shared collaboration of classes. All information is needed for both a high level of abstraction, and for a concrete level of detail.

The documentation must be descriptive; it cannot be prescriptive, sine the original designers can rarely predict how a framework might be extended through additional flexibility.

3.4.2 Application Developer

An application developer may be neither a domain expert, nor an experienced software developer. But he should know how to customize the framework to produce the desired application. This is a very goal-directed activity, where the main priority is to know how to do something, rather than to understand why it is done that way. The application developer needs to know which classes should be inherited, which methods will be override, and whether combinations of classes and methods need to be specialized to maintain a protocol of collaboration among the classes. So the need for documentation is the prescriptive aspects that an application developer requires.

3.4.3 Developer of Another Framework

Framework developers often find good ideas from existing frameworks, even though the frameworks are for another application domains. Their particular interests are the design patterns, which provide flexibility. The developers require information primarily at a high level of abstraction, though the kinds of information needed are similar to that for the framework maintainers.

3.4.4 Verifier

Some application developers and framework developers may be concerned with the hardness of their system. They may have a need to verify certain properties of the system in order to satisfy stringent customer requirements. This requires formal methods of specification and verification.

Specification for reuse is generally more descriptive than prescriptive: the reuser is left to figure out the implications of the specification in terms of the desired customization. The main concerns are to clearly specify the obligations on a subclass and its methods that a developer may write, to specify any protocols that the developer can customize, and to specify the collaborations must be supported by the developers' new subclasses.

3.5 MVC – an Example of Framework

The Model/View/Controller (MVC) is the "classic" framework because it was one of the first to be introduced. The MVC framework was designed as a tool to help system designers and programmers develop graphical user interfaces (GUI) through the reuse of software components. It pays particular attention to the presentation and interaction aspects of the application. In the framework, an important point is that the model, view, and controller are independent.

The developer of an application system can improve software productivity by reusing user interfaces for different applications, providing support for multiple ways of viewing, and interacting with the application model. For instance, for a given model, some users may prefer to view their data in the form of tables, while others may prefer to view them as diagrams. However, for a given application, the different views can be mapped into one model. Thus, the development cost included in implementing the model is met with only one time; and different models can be presented with similar user interfaces.

3.5.1 Development of the MVC Framework

The principle of the Model-View-Controller framework allows the systematic development of an interactive application. Any interface within this kind of framework consists of the following three components.

Model: A collection of objects which represent the application domain of the user interface. These objects form the main composition of the system and respond for informing Views and Controllers of changes in their state.

View: A collection of objects that contribute to the user interface. It is a specification of how different aspects of the model are presented to the user.

Controller: A collection of objects that control the flow of information between the Model and the View. It specifies how the user can interact with the application through requesting changes in the View and in the Model.

These three components establish the MVC framework. As illustrated in the figure 3.1, the Controller is responsible for handling the user input and communicating with the Model and View through message passing.

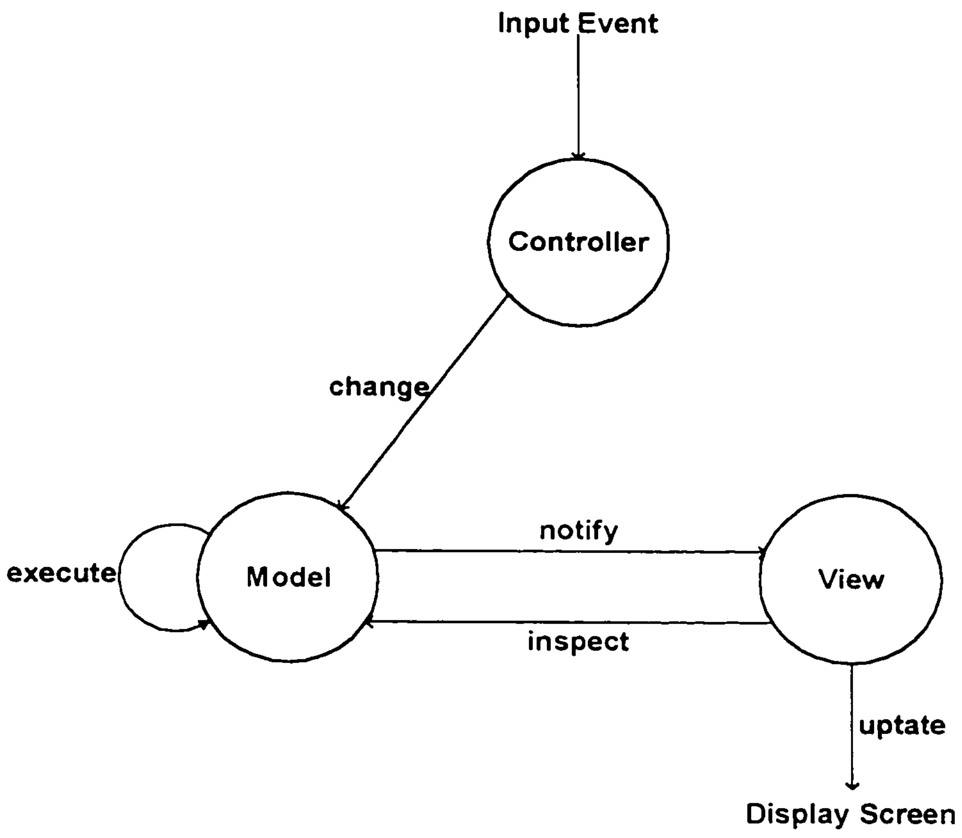


Figure 3.1: MVC framework

For example, via the keyboard or mouse, a user issues a command to change the state of a 3-dimention graphics displayed on screen. The controller receives the user's input event and sends a message to the model requesting a change. The model executes the appropriate methods and then notifies the view that it has changed. As a result, the view inspects the current state of the model and updates the effects of a change in a new graphic state to the user on screen. The simple animations developed in this Major Report are designed as this kind of MVC framework.

To develop a software application under the MVC framework, the software developers concentrate upon the design and implementation of components with respect to the model. After completion of the model, the developers build the appropriate user interface components according to the user's requirements. The components can be selected from the existed reusable frameworks. Thus through changing the user interface, different system models are quickly produced.

3.5.2 Benefits of the MVC Framework

Using the MVC framework offers the advantage of multiple viewing, development productivity, and quality for the development of an application system.

Multiple viewing: Since the Model and View are independent, a model can be mapped into several interfaces. For example, an elevator control system model can have different graphical user interface such as analog, digital, graph etc.

Development productivity: When retaining the model, it is possible to create new views and controllers through the refinement of existing one. Thus the reusability of existing MVC framework components can significantly reduce the amount of programming required in order to develop a system.

Quality: Through the reuse of existing components, the quality of these components improves as they undergo a refinement process under different conditions and environments.

Chapter 4: Introduction to OpenGL

This section provides a basic viewpoint for the graphics system OpenGL. Here will state what OpenGL is, several libraries that OpenGL supports. Moreover, it also gives out the basic structure of an OpenGL application program, and the syntax of some OpenGL commands used in this Major Report.

4.1 What is OpenGL?

OpenGL is a widely used standard for graphics programming. OpenGL means "Open Graphics Library." It is a software interface to graphics hardware. This interface contains more than 150 different routines that allow production of interactive three-dimensional graphics applications. Based on IRIS GL, SILICON GRAPHICS INC (SGI) developed the source code of the library for graphics workstations. In 1992, Microsoft and SGI decided to support this standard. It took four years to make the dream come true: OpenGL now runs bug-free on several platforms, including on Windows9X, Windows NT, Alpha-Stations (Digital workstations), and on all SILICON GRAPHICS workstations.

Obviously, there are many advantages of such a standard library:

- First, the commands cover a large part of the typically used graphics commands. This saves quite a lot of programming work; for instance, the simple OpenGL command *glEnable(GL_DEPTH_TEST)* allows the user to work with a very efficient hidden surface algorithm called "depth-buffering" or "z-buffering". From the moment you have called this routine, any further drawing is done considering depth values.
- Second, OpenGL is a hardware-independent interface, and therefore an excellent prerequisite for animated graphics via the Internet, e.g., Virtual

Reality Modeling Language (VRML) is based on OpenGL. With OpenGL, users can design graphics programs on any computer and display the graphics on any other computer.

- More and more hardware manufactures support OpenGL. Thus, 3D graphics can be done in real time, even on otherwise less powerful computers like PCs.

Of course, OpenGL also have some drawbacks:

- In order to achieve hardware-independency, OpenGL does not offer commands for performing windowing tasks or obtaining user input.
- OpenGL doesn't provide high-level commands for describing models of 3D objects; instead, it only supports the drawing of a small set of geometric primitives like points, lines, and polygons. Users must build up their desired model from this small set.

However, these drawbacks can be decreased – if not compensated for – by some OpenGL-related libraries that allow high-level drawing, among them the OpenGL Utility Library (GLU), the Auxiliary Library, and the OpenGL Utility Toolkit (GLUT).

4.2 OpenGL-Related Libraries

In OpenGL, there are several libraries that contain a powerful and primitive set of rendering commands. All higher-level drawing can be done in terms of these commands. It allows the OpenGL programmers to simplify their programming tasks. These libraries were organized as header files; they can be included in any OpenGL applications, conveniently.

- *gl.h* – this module contains procedure declarations, constant definitions and macros for the OpenGL component. *gl* is the prefix of these procedure, for example, *glPushMatrix()*.

- *glu.h* – it includes procedure declarations, constant definitions and macros for the OpenGL Utility Library. The OpenGL Utility Library (GLU) includes some routines that use lower-level OpenGL commands to perform such task as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. These commands all use the prefix *glu*, such as, *gluCylinder()*.
- *glx.h* – it defines procedure declarations, constant definitions and macros for the OpenGL Extension Library for the X window System. For every window systems, the OpenGL Extension Library extents the functionality of that window system to support OpenGL rendering. In this library, GLX routines use the prefix *glX*, for instance, *glXCreateContext()*.
- *glut.h* – this header file is for the OpenGL Utility Toolkit (GLUT) Library. GLUT is a window system-independent toolkit. It has become a popular library for OpenGL programmers because it standardizes and simplifiers window and event management.

  First, the GLUT contains the commands for creating and opening windows and reading events from the keyboard or mouse. These routines enable the OpenGL programmers to simplify opening windows, detecting input, and so on. In addition, GLUT includes some commands that create more complicated three-dimensional objects, such as, a cube, a sphere, a cylinder, a torus, and so on. Using them, the OpenGL programmers can more easily complete their complex tasks. All GLUT commands start with the prefix *glut* such as *glutCreateWindow()*, *glutKeyboardFunc()*, *glutSolidSphere()*.

4.3 OpenGL Command Syntax

Generally, an OpenGL command is composed of the prefix *gl*, *glu*, or *glut* related the names of the OpenGL libraries, and initial capital letters for each word making up the command name, for example:

```
glClearColor();
gluCylinder();
glutDisplayFunc().
```

Additionally, some OpenGL commands can have different types or numbers of arguments. Such as the command *glColor*()*, it can be given three-float arguments, or an array with four-float elements, respectively. These two-type commands can be written as:

```
glColor3f(1.0, 1.0, 1.0);
GLfloat col_array[] = {0.0, 1.0, 0.5, 0.0};
glColor4fv(col_array).
```

Here the seemingly extraneous letters indicate the different argument types of these commands. The *f* part of suffix shows that the arguments are floating-point numbers. Some OpenGL commands can accept as many as eight different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are illustrated in table 4.1.

Suffix Data Type	OpenGL Type Definition
b	GLbyte
s	GLshort
i	GLint, GLsizei
f	GLfloat, GLclampf
d	GLdouble, GLclampd
ub	GLubyte, GLboolean
us	GLushort
ui	GLuint, GLenum, GLbitfield

Table 4.1: OpenGL command suffixes and argument data types

OpenGL is a state machine. Thus there are many state-value constants for managing and controlling the different kinds of states. OpenGL defined them begin with *GL*, *GLU*, or *GLUT*, use all capital letters, and use underscores to separate words. For instance, the state values for defining the window display mode are named as *GLUT_DOUBLE*, *GLUT_RGB*, and *GLUT_DEPTH*.

4.4 The Basic Structure of an OpenGL Application

OpenGL is a powerful graphics programming system. The programmers can do so many things with it, so an OpenGL program can be very complicated. However, the basic structure of an application program may be simple.

4.4.1 Initializing and Creating a Window

Before an OpenGL programmer draws any objects in OpenGL graphics system, the first step that should be done is to initialize and create a window for display the objects. The programmer must specify the characteristics of the window, and where the window should appear on the display screen. Usually, to complete this step, following procedures should be performed.

- Initialize the OpenGL library.
- Specify a display mode such as single-buffered or double buffered, or RGBA or colour-index.
- Specify the screen location for the upper-left corner of the window.
- Specify the size, in pixels, of the window.
- Create the window.

For these procedures, the main OpenGL commands include:

```
void glutInit();
void glutInitDisplayMode(unsigned int mode);
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
int glutCreateWindow(char* name);
```

4.4.2 Handling Window

After the window is created, the programmer should register some callback functions for handling the window. The roles of these callback functions mainly include following two aspects.

- Redraw the window and objects in the window when the window is popped and window damage is exposed, or when *glutPostRedisplay()* is explicitly called.
- Redisplay the window and redefine the projection matrix when the window is resized or moved.

And the related routines for handling window are as following (here *func* is the name of the related callback function):

```
void glutDisplayFunc(void (*func)(void));
void glutReshapeFunc(void (*func)(int width, int height));
```

4.4.3 Inputting events

To handle input events, OpenGL also call some callback functions. So the programmer must register these callback functions and specify what events which will input. These functions include:

- The function for handling the keys of keyboard.
- The function invoked when a mouse button is pressed or released.
- The function called when the mouse pointer moves within the window while one or more buttons is pressed.

To perform these callback functions for handling input events, following statements usually be used.

```
void glutKeyboardFunc(void (*func)(unsigned int key, int x, int y));
void glutSpecialFunc(void (*func)(unsigned int key, int x, int y));
```

*void glutMouseFunc(void (*func)(int button, int state, int x, int y));*
void GlutPostRedisplay(void);

4.4.4 Drawing Objects

OpenGL provides a set of geometric primitives, including points, lines, and polygons such as cube, sphere, cylinder, and so on. Using these routines, a programmer can build up any desired objects, either two-dimensional or three-dimensional. And the programmer can use smooth shading to draw a single polygon with more than one colour, or render illuminated objects by defining the desired light sources and lighting model. And he can also blend colours to achieve such effects as making objects appear translucent, smooth jagged edges of lines and polygons with antialiasing, or create scenes with realistic atmospheric effects.

One of the most exciting things in graphics field is draw pictures that move. Of course, the programmer can also draw any movable pictures on computer display screen, which is three-dimensional animation. OpenGL provides some routines for displaying smoothly animated graphics. One of the goals of this Major Report is to achieve some simple colour animation pictures on screen.

4.4.5 Managing a Background Process

There is another callback function in OpenGL library. It will be executed if no other events are pending. This is particularly useful for continuous animation or other background processing. For performing this callback function, call the OpenGL routines: *void glutIdleFunc(void (*func)(void));*.

4.4.6 Running the Program

After all the above setup is completed, the very last thing which must be done in an application program is call the OpenGL function *void glutMainLoop(void)*. And only till now, all windows that have been created are shown, and rendering to those windows is effective. And event processing begins, and the registered display callback is triggered. Once this loop is entered, it is never to return.

4.5 OpenGL and OO Framework

OpenGL is one of the widely used graphics languages. It is same as most of these graphics languages, simple use of OpenGL functions rends to lead to badly structured programs with many global variables. In another word, there are many global variables in an OpenGL application program. As mentioned above, OpenGL uses callback functions to handle windows and input events for objects. Since the parameters and result type of a callback function are predefined by OpenGL itself, the easiest way to provide communication between callback functions is to use global variables. This makes it difficult to implement a new graphics program and maintain an exist one.

However, with the appearance of the object-oriented programming language, and the application of framework, it is becoming possible to combine an OOPL such as C++ with OpenGL. And it is possible to eliminate most of the global variables and construct programs which have better encapsulation and are easier to read. Of course, since the characteristic of OpenGL, it is difficult to let all global variables disappear completely in an application. But the number of the global variables can at least be kept as small as possible.

Based on the above idea, this Major Report will develop an object-oriented framework: two simple animation systems implemented with C++ in conjunction with OpenGL. The goals of the framework are to reduce the use of global variables, which will be replaced by global objects, and to reduce the use of OpenGL commands.

Chapter 5: OO Programming Language C++

This section gives an overview of the object-oriented programming language C++. C++ was invented at AT&T Bell Labs by Bjarne Stroustrup. Dr. Stroustrup was an enthusiastic user of Simula, which is a simulation language that had many object-oriented capabilities. When it became necessary for him to write projects in C, he missed the abstract data type and other OOP features from Simula. So he created *C with Classes* that came to be known as C++.

C++ can be viewed just as C with classes. A class is the C++ term for an abstract data type. Of course, C already has a way of aggregating data into a complex type: the *struct*. A class is a *struct* with a few more features.

5.1 Why select C++

Graphics programming can be done with many high-level computer languages. There are, however, several reasons why choose C++:

- C++ is highly portable and efficient. It is almost a superset of C.
- C++ compilers are available on any computer platform.
- OpenGL is written in C and C++, respectively. This makes it easy to include OpenGL code.
- C++ is very suitable for geometric thinking. It allows overwriting operators; e.g., the union of two geometric primitives can be overwritten by the plus-sign. So, if p is a point and G is a straight line, and P is the plane that connects p and G, this can be written very clearly by P = p + G.

5.2 Modules of C++ Programming Language

This section is a fairly intense tour through the main fundamental features of C++ language, some of which are inherited from and in common with C.

5.2.1 Structure of a C++ Application

A C++ program is a collection of variables, function definitions, and function calls. In General, the source code of a large program package is split up into several files file1.cpp, file2.cpp, file3.cpp, etc. Normally, one of these files contains the *main()*-function. When the program starts, it executes initialization code and calls the special function "*main()*." This concept, inherited from C, has been true for decades.

5.2.2 Data Types

Data types define the way using storage in the application programs. In C++, there are two kinds of data types; they are built-in and abstract data type. A built-in data type is one that the compiler inherently understands and is wired directly into the compiler. These built-in data types are almost exactly the same in C and C++. An abstract data type, that is user-defined data type, is a class that a programmer creates.

5.2.2.1 Basic Built-in Data Types

In the standard C and C++, there are only four basic built-in data types. An *int* stores an integral number and uses a minimum of two bytes of storage. A *char* is for character storage and uses a minimum of 8 bits of storage. A *float* is for single-precision floating point. And a *double* is for double-precision floating point. The float and double types are usually in IEEE floating-point format.

In the standard C++, there is another built-in data type: *bool*. The *bool* type can have two states expressed by the built-in constants *true* (which converts to an integral one) and *false* (which converts to an integral zero).

There are four specifiers: *short*, *long*, *signed*, and *unsigned*; they can modify the meanings of the basic built-in types and spread them out to a much larger set. The *long* and *short* modify the maximum and minimum values that a data type hold. The signed and unsigned specifies tell the compiler how to use the sign bit with integral types. For instance, *short int*, *long int*, and *long double*, they expand the size of integral type and floating point numbers, respectively. Moreover, an *unsigned short int* number does not keep the track of the sign and thus has an extra bit available, so it can store positive numbers twice as large as the positive numbers that can be stored in a *signed short int* number.

OpenGL defines its own basic data types, essentially starting with capitalized *GL*, followed by the intended types. The table 1 illustrates the relationships between the C++ built-in data types and OpenGL data types.

OpenGL data type	C++ built-in data type
GLenum	unsigned int
GLboolean	unsigned char
GLbitfield	unsigned int
GLbyte	signed char
GLshort	short
GLint	int
GLsizei	int
GLubyte	unsigned char
GLushort	unsigned short
GLuint	unsigned int
GLfloat	float
GLclampf	float
GLdouble	double
GLclampd	double
GLvoid	void

Table 5.1: The relationship between OpenGL and C++ built-in data types

5.2.2.2 User-defined Data Type

The programming language C, and of course also C++, allows programmers to define new data types by the means of syntax:

typedef <predefined type> <new type> [<optional dimensions>].

For example, the statement

typedef Glfloat Real;

means a new data type *Real* has been defined. It can be used same as any other built-in data types.

Another type of user-defined data type in C and C++ is defined by the keyword *struct*. In fact, a *struct* is a way to collect a group of variables and even functions into a structure. This is often called encapsulation, too. Once a *struct* is created, the programmer can make many instances of this new type of variable.

The concept of abstract data types which also refers to as user-defined data types is a basic concept in Object-Oriented programming. And it is one of the important features of C++ program language. It allows programmers to create a new data type, that is class.

5.2.3 Access Control in C++

In a class of C++ application, any variables and methods can be modified by *private*, *public*, or *protected*. Their use and meanings are remarkably straightforward. They change the limit for all the declarations that follow them.

Public means all member declarations that follow are available to everyone. The *private* keyword means that no one can access that member except the creator of the type, inside function members of that type. *Private* is a brick wall between the creator and the client programmer. If someone tries to access a *private* member, they will get a compile-time error. The protected keyword acts just like private as far as this class user is concerned, but available to anyone who inherits from this class. Figure 5.1 shows the relationships of the access control.

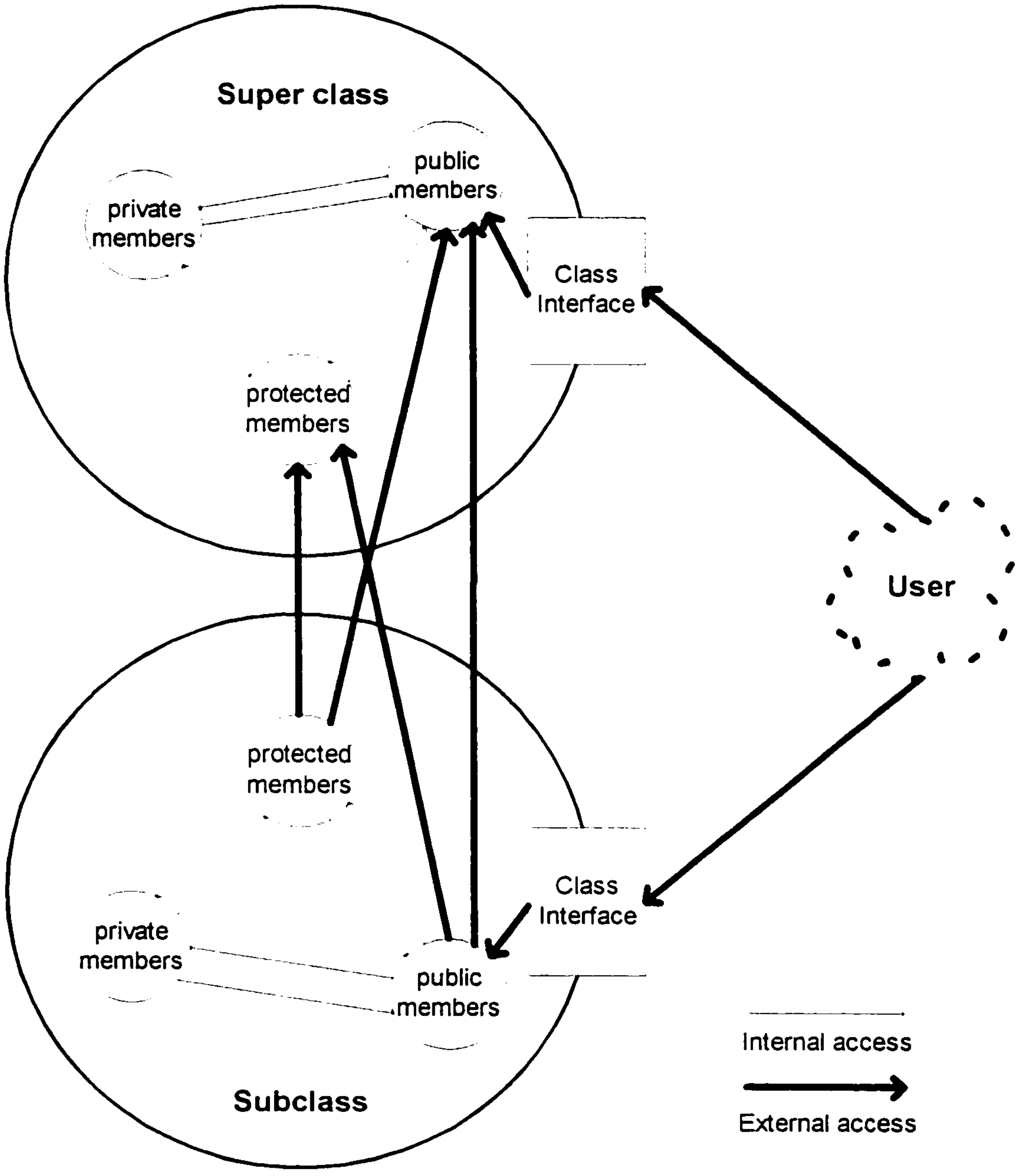


Figure 5.1: Access control in C++

5.3 OO Features in C++

C++, based on the C programming language, is one of the most widely used object-oriented programming languages. Although it does not belong to the purest OOPL, there are many object-oriented characteristics developed in C++. They also construct the cornerstones of C++ programming language. These features are known as *abstract data type*, *inheritance*, *polymorphism*, *templates*, and *exception handling*. Here will give an overview for these essential features.

5.3.1 Data Abstraction

Data Abstraction is one of the fundamental of C++; and it is one of the basic concepts in object-oriented design and development. In C++, the abstract data type was developed from the type *struct* in C. The C++ programmers place functions inside *struct*, thus package data and functions together. That builds up the new data type, which is referred to as encapsulation in object-oriented development. Variables created using the abstract data type are called *objects*, or *instances*, of that type. And calling a member function for an object is called *sending a message* to that object. While the primary action in object-oriented programming is sending messages to objects.

In C++, access specifies (stated above) are part of the structure *struct* and don't affect the objects created from the structure. They establish what the client programmers can and can't use, and separate the interface from the implementation. Figure 5.2 shows an example of the *struct* in C and C++.

`struct Shap { int x, y, z; }`	`struct Shap { private: int x, y, z; public: int getPosition(); void move(); }`
struct in C	struct in C++

Figure 5.2: The struct type in C and C++

In the original OOP language, Simula-67, the keyword *class* was used to describe a new data type. This apparently inspired Stroustrup to choose the same keyword for C++, to emphasize that this was focal point of the whole language: the creation of new data types that are more than just C *struct* with functions. In fact, the keyword *class* in C++ is identical to the *struct* keyword in absolutely every way except one: *class* defaults to *private*, whereas *struct* defaults to *public*. However, it is just because of the introduction of the data abstract, access specifiers, and class keyword, the C++ programming language becomes one of the object-oriented domains.

5.3.2 Inheritance and Composition

One of the most compelling features about C++ is code reuse. C++ programmers reuse code through creating new classes, in detail using existing classes that someone else has built and debugged.

There are two ways to use the classes without spoiling the existing code in C++. One is class inheritance. Using it, a new class can be created as a type of an existing one. It is *is-a* relationship between the subclass and the parent class. With inheritance, the internals of parent classes are often visible to subclasses. This style of reuse is often referred to as white-box reuse. Inheritance in C++ may be single or multiple.

The second approach for reusing code in C++ is object composition, which is an alternative to class inheritance. Here, new class is obtained by assembling or composing objects to get more complex functionality. In another, the programmers reuse existing types as part of the underlying implementation of the new type. Object composition requires that the objects being composed have well-defined interfaces. It built the has-a relationship between the new type and the existing type. This kind of reuse is known as black-box reuse, because no internal details of objects are visible.

5.3.3 Polymorphism and Virtual Functions

Polymorphism is the third fundamental feature of C++ after data abstraction and inheritance. It provides another dimension of separation of interface from implementation. Polymorphism allows improved code organization and readability as well as the creation of extensible programs.

Polymorphism is implemented with virtual functions, which can cause late binding or dynamic binding to occur for a particular function. Virtual function means "different form". It allows the application of the different versions of one interface. C++ requires that the programmers use the *virtual* keyword when declaring the function in the baseclass. An example is shown in Figure 5.3. Polymorphism and dynamic binding are all basic and important concepts in object-oriented programming. It can be said that the C++ programmer doesn't understand OOP yet if he cannot use virtual functions well in his programs. It is just because of the application of virtual and pure virtual functions in C++ programs, the frameworks can be applied more widely.

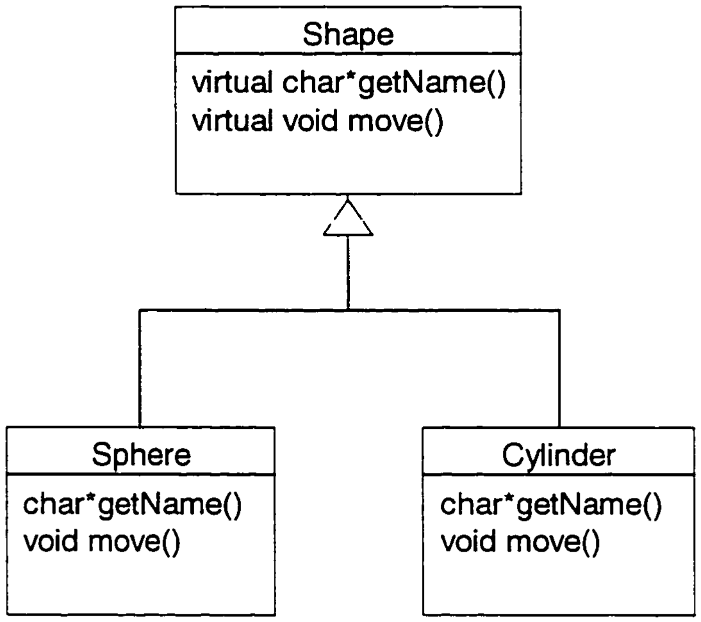
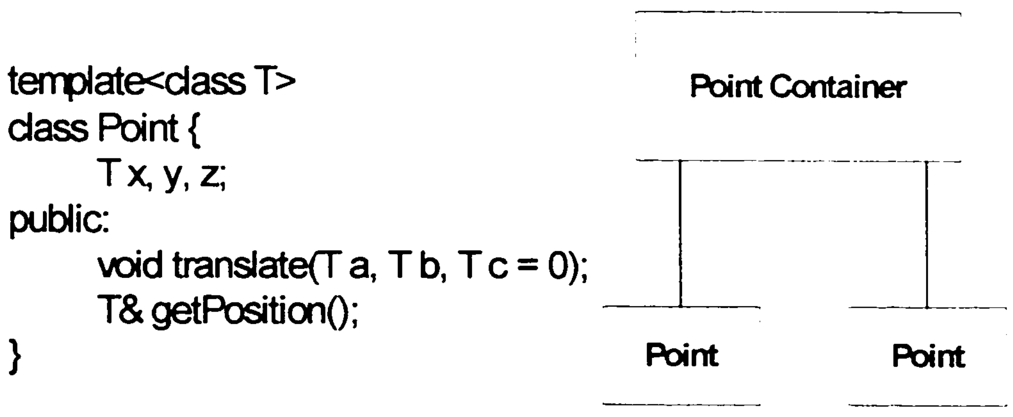


Figure 5.3: Virtual function in C++

5.3.4 Templates

Inheritance and composition provide a way to reuse object code. While the C++ template feature provides a way to reuse source code. In C, in order to reuse source code, the programmer copy the source code for a new program and make modifications by hand. This can certainly introduce new errors in the process. To solute this problem, Stroustrup developed a new technique in C++, that is template classes. This kind of classes no longer holds a generic base class called object; but instead it holds an unspecified parameter. When a programmer wants to use a template class, the parameter is substituted by the compiler. For example, the template class *Point* shown as Figure 5.4. Using it, the client user can create an *integer*-type *Point* generally in 2-dimention graphic system; and he can make a *float*-type *Point* usually in 3-dimention graphic system.



```
template<class T>
class Point {
      T x, y, z;
public:
      void translate(T a, T b, T c = 0);
      T& getPosition();
}
```

Figure 5.4: Template class in C++

5.3.5 Exception Handling

Exception handling is one of the most concepts in object-oriented development. And it is one of the five cornerstones of C++. In fact, error recovery is a fundamental concern for every program, and it's especially important in C++. The goals for exception handling in C++ are to simplify the creation of large, reliable programs using less code than currently possible, with more confidence that the application doesn't have an unhandled error. This is accomplished with little or no performance penalty, and with low impact on existing code.

The implementation of exception handling in C++ is using the *try* and *catch* block, and *throw* statement. Here's a small example that shows a simple use of the features for exception handling:

```
#include <exception>
#include <iostream>
#include <cstdlib>
#include <cstring>

class First {};
class Second {};
//void g();

void f(int i) throw (First, Second) {
  switch(i) {
    case 1: throw First();
    case 2: throw Second();
  }
  //g();
}

// void g() {} // Version 1
// void g() { throw 47; } // Version 2

int main() {
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    } catch(First) {
      cout << "Class First caught" << endl;
    } catch(Second) {
      cout << "Class Second caught" << endl;
    }

}
```

Chapter 6: The OpenGL Frameworks

Given an application software system, a framework can be constructed and then object-oriented concepts applied to develop the system in an object-oriented methodology. But do graphic systems have enough common features for a framework? How can the graphic systems be developed in object-oriented framework? To answer these questions, two simplified three-dimensional animations are designed and implemented through combining C++ programming language with OpenGL software system, the car running along a round track and the planet with a moon. They provide the examples for combining the object-oriented development with a graphics software system.

6.1 The Architecture of a Graphics System

An object-oriented framework is a collection of concrete and abstract classes. In the graphics system combining C++ programming language with OpenGL, a framework is designed and constructed from the C++ standard library and the OpenGL standard library. It constructs the application library in this application domain. In order to establish an application system, the programmer inherits the abstract classes and concrete classes, or composes the concrete classes in the application library, and/or uses classes in the C++ standard library and commands in the OpenGL libraries.

Figure 6.1 illustrates the architecture of the graphics domain combined C++ and OpenGL software.

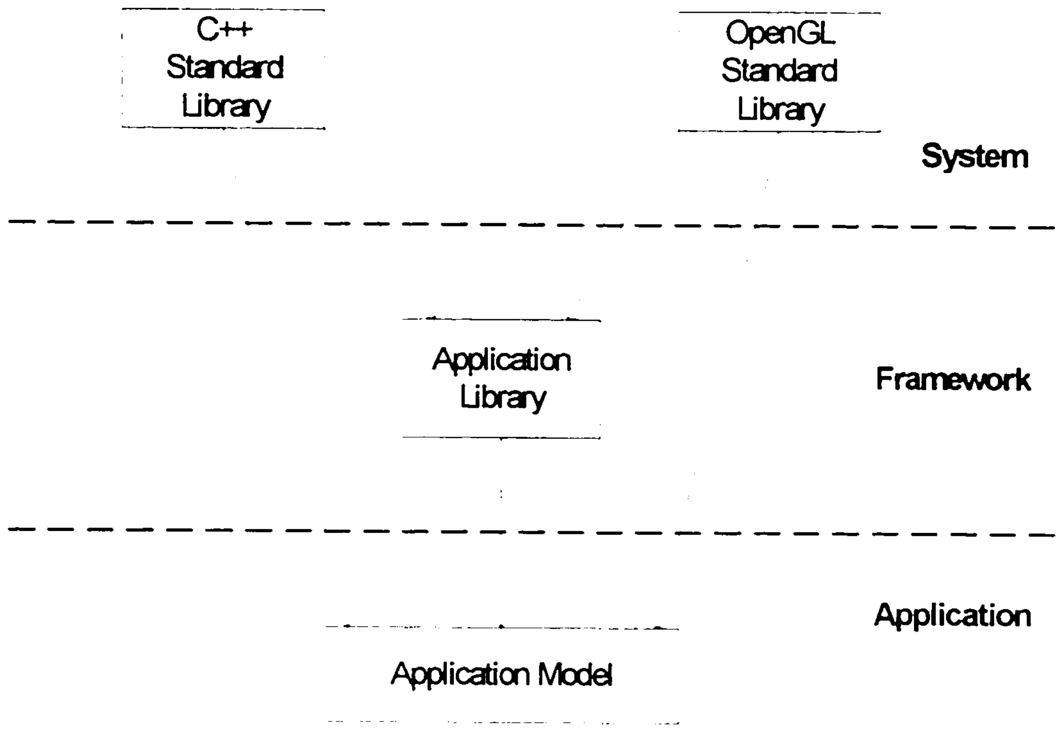


Figure 6.1: The architecture of a graphics system

6.2 The Design of the Application Library

6.2.1 The Abstract Classes Design

6.2.1.1 The Abstract *Object* Class

In any graphics systems, no matter how complex it is, they are all structured by the basic graphics elements – such as points, lines, sphere, or cylinder, and so on – that is a shape. Although we don't know what a real shape is and how it can move, it can be imagined that a shape has a name, has a place, may be colour, can be rotated or scaled, and so on. Thus, we can design a shape class *Object* as an abstract class. It will be the base class in our next two animation systems. Following is the structure of the abstract class.

```
class Object:
protected attributes:
        char* name                                  // Object's name.
        GLfloat x, y, z                             // Object's position.
        GLfloat r, g, b                             //RGB color of object
public methods:
        Object(char* inam = "Object")
        ~Object()

        virtual void draw() = 0                     //pure virtual function
        char* get_name()
        virtual void setPosition(GLfloat xx, GLfloat yy, GLfloat zz)
        virtual void setRGB(GLfloat rr, GLfloat gg, GLfloat bb)
        virtual void setRGB(Color col)
        virtual void translate()
        virtual void rotate(GLfloat angle, GLfloat x0, GLfloat y0, GLfloat z0)
        virtual void scale(GLfloat x1, GLfloat y1, GLfloat z1)
```

6.2.1.2 The Abstract *KeyReaction* Class

In a graphics system, the users generally control the animation through the keyboard and mouse. So we design another abstract class: *KeyReaction*. This class has a *pure virtual* method which will be inherited by the users for stating the functions of every keys and mouse buttons. The members of the class are as following.

```
class KeyReaction:
attributes:
public methods:
        KeyReaction()
        ~KeyReaction()

        virtual void explain() = 0
        virtual void setKeys(unsigned char key)
        virtual void setFunctionKeys(unsigned char key)
        virtual void setMouseButton(unsigned char button)
```

6.2.2 The Other Base Classes in Application Library

The other base classes in application library include following classes.

- Vector: the basic class in this framework. It can create two-dimensional or three-dimensional vectors for the graphics system.
- Point: the basic class of this domain. It states a point in two-dimensional or three-dimensional graphics system.
- Color: another basic class in the system. It defines the RGB color model for the graphics system.
- Window: the basic class for this domain. It defines the needed features of a window, such as position, color, size, and how to create a window using the OpenGL statements.
- Camera: the basic class in this domain. It defines a simulation camera for a graphics system. The user can change the position and target of the camera through its interface methods.

6.3 The Running Car Animation

6.3.1 Requirement Analysis

This system is a three-dimensional animation about a car running along a round track. It is based on an example provided by Dr. Peter Grogono. The car is simply made up of quadric surfaces. It includes a car body, which is a tapered cylinder and disks, and four wheels with two axles. This system also has a circular track for the car driving on it, and grass ground inside the track. And several stimulated houses dot the landscape.

Through keyboard, the users of this animation can change the viewpoints which includes from a static point far from the whole animation, from a static point inside the car, from a static point outside the car, from the driver's point, from a point driving around while looking at a house, from the point looking backwards from another car, from the point beside the car, from a balloon fixed over the scene, and from a helicopter flying around the scene. And they can control the bumpiness and fog, and their increase and decrease.

6.3.2 Classes Design

In an object-oriented framework, everything is an object. The objects with the same attributes form a class. The methods are encapsulated in the class invocation of methods by passing message to objects. From the requirement of the running car system, we design the classes as following. They will inherit or compose the classes in the application library and system libraries.

- Disk: one of the basic classes. It specifies the attributes and methods related with drawing a disk.
- Cylinder: the basic class in this domain. It defines member data and functions for drawing the object cylinder.
- House: an application class in this domain. Using it, a house of any size simply constructed with walls, door, and roofs can be drawn.
- Car: another application class. It specifies how to draw a simple car using some disks and cylinder.
- Keyboard: an application class. It specifies how to control the animation using keys and function keys on keyboard.
- Mouse: an application class. It specifies how to control the animation using the mouse buttons.

Figure 6.2 illustrates the class diagram for the running car animation system.

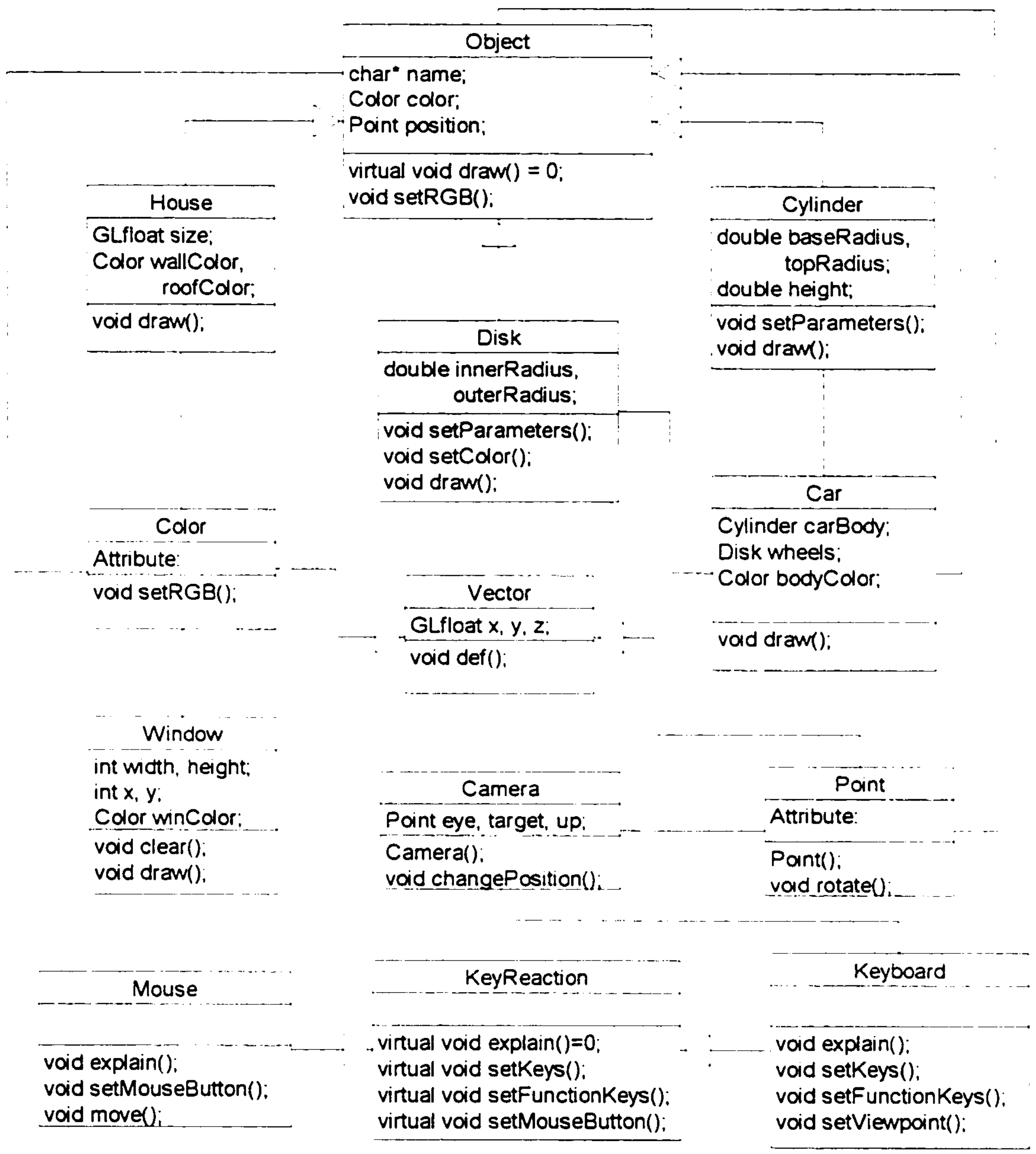


Figure 6.2: The class diagram of the running car system

6.4 The Planet and its Moon System

6.4.1 Requirement Analysis

The planet system is made up of one planet and its only one moon. The planet and the moon are represented using sphere.

The users can control the system rotate round the X, Y, and Z axis through the keyboard. And they can also change the viewpoint between forward, upward, leftward, and rightward. Using the mouse buttons, the user may switch the light effect like a sun, and can let the sun rotate to up, down, left, or right. And zoom in and zoom out are need for the window in this animation.

6.4.2 Classes Design

This system is designed by reusing the above object-oriented framework. Several basic classes will be reused without any change in here. They are Vector, Point, Color, Object, Camera, Window, KeyReaction, Mouse, and Keyboard. From the problem, other needed classes are added as following.

- Sphere: the basic class in this system. It specifies the attributes and methods for drawing a color sphere.
- LightSouce: another basic class for graphics domain. The users can control the light effect in a three-dimensional animation system using this class.

Figure 6.3 illustrates the class diagram for the planet and its moon animation system.

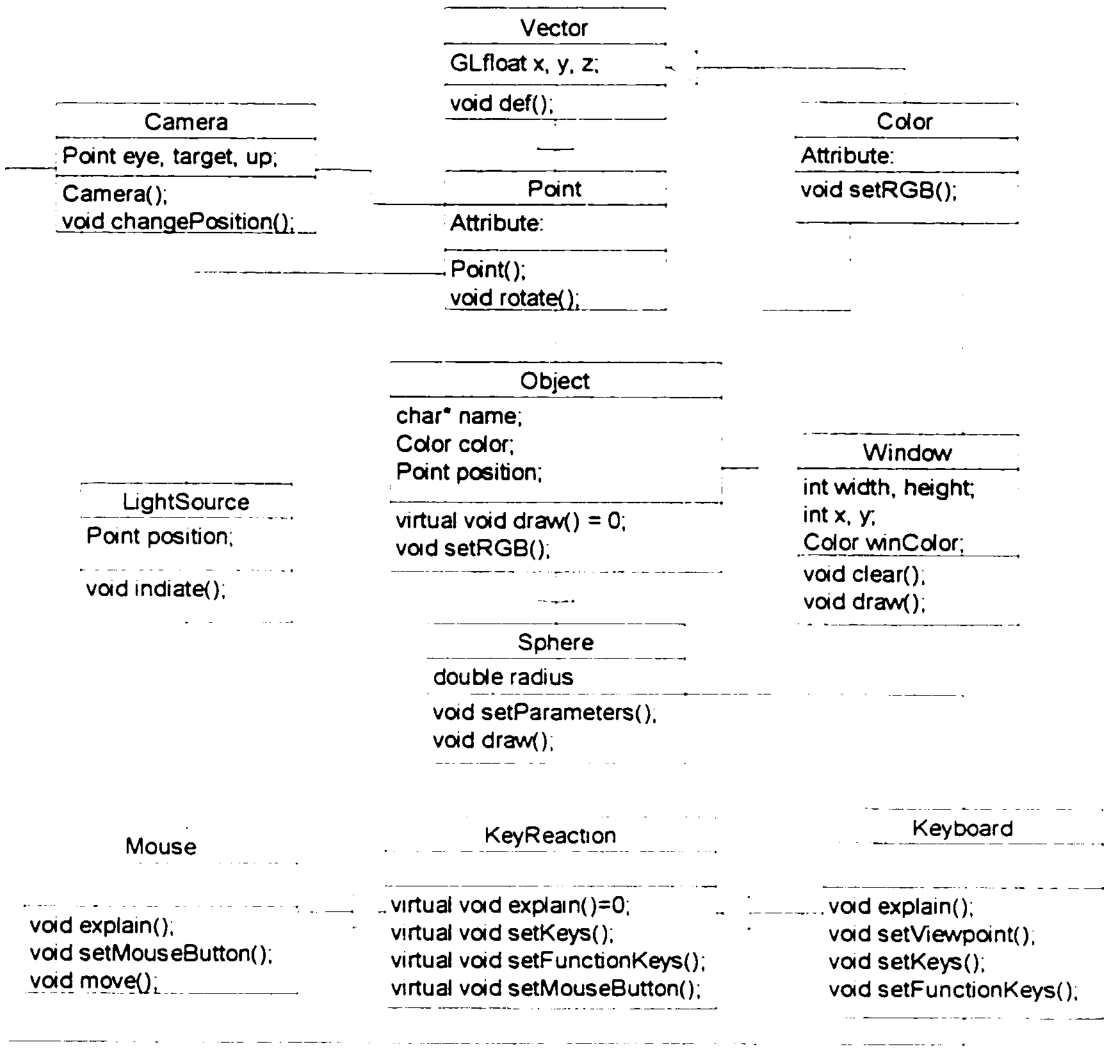


Figure 6.3: The class diagram of the planet and its moon system

Chapter 7: System Implementation

This chapter will state the main structures of the two animation systems, including the detail members of the classes in the framework and the application models. The source codes of them will be given in appendix B.

7.1 The Basic Classes and header Files

7.1.1 Some Needed Header Files

In order to reduce the use of the statements and type definitions of OpenGL software, we firstly specify several header files for our application.

7.1.1.1 types.h

In this header file, some common types of OpenGL are redefined for the user to use them conveniently. They are as follows:

```
#define Real Glfloat
#define Double Gldouble
#define QuadricObj GLUquadricObj
```

7.1.1.2 specialkeys.h

Same as the file types.h, here gives the redefines for the OpenGL function keys. Thus we can reduce to use the original definition in the system libraries.

```
#define LEFTARROWKEY        GLUT_KEY_LEFT
#define RIGHTARROWKEY       GLUT_KEY_RIGHT
#define DOWNARROWKEY        GLUT_KEY_DOWN
#define UPARROWKEY          GLUT_KEY_UP
#define ENDKEY              GLUT_KEY_END
#define PAGEUPKEY           GLUT_KEY_PAGE_UP
#define PAGEDOWNKEY         GLUT_KEY_PAGE_DOWN
```

```
#define F1KEY         GLUT_KEY_F1
#define F2KEY         GLUT_KEY_F2
#define F3KEY         GLUT_KEY_F3
#define F4KEY         GLUT_KEY_F4
#define F5KEY         GLUT_KEY_F5
#define F6KEY         GLUT_KEY_F6
#define F7KEY         GLUT_KEY_F7
#define F8KEY         GLUT_KEY_F8
#define F9KEY         GLUT_KEY_F9
#define F10KEY        GLUT_KEY_F10
#define F11KEY        GLUT_KEY_F11
#define F12KEY        GLUT_KEY_F12

#define LEFTBUTTON            GLUT_LEFT_BUTTON
#define MIDDLEBUTTON          GLUT_MIDDLE_BUTTON
#define RIGHTBUTTON           GLUT_RIGHT_BUTTON

#define BUTTONDOWN            GLUT_DOWN
```

7.1.2 Basics Classes

7.1.2.1 Class Vector

This is basic class for a graphics domain. It defines and implements the related data and operation members for setting a two-dimensional vector or a three-dimensional vector.

```
Public attributes: Real x, y, z;
Public operations: Vector();
                   Vector(Real x0, Real y0, Real z0 = 0.0);
                   ~Vector();

                   Real getX();
                   Real getY();
                   Real getZ();
                   void setX(Real x0);
                   void setY(Real y0);
                   void setZ(Real z0);
                   void operator() (Real x0, Real y0, Real z0);
                   void def(Real x0, Real y0, Real z0);
```

7.1.2.2 Class Point

This class inherits the class Vector. It can set the coordinate level for any point instance in two-dimensional or three-dimensional graphics system.

```
Attributes:

Public operations: Point( );
                   Point( Real x0, Real y0, Real z0 = 0.0 );
                   Point( Vector &v );
                   ~Point();

                   void rotate( Real angle_in_deg, const Vector &axis );
                   void rotate( Real angle_in_deg, Real dx, Real dy, Real dz );
                   void translate( Real dx, Real dy, Real dz );
                   void translate( const Vector &v );
                   void scale( Real kx, Real ky, Real kz );
```

7.1.2.3 Class Color

This is a subclass inherited from the class Vector publicly. It defines the RGBA (red, green, blue, and alpha) color-display model, which is very commonly used in a graphics system.

```
Attributes:

Public operations: Color();
                   Color(Real r, Real g, Real b, Real a = 0.0);

                   void def() (Real r0, Real g0, Real b0, Real a0 = 0.0);
                   void setRGB();
```

7.1.2.4 Class Object

This an important abstract class in our framework. It defines a pure virtual operation and some other virtual functions which can be inherited by the users of it.

```
Protected attributes: char* name;
                   GLfloat x, y, z;
                   GLfloat r, g, b;
Public operations: Object(char* inam = "Object")
```

```
~Object()

virtual void draw() = 0;
char* get_name();
virtual void setPosition(GLfloat xx, GLfloat yy, GLfloat zz);
virtual void setRGB(GLfloat rr, GLfloat gg, GLfloat bb);
virtual void setRGB(Color col);
virtual void translate();
virtual void rotate(GLfloat angle, GLfloat x0, GLfloat y0,
            GLfloat z0);
virtual void scale(GLfloat x1, GLfloat y1, GLfloat z1);
```

7.1.2.5 Class Camera

This class simulates the camera in real world. Through the instance of this class, a user can define the position of the camera and its target, and can change them at anytime.

```
Private attributes: Point eye, target, up;

Public operations: Camera();
                   Camera(Real ex, Real ey, Real ez, Real cx, Real cy, Real cz,
                              Real ux, Real uy, Real uz);
                   ~Camera() { }

                   void changePosition( const Point &newEye );
                   void changePosition( Real eye_x, Real eye_y, Real eye_z );
                   void changeTarget( const Point &newTarget );
                   void changeTarget( Real target_x, Real target_y,
                              Real target_z );

                   void changeUpDirection( const Point &newUp );
                   void changeUpDirection( Real up_x, Real up_y,
                              Real up_z );

                   void view();
```

7.1.2.6 Class Window

The Window class is a very common class in a graphics system, because any objects must be display in a window in computer system. In this class, the related attributes and operations are defined and implemented for setting a window. A use can conveniently create a needed window for his application.

```
Private attributes: int width;
                    int height;
                    int x, y;
                    char* title;
                    Color winCol;
                    Real alpha;
Public operations:  Window(int w = 600, int h = 400, int ix = 10,
                             int iy = 10, char* t = "");
                    Window(char* t);
                    ~Window();

                    void setSize(int w, int h);
                    void setPosition(int xx, int yy);
                    void setBGColor(Real br, Real bg, Real bb, Real ba = 1.0);
                    void setBGColor(Color c);
                    void draw();
                    void clear();
```

7.1.2.7 Class KeyReaction

Class KeyReaction is an abstract class in our framework. It sets the virtual operations for controlling animation through keyboard or mouse. The programmer of a graphics domain should inherit this class and establish himself subclasses and implement them in his application model.

```
Attributes:
Public operations:  KeyReaction();
                    ~KeyReaction();

                    virtual void explain() = 0;

                    virtual void setKeys(unsigned char key);
                    virtual void setFunctionKeys(unsigned char key);
                    virtual void setMouseButton(unsigned char button);
```

7.2 The Running Car Animation Implementation

7.2.1 Class implementation

7.2.1.1 Class Disk

Class Disk is a subclass of Object. Using it, the user can draw the basic color graphic disk that has an inner radius and an outer radius. For example Figure 7.1 is a light green disk, which the inner radius is 40.0 and the outer radius is 60.0.

```
Private attributes: QuadricObj *dobj;
                    Double innerRadius, outerRadius;
                    int slices, rings;
Public operations:  Disk();
                    Disk(char* dn);

                    void setParameters(QuadricObj *p, Double ir, Double or,
                              int sl, int rg);
                    void setColor(const Color &col);
                    void draw();
```

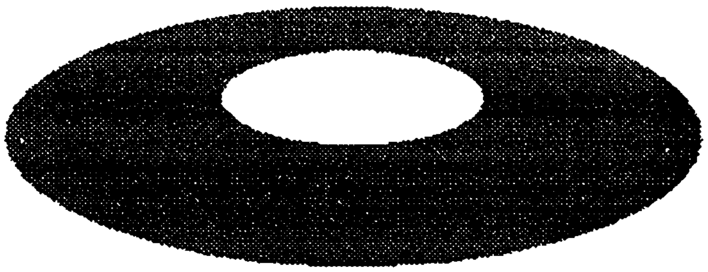


Figure 7.1: An example of disk

7.2.1.2 Class Cylinder

This is another subclass of class Object. It is for drawing the basic color graphic cylinder which basic elements include the base radius, top radius, and height.

```
Private attributes: QuadricObj *cobj;
                    Double baseRadius, topRadius, height;
                    int slices, stacks;
Public operations:  Cylinder(char* dn);
                    ~Cylinder() {}

                    void setParameters(QuadricObj *p, Double br, Double tr,
                             Double h, int sl, int st);
                    void setColor(const Color &col);
                    void draw();
```

7.2.1.3 Class Car

Class Car is also a subclass of the Object. It is simply made up of a cylinder as the car body, two cylinders as the axles, and four disks as its wheels. Figure 7.2 gives an example of this kind of car.

```
Private attributes: Cylinder carBody;
                    Disk bodyFront, bodyRear;
                    Cylinder frontAxle, rearAxle;
                    Disk frontLeftWheel, frontRightWheel;
                    Disk rearLeftWheel, rearRightWheel;
                    Color bodyColor, axleColor, wheelColor;
                    Real length;
                    Real frontRadius, rearRadius;
                    Real axleRadius;
                    Real wheelRadius;
                    QuadricObj *p;
Public operations:  Car(char *nm);
                    ~Car() {}

                    void setBodyColor(Color c);
                    void setAxleColor(Color c);
                    void setWheelColor(Color c);
                    void setStartPosition(Real x0, Real y0, Real z0);
                    void setStartPosition(Vector &pos);
                    void setBodyParam(Real l, Real fr, Real rr);
                    void setAxleRadius(Real ar);
                    void setWheelRadius(Real wr);
                    void setQuadricObj(QuadricObj *p1);
                    void draw();
```

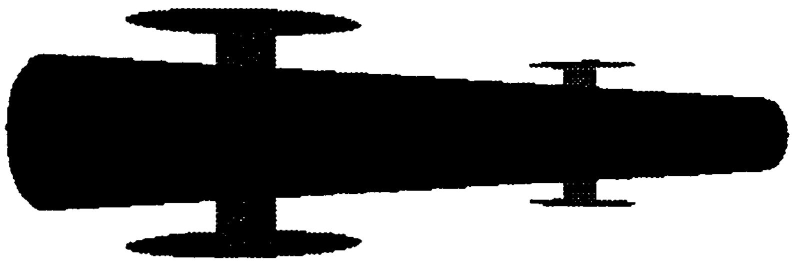


Figure 7.2: An example of Car

7.2.1.4 Class House

The class House inherits the members from the class Object. It draws the simply house with four walls, one door, and two roofs. The user can define any size of this kind of house. Figure 7.3 illustrates an example of it.

```
Private attributes: Real size;
                    Color wallColor, doorColor, roofColor;
Public operations:  House( char* nm );
                    ~House();

                    void setPosition( Real x0, Real y0, Real z0 = 0.0 );
                    void setPosition( Point &sp );
                    void setSize( Real sz );
                    void setWallColor( Color &wc );
                    void setDoorColor( Color &dc );
                    void setRoofColor( Color &rc );
                    void draw();
```

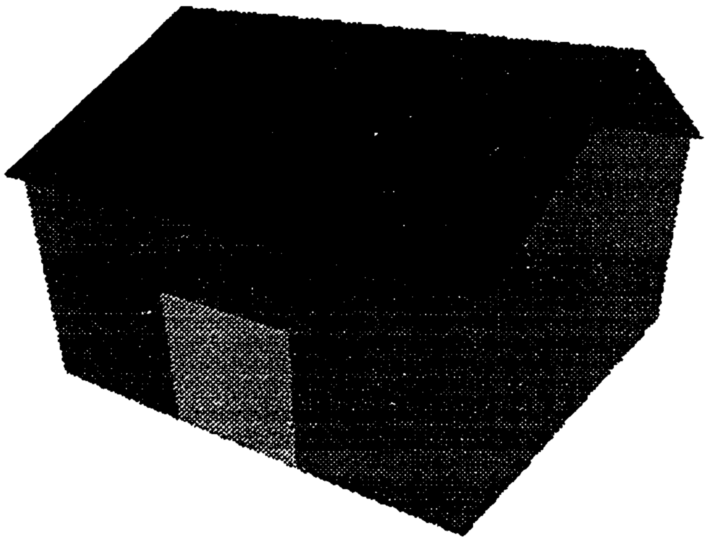


Figure 7.3: An example of House

7.2.1.5 Class Mouse

This class is a subclass of the KeyReaction. It defines the operations for controlling the mouse. Because in a graphics system the operations of an instance of the class Mouse should relate many other objects in the system, most of them must be implemented with the graphics application model.

```
Attributes:
Public operations: Mouse();
                   ~Mouse();

                   void explain();

                   void setMouseButton(unsigned char Button);
                   void move(int x, int y);
```

7.2.1.6 Class Keyboard

Class Keyboard is another subclass of class KeyReaction. It specifies the needed operations for controlling an animation system through the keyboard, including keys and function keys. Because of the same reasons as the class Mouse, most of operations must also be implemented with the graphics application model.

```
Attributes:
Public operations: Keyboard();
                   ~Keyboard();


                   void explain();
                   void setViewpoint();
                   void setKeys(unsigned char key);
                   void setFunctionKeys(unsigned char key);
```

7.2.2 Application Model Implementation

In this application domain, we implement a three-dimensional animation which is a car running along a round road. Inside the circle trace, there is a piece of grass dotted with several houses. The user can set ten different viewpoints through the keyboard:

1. setting the viewpoint on a static distant point; and this is the default display;
2. setting the viewpoint inside the track, respectively; and the camera follows the car;
3. setting the viewpoint outside the track, respectively; and the camera follows the car;
4. putting the viewpoint on the place of driver;
5. setting the effect same as the driver is staring at one house;
6. setting the viewpoint on another car driving before the car, and the man in that car is observing backwards;
7. placing the viewpoint beside the car and is viewing the car;
8. setting the viewpoint on a balloon drifting over the area; moving the mouse can change the height of the balloon;
9. setting the viewpoint on a helicopter flying in the sky; moving the mouse can change the height of the helicopter.
10. setting a random viewpoint of one to nine above.

Figures from 7.4 to 7.8 illustrate some effects of these viewpoints.

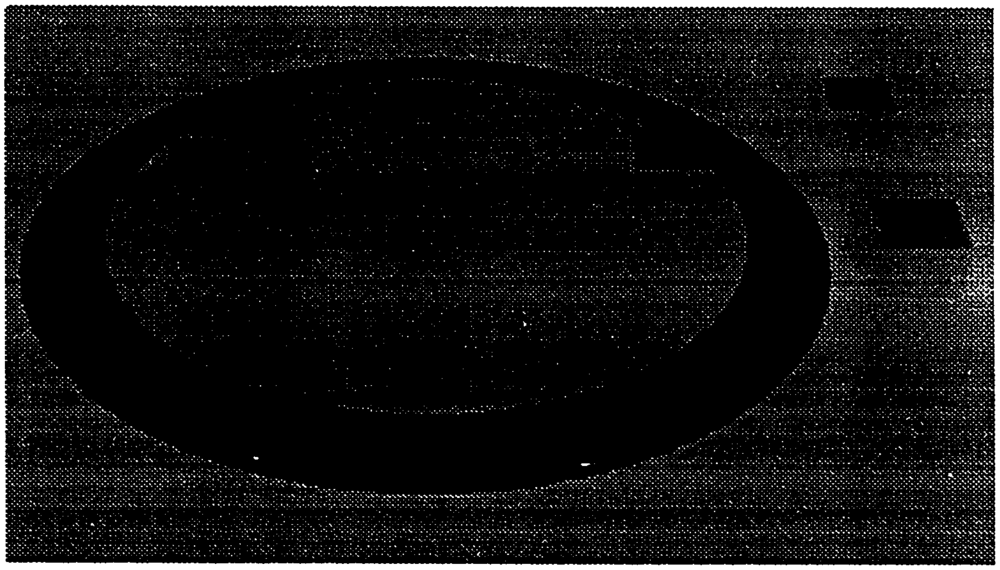


Figure 7.4: Viewing from a distant point

Figure 7.5: Viewing from the driver

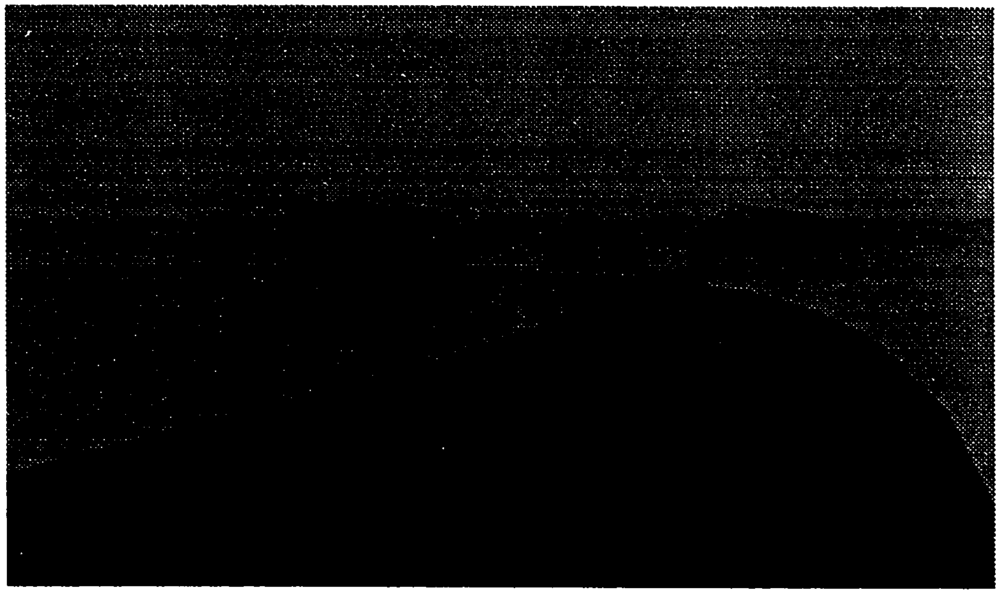


Figure 7.6: Viewing from another car

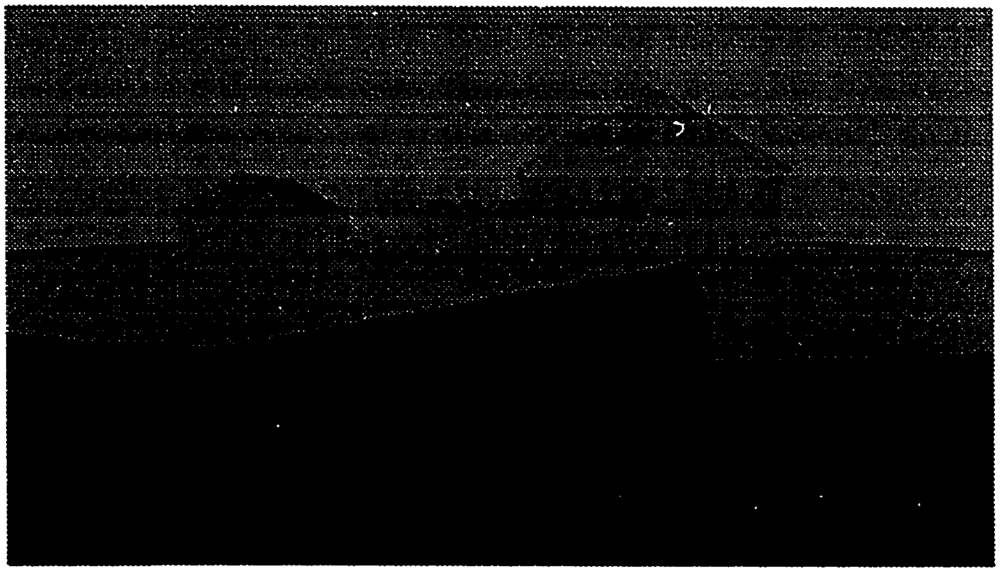


Figure 7.7 Viewing beside the car

Figure 7.8: Viewing from a helicopter

Except the viewpoint, a user can also control the effect of fog or let the car drive on bumpy road, and can regulate the fog density or the bumpiness of surface through the keys. Figure 7.9 shows an example of fog effect.

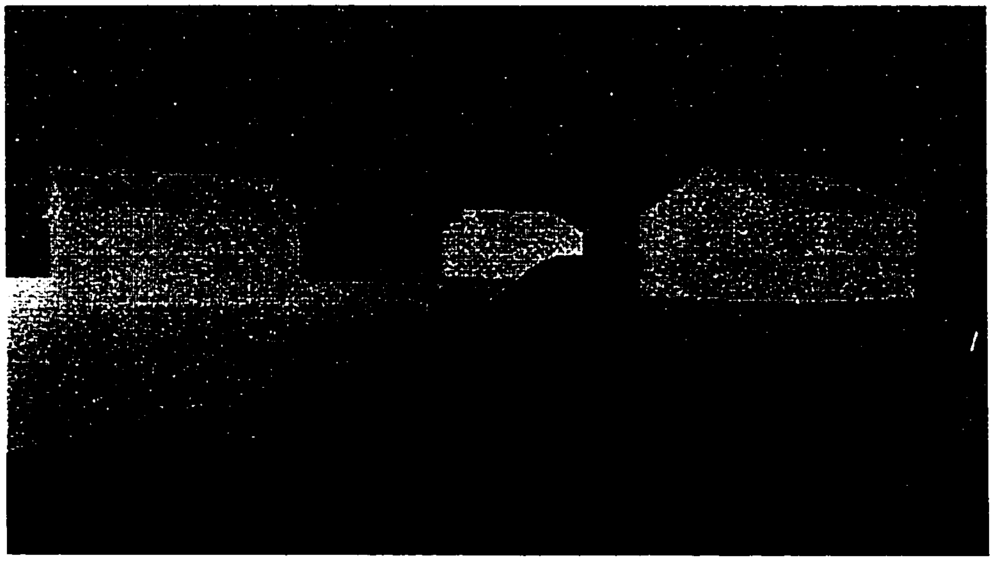


Figure 7.9: Fog effect

Moreover, user can change the height of viewpoint or zoom in or zoom out the window by using the arrow keys.

7.3 The Planet and its Moon System

7.3.1 Classes implementation

7.3.1.1 Class Sphere

Class Sphere is a subclass of the abstract class Object. Using it, the user can draw the color sphere with any radius.

```
Private attributes: Double radius;
                    int slices, stacks;
Public operations:  Sphere(char* sp);
                    ~Sphere() { }

                    void setParameters(Double ra, int sl, int st);

                    void setColor(const Color &col);
                    void setColor(Real r, Real g, Real b);
                    void draw();
```

7.3.1.2 Class LightSource

Light is an important part in OpenGL software. And it is one of the important items in graphics system. Class LightSource defines the basic attributes and operations for a light source in an animation system. Figure 7.10 shows a blue sphere in white light.

```
Private attributes: Real ambient[4];
                    Real diffuse[4];
                    Real specular[4];
                    Point position;
                    Real dull[1];
Public attributes:  bool show;
Public operations:  LightSource();
                    ~LightSource();
```

```
void setLightAmbient(const Real* amb);
void setLightDiffuse(const Real* diff);
void setLightSpecular(const Real* spec);
void setLightShininess(const Real* d);
void setLightPosition(Real* pos);
void setLightPosition(Point pos);
void indicate();
```

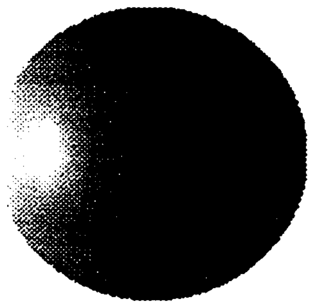


Figure 7.10: A blue sphere in white light

7.3.2 Application Model Implementation

This model is implemented mainly reusing the definition of the base classes. That is two spheres simulating a planet and its one moon. Through the keyboard, the user can set different viewpoints such as forward, upward, leftward, and rightward. He can also control the moon's rotation round the planet; or zoom in or zoom out the window. And the user can set the light effect through the mouse buttons and let the light rotate leftwards, rightwards, upwards, or downwards. Figures from 7.11 to 7.13 illustrate some effects of this animation.

Figure 7.11: The original sate of the planet system

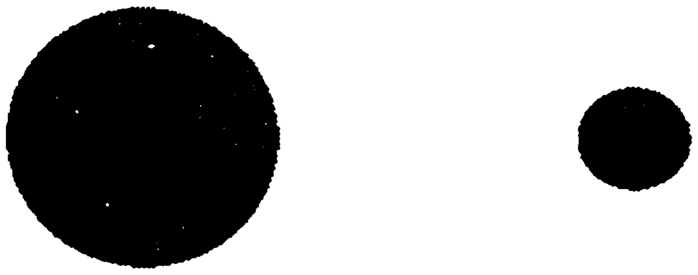


Figure 7.12: The light effect of the Planet

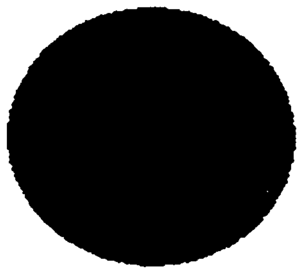


Figure 7.13: The rightward viewpoint of the Planet

Chapter 8: Conclusion and Future Work

As a result of this major report, it can be seen that the OpenGL framework used in the simulation of the two simple animation control system, has enough common features to constitute an object-oriented framework.

Here we only designed several classes, and developed two very simple animation system. But many global variables have been avoided; and we can reduce the use of OpenGL commands and constants. From the examples, we can also receive some ideas for solving some problems in large and complex graphics system. For instance, it is possible decomposing an entire graphic system into subsequent modules and tackling each one separately, just like the running car animation was broken down into many objects color, car, road, house, etc. each with their own attributes and methods. And the MVC framework can be used to simulate graphics system. The MVC triple is easier to understand and programmers can make necessary changes to an application by concentrating on the respective category Model, View, or Controller. While the key to reusability in software methodologies lies within the object-oriented paradigm and MVC framework. So applying the framework design technology, combining the object-oriented programming language such as C++ with the OpenGL software system, we can develop maintainable and reusable graphics application software.

In this version of the OpenGL framework, we have created more than ten classes for the animation application. The implementation of the system design is just focused on creating some basic component objects that represent basic structures for animation. There are a number of elements of the initial design left to implement. Remedying them can make the framework more practical. In the future, we will also further reduce the use of global variables and OpenGL commands.

References

[1] George Wilkie, *Object-Oriented Software Engineering: The Professional developer's Guide*, The institute of Software Engineering, Addison-Wesley Publishing Company, 1994

[2] Dino Mandrioli and Bertrand Meyer, *Advances in Object-Oriented Software Engineering*, Interactive Software Engineering, Santa Barbara and Societe des Outils du Logiciel, Paris, 1992

[3] Adam Blum, *Neural Networks in C++: An Object-Oriented Framework for Building Connectionist Systems*, John Wiley & Sons, Inc., 1992

[4] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*, Johannes Kepler University Linz, Addison-Wesley Publishing Company, 1995

[5] Erich Gamma, Richard Helm, Ralph, Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Objected-Oriented Software*, Addison-Wesley Publishing Company, 1995

[6] Peter Wisskirchen, *Object-Oriented Graphics: from GKS and PHIGS to Object-Oriented Systems*, Springer-Verlag Berlin Heidelberg, New York, 1990

[7] Karon M. Gardner, Alexander Rush, Michael K. Crist, Robert Konitzer, and Bobbin Teegarden, *Cognitive Patterns: Problem-Solving Frameworks for Object Technology*, Cambridge University Press, 1998

[8] Mason Woo, Jackie Neider, and Tom Davis, *OpenGL: Programming Guide, Second Edition, The Official Guide to Learning OpenGL, Version 1.1*, Addison-Wesley Developers Press, 1996

[9] Bruce Eckel, *Thinking in C++, Volume1 and Volume 2, 2nd Edition*, MindView, Inc., January 2000

[10] Youchen Lou, *VDM C++: A Design And Implementation Framework*, Concordia University, February 1994

[11] Desmond Francis D'Souza and Alan Cameron Wills, *Objects, Components, and Frameworks with UML: The Catalysis^{SM} Approach*, Addison-Wesley, An imprint of Addison Wesley Longman, Inc., 1998

[12] Grady Booch, James umbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, An imprint of Addison Esley Longman, Inc., 1990

[13] Taligent, Inc., *Building Object-Oriented Framework*, A Taligent White Paper, 1994.

Glossary

abstract class

A class that contains at least one abstract method. Abstract classes cannot be instantiated; and they must be extended by a class that implements the abstract methods.

abstract data type

An entity that consists of data and routines and represents a type so that any number of variables can be declared.

abstract method

A method that has no implementation. The implementation is deferred to subclasses.

abstraction

The principle of characterizing the behaviours, or functions, of a module in a succinct and precise description known as the contractual interface.

aggregation

A special form of association in which one class is a part of or belongs to another class. It represents the *has-a* or *part-of* relationship.

animation

Generating repeated rendering of a scene, with smoothly changing viewpoint and/or object positions, quickly enough so that the illusion of motion is achieved. OpenGL animation is almost done using double-buffering.

association

A general binary relationship among classes.

C

God's programming language.

C++

The object-oriented programming language of a pagan deity.

class

An extensible abstract data type.

coordinate system

In n-dimensional space, a set of n linearly independent vectors anchored to a point (called the origin). A group of coordinates specifies a point in space by indicating how far to travel along each vector to reach the point.

design patterns

A means of capturing and communicating the design of object-oriented systems.

double-buffering

OpenGL contexts with both front and back colour buffers are double-buffered. Smooth animation is accomplished by rendering into only the back buffer, then causing the front and back buffers to be swapped.

dynamic binding

The binding of a method invocation to a specific implementation at run time instead of at compile time.

encapsulation

The principle of separating the implementation of an object from its contractual interface and hiding the implementation from its clients.

framework

A semifinished software architecture that can be adapted to specific needs by applying object-oriented programming concepts; generic term for application framework and small framework.

inheritance

The relationship among classes and interfaces that models the *is-a* relationship in the real world; the aspect of the *is-a* relationship that permits the reuse of class definitions.

interface

A special form of class that declares features but provides no implementation. An interface declares only constants and abstract methods.

multiple inheritance

A form of inheritance that allows a class to have multiples superclasses.

object

An instance of a class.

OOP

Object-Oriented Programming.

OOPL

Object-Oriented Programming Language.

OpenGL

"Open Graphics Library." It is a software interface to graphics hardware. Using the interface, a programmer can build up any desired pictures.

polymorphism

The ability of dynamically interchanging modules without affecting clients.

RGBA

Red, Green, Blue, Alpha.

RGBA mode

An OpenGL context is in RGBA mode if its colour buffers store red, green, blue, and alpha colour components, rather than colour indices.

single-buffering

OpenGL contexts that don't have back colour buffers are single-buffered. You can use these contexts for animation, but take care to avoid visually disturbing flashes when rendering.

single inheritance

A form of inheritance in which each class may inherit from only one superclass.

UML

Unified Modeling Language which is a graphical notation for describing object-oriented analysis and design models.

viewpoint

The origin of either the eye- or the clip-coordinate system, depending on context. With a typical projection matrix, the eye-coordinate and clip-coordinate origins are at the same location.

window

A subregion of the frame buffer, usually rectangular, whose pixels all have the same buffer configuration. An OpenGL context renders to a single window at a time.

Appendix A: User's Guide

A.1 System Requirements

This application is completed using the Visual C++ system (version 6.0) and OpenGL (using the GLUT library) software system in Microsoft Windows'98 or above system platform. In order to apply the OpenGL software library, before compile the application programs, the system LIB files *glu32.lib*, *glut32.lib*, and *opengl32.lib* should be added under the directory of *vc98 lib* . And the DLL files *glu32.dll*, *glut32.dll*, and *opengl32.dll* must located in the directory windows/system/. And also the include files *gl.h*, *glaux.h*, *glu.h*, and *glut.h* should be put into vc98/include/gl/. These files can be downloaded from Microsoft.

Moreover before compiling the source codes, the LIB files *opengl32.lib*, *glu32.lib*, and *glut32.lib* must be added in the term of *projects settings link object-library modules* on the compile window.

A.2 The Structure of the Application

The whole application including documents and its implementation are organized under one director named report. This is clearly illustrated by the figure A.A.1. The user only need to decompresses the zip file on disk.

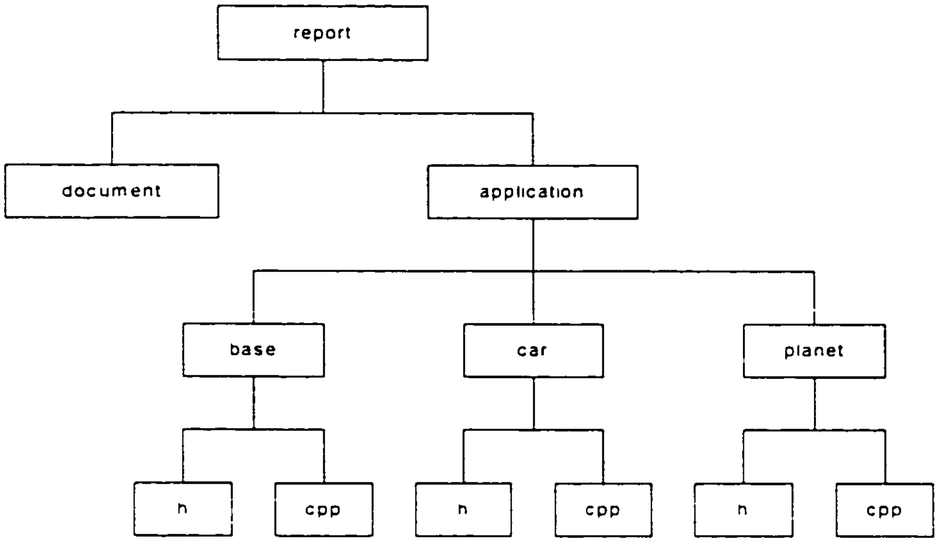


Figure A.A.1: The structure of the application

A.3 Operation Guide

A.3.1 The Running Car Animation

In order to controlling the animation, the system sets the following operations of keyboard and mouse for its users.

- F1 – Set distant viewpoint.
- F2 – Set viewpoint inside the track; and camera follows the car.
- F3 – Set viewpoint outside the track; and camera follows the car.
- F4 – Set the driver's point of view.
- F5 – Set viewpoint while looking at a house.
- F6 – Set viewpoint looking backwards from another car.
- F7 – Set viewpoint from beside the car.
- F8 – Set viewpoint from a balloon.
- F9 – Set viewpoint from a helicopter.
- f10 – Set viewpoints changing at random intervals.

- Left arrow – Zoom out the window.
- Right arrow – Zoom in the window.

- Up arrow – Increase the height of viewpoint.
- Down arrow – Decrease the height of viewpoint.

- b (B) – Bumpiness control switch.
- < (,) – More bumpiness.
- > (.) – Less bumpiness.

- f (F) – Fog control switch.
- + (=) – Increase fog density.
- - (_) – Decrease fog density.

- End – Exit program.
- ESC – Exit program.

A.3.2 The Planet and its Moon System

In this system, the user can control the animation using the following keyboard and mouse button definitions.

- F1 – Set the forward viewpoint.
- F2 – Set the upward viewpoint.
- F3 – Set the leftward viewpoint.
- F4 – Set the rightward viewpoint.

- Left button – Close the light effect.
- Right button – Open the light effect.

- Left arrow key – Rotate the light source to left.
- Right arrow key – Rotate the light source to right.
- Up arrow key – Rotate up the light source.
- Down arrow key – Rotate down the light source.

- x (X) – Rotate the system round x axis.
- y (Y) – Rotate the system round y axis.
- z (Z) – Rotate the system round z axis.
- s (S) – Stop the rotating effect.

- + (=) – Zoom in the window.
- - (_) – Zoom out the window.

- ESC – Exit the program.

Appendix B: The Source Code

B.1 The Created Header Files

B.1.1 types.h

```
//types.h
//define some types and global veriables

#include <gl/glut.h>

#ifndef Int
#define Int unsigned char
#endif

#ifndef Real
#define Real GLfloat
#endif

#ifndef Double
#define Double GLdouble
#endif

//OpenGL types: Pointer for quadric objects
#ifndef QuadricObj
#define QuadricObj GLUquadricObj
#endif
```

B.1.2 specialkeys.h

```
//: specialkeys.h
//define the special keys on keyboard and mouse

#ifndef SPECIALKEYS_H
#define SPECIALKEYS_H

#define LEFTARROWKEY        GLUT_KEY_LEFT
#define RIGHTARROWKEY       GLUT_KEY_RIGHT
#define DOWNARROWKEY        GLUT_KEY_DOWN
#define UPARROWKEY          GLUT_KEY_UP
#define ENDKEY              GLUT_KEY_END
#define PAGEUPKEY           GLUT_KEY_PAGE_UP
#define PAGEDOWNKEY         GLUT_KEY_PAGE_DOWN
```

```
#define F1KEY        GLUT_KEY_F1
#define F2KEY        GLUT_KEY_F2
#define F3KEY        GLUT_KEY_F3
#define F4KEY        GLUT_KEY_F4
#define F5KEY        GLUT_KEY_F5
#define F6KEY        GLUT_KEY_F6
#define F7KEY        GLUT_KEY_F7
#define F8KEY        GLUT_KEY_F8
#define F9KEY        GLUT_KEY_F9
#define F10KEY       GLUT_KEY_F10
#define F11KEY       GLUT_KEY_F11
#define F12KEY       GLUT_KEY_F12

//mouse button
#define LEFTBUTTON              GLUT_LEFT_BUTTON
#define MIDDLEBUTTON            GLUT_MIDDLE_BUTTON
#define RIGHTBUTTON             GLUT_RIGHT_BUTTON

//the state DOWN of mouse button
#define BUTTONDOWN              GLUT_DOWN

#define ESC                     27

#endif //SPECIALKEYS_H
```

B.2 Base Classes

B.2.1 Class Vector

1. vector.h

```
//vector.h
//class Vector

#ifndef VECTOR1_H
#define VECTOR1_H

#include "types.h"

class Vector {
public:
   Real x, y, z;
public:
	Vector();
	Vector(Real x0, Real y0, Real z0 = 0.0);
	~Vector();

	Real getX();
	Real getY();
	Real getZ();
	void setX(Real x0);
	void setY(Real y0);
	void setZ(Real z0);

	void operator() (Real x0, Real y0, Real z0);
	void def(Real x0, Real y0, Real z0);
};

#endif //VECTOR_H
```

2. vector.cpp

```
//vector.cpp
//implementation for class Vector

#include "../h/vector.h"

Vector::Vector() {
	x = y = z = 0.0;
}

Vector::Vector(Real x0, Real y0, Real z0) {
	def( x0, y0, z0 );
}

Vector::~Vector() {
}

void Vector::operator( ) ( Real x0, Real y0, Real z0 ) {
	def( x0, y0, z0 );
}

Real Vector::getX() {
	return x;
}

Real Vector::getY() {
	return y;
}

Real Vector::getZ() {
        return z;
}

void Vector::setX(Real x0) {
        x = x0;
}

void Vector::setY(Real y0) {
        y = y0;
}

void Vector::setZ(Real z0) {
        z = z0;
}

void Vector::def( Real x0, Real y0, Real z0 ) {
        x = x0;
        y = y0;
        z = z0;
}
```

B.2.2 Class Point

1. point.h

```
// point.h
// Describes a point (x, y, z) in 3-dimension.
// If z=0, it is a point in 2-dimension.

#ifndef POINT_H
#define POINT_H

#include "../h/vector.h"

class Point : public Vector {
public:
        Point( );
        Point( Real x0, Real y0, Real z0 = 0.0 );
        Point( Vector &v );
        ~Point();
        void rotate( Real angle_in_deg, const Vector &axis );

        void rotate( Real angle_in_deg, Real dx, Real dy, Real dz );

        void translate( Real dx, Real dy, Real dz );

        void translate( const Vector &v );

        void scale( Real kx, Real ky, Real kz );
};

#endif //POINT_H
```

2. point.cpp

```
// point.cpp.
// implementation for class Point
// Describes a point (x, y, z) in 3-dimension.
// If z = 0, it is a point in 2-dimension.

#include <gl/glut.h>

#include "../h/types.h"
#include "../h/point.h"
#include "../h/vector.h"

Point::Point( ) {
        x = 0.0;
        y = 0.0;
        z = 0.0;
}

Point::Point( Vector &v ) {
        def( v.x, v.y, v.z );
}

Point::Point( Real x0, Real y0, Real z0 ) {
        def( x0, y0, z0 );
}

Point::~Point() { }

void Point::rotate( Real angle_in_deg, const Vector &axis ) {
        glRotatef(angle_in_deg, axis.x, axis.y, axis.z);
}

void Point::rotate( Real angle_in_deg, Real dx, Real dy, Real dz ) {
        glRotatef(angle_in_deg, dx, dy, dz);
}

void Point::translate( Real dx, Real dy, Real dz ) {
        x += dx;
        y += dy;
        z += dz;
}

void Point::translate( const Vector &v ) {
        x += v.x;
        y += v.y;
        z += v.z;
}

void Point::scale( Real kx, Real ky, Real kz ) {
        x *= kx;
        y *= ky;
        z *= kz;
}
```

B.2.3 Class Color

1. color.h

```
//color.h
// define the RGB color of OpenGL software

#ifndef COLOR_H
#define COLOR_H

#include "vector.h"
#include "types.h"

class Color : public Vector {

public:
        Color();
        Color(Real r, Real g, Real b);

        void operator() (Real r0, Real g0, Real b0);
```

```
	void setRGB();
};

#endif //COLORS_H
```

2. color.cpp

```
//color.cpp
// implementation for class Color

#include <gl/glut.h>

#include "../h/color.h"
#include "../h/vector.h"

Color::Color() : Vector() {
}

Color::Color(Real r, Real g, Real b) : Vector(r, g, b) {
}

void Color::setRGB() {
	glColor3f(x, y, z);
}

void Color::operator( ) ( Real r0, Real g0, Real b0 )
{
	def( r0, g0, b0 );
}
```

B.2.4 Class Object

1. object.h

```
// object.h
// An abstract class for objects that can be drawn.

#ifndef OBJECT_H
#define OBJECT_H

#include "types.h"
#include "color.h"
#include "point.h"
```

```
class Object {
protected:
        char* name;                     // Object's name.
        Point position;                 //the start coordinator of a project
        Color color;                    //the color of a project
public:
        Object(char* inam = "Object");
        ~Object() { }

        virtual void draw() = 0;

        char* get_name() { return name; }

        virtual void setPosition(Real xx, Real yy, Real zz);

        virtual void setRGB(Real rr, Real gg, Real bb);

        virtual void setRGB(Color col);

        virtual void translate();

        virtual void rotate(Real angle, Real x0, Real y0, Real z0);

        virtual void scale(Real x1, Real y1, Real z1);
};

#endif //OBJECT_H
```

2. object.cpp

```
// Drawable.cpp
// The implementation of class Drawable.

#include <gl\glut.h>
#include <ctype.h>

#include "../h/object.h"
#include "../h/types.h"

// Constructor sets initial values.
Object::Object(char* iname) {
        //the name of object
        name = iname;
        //set start position value of object
        position.x = 0.0;
        position.y = 0.0;
        position.z = -10.0;

        //set color value of object
        color.x = 1.0;
        color.y = 1.0;
        color.z = 1.0;
}

void Object::setPosition(Real xx, Real yy, Real zz) {
        position.x = xx;
        position.y = yy;
        position.z = zz;
}

void Object::setRGB(Real rr, Real gg, Real bb) {
        color.x = rr;
        color.y = gg;
        color.z = bb;
}

void Object::setRGB(Color col) {
        color.x = col.x;
        color.y = col.y;
        color.z = col.z;
}

void Object::translate() {
        // Move the object to its current position.
        glTranslatef(position.x, position.y, position.z);
}

void Object::rotate(Real angle, Real x0, Real y0, Real z0) {
        //Rotate the object round (x0, y0, z0).
        glRotatef(angle, x0, y0, z0);
}

void Object::scale(Real x1, Real y1, Real z1) {
        // Scale the object,
        // And every point in the object is multiplied
        //  by the corresponding argument x1, y1, z1.
        setPosition(position.x * x1, position.y * y1, position.z * z1);
        glScalef(x1, y1, z1);
}
```

B.2.5 Class Camera

1. camera.h

```
//camera.h
//header file for class Camera

#ifndef CAMERA_H
#define CAMERA_H

#include "types.h"
#include "point.h"

class Camera {
private:
        Point eye, target, up;
public:
        Camera();
        Camera(Real ex, Real ey, Real ez, Real cx, Real cy, Real cz,
                Real ux, Real uy, Real uz);
        ~Camera() {}

        // Change eye point:
        void changePosition( const Point &newEye );
        void changePosition( Real eye_x, Real eye_y, Real eye_z );

        // Change target point:
        void changeTarget( const Point &newTarget );
        void changeTarget( Real target_x, Real target_y, Real target_z );

        // Change up direction point:
        void changeUpDirection( const Point &newUp );
        void changeUpDirection( Real up_x, Real up_y, Real up_z );

        void view();
};

#endif //CAMERA_H
```

2. camera.cpp

```
//camera.cpp
//implementation of class Camera

#include <gl/glut.h>

#include "../h/camera.h"

Camera::Camera() : eye(0.0, 0.0, 0.0), target(0.0, 0.0, -100.0),
                        up(0.0, 1.0, 0.0) {}

Camera::Camera(Real ex, Real ey, Real ez, Real cx, Real cy, Real cz,
                        Real ux, Real uy, Real uz): eye(ex, ey, ez), target(cx, cy, cz),
                        up(ux, uy, uz) { }

void Camera::changePosition( const Point &newEye ) {
        eye = newEye;
}

void Camera::changePosition( Real eye_x, Real eye_y, Real eye_z ) {
        changePosition( Point(eye_x, eye_y, eye_z) );
}

void Camera::changeTarget( const Point &newTarget ) {
        target = newTarget;
}

void Camera::changeTarget( Real target_x, Real target_y, Real target_z ) {
        changeTarget( Point(target_x, target_y, target_z) );
}

void Camera::changeUpDirection( const Point &newUp ) {
        up = newUp;
}

void Camera::changeUpDirection( Real up_x, Real up_y, Real up_z ) {
        changeUpDirection( Point(up_x, up_y, up_z) );
}

void Camera::view() {
        gluLookAt(eye.x, eye.y, eye.z, target.x, target.y, target.z,
                up.x, up.y, up.z);
}
```

B.2.5 Class Window

1. window.h

```
//Window.h
//Make preparing for creating a window.

#ifndef WINDOW_H
#define WINDOW_H

#include "types.h"
#include "vector.h"
#include "color.h"

class Window {
private:
        int width;          //the size of the window (width, height)
        int height;

        int x, y;           //start position (x, y)

        char* title;        //display title on top of the window.
        Color winCol;
        Real alpha;
public:

        Window(int w = 600, int h = 400, int ix = 10, int iy = 10, char* t = "");

        Window(char* t);

        ~Window();

        //set the original size of the window.
        void setSize(int w, int h);

        //set the start position (x, y).
        void setPosition(int xx, int yy);

        //set the background color of the window.
        void setBGColor(Real br, Real bg, Real bb, Real ba = 1.0);

        void setBGColor(Color c);

        void draw();
        void clear();
};

#endif //WINDOW_H
```

2. window.cpp

```
// Window.cpp.
// implementation of the class Window

#include <gl/glut.h>

#include "../h/window.h"
#include "../h/types.h"
#include "../h/vector.h"
#include "../h/color.h"

Window::Window(int w, int h, int ix, int iy, char* t) {
        width = w;
        height = h;

        x = ix; y = iy;
        title = t;

        //default background color is black.
        winCol.x = 0.0;
        winCol.y = 0.0;
        winCol.z = 0.0;

        alpha = 1.0;
}

Window::Window(char* t) : title(t) {
        width = 600;
        height = 400;

        x = y = 50;

        winCol.x = 0.0;
        winCol.y = 0.0;
        winCol.z = 0.0;

        alpha = 1.0;
}
```

```
Window::~Window() {
        delete title;
}

void Window::setSize(int w, int h) {
        width = w;
        height = h;
}

void Window::setPosition(int xx, int yy) {
        x = xx;
        y = yy;
}

void Window::setBGColor(Real br, Real bg, Real bb, Real ba) {
        winCol.x = br;
        winCol.y = bg;
        winCol.z = bb;

        alpha = ba;
}

void Window::setBGColor(Color c) {
        winCol.x = c.x;
        winCol.y = c.y;
        winCol.z = c.z;
}

void Window::clear() {
        glClearColor(winCol.x, winCol.y, winCol.z, alpha);
        glColor4f(winCol.x, winCol.y, winCol.z, alpha);

   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

void Window::draw() {
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(width, height);
        glutInitWindowPosition(x, y);

        glutCreateWindow(title);

        clear();
}
```

B.2.6 Class KeyReaction

1. keyreact.h

```
//key_react.h
//An abstract class for keyboard and mouse

#ifndef KEY_REACT_H
#define KEY_REACT_H

class KeyReaction {

public:
	KeyReaction() { }
	~KeyReaction() { }

	virtual void explain() = 0;

	virtual void setKeys(unsigned int key){ }
	virtual void setFunctionKeys(unsigned int key){ }
	virtual void setMouseButton(int button, int state){ }
};

#endif //KEY_REACT_H
```

2. keyreact.cpp

B.3 The Running Car Animation

B.3.1 Class Disk

1. disk.h

```
//disk.h
//head file for class Disk

#ifndef DISK_H
#define DISK_H

#include "../../base/h/object.h"
```

```
#include "../../base/h/types.h"

class Disk : public Object {
private:
	QuadricObj *dobj;
	Double innerRadius, outerRadius;
	int slices, rings;
public:
	Disk();
	Disk(char* dn);

	void setParameters(QuadricObj *p, Double ir, Double or, int sl, int rg);

	void setColor(const Color &col);

	void draw();
};

#endif //DISK_H
```

2. disk.cpp

```
//disk.cpp
//implementation for class Disk

#include <gl/glut.h>
#include "../h/disk.h"

Disk::Disk() : Object() {
}

Disk::Disk(char* dn) : Object(dn) {
	dobj = 0;
	innerRadius = 0.0;
	outerRadius = 1.0;
	slices = 25;
	rings = 5;
}

void Disk::setParameters(QuadricObj *p, Double ir, Double or, int sl, int rg) {
	dobj = p;
	innerRadius = ir;
	outerRadius = or;
	slices = sl;
	rings = rg;
}

void Disk::setColor(const Color &col) {
        setRGB(col);
}

void Disk::draw() {
        glPushMatrix();
        glColor3f(color.x, color.y, color.z);
        gluDisk(dobj, innerRadius, outerRadius, slices, rings);
        glPopMatrix();
}
```

B.3.2 Class Cylinder

1. cylinder.h

```
//cylinder.h
//head file for class Cylinder

#ifndef CYLINDER_H
#define CYLINDER_H

#include "../../base/h/object.h"
#include "../../base/h/types.h"

//Draw a cylinder oriented along the z axis, with the base of the cylinder
//at z = 0 and the top at z = height. And it is subdivided around the z axis
//into a number of slices and alond the z axis into a number of stacks.
//baseRadius is the radius of the cylinder at z = 0.
class Cylinder : public Object {
private:
        QuadricObj *cobj;
        Double baseRadius, topRadius, height;
        int slices, stacks;
public:
        Cylinder(char* dn);
        ~Cylinder() { }

        void setParameters(QuadricObj *p, Double br, Double tr, Double h,
                int sl, int st);

        void setColor(const Color &col);
```

```
	void draw();
};

#endif //CYLINDER_H
```

2. cylinder.cpp

```
//cylinder.cpp
//implementation for class Cylinder

#include <gl/glut.h>

#include "../h/cylinder.h"

Cylinder::Cylinder(char* dn) : Object(dn) {
	cobj = 0;
	baseRadius = 1.0;
	topRadius = 1.0;
	height = 1.0;
	slices = 10;
	stacks = 10;
}

void Cylinder::setParameters(QuadricObj *p, Double br, Double tr, Double h,
			int sl, int st) {
	cobj = p;
	baseRadius = br;
	topRadius = tr;
	height = h;
	slices = sl;
	stacks = st;
}

void Cylinder::setColor(const Color &col) {
	setRGB(col);
}

void Cylinder::draw() {
	glPushMatrix();
	glColor3f(color.x, color.y, color.z);
	gluCylinder(cobj, baseRadius, topRadius, height, slices, stacks);
	glPopMatrix();
}
```

B.3.3 Class Car

1. car.h

```
//car.h
//definition of class Car

#ifndef CAR_H
#define CAR_H

#include "../../base/h/object.h"
#include "../../base/h/types.h"
#include "../../base/h/color.h"

#include "disk.h"
#include "cylinder.h"

class Car : public Object {
private:
        Cylinder carBody;
        Disk bodyFront, bodyRear;
        Cylinder frontAxle, rearAxle;
        Disk frontLeftWheel, frontRightWheel;
        Disk rearLeftWheel, rearRightWheel;
        Color bodyColor, axleColor, wheelColor;

        Real length;   //length of a car body
        Real frontRadius, rearRadius; //the radius of two side of a car body
        Real axleRadius; //the axle radius of a car
        Real wheelRadius; //wheel radius of a car

        QuadricObj *p;   //quadrics object
public:
        Car(char *nm);
        ~Car() { }

        void setBodyColor(Color c);
        void setAxleColor(Color c);
        void setWheelColor(Color c);

        void setStartPosition(Real x0, Real y0, Real z0);
        void setStartPosition(Vector &pos);
        void setBodyParam(Real l, Real fr, Real rr);
        void setAxleRadius(Real ar);
        void setWheelRadius(Real wr);
```

```
	void setStartPosition(Real x0, Real y0, Real z0);
	void setStartPosition(Vector &pos);
	void setBodyParam(Real l, Real fr, Real rr);
	void setAxleRadius(Real ar);
	void setWheelRadius(Real wr);
	void setQuadricObj(QuadricObj *p1);

	void draw();
};

#endif //CAR_H
```

2. car.cpp

```
//car.cpp
//implementation for class Car

#include <gl/glut.h>

#include "../h/car.h"

Car::Car(char* nm) : Object(nm), carBody("carBody"), bodyFront("bodyFront"),
	bodyRear("bodyRear"), frontAxle("frontAxle"), rearAxle("rearAxle"),
	frontLeftWheel("frontLeftWheel"),                   frontRightWheel("fRW"),
rearLeftWheel("rLW"),
	rearRightWheel("rRW")
{
	bodyColor(0.4f, 0.0f, 0.6f);
	axleColor(0.4f, 0.4f, 0.4f);
	wheelColor(1.0f, 0.0f, 0.0f);

	position.x = 0.0f;
	position.y = 0.0f;
	position.z = 3.0f;

	length = 12.0f;
	frontRadius = 1.0f;
	rearRadius = 2.0f;
	axleRadius = 0.5f;
	wheelRadius = 2.0f;
	p = 0;
}

void Car::setBodyColor(Color c) {
}

void Car::setWheelColor(Color c) {
        wheelColor = c;
}

void Car::setStartPosition(Real x0, Real y0, Real z0) {
        Object::setPosition(x0, y0, z0);
}

void Car::setStartPosition(Vector &pos) {
        position.x = pos.x;
        position.y = pos.y;
        position.z = pos.z;

}

void Car::setBodyParam(Real l, Real rr, Real fr) {
        length = l;
        frontRadius = fr;
        rearRadius = rr;
}

void Car::setAxleRadius(Real ar) {
        axleRadius = ar;
}

void Car::setWheelRadius(Real wr) {
        wheelRadius = wr;
}

void Car::setQuadricObj(QuadricObj *p1) {
        p = p1;
}

void Car::draw() {
        // Draw the car: body first.  The car is facing +X,
        // and up is +Z.  Save current transformation.
        glPushMatrix();
                glTranslatef(position.x, position.y, position.z);
                glRotatef(95.0, 0.0, 1.0, 0.0);

                carBody.setColor(bodyColor);
                carBody.setParameters(p, rearRadius, frontRadius, length, 10, 12);
                carBody.draw();
        bodyRear.setColor(bodyColor);
        bodyRear.setParameters(p, 0.0, rearRadius, 25, 5);
        bodyRear.draw();

        glTranslatef(0.0, 0.0, length);

        bodyFront.setColor(bodyColor);
        bodyFront.setParameters(p, 0.0, frontRadius, 20, 5);
        bodyFront.draw();
glPopMatrix();

// Rear axle and wheels.
glPushMatrix();
        glTranslatef( position.x + length * 0.25f,
                position.y - (rearRadius + 2.0f * axleRadius),
                rearRadius);

        glRotatef(90.0, -1.0, 0.0, 0.0);

        rearAxle.setColor(axleColor);
        rearAxle.setParameters(p, axleRadius, axleRadius,
                2.0 * rearRadius+rearRadius, 10, 12);
        rearAxle.draw();

        rearLeftWheel.setColor(wheelColor);
        rearLeftWheel.setParameters(p, 0.0, wheelRadius, 25, 5);
        rearLeftWheel.draw();
        glTranslatef(0.0, 0.0, 2.0*rearRadius+rearRadius);

        rearRightWheel.setColor(wheelColor);
        rearRightWheel.setParameters(p, 0.0, wheelRadius, 25, 5);
        rearRightWheel.draw();
glPopMatrix();

// Front axle and wheels.
glPushMatrix();
        glTranslatef(position.x + length * 0.75f,
                position.y - (frontRadius + 2.0f * axleRadius),
                frontRadius);

        glRotatef(90.0, -1.0, 0.0, 0.0);

        frontAxle.setColor(axleColor);
        frontAxle.setParameters(p, axleRadius, axleRadius,
                3.0 * frontRadius + frontRadius, 10, 12);
            frontAxle.draw();

            frontLeftWheel.setColor(wheelColor);
            frontLeftWheel.setParameters(p, 0.0, wheelRadius/2.0f, 25, 5);
            frontLeftWheel.draw();

            glTranslatef(0.0, 0.0, 3.0*frontRadius+frontRadius);

            frontRightWheel.setColor(wheelColor);
            frontRightWheel.setParameters(p, 0.0, wheelRadius/2.0f, 25, 5);
            frontRightWheel.draw();
        glPopMatrix();
}
```

B.3.4 Class House

1. house.h

```
//house.h
//the header file for class House

#ifndef HOUSE_H
#define HOUSE_H

#include <gl/glut.h>

#include "../../base/h/object.h"
#include "../../base/h/color.h"
#include "../../base/h/point.h"

class House : public Object {
        Real size;
        Color wallColor, roofColor, doorColor;
public:
        House( char* nm );
        ~House() { }

        void setPosition( Real x0, Real y0, Real z0 = 0.0 );
        void setPosition( Point &sp );
        void setSize( Real sz );
        void setWallColor( Color &wc );
        void setDoorColor( Color &dc );
        void setRoofColor( Color &rc );
```

```
	void draw();
};

#endif //HOUSE_H
```

2. house.cpp

```
//house.cpp
//implementation of class House

#include "../h/house.h"

House::House( char* nm ) : Object( nm ) {
	position.x = 0.0;
	position.y = 0.0;
	position.z = 0.0;

	wallColor( 0.7f, 0.4f, 0.2f );	//default color of wall
	roofColor( 0.8f, 0.0f, 0.2f );	//default color of roof
	size = 1.0;
}

void House::setPosition( Real x0, Real y0, Real z0 ) {
	Object::setPosition( x0, y0, z0 );
}

void House::setPosition( Point &sp ) {
	setPosition( sp.x, sp.y, sp.z );
}

void House::setSize( Real sz ) {
	size = sz;
}

void House::setWallColor( Color &wc ) {
	wallColor = wc;
}

void House::setDoorColor( Color &dc ) {
	doorColor = dc;
}

void House::setRoofColor( Color &rc ) {
	roofColor = rc;
```

```
}

void House::draw() {
        //Draw a house
        glPushMatrix();

        glTranslatef( position.x, position.y, position.z );
        glScalef( size, size, size );

        glBegin( GL_QUADS );
                wallColor.setRGB();
                glVertex3f( 1.0, -1.0, 0.0 );          //Wall 1
                glVertex3f( 1.0, -1.0, 1.0 );
                glVertex3f( -1.0, -1.0, 1.0 );
                glVertex3f( -1.0, -1.0, 0.0 );

                glVertex3f( 1.0, -1.0, 0.0 );          // Wall 2
                glVertex3f( 1.0, 1.0, 0.0 );
                glVertex3f( 1.0, 1.0, 1.0 );
                glVertex3f( 1.0, -1.0, 1.0 );

                glVertex3f( -1.0, 1.0, 0.0 );          // Wall 3
                glVertex3f( -1.0, 1.0, 1.0 );
                glVertex3f( 1.0, 1.0, 1.0 );
                glVertex3f( 1.0, 1.0, 0.0 );

                glVertex3f( -1.0, -1.0, 1.0 );         // Wall 4
                glVertex3f( -1.0, 1.0, 1.0 );
                glVertex3f( -1.0, 1.0, 0.0 );
                glVertex3f( -1.0, -1.0, 0.0 );

                doorColor.setRGB();
                glVertex3f( 1.0, 0.3f, 0.0 );          // door
                glVertex3f( 1.0, 0.3f, 0.6f );
                glVertex3f( 1.0, -0.3f, 0.6f );
                glVertex3f( 1.0, -0.3f, 0.0 );
        glEnd();

        glBegin( GL_TRIANGLES );
                wallColor.setRGB();
                glVertex3f( -1.0, -1.0, 1.0);          // Roof side 1
                glVertex3f( 0.0, -1.0, 1.5 );
                glVertex3f( 1.0, -1.0, 1.0 );

                glVertex3f( -1.0, 1.0, 1.0 );          // Roof side 2
                glVertex3f( 0.0, 1.0, 1.5 );
                glVertex3f( 1.0, 1.0, 1.0 );
        glEnd();

        glBegin( GL_QUADS );
                roofColor.setRGB();

                glVertex3f( 1.05f, -1.05f, 0.95f );          // Roof side 1
                glVertex3f( 0.0f, -1.05f, 1.5f );
                glVertex3f( 0.0f, 1.05f, 1.5f );
                glVertex3f( 1.05f, 1.05f, 0.95f );

                glVertex3f( -1.05f, -1.05f, 0.95f );         // Roof side 2
                glVertex3f( 0.0f, -1.05f, 1.5f );
                glVertex3f( 0.0f, 1.05f, 1.5f );
                glVertex3f( -1.05f, 1.05f, 0.95f );
        glEnd();

        glPopMatrix();
}
```

B.3.5 Class Mouse

1. mouse.h

```
//mouse.h
//definition of class Mouse

#ifndef MOUSE_H
#define MOUSE_H

#include "../../base/h/keyreact.h"

class Mouse : public KeyReaction {
public:
        Mouse() : KeyReaction() { }
        ~Mouse() { }

        void explain();

        //following methods have to use some global objects,
        //they should be implemented in the application model.
        void setMouseButton(int Button, int state) { }
        void move(int x, int y);
};

#endif //MOUSE_H
```

2. mouse.cpp

```
//mouse.cpp
//implementation of the method in class Mouse

#include <stdio.h>
#include "..\h\mouse.h"

void Mouse::explain() {
	printf("\a\nHere are the meanings for using mouse:\n");
	printf("Moving mouse can change the position of the picture display.\n");
}
```

B.3.6 Class Keyboard

1. keyboard.h

```
//keyboard.h
//definition of class Keyboard

#ifndef KEYBOARD_H
#define KEYBOARD_H

#include "../../base/h/keyreact.h"

class Keyboard : public KeyReaction {
public:
	Keyboard() : KeyReaction() { }
	~Keyboard() { }

	void explain();

	//following methods have to use some global objects,
	//they should be implemented in the application model.
	void setViewpoint();
	void setKeys(unsigned int key);
	void setFunctionKeys(unsigned int key);
};

#endif //KEYBOARD_H
```

2. keyboard.cpp

```
//keyboard.cpp
//implementation of method of class Keyboard

#include <stdio.h>
#include "..\h\keyboard.h"

void Keyboard::explain() {
   printf("\a\nHere are the keys you can use:\n");
   printf("F1  ...  Static viewpoint and stationary camera.\n");
   printf("F2  ...  Static viewpoint inside the track; camera follows car.\n");
   printf("F3  ...  Static viewpoint outside the track; camera follows car.\n");
   printf("F4  ...  Driver's point of view.\n");
   printf("F5  ...  Drive around while looking at a house.\n");
   printf("F6  ...  View looking backwards from another car.\n");
   printf("F7  ...  View from beside the car.\n");
   printf("F8  ...  View from a balloon.\n");
   printf("F9  ...  View from a helicopter.\n");
   printf("f10 ...  views change at random intervals.\n");
   printf("end ...  Exit program\n\n");

   printf("Left/right arrow  ... Zoom out/in.\n");
   printf("Up/down arrow     ... Change the height of viewpoint.\n\n");

   printf("b (B)          ... Bumpiness control switch.\n");
   printf("< (,)          ... More bumpiness.\n");
   printf("> (.)          ... Less bumpiness.\n\n");

   printf("f (F)          ... Fog control switch.\n");
   printf("+ (=)          ... Increase fog density.\n");
   printf("- (_)          ... Decrease fog density.\n\n");

   printf("\nESC          ... Exit program\n");
}
```

B.3.7 Header File includefiles.h

```
//includes.h
//one header file including all application files

#ifndef INCLUDES_H
#define INCLUDES_H
```

```
#include "../../base/h/specialkeys.h"
#include "../../base/h/types.h"

#include "../../base/h/window.h"
#include "../../base/h/object.h"
#include "../../base/h/vector.h"
#include "../../base/h/point.h"
#include "../../base/h/color.h"
#include "../../base/h/camera.h"

#include "disk.h"
#include "cylinder.h"
#include "car.h"
#include "house.h"
#include "keyboard.h"
#include "mouse.h"

#endif //INCLUDES_H
```

B.3.8 File carmodel.cpp

```
//application.cpp
//define the needed constants, global variables, callback functions
//draw the three-dimensional animation

#include <gl/glut.h>

#include <iostream.h>
#include <math.h>
#include <stdlib.h>

#include "h/includefiles.h"

//some constant variables
const Real PI = 3.1415926f;
const Real TWO_PI = 2.0f * PI;
const Real RAD_TO_DEG = 180.0f / PI;
const Real INNER_RADIUS = 90.0;
const Real TRACK_WIDTH = 20.0;
const Real TRACK_MIDDLE = INNER_RADIUS + 0.5 * TRACK_WIDTH;

enum {          // Constants for different views
        DISTANT, INSIDE, OUTSIDE, DRIVER, HOUSE,
        OTHER, BESIDE, BALLOON, HELICOPTER, AUTO
} view = DISTANT;

const Color bg_col(0.5, 0.5, 0.5);         //color of background

const Color grass_color( 0.0, 0.8f, 0.1f );        //color of grass
const Color road_color( 0.2f, 0.2f, 0.2f );        //color of road

const Real fog[4] = { 0.7f, 0.7f, 0.7f, 1.0 };

// Global variables.
Real car_direction = 0.0;               // Variables for moving car.
Real car_x_pos = 100.0;
Real car_y_pos = 0.0;
Real car_z_pos = 0.0;

Real height = 5.0;                      // Viewer's height
Real zoom = 50.0;                       // Camera zoom setting

Real mouse_x = 0.0;                     // Mouse coordinates
Real mouse_y = 0.0;

int win_width = 600;                    // Window dimensions
int win_height = 400;

bool movie_mode = false;                // Change viewpoint periodically.
long clock = 0;
long next_switch_time = 0;

bool fog_enabled = false;               // Fog data.
Real fog_density = 0.01f;
Real bumpiness = 0.0;                   // Bumpy road.

//some objects
Car car("car");
Disk grass("grass");
Disk road("road");

QuadricObj *p;                                  // Pointer for quadric objects.

Camera camera(250.0, 0.0, 20.0 * height,
                        0.0, 0.0, 0.0,
                        0.0, 0.0, 1.0 );

void drawCar() {
```

```
Camera camera(250.0, 0.0, 20.0 * height,
                        0.0, 0.0, 0.0,
                        0.0, 0.0, 1.0 );

void drawCar() {
        // Draw the car: The car is facing +X, and up is +Z.
        car.setQuadricObj(p);
        car.draw();
}

void drawHouse (char* nm, GLfloat x, GLfloat y, GLfloat size) {
        // Draw a house.
        House house( nm );
        house.setSize(size);
        house.setPosition(x, y);

        house.draw();
}

// Draw a patch of grass, a circular road, and some houses.
void drawScenery () {
        grass.setColor(grass_color);
        grass.setParameters(p, 0.0, INNER_RADIUS, 50, 5);
        grass.draw();
        road.setColor(road_color);
        road.setParameters(p, INNER_RADIUS,
                        INNER_RADIUS + TRACK_WIDTH, 50, 5);
        road.draw();

        drawHouse("house1", -20.0, 50.0, 5.0);
        drawHouse("house2", 0.0, 70.0, 10.0);
        drawHouse("house3", 20.0, -10.0, 8.0);
        drawHouse("house4", 40.0, 120.0, 10.0);
        drawHouse("house5", -30.0, -50.0, 7.0);
        drawHouse("house6", 10.0, -60.0, 10.0);
        drawHouse("house7", -20.0, 75.0, 8.0);
        drawHouse("house8", -40.0, 140.0, 10.0);
}

// Reset the projection when zoom setting or window shape changes.
void setProjection () {
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(zoom, GLfloat(win_width) / GLfloat(win_height),
                1.0, 500.0);
        glFogf(GL_FOG_DENSITY, fog_density);
}

//the implementation of some member functions of class Keyboard
void Keyboard::setViewpoint () {
        // Use the current viewpoint to display.
        switch (view) {

        case DISTANT:
                // Static viewpoint and stationary camera.

                camera.changePosition(250.0, 0.0, 20.0*height);
                camera.changeTarget(0.0, 0.0, 0.0);
                camera.changeUpDirection(0.0, 0.0, 1.0);
                camera.view();

                drawScenery();
                // Move to position of car.
                glTranslatef(car_x_pos, car_y_pos, car_z_pos);

                // Rotate so car stays parallel to track.
                glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, -1.0);

                drawCar();
                break;

        case INSIDE:
                // Static viewpoint inside the track; camera follows car.
                camera.changePosition(85.0, 0.0, height);
                camera.changeTarget(car_x_pos, car_y_pos, 0.0);
                camera.changeUpDirection(0.0, 0.0, 1.0);
                camera.view();

                drawScenery();

                glTranslatef(car_x_pos, car_y_pos, car_z_pos);
                glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, -1.0);

                drawCar();
                break;

        case OUTSIDE:
                // Static viewpoint outside the track; camera follows car.
                camera.changePosition(115.0, 0.0, height);
                camera.changeTarget(car_x_pos, car_y_pos, 0.0);
        camera.changeUpDirection(0.0, 0.0, 1.0);
        camera.view();

        drawScenery();

        glTranslatef(car_x_pos, car_y_pos, car_z_pos);
        glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, -1.0);

        drawCar();
        break;

    case DRIVER:
        // Driver's point of view.  gluLookAt() is defined in "car space".
        // After drawing the car, we use inverse transformations to show
        // the scenery.  The same idea is used for OTHER and BESIDE.
        camera.changePosition(2.0, 0.0, height);
        camera.changeTarget(12.0, 0.0, 2.0);
        camera.changeUpDirection(0.0, 0.0, 1.0);
        camera.view();

        drawCar();

        glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, 1.0);
        glTranslatef(- car_x_pos, - car_y_pos, car_z_pos);

        drawScenery();
        break;

    case HOUSE:
        // Drive around while looking at a house.  The first rotation
        // couteracts the rotation of the car.  gluLookAt() looks from
        // the driver's position to the house at (40,120).
        glRotatef(RAD_TO_DEG * car_direction, 0.0, -1.0, 0.0);

        camera.changePosition(2.0, 0.0, height);
        camera.changeTarget(40.0-car_x_pos, 120.0-car_y_pos, car_z_pos);
        camera.changeUpDirection(0.0, 0.0, 1.0);
        camera.view();

        drawCar();

        glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, 1.0);
        glTranslatef(- car_x_pos, - car_y_pos, car_z_pos);

        drawScenery();
        break;

case OTHER:
        // View looking backwards from another car.
        camera.changePosition(25.0, 5.0, height);
        camera.changeTarget(0.0, 0.0, 3.0+car_z_pos);
        camera.changeUpDirection(0.0, 0.0, 1.0);
        camera.view();

        drawCar();

        glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, 1.0);
        glTranslatef(- car_x_pos, - car_y_pos, 0.0);

        drawScenery();
        break;

case BESIDE:
        // View from beside the car.
        camera.changePosition(5.0, 15.0, height);
        camera.changeTarget(5.0, 0.0, 3.0+car_z_pos);
        camera.changeUpDirection(0.0, 0.0, 1.0);
        camera.view();

        drawCar();

        glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, 1.0);
        glTranslatef(- car_x_pos, - car_y_pos, 0.0);

        drawScenery();
        break;

case BALLOON:
        // View from a balloon.
        camera.changePosition(150.0, 75.0, 250.0);
        camera.changeTarget(200.0*mouse_x, 200.0*mouse_y, 0.0);
        camera.changeUpDirection(0.0, 0.0, 1.0);
        camera.view();

        drawScenery();

        glTranslatef(car_x_pos, car_y_pos, car_z_pos);
        glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, -1.0);

        drawCar();
```

```
                break;

        case HELICOPTER:
                // View from a helicopter.
                camera.changePosition(200.0 * mouse_x, 200.0 * mouse_y, 200.0);
                camera.changeTarget(0.0, 0.0, 0.0);
                camera.changeUpDirection(0.0, 0.0, 1.0);
                camera.view();

                drawScenery();

                glTranslatef(car_x_pos, car_y_pos, car_z_pos);
                glRotatef(RAD_TO_DEG * car_direction, 0.0, 0.0, -1.0);

                drawCar();
                break;
        }
}

void Keyboard::setKeys(unsigned int key) {
        switch (key) {

        case 'b':
        case 'B':
                // Bumpiness control.
                if (bumpiness == 0.0) {
                        cout << "Bumpy road!  Use ',' and '>' to change bumpiness."
                             << endl;
                        bumpiness = 0.1f;
                }
                else {
                        cout << "Smooth road again!" << endl;
                        bumpiness = 0.0;
                }
                break;

        case '<':
        case ',':
                // More bumpiness.
                bumpiness /= 2.0;
                break;

        case '>':
        case '.':
                // Less bumpiness.
                bumpiness *= 2.0;
                break;

        case 'f':
        case 'F':
                // Fog control.
                if (fog_enabled) {
                        glDisable(GL_FOG);
                        cout << "It's a clear day!" << endl;
                        fog_enabled = false;
                }
                else {
                        glEnable(GL_FOG);
                        cout << "It's foggy: use +/- to change fog density." << endl;
                        setFogDensity(0.01f);
                        fog_enabled = true;
                }
                break;

        case '+':
        case '=':
                // Increase fog density.
                setFogDensity(2.0 * fog_density);
                break;

        case '-':
        case '_':
                // Decrease fog density.
                setFogDensity(0.5f * fog_density);
                break;

        case ESC:
                exit(0);
                break;

        default:
                break;
        }
}

void Keyboard::setFunctionKeys(unsigned int key) {
        // Set viewpoint from function keys.
        switch (key) {

        case LEFTARROWKEY:
```

```
        zoom *= 1.2f;

        setProjection();
        break;

case RIGHTARROWKEY:
        zoom /= 1.2f;

        setProjection();
        break;

case UPARROWKEY:
        height += 1.0;
        break;

case DOWNARROWKEY:
        height -= 1.0;
        break;

case F1KEY:
        movie_mode = false;
        view = DISTANT;
        break;

case F2KEY:
        movie_mode = false;
        view = INSIDE;
        break;

case F3KEY:
        movie_mode = false;
        view = OUTSIDE;
        break;

case F4KEY:
        movie_mode = false;
        view = DRIVER;
        height = 6.0;
        zoom = 75.0;
        setProjection();
        break;

case F5KEY:
        movie_mode = false;
        view = HOUSE;
```

```
                break;

        case F6KEY:
                movie_mode = false;
                view = OTHER;
                break;

        case F7KEY:
                movie_mode = false;
                view = BESIDE;
                break;

        case F8KEY:
                movie_mode = false;
                view = BALLOON;
                break;

        case F9KEY:
                movie_mode = false;
                view = HELICOPTER;
                break;

        case F10KEY:
                movie_mode = true;
                clock = 0;
                next_switch_time = 0;
                break;

        case ENDKEY:
                exit(0);
        }
}

//the implementation of the method of class Mouse
void Mouse::move (int x, int y) {
        // Get mouse position and scale values to [-1, 1].
        mouse_x = (2.0 * x) / win_width - 1.0;
        mouse_y = (2.0 * y) / win_height - 1.0;
}

//an object of class Keyboard
Keyboard kb;

//an object of class Mouse
Mouse mouse;
```

```
//callback functions
void display (void) {
        // The display callback function.
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        kb.setViewpoint();

        glutSwapBuffers();
}

void reshape (GLint new_width, GLint new_height) {
        // Window reshaping function.
        win_width = new_width;
        win_height = new_height;
        glViewport(0, 0, win_width, win_height);

        setProjection();
}

void drive () {
        // Idle callback function moves the car. Since this function
        // posts redisplay whenever there is nothing else to do,
        // we do not need any other calls to glutPostRedisplay().
        car_direction += 0.05f;

        if (car_direction > TWO_PI)
                car_direction -= TWO_PI;

        car_x_pos = TRACK_MIDDLE * sin(car_direction);
        car_y_pos = TRACK_MIDDLE * cos(car_direction);
        car_z_pos = (bumpiness * rand()) / RAND_MAX;

        if (movie_mode) {
                clock++;

                if (clock > next_switch_time) {
                        next_switch_time += 20 + rand() % 200;

                        switch (rand() % 7) {
                        case 0:
                                view = DISTANT;
                                break;
```

```
                    case 1:
                            view = INSIDE;
                            break;
                    case 2:
                            view = OUTSIDE;
                            break;
                    case 3:
                            view = DRIVER;
                            break;
                    case 4:
                            view = HOUSE;
                            break;
                    case 5:
                            view = OTHER;
                            break;
                    case 6:
                            view = BESIDE;
                            break;
                    }
            }
        }
        glutPostRedisplay();
}

//some callback function for controlling the animation
void mouseMove (int x, int y) {
        mouse.move(x, y);
}

void keyboard(unsigned int key, int x, int y) {

        kb.setKeys(key);
}

void functionKeys(int key, int x, int y) {

        kb.setFunctionKeys(key);
}

void showGraphics() {
        // Initialize and create a window.
        Window theWindow("Major Report");
        theWindow.setSize(win_width, win_height);
        theWindow.setBGColor(bg_col);
        theWindow.draw();
        // handling window and input events.
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutIdleFunc(drive);
        glutPassiveMotionFunc(mouseMove);
        glutKeyboardFunc(keyboard);
        glutSpecialFunc(functionKeys);


        // Select GL options.
        glEnable(GL_DEPTH_TEST);

        p = gluNewQuadric();

        gluQuadricDrawStyle(p, GLU_FILL);

        glFogi(GL_FOG_MODE, GL_EXP);
        glFogfv(GL_FOG_COLOR, fog);
        glFogf(GL_FOG_DENSITY, fog_density);

        // Initialize projection.
        setProjection();

        // draw the window and projects.
        glutMainLoop();
}
```

B.3.9 File runningcar.cpp

```
//main.cpp

#include <iostream.h>
//#include <gl/glut.h>

//#include "h/globals.h"
//#include "globals.h"

extern void showGraphics();

int main () {

        // Instruct user.
        cout <<"Change view point:"                              << endl <<
        "  Press keys F1, F2, ..., F9.  Movie mode: F10."     << endl <<
        "Bumpiness:"                                          << endl <<
        "  Press 'b' to toggle bumpy road conditions."        << endl <<
        "  Press '<' ('>') to decrease (increase) bumpiness." << endl <<
        "Fog:"                                                << endl <<
        "  Press 'f' to toggle fog effect."                   << endl <<
        "  Press '+' ('-') to increase (decrease) fog."       << endl <<
        "Quit:"                                               << endl <<
        "  Press ESC to stop program."                        << endl <<
        "Keep window small for fast action!"                  << endl;

    //draw the running car system
    showGraphics();

    return 0;
}
```

B.4 The Planet and its Moon System

This system use same basic classes and header files as the running car animation, including the header files types.h, specialkeys.h, and the class Vector, Point, Color, Object, Camera, Window, KeyReaction, Mouse, and Keyboard. The codes of other files are listed as following.

B.4.1 Class Sphere

1. sphere.h

```
//sphere.h
//head file for class Sphere

#ifndef SPHERE_H
#define SPHERE_H

#include "object.h"
#include "types.h"

//Using particular radius draw a color solid sphere.
//And its surface is subdivided into a number of slices
```

```
//  and a number of stacks.

class Sphere : public Object {
private:
        Double radius;
        int slices, stacks;
public:
        Sphere(char* sp);
        ~Sphere() { }

        void setParameters(Double ra, int sl, int st);

        void setColor(const Color &col);
        void setColor(Real r, Real g, Real b);

        void draw();
};

#endif //SPHERE_H
```

2. sphere.cpp

```
//sphere.cpp
//implementation for class Sphere

#include <gl/glut.h>

#include "../h/sphere.h"

Sphere::Sphere(char* sp) : Object(sp) {

        radius = 1.0;

        slices = 20;
        stacks = 10;
}

void Sphere::setParameters(Double ra, int sl, int st) {
        radius = ra;
        slices = sl;
        stacks = st;
}

void Sphere::setColor(const Color &col) {
        setRGB(col);
}

void Sphere::setColor(Real r, Real g, Real b) {
	setRGB(r, g, b);
}

void Sphere::draw() {
	glPushMatrix();
	glColor3f(color.x, color.y, color.z);

	glutSolidSphere(radius, slices, stacks);
	glPopMatrix();
}
```

B.4.2 Class LightSource

1. light.h

```
//light.h
//the definition of class LightSource

#ifndef LIGHT_H
#define LIGHT_H

#include "../h/types.h"
#include "../h/point.h"

class LightSource {
private:
	Real ambient[4];
	Real diffuse[4];
	Real specular[4];

	Point position;
	Real dull[1];
public:
	bool show;
public:
	LightSource();
	~LightSource();

	void setLightAmbient(const Real* amb);
```

```
        void setLightDiffuse(const Real* diff);

        void setLightSpecular(const Real* spec);

        void setLightShininess(const Real* d);

        void setLightPosition(Real* pos);
        void setLightPosition(Point pos);

        void indicate();
};

#endif //LIGHT_H
```

2. light.cpp

```
//light.cpp
//implementation of class LightSource

#include <glut.h>
#include <math.h>

#include "../h/light.h"

LightSource::LightSource() {

        show = false;
}

LightSource::~LightSource() {
}

void LightSource::setLightAmbient(const Real* amb) {
        for (int i = 0; i < 4; i++)
                ambient[i] = amb[i];
}

void LightSource::setLightDiffuse(const Real* diff) {
        for (int i = 0; i < 4; i++)
                diffuse[i] = diff[i];

}

void LightSource::setLightSpecular(const Real* spec) {
        for (int i = 0; i < 4; i++)
```

```
                ambient[i] = spec[i];

}

void LightSource::setLightPosition(Point pos) {
        position.x = pos.x;
        position.y = pos.y;
        position.z = pos.z;
}

void LightSource::setLightPosition(Real* pos) {

        position.x = pos[0];
        position.y = pos[1];
        position.z = pos[2];
}

void LightSource::setLightShininess(const Real* d) {
        dull[0] = d[0];
}

void LightSource::indicate() {

        Real lightPos[4] = {position.x, position.y, position.z, 0.0f};

        if (show == true) {
                glEnable(GL_DEPTH_TEST);
                glEnable(GL_LIGHTING);
                glEnable(GL_LIGHT0);

                glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
                glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);

                glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
                glMaterialfv(GL_FRONT, GL_SHININESS, dull);

                glPushMatrix();         // position of the light
                        glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
                glPopMatrix();
        }
        else
                glDisable(GL_LIGHT0);
}
```

B.4.3 Header File includefiles.h

```
//includefiles.h
//

#ifndef INCLUDEFILES_H
#define INCLUDEFILES_H

#include "window.h"
#include "color.h"
#include "sphere.h"
#include "light.h"
#include "camera.h"
#include "specialkeys.h"
#include "types.h"
#include "keyreact.h"
#include "keyboard.h"
#include "mouse.h"

#endif //INCLUDEFILES_H
```

B.4.4 File planetmodel.cpp

```
//planet.cpp
//define needed constants, global variables, callback functions
//draw the planet and its moon

#include <GL/glut.h>
#include <math.h>

#include "h/includefiles.h"

//some constant variables
const Color bg_color(1.0, 1.0, 1.0);    //background color
const int win_width = 600;              //size of window
const int win_height = 500;

const Color earth_col(0.0, 0.0, 0.0);
const Color moon_col(0.0, 0.0, 0.0);

enum { X, Y, Z } axis=X;
enum {FORWORD, UPWORD, LEFT, RIGHT} viewpoint = FORWORD;
```

```
const Real white[] = {1.0, 1.0, 1.0, 1.0};    // for lighting
const Real blue[] = {0.0, 0.0, 1.0, 1.0};
const Real dull[] = {100.0};

const Real rootTwo = sqrt(2.0);

//some needed global variables
Real vxpos = 7.0, vypos = 7.0, vzpos = 7.0;
Real alpha = 30.0;
Real w = 600, h = 500;                        //for zoom in/out

bool rot = false;                   //for rotate
Real x_angle = 0.0;
Real y_angle = 0.0;
Real z_angle = 0.0;

Real pos[] = {1.0, 1.0, 1.0, 0.0};       //default light position
Real la = 45.0;                          //original value for change light direction

//define objects
Sphere earth("earth");
Sphere moon("moon");

Camera camera(1.0, 0.0, 0.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0);

LightSource light;

//some member functions of class Keyboard
void Keyboard::setViewpoint()
{
        switch (viewpoint)
        {
        case FORWORD:
                camera.changePosition(0.0, 0.0, vzpos);
                camera.changeTarget(0.0, 0.0, 0.0);
                camera.changeUpDirection(0.0, 1.0, 0.0);
                camera.view();
                break;

        case UPWORD:
                camera.changePosition(0.0, vypos, 0.0);
                camera.changeTarget(0.0, 0.0, 0.0);
                camera.changeUpDirection(0.0, 0.0, -1.0);
                camera.view();
                break;

        case LEFT:
                camera.changePosition(-vxpos, 0.0, 0.0);
                camera.changeTarget(0.0, 0.0, 0.0);
                camera.changeUpDirection(0.0, 1.0, 0.0);
                camera.view();
                break;

        case RIGHT:
                camera.changePosition(vxpos, 0.0, 0.0);
                camera.changeTarget(0.0, 0.0, 0.0);
                camera.changeUpDirection(0.0, 1.0, 0.0);
                camera.view();
                break;

        default:
                break;
        }

}

void Keyboard::setFunctionKeys(unsigned int key)
{
   switch(key)
   {
        case F1KEY:
                viewpoint = FORWORD;
                break;

        case F2KEY:
                viewpoint = UPWORD;
                break;

        case F3KEY:
                viewpoint = LEFT;
                break;

        case F4KEY:
                viewpoint = RIGHT;
                break;

        case LEFTARROWKEY:
            if (light.show == true) {
                  la = la + 0.1;
                  pos[0] = rootTwo * cos(la);
                  pos[2] = rootTwo * sin(la);
            }
            break;

      case RIGHTARROWKEY:
            if (light.show == true) {
                  la = la - 0.1;
                  pos[0] = rootTwo * cos(la);
                  pos[2] = rootTwo * sin(la);
            }
            break;

      case UPARROWKEY:
            if (light.show == true) {
                  la = la - 0.1;
                  pos[1] = rootTwo * cos(la);
                  pos[2] = rootTwo * sin(la);
            }
            break;

     case DOWNARROWKEY:
            if (light.show == true) {
                  la = la + 0.1;
                  pos[1] = rootTwo * cos(la);
                  pos[2] = rootTwo * sin(la);
            }
            break;

     default:
            break;
  }
}

void Keyboard::setKeys(unsigned int key)
{
  switch(key)
  {
     case ESC:
            exit(0);
            break;

     case '+':          // for zoom-in
```

```
        case '=':
                alpha -= 0.25;
                if(alpha <= 0)
                        alpha = 0;
                break;

        case '-':               // for zoom-out
        case '_':
                alpha += 0.25;
                if(alpha >= 180)
                        alpha = 180;
                break;

        case 'x':
        case 'X':
                rot = true;
                axis = X;
                break;

        case 'y':
        case 'Y':
                rot = true;
                axis = Y;
                break;

        case 'z':
        case 'Z':
                rot = true;
                axis = Z;
                break;

        case 's':
        case 'S':
                rot = false;
                break;
        }
}

void Mouse::setMouseButton(int button, int state)
{
        if(button==LEFTBUTTON && state==BUTTONDOWN)
                light.show = false;

        else if(button==RIGHTBUTTON && state==BUTTONDOWN)
                light.show = true;
```

```
}


//an object of class Keyboard
Keyboard kb;

//an object of class Mouse
Mouse mouse;

//draw the planet and its moon
void drawPlanet()
{
        glPushMatrix();

        earth.setColor(earth_col);
        earth.setParameters(0.5, 200, 100);
        earth.draw();

        glRotatef(0.0, 0.0, 1.0, 0.0);
        glTranslatef(1.8f, 0.0, 0.0);
        glRotatef(0.0, 0.0, 1.0, 0.0);

        moon.setColor(moon_col);
        moon.setParameters(0.2, 200, 100);
        moon.draw();

        glutSwapBuffers();
        glPopMatrix();
}

void display ()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glMatrixMode(GL_PROJECTION | GL_MODELVIEW);
        glLoadIdentity();

        gluPerspective(alpha,w/h,1,20); //zoom in/out with changing alpha

        //set viewpoint
        kb.setViewpoint();

        if (rot)        //rotate round X/Y/Z aix
        {
                glRotatef(x_angle, 1.0, 0.0, 0.0);
                glRotatef(y_angle, 0.0, 1.0, 0.0);
                glRotatef(z_angle, 0.0, 0.0, 1.0);
        }

        drawPlanet();   // draw the planet system

        //set and display light
        light.setLightAmbient(blue);
        light.setLightDiffuse(blue);
        light.setLightSpecular(white);
        light.setLightShininess(dull);
        light.setLightPosition(pos);
        light.indicate();

        glFlush();
        glutSwapBuffers();
}

void spin(void)
{
        switch(axis)
        {
           case X:
             x_angle = x_angle + 1.0;
             break;

           case Y:
             y_angle = y_angle + 2.0;
             break;

           case Z:
             z_angle = z_angle + 2.0;
             break;

        }

        glutPostRedisplay();
}

//callback functions for controlling the animation
void special(int key, int x, int y)
{
        kb.setFunctionKeys(key);
        glutPostRedisplay();
}
```

```
void keyboard(unsigned char key, int x, int y)
{
        kb.setKeys(key);
        glutPostRedisplay();
}

void mousePress(int button, int state, int x, int y) {
        mouse.setMouseButton(button, state);
        glutPostRedisplay();
}

void reshape(int width, int height)
{
        w=width;
        h=height;

        glViewport(0,0,width, height);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(30.0,(GLfloat)width/(GLfloat) height, 1, 20);
        glutPostRedisplay();
}

void drawPlanetSystem()
{
        //craete a window for display graphic
        Window theWindow("Planet and its one moon");
        theWindow.setSize(win_width, win_height);
        theWindow.setBGColor(bg_color);
        theWindow.draw();

        //callback functions
        glutDisplayFunc(display);
        glutIdleFunc(spin);
        glutSpecialFunc(special);
        glutKeyboardFunc(keyboard);
        glutMouseFunc(mousePress);
        glutReshapeFunc(reshape);

        glEnable(GL_DEPTH_TEST);

        //draw the graphics system
        glutMainLoop();
}
```

B.4.5 File planet.cpp

```
//main.cpp

#include <iostream.h>

extern void drawPlanetSystem();

int main () {

        // Instruct user.
        cout <<
                "Change view point:"                                              << endl <<
                "   Press keys F1, F2, F3, F4."                                   << endl <<
                "Light:"                                                          << endl <<
                "Press mouse left and right button to toggle light effect."       << endl <<
                "Change the direction of light source:"                           << endl <<
                "   Press arrow keys to toggle light directions."                 << endl <<
                "Rotate:"                                                         << endl <<
                "   Press 'x' ('X') to rotate the system round x axis."           << endl <<
                "   Press 'y' ('Y') to rotate the system round y axis."           << endl <<
                "   Press 'z' ('Z') to rotate the system round z axis."           << endl <<
                "   Press 's' ('S') to stop rotating effect."                     << endl <<
                "Zoom in/out"                                                     << endl <<
                "           Press '+' ('=') to zoom in the window."               << endl <<
                "           Press '-' ('_') to zoom out the window."              << endl <<
                "Quit:"                                                           << endl <<
                "   Press ESC to stop program."                                   << endl;

        //draw the planet system
        drawPlanetSystem();

        return 0;
}
```