INFORMATION TO USERS

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

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

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

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

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

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

The Implementation of a 3D-Object Simulator Using Open-GL

Ping Ma

A Major Report
In
The Department
Of
Computer Science

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

May 2001

Abstract

The Implementation of a 3D-Object Simulator Using Open-GL

Ping Ma

This project demonstrates a prototype of a simple 3D objects simulation coded using Visual C++ Version 6 and OpenGL on Windows 98 environment. The purpose of this project is to design and implement a framework of 3D objects using object-oriented modeling and design methodology; simulate behavior of 3D objects and show a virtual world of objects. This report introduces a theoretical basis for the project and presents the implementation of 3D objects simulation. Finally, the simulation results and future work are presented.

Acknowledgments

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

Second I would like to express my sincere thanks to my partner, Mr. Shangbin Zou, for his contribution to this project -- described in the design of a 3D-Object Simulator.

Finally, I would like to dedicate this project to my family for their full supports, great patience, and encouragement.

Contents

1. Introduction

This project demonstrates a prototype of a simple 3D objects simulation coded using Visual C++ Version 6 and OpenGL on Windows environment. The purpose of this project is to design and implement a framework of 3D objects using object-oriented modeling and design methodology; simulate behavior of 3D objects; and show a virtual world of objects. This report describes the implementation of 3D objects simulation and another report "The Design of a 3D-Object Simulator Using Open-GL" [referring to the major report of Mr. Shangbin Zou] describes the design part of this project.

1.1 Aim of the Project

This project is about a simple 3D objects simulation. The aim of this project is to present an Object-Oriented approach to simulate interactions among 3D objects. Our intention is to design and implement a simulation environment that is as realistic as possible. A graphics user interface is provided so that a user can use it to set up initial simulation parameters, control simulation processing and view the simulation result.

1.2 Objectives

In our project, we use object-oriented programming (OOP) because of the following reasons:

- It provides us with experience in OOP techniques.
- For 3D objects simulation, it can take advantage of the strengths of object orientation, objects should usually represent real-world things.
- Build up a framework, easy to expand this system.

Since this program is intended to be an extendable framework, we think there are phase objectives and if this goes well there should be further phases.

- Allow people, who are not experts in 3D work, to create and manipulate interesting 3D worlds.
- The user should be able to create and manipulate simple shapes like cube, cone and sphere.
- The system should be able to simulate simple behaviors such as objects moving, objects rotating, objects colliding, etc.
- The program should use existing open standards such as: OpenGL and other storage and rendering standards.
- The system should be easily expandable by extending classes at build time. It should allow the addition of:
  - New shapes
  - New storage formats
  - New behaviors
  - Performance should be good enough for simple simulations.

1.3 Organization of the Report

This report is divided into five chapters. Chapter one is an introduction. It briefly describes the aim and objective of this project. Chapter two describes the background and all concepts of the project. Chapter three starts with an introduction to our implementation tools OpenGL and C++, and part codes of implementation are listed in

this chapter. Chapter four gives four simulation scenarios, each one presents a simulation result. The last chapter concludes the report and suggests further work for the project.

2. Theory behind Project

This chapter describes the background knowledge necessary for designing and implementing the 3D objects simulation.

2.1 Object-Oriented Programming

Object-oriented programming has become the dominant programming style in the software industry over the last ten years or so. The reason for this has to do with the growing size and scale of software projects. It becomes extremely difficult to understand a procedural program once it gets above a certain size. Object-oriented programs scale up better, meaning that they are easier to write, understand and maintain than procedural programs of the same size.

Object-oriented programming appeals at multiple levels. For managers, it promises faster and cheaper development and maintenance. For analysts and designers, the modeling process becomes simpler and produces a clear, manageable design. For programmers, the elegance and clarity of the object model and the power of object-oriented tools and libraries makes programming a much more pleasant task, and programmers experience an increase in productivity. [JMWFW97]

Object-oriented programs are organized around data. The three principles of object-oriented programming are: encapsulation, polymorphism, and inheritance.

2.1.1 Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.

Within an object, code, data, or both may be private to that object or public. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of program that access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object. [JMWFW97]

For all intents and purpose, an object is a variable of a user-defined type. Each time we define a new type of object, we are creating a new data type. Each specific instance of this data type is a compound variable.

2.1.2 Polymorphism

Object-oriented programming languages support polymorphism, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation. A real-world

example of polymorphism is a thermostat. No matter what type of furnace our house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) we have. For example, if we want a 70-degree temperature, we set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat.

This same principle can also apply to programming. For example, we might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, we can define one set of names, push() and pop(), that can be used for all three stacks. In our program we will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored. Thus, the interface to a stack -- the function push() and pop() -- are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data.

Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the specific action (i.e., method) as it applies to each situation. The programmer doesn't need to do this selection manually.

2.1.3 Inheritance

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

2.2 3D Graphics Fundamentals

Before getting into our project, we give the fundamental concepts of 3D graphics and coordinate system.

2.2.1 3D Perception

"3D Computer graphics" is actually two-dimensional images on a flat computer screen that provide an illusion of depth, or a third "dimension." In order to truly see in 3D, we need to actually view the object with both eyes, or supply each eye with separate and unique images. Each eye receives a two-dimensional image that is much like a temporary photograph on the retina. These two images are slightly different because they are

received at two different angles. The brain then combines these slightly different images to produce a single, composite 3D picture in our head, as shown in Figure 2-1.

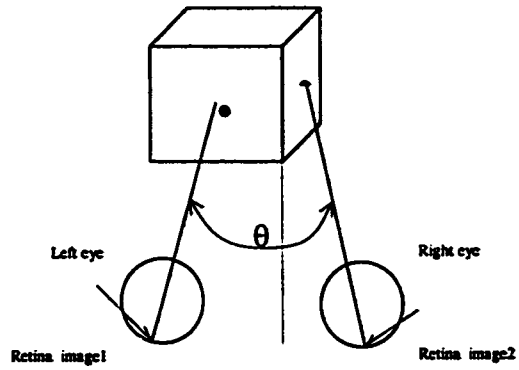

Figure 2.1 How the eyes "see" three dimensions

2.2.1.1 2D + Perspective = 3D

Why doesn't the world suddenly flatten when we cover one eye? The reason is that many of a 3D world's effects are also present in a 2D world. This is just enough to trigger our brain's ability to discern depth. The most obvious cue is that nearby objects appear larger than distant objects and parallel lines do not appear parallel, etc. This effect is called *perspective*. Figure 2-2 presents a simple wire frame cube. [Rsw96]

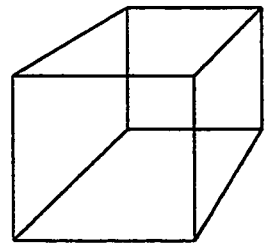

Figure 2-2 This simple wire frame cube demonstrates perspective.

2.2.1.2 Hidden Line Removal

A wire frame cube like figure 2-2 contains enough information to lend the appearance of three dimensions, but not enough to let us discern the front of the cube from the back. When viewing a real object, how do tell the front from the back? Simple – the back is obscured by the front. To simulate this in a two-dimensional drawing, lines that would be obscured by surfaces in front of them must be removed. This is called hidden line removal, as shown in figure 2-3.

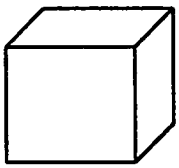

Figure 2-3 The cube after hidden lines are removed

2.2.1.3 Colors and Shading

Figure 2-3 still doesn't look much like a real-world object. The faces of the cube are exactly the same as the background. A real cube would have some color and / or texture. Unless we specifically draw the edges in a different color, there is no perception of three dimensions at all, as shown in Figure 2-4. In order to regain the perspective of a solid object, we need to either make each of the three visible sides a different color, or make them the same color with shading to produce the illusion of lighting.

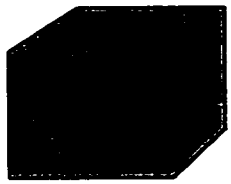

Figure 2-4 The cube with color, but no shading

2.2.1.4 Lights and Shadows

Lighting has two important effects on objects viewed in three dimensions. First, it causes a surface of a uniform color to appear shaded when viewed or illuminated from an angle. Second, objects that do not transmit light cast shadows when they obstruct the path of a ray of light, as shown in figure 2-5.

Figure 2-5 A solid cube illuminated by a single light

Two sources of light can influence our three-dimensional objects. Ambient light, which is undirected light, is simply a uniform illumination that can cause shading effects on objects of a solid color; ambient light causes distant edges to appear dimmer. Another kind of light is from a light source, called a lamp. Lamps can be used to change the shading of solid objects and shadow effects.

2.2.2 Coordinate Systems

When drawing points, lines, or other shapes on a computer screen, we usually specify a position in terms of a row and column. If a standard VGA screen has 640 x 480 pixels, the middle of screen should be plotted at (320,240)- that is, 320 pixels from the left of the screen and 240 pixels down from the top of the screen.

2.2.2.1 2D Cartesian Coordinates

The most common coordinate system for two-dimensional plotting is the *Cartesian* system. Cartesian coordinates are specified by an x-coordinate and a y-coordinate. The x-coordinate is a measure of position in the horizontal direction and y is a measure of position in the vertical direction.

The origin of the Cartesian system is at x=0, y=0. Cartesian coordinates are written as coordinate pairs, in parentheses, with the x-coordinate first and the y-coordinate second, separated by a comma. The x and y lines with marks are called the axes and can extend from negative to positive infinity. The x-axis and y-axis are perpendicular and together define the xy plane, as shown in figure 2-6. [Rsw96]

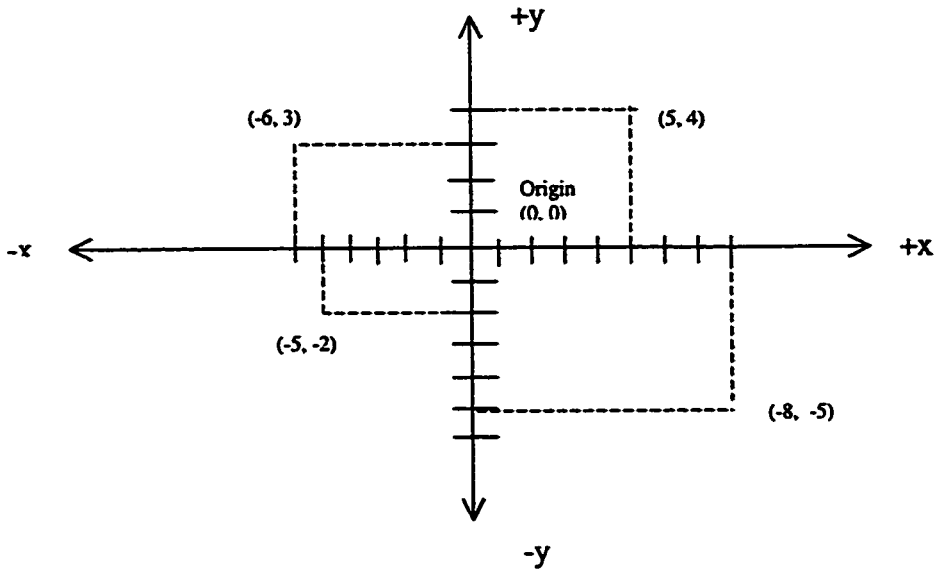

Figure 2-6 The Cartesian plane

2.2.2.2 Coordinate Clipping

A window is measured physically in term of pixels. Before plotting any shapes in a window, we must translate specified coordinate pairs into screen coordinates. This is done by specifying the region of Cartesian space that occupies the window; this region is known as the clipping area. In two-dimensional space, the clipping area is the minimum and maximum x and y values that are inside the window. Figure 2-7 shows a common clipping area, x-coordinates in the window range left to right from 0 to +150, and y-coordinates range bottom to top from 0 to +100. A point in the middle of the screen would be represented as (75,50).

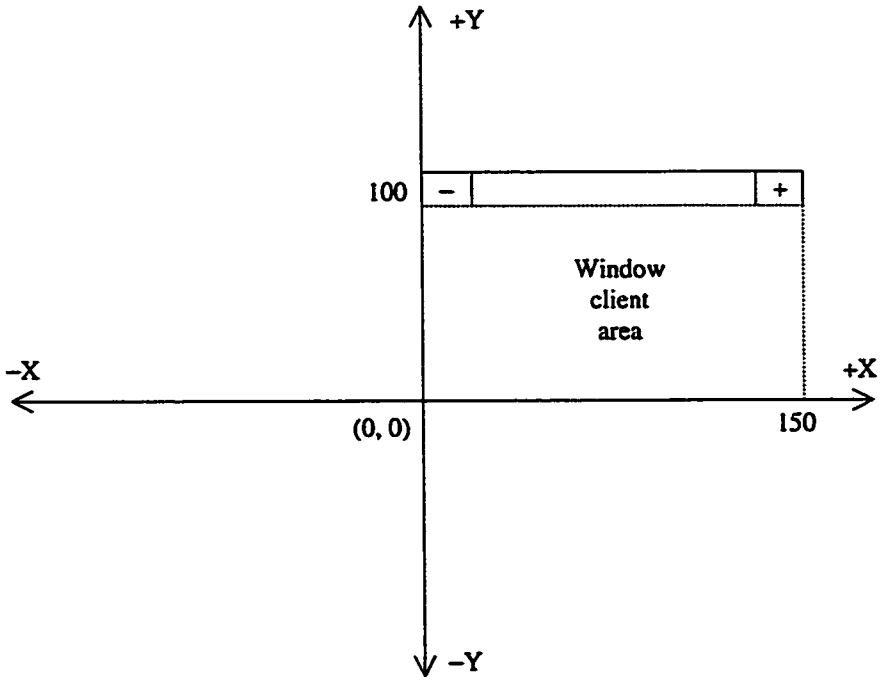

Figure 2-7 One clipping area

2.2.2.3 Viewports

Rarely will a clipping area width and height exactly match the width and height of the window in pixels. The coordinate system must therefore be mapped from logical Cartesian coordinates to physical screen pixel coordinates. This mapping is specified by a setting known as the viewport. The viewport simply maps the clipping area to a region of the window. Usually the viewport is defined as the entire window, as shown in Figure 2-8.

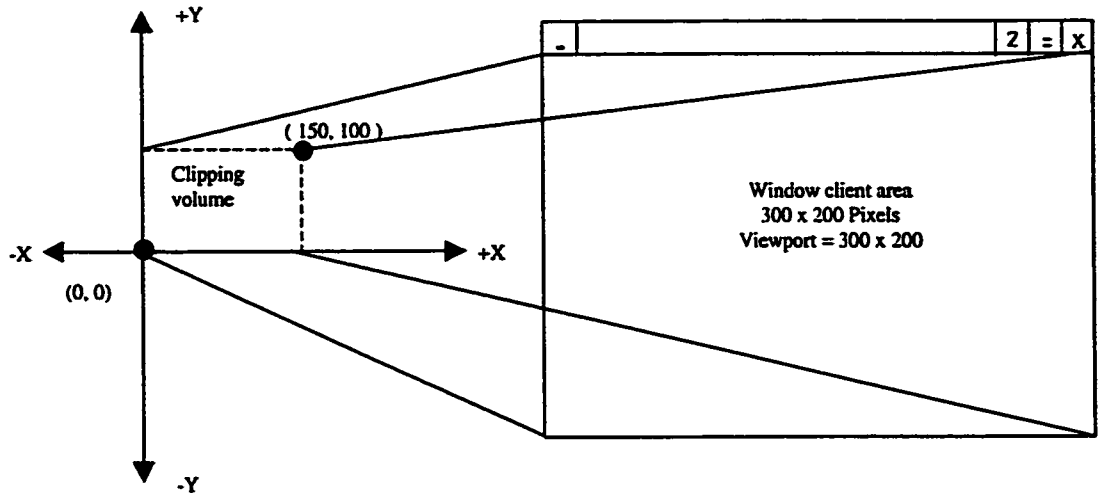

Figure 2-8 A viewport

2.2.2.4 Drawing Primitives

When drawing an object, we actually compose it with several smaller shapes called *primitives*. Primitives are two-dimensional surfaces such as points, lines, and polygons that are assembled in 3D space to create 3D objects. A three-dimensional cube is made up of six two-dimensional squares, each placed on a separate face. Each corner of the square is called a vertex. These vertices are then specified to occupy particular coordinates in 2D or 3D space.

2.2.2.5 3D Cartesian Coordinates

A three-dimensional coordinate system adds a new axis z to the two-dimensional coordinate system. The z-axis is perpendicular to both the x- and y-axes. It represents a line drawn perpendicularly from the center of the screen heading toward the viewer, as shown in figure 2-9.

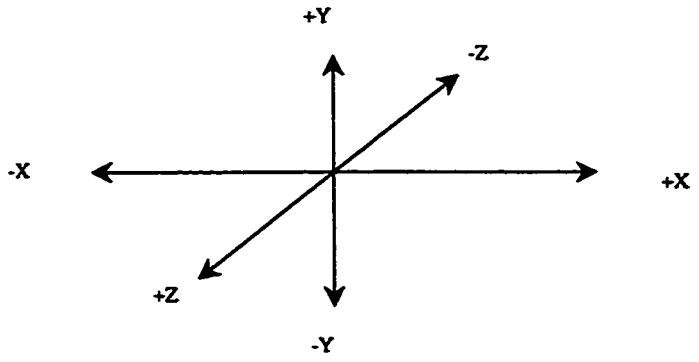

Figure 2-9 Cartesian coordinates in three dimensions

2.2.3 Projections

A mapping of 3D coordinates onto a 2D surface is called a projection. The two main types of projections are orthographic and perspective projections.

2.2.3.1 Orthographic Projections

This projection specifies a square or rectangular clipping volume. Anything outside this clipping area is not drawn. Furthermore, all objects that have the same dimensions appear the same size, regardless of whether they are far away or nearby. This produces a parallel projection, which is useful for drawings of specific objects that do not have any foreshortening when viewed from a distance. A clipping volume in an orthographic projection is defined by specifying the far, near, left, right, top, and bottom clipping planes, as shown in figure 2-10.

In order to view a three-dimensional object on a two-dimensional screen, we must project it. For orthographics projections, the simplest kind of projection simply ignores one

coordinate, the transformation $[x, y, z, 1]^T \rightarrow (x, y)$ provides a screen position for each point by directly ignoring its z coordinate. Here, (x , y, z) defines a point.

Orthographic projections do not give a strong sensation of depth because the size of an object does not depend on its distance from the view. An orthographic projection simulates a view from an infinite distance away.

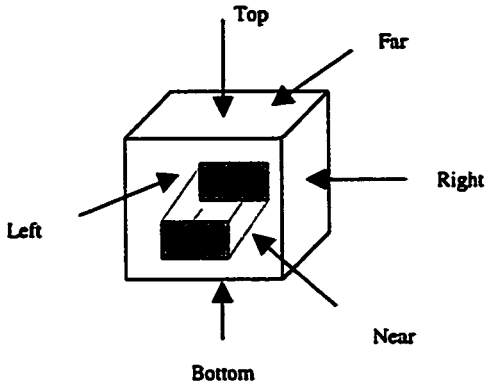

Figure 2-10 The clipping volume for an orthographic projection

2.2.3.2 Perspective Projections

A more common projection is the perspective projection. This projection adds the effect that distant objects appear smaller than nearby objects. The viewing volume as shown in figure 2-11 is something like a pyramid with the top shaved off. This shaved off part is called the frustum. The top and bottom of the frustum determine the distances of the nearest and furthest points that are visible. The sides of the frustum determine the visibility of objects in the other two dimensions. If we think of the closer vertical plane as the screen, the closest objects lie in the plane of the screen and other objects lie behind it.

Objects nearer to the front of viewing volume appear close to their original size, while objects near the back of the volume shrink as they are projected to the front of the volume. This type of projection gives the most realism for simulation and 3D animation.

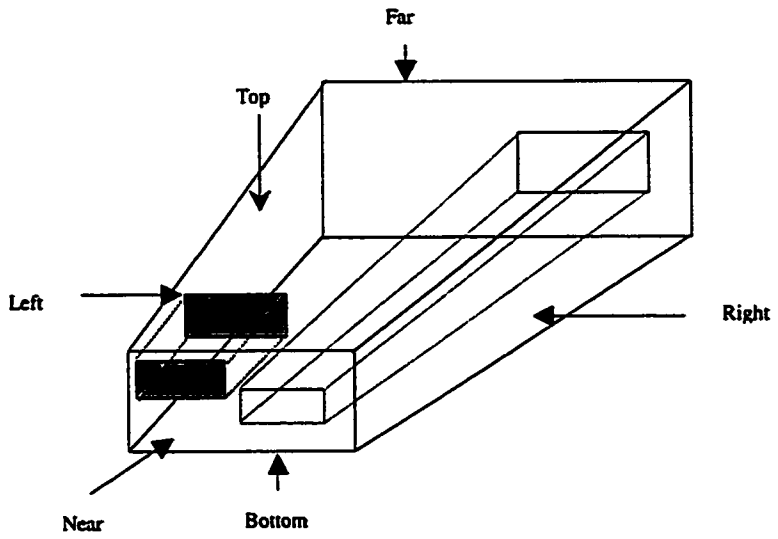

Figure 2-11 The clipping volume for a perspective projection

How can we map 3D coordinates onto a 2D surface? The simplest way is still to ignore one coordinate. Perspective transformations provide a better sense of depth by distant objects with reduced size. The size reduction is a natural consequence of viewing the object from a finite distance. In figure 2-12, E represents the eye of the viewer viewing a screen, the model is behind the screen. An object at P in the model appears on the screen at the point P'. [Pg98]

The point P in the model is $[x, y, z, 1]^T$. The x value does not appear in the diagram becase it is perpendicular to the paper. The transformed coordinates on the screen are (x', y'): there is no z coordinate because the screen has only two dimensions. By similar triangles :

$$x' / d = x / (z+d)$$

$$y' / d = y / (z+d)$$

and hence

$$x' = x*d / (z + d)$$

$$y' = y*d / (z + d)$$

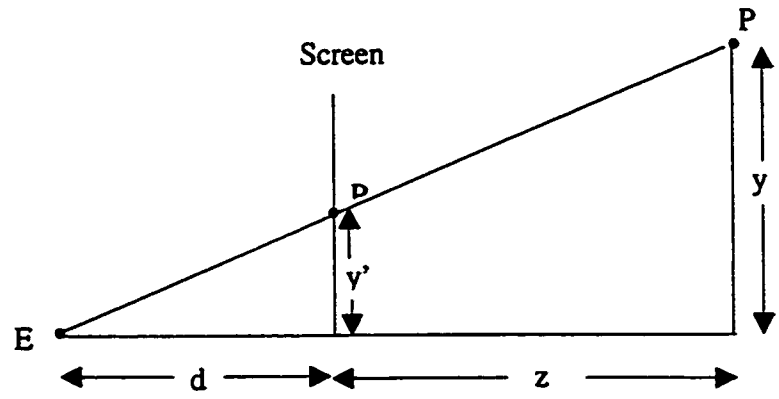

Figure 2-12 Perspective transformation

2.3 3D Physics Concepts

How do we build a 3D simulation of physical object? 3D programs like OpenGL use transformations to render to the screen, but how to we to simulate object's interactions?

2.3.1 Modeling Newtonian Mechanics

To model Newtonian mechanics, the following things are needed:

- Model motion
- Collision detection
- Collision response

2.3.1.1 Modeling Motion

Transformations can be used to model motion, an object's movement could be placed under a transform matrix and its translation modified each frame. In order to calculate these, the parameters such as energy, mass, linear and angular momentum, centre-of-mass, etc. would be needed.

2.3.1.2 Collision Detection

To prevent two objects from occupying the same volume of space, collision detection is needed. Collision detection is quite difficult for arbitrary shapes. Even for rigid objects, there is also considerable overhead when there are a lot of objects to be tested if a collision may happen between every object and any other object.

2.3.1.3 Collision Response

Once a collision happens, the effect of the collision should be calculated, new velocities are generated.

2.3.1.4 Analytical verses Numerical Methods

There are generally two methods to solve the problems outlined above: analytical methods or numerical methods. Analytical methods are used to solve some complex equations, and therefore require a lot of work at design time; numerical methods involve less complex equations but more calculation is done at runtime. In the most general case numerical methods probably are used, but in more restricted cases analytical methods are used.

2.3.2 3D Physics - Solid Objects

Standards such as OpenGL allow the geometry and appearance of solid objects to be defined.

2.3.2.1 Kinematics

In this section, we will discuss some kinematics theory.

2.3.2.1.1 Translation

Translations or relative positions can be represented by vectors of dimension three. In the case of representing a new position relative to the old position, translations can be used mathematically by adding the vectors. If a translation [ax, ay, az] is applied and then another translation [bx, by, bz], the resulting translation will be [ax+bx, ay+by, az+bz]. The usual rules of vector addition are:

[A] + [B] = [B] + [A] = [ax+bx, ay+by, az+bz].

A complex object can be translated by applying the translation to each point on the object. These translations can be used to describe the linear movement of an object.

2.3.2.1.2 Rotation

If an object is not restrained then it can rotate in addition to moving linearly. This gives six degrees of freedom: three for linear movement, three for rotation.

2.3.2.1.3 Uniform Velocity

An object is moving at a constant speed in the x-direction, without rotation, its motion is given by:

$$v = \partial x / \partial t$$

Here, x means moving distance of an object, v means moving speed of an object.

Similarly, an object is moving at a constant speed in the y-direction, without rotation, its motion is given by:

$$v = \partial y / \partial t$$

and an object is moving at a constant speed in the z-direction, without rotation, its motion is given by:

$$v = \partial z / \partial t$$

2.3.2.1.4 Uniform Acceleration

If an object is moving at a constant acceleration, its acceleration is given by:

$$a = \partial v / \partial t$$

Where v means moving speed of an object and t means moving time of an object.

In the more general case of translation in three dimensions:

$$[a] = \begin{bmatrix} \partial^2 x / \partial t^2 \\ \partial^2 y / \partial t^2 \\ \partial^2 z / \partial t^2 \end{bmatrix}$$

Where x, y and z stand for respectively the moving distances of an object in x-direction, y-direction and z-direction.

2.3.2.2 Newtonian Laws

If objects are representing physical objects, we probably calculate the kinematics from the dynamics. For applications such as games and simulations of normal objects we can use Newtonian methods.

Newton defines three laws:

- If no forces act on a particle, the particle retains its linear momentum.
- The rate of change of the linear momentum of a particle is equal to the sum of all forces acting on it.
- When two particles exert forces upon each other, these forces are equal in magnitude and opposite in direction.

2.3.2.3 Motion with No External Torque

What are the equations for the movement of a rigid object with no external forces or torques acting on it? This is the simplest case that we can imagine for physics simulation, so we would like to make sure that we fully understand it and can represent it

programmatically before going on to more complex situations like collisions and jointed structures.

For a rigid object rotating in free space, by Newton's first law.

- Linear momentum of centre-of-mass in x dimension = constant.
- Linear momentum of centre-of-mass in y dimension = constant.
- Linear momentum of centre-of-mass in z dimension = constant.
- Angular momentum about x-axis through centre-of-mass = constant.
- Angular momentum about y-axis through center-of-mass = constant.
- Angular momentum about z-axis through mass-of-mass = constant.

So the linear position of the center of mass is given by:

$$[p] = [p0] + [v]t$$

where:

[p] = position vector in x, y and z dimensions

[p0] = position vector at t=0

[v] = velocity

t = time (scalar)

2.3.2.4 Dynamics Collision

The dynamic equations for a single object were covered in the previous section. Here we extend this analysis to two objects so that we can calculate the result of a collision between the objects.

Linear momentum, angular momentum and energy is passed between the colliding shapes at the point of collision by means of impulse. Impulse is the integral of force over time. This is measured in Newton-seconds. For rigid body collisions, the impact is assumed to happen in an infinitesimally small time.

Collisions in three dimensions are quite a complex problem. If a body is solid and it collides with another body then it may bounce off the other, or slide and/or deform depending on the conditions such as coefficient of friction, elasticity of the bodies, etc. [Web3D]

3. Implementation

We implement this project using visual C++ v6.0 and OpenGL. This part involves the basic concepts of C++ and OpenGL, definitions and implementations of classes, the detections of boundary and collision, collision response, algorithms for dropping down, and so on.

3.1 What Is C++?

C++ is an object-oriented programming language. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types. A programmer can partition an application into manageable pieces by defining new types that closely match the concepts of the application. This technique for program construction is often called data abstraction. Objects of some user-defined types contain type information. Such objects can be used conveniently and safely in contexts in which their type cannot be determined at compile time. Programs using objects of such types are often called object based. When used well, these techniques result in shorter, easier to understand, and easier to maintain programs.

The key concept in C++ is the *class*. A class is a user-defined type. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management, and mechanisms for overloading operators. C++ provides much better facilities for type checking and for expressing modularity than C does. It also contains improvements that are not directly related to

classes, including symbolic constants, inline substitution of functions, default function arguments, overloaded function names, free store management operators, and a reference type. C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.). This allows the user-defined types to be implemented with a pleasing degree of efficiency.

C++ and its standard libraries are designed for portability. The current implementation will run on most systems that support C. C libraries can be used from a C++ program, and most tools that support programming in C can be used with C++.

3.2 What is OpenGL?

OpenGL is a powerful and sophisticated application programming interface (API) for creating 3D graphics, with over 300 functions that cover everything from setting material colors and reflective properties to doing rotations and complex coordinate transformations.

3.2.1 OpenGL-Related Libraries

In fact, OpenGL not only provides a powerful primitive set of rendering functions, but also provide specialized features such libraries and routines.

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

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

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

3.2.2 OpenGL Function Syntax

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

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

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

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

Where :

gl : gl library

Color : RootCommand

3 : Number of arguments

f : Type of arguments

3.2.3 OpenGL Data Types

OpenGL functions accept as many as eight different data types for their arguments, shown in table 3-1.

Suffix	Data Type	OpenGL Type Definition
b	8-bit integer	Glbyte
s	16-bit integer	Glshort
i	32-bit integer	GLint, Glsizei
f	32-bit floating-point	GLfloat, Glclampf
d	64-bit floating-point	GLdouble, Glclampd
ub	8-bit unsigned integer	GLubyte, Glboolean
us	16-bit unsigned integer	Glushort
ui	32-bit unsigned integer	GLuint, GLenum, Glbitfield

Table 3-1 Function suffixes and argument data types

Thus, the two functions

glVertex2i(1, 3);

glVertex2f(1.0, 3.0);

are equivalent, except that the first specifies the vertex's coordinates as 32-bit integers and the second specifies them as single-precision floating-point numbers. Both functions have two argumants.

3.3 Project Implementation

In this project, we implemented four programs. The first one demonstrates a cube, which moves on a table; the second one shows two cubes, which are apart from each other in the initial state, and one cube would collide with another, and fall down from the table; the third one indicates how to simulate the movement of two cubes, where one cube is on another; the last one illustrates two spheres. In the following sections, we will discuss the implementations in detail.

3.3.1 Create an OpenGL Window

Before displaying anything by OpenGL, first we should set up a display window. In this project we use following codes to build the window.

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize (Window_Width= 600, Window_Height = 400 );
glutInitWindowPosition (Init_Position = 20, Init_Position = 20);
glutCreateWindow ("My Project");
```

Here we create an OpenGL display window with following characteristics:

- Double buffer. To avoid flicker, we set the window with attribute of double buffer, thus OpenGL renders the image into one buffer while displaying the contents of the other buffer.

- Depth buffer. Using depth buffer, we can eliminate hidden surface. That means that OpenGL displays only the objects that the viewer can see.
- Use RGB color mode.
- The window size is 600 x 400, and its initial position is at (20 x 20).
- The name of the window is "My Project"

3.3.2 Menus and Hot Keys

Menus and hot keys are used to operate our applications; we interact the applications by menu system or hot keys. The menu system and a group of hot keys might issue the same actions, the menu system is driven by mouse, while hot keys are driven by keyboard.

3.3.2.1 Menu System

GLUT provides menus. The format of menus is restricted but the functions are easy to use. In our project, we use two levels of menus – the first level is main menu, and the second level is submenu. The structure of menu system is shown as figure 3-1. [Pg98]

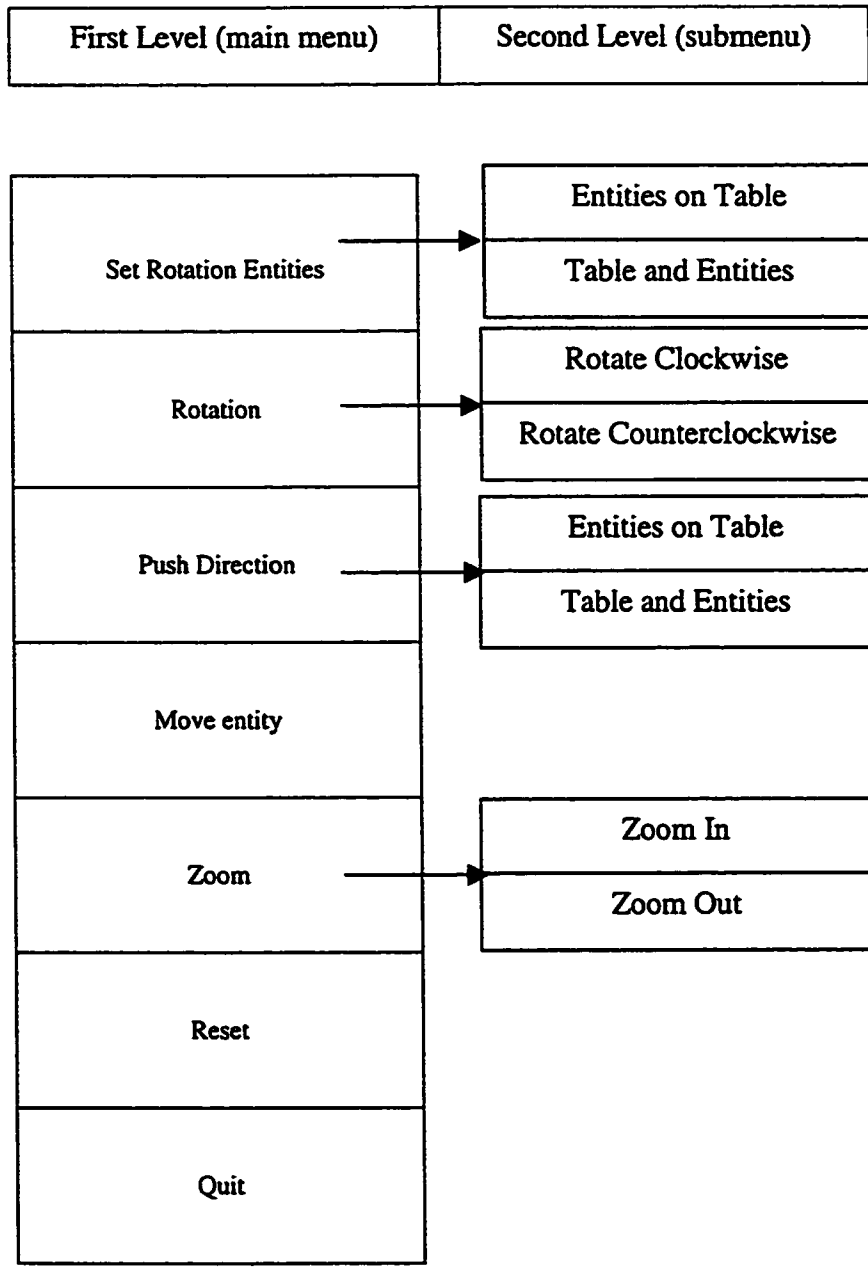

Figure 3-1 Structure of menu system

Several GLUT functions are very useful for an OpenGL menu system.

- Function *glutCreateMenu(func)* is used to create a menu. *func* is the name of a function that will handle mouse-selection events.

- Function *glutAddMenuEntry(string, num)* is used to add an item to the menu system. *string* is the string that will be displayed in the menu, while num is a constant.
- Function *glutAttachMenu(button)* is used to attach the menu to a mouse button. *button* should be one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON.
- To create a submenu, we should keep the value returned by the function *glutCreateMenu(func)*, complete the submenu with calls to *glutAddMenuEntry()*, but do not attach it to a button. [Pg98]

By combinations of these function calls, we can set up any menu system we want. Figure 3-2 shows part of the implementation of our menu system, where function *InitializeMenu()* is used to set up a menu system, function *MainMenu(int sel)* is the name of function that will handle menu-selection events, while function *Zoom(sel)* is the implementation of submenu. The variable *PerspectiveAngle* which is referenced in function *Zoom(sel)* is a field-of-view angle in the vertical direction of perspective projection; it is the first parameter of function *gluPerspective(PerspectiveAngle, aspect, zNear, zFar)*.

```
#defineM_ZOOMIN                 410
#define M_ZOOMOUT               411
#defineM_QUIT                   99

void Zoom(int sel)
{
```

```
        switch(sel) {
        case M_ZOOMIN:
                if(PerspectiveAngle >= 20 && PerspectiveAngle <= 70)
                        PerspectiveAngle -= 10;
                break;
        case M_ZOOMOUT:
                if(PerspectiveAngle <= 70)
                        PerspectiveAngle += 10;
                break;
        }
        glutPostRedisplay();
}


void MainMenu(int sel)
{
        switch(sel) {
                case M_QUIT:        exit(0);
                default:            break;
        }
        glutPostRedisplay();
}

void InitializeMenu(void)
```

```
{
    int SubMenuZoom;

    SubMenuZoom = glutCreateMenu(Zoom);
    glutAddMenuEntry("Zoom In", M_ZOOMIN);
    glutAddMenuEntry("Zoom Out", M_ZOOMOUT);
    glutCreateMenu(MainMenu);
    glutAddSubMenu("Zoom", SubMenuZoom);
    glutAddMenuEntry("Quit", M_QUIT);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
}
```

Figure 3-2 Part of the implementation of the menu system

3.3.2.2 Hot-keys

For ease of operating the applications, it's better to use a group of hot-keys instead of menus. GLUT provides a callback function *glutKeyboardFunc(keyboard)* to define hot-keys. Once the program has registered a keyboard callback function, this function would be called whenever a key is pressed.

3.3.3 Magnifying or Reducing an Image

Normally, each pixel in an image is written to a single pixel on the screen. However, you can arbitrarily magnify or reduce an image by changing the perspective angle of perspective projection.

We can implement zoom-in and zoom-out by changing the perspective angle. If a request of magnifying an image is issued, we reduce the perspective angle by a value, so that it results in magnifying an image. However, if a request of reducing an image is issued, we increase the perspective angle by a value, so that it results in reducing the image.

3.3.4 Entity's Movement

Once a force is applied to an entity, the entity should move a distance. Suppose the entity is a cube (the algorithm is the same for all other entities), it may rotate any angle about the Y-axis on a table. That makes it possible that there is an angle between the force and the X-axis, as shown in figure 3-3.

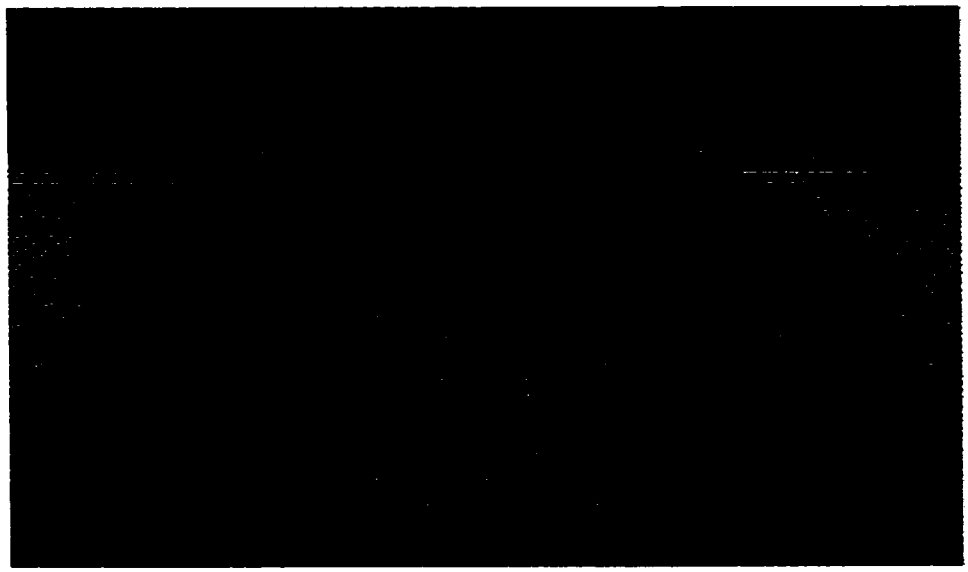

Figure 3-3 Angle between a force and X-axis

To simplify the description of a cube's movement, we name the four surfaces of the cube. When a cube object is created (initial position), the surface towards (–X) direction is called as *LeftSurface*; the surface towards (+X) direction is called as *RightSurface*; the surface towards (-Z) direction is called as *BackSurface*; and the surface towards (+Z) direction is called as *FrontSurface*. Whatever the cube rotates, the surface names keep unchanged, as shown in figure 3-4.

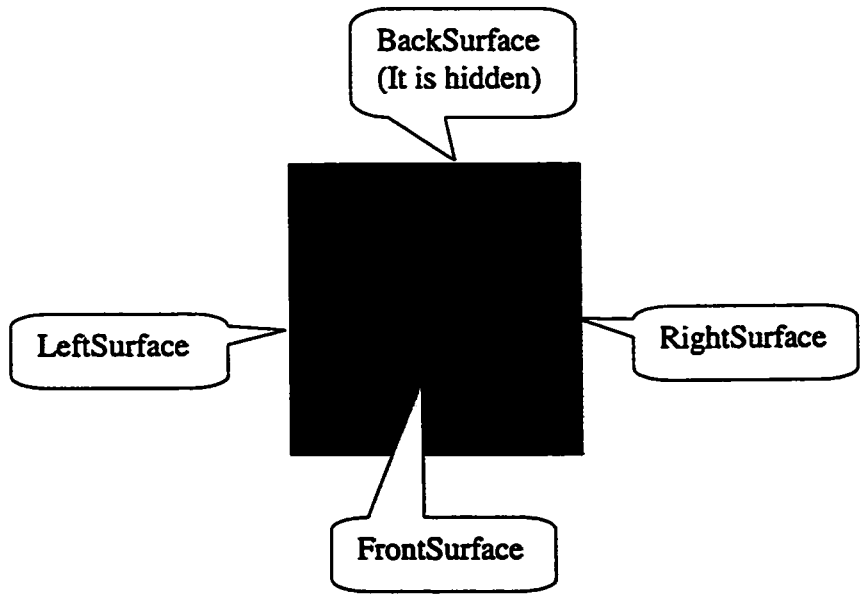

Figure 3-4 Naming the surfaces of a cube

Suppose that there is a force acted on the surface *LeftSurface* of a cube, the angle between force and X-axis be α, then the entity would move a distance *Step* after the force, as shown in figure 3-5.

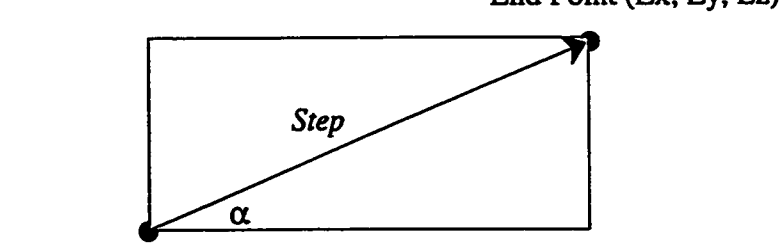

Figure 3-5 There is an angle between force and X-axis

The target coordinate of the cube after moving may be calculated by:

$$Ex = Sx + Step * \cos(\alpha)$$

$$Ey = Sy$$

$$Ez = Sz - Step * \sin(\alpha)$$

Similarly, we can calculate how a cube moves when there is a force that acts on one of the surfaces *RightSurface*, *FrontSurface* and *BackSurface*. The codes to implement this are shown as figure 3-6.

- If a force acts on the surface *LeftSurface* of a cube

```
TargetPositionX = CurrentPositionX + Step *
                  cos((double)ConvertAngle(itsCube.GetRotationAngle()));
TargetPositionY = CurrentPositionY;
TargetPositionZ = CurrentPositionZ - Step *
                  sin((double)ConvertAngle(itsCube.GetRotationAngle()));
```

- If a force acts on the surface *RightSurface* of a cube

```
TargetPositionX = CurrentPositionX - Step *
                  cos((double)ConvertAngle(PI - itsCube.GetRotationAngle()));
TargetPositionY = CurrentPositionY;
TargetPositionZ = CurrentPositionZ - Step *
                  sin((double)ConvertAngle(PI - itsCube.GetRotationAngle()));
```

- If a force acts on the surface *FrontSurface* of a cube

```
TargetPositionX = CurrentPositionX - Step *
                  sin(double)ConvertAngle(itsCube.GetRotationAngle()));
TargetPositionY = CurrentPositionY;
TargetPositionZ = CurrentPositionZ - Step *
                  cos((double)ConvertAngle(itsCube.GetRotationAngle()));
```

- If a force acts on the surface *BackSurface of* a cube

```
TargetPositionX = CurrentPositionX + Step *
                  sin(double)ConvertAngle(itsCube.GetRotationAngle()));
TargetPositionY = CurrentPositionY;
TargetPositionZ = CurrentPositionZ + Step *
                  cos((double)ConvertAngle(itsCube.GetRotationAngle()));
```

Figure 3-6 Codes for an entity's movement

Where (*CurrentPositionX, CurrentPositionY, CurrentPositionZ*) is the entity's start position; (*TargetPositionX, TargetPositionY, TargetPositionZ*) is the entity's target position; *Step* is a distance between the start point to the end point; *ConvertAngle()* is a function, which convert an angle to radian; *PI* is a constant with the value 3.1415926.

3.3.5 Boundary Check

Boundary checking is very important to the project. As we know, whenever an entity moves on a table, it is necessary to check whether the entity has reached the boundary of the table - we can then decide whether the entity will keep moving on the table or drop down from the table.

We say an entity reaches a boundary of the table, that means its center of gravity has reached an edge of the table. With a table, let the X-coordinate of the left edge be X_L, the X-coordinate of the right edge be X_R, the Z-coordinate of the front edge be Z_F, and the Z-coordinate of the back edge be Z_E, as shown in figure 3-7.

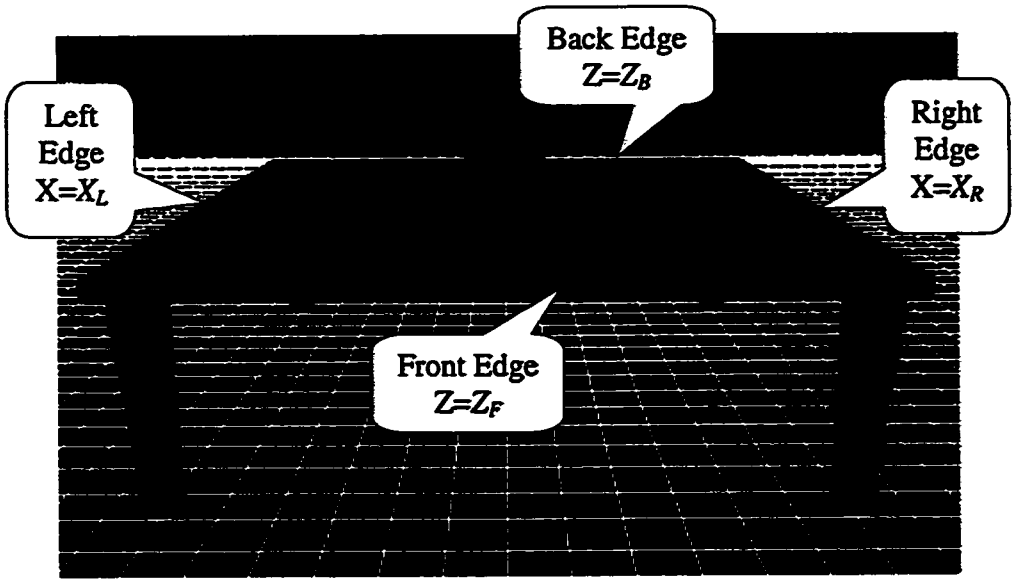

Figure 3-7 Description of table edges

For example, if the X-coordinate of the cube equals to (or greater than) the right edge of the table X_R, that means the cube has reached the boundary of the table, and would drop down from the table. Therefore, the conditions that a cube reaches table's boundary are

Cube's X-coordinate equals to (or greater than) X_R, or

Cube's X-coordinate equals to (or less than) X_L, or

Cube's Z-coordinate equals to (or greater than) Z_T, or

Cube's Z-coordinate equals to (or less than) Z_B.

A data type called *EntityPositionStatus* is introduced to describe the position status of an entity:

```
typedef enum {
        OnTable,
        DropFromLeftEdge,
```

DropFromFrontEdge,

DropFromRightEdge,

DropFromBackEdge,

FinishDropSimulation

} EntityPositionStatus;

OnTable means an entity never reaches table's boundary; *DropFromLeftEdg* means an entity has reached the table's left edge, and will drop down from the table; *DropFromFrontEdge* means an entity has reached table's front edge, and will drop down from the table; *DropFromRightEdge* means an entity has reached table's right edge, and will drop down from the table; *DropFromBackEdge* means an entity has reached table's back edge, and will drop down from the table; *FinishDropSimulation* means an entity has finished the whole dropping process, and has dropped to the ground.

Figure 3-8 shows an implementation of boundary checking. It is a member function of class *Cube*, the four parameters of the function *CheckBoundary()* stand for the values of X_L, X_R, Z_T and Z_B respectively; member function *SetPositionStatus()* is used to set the object's position status; while the variables of *PositionX, PositionY, PositionZ* are the values of the object.

```
void Cube :: CheckBoundary(GLfloat BoundaryMinX,
                           GLfloat BoundaryMaxX,
                           GLfloat BoundaryMinZ,
                           GLfloat BoundaryMaxZ)
```

```
{
        if(PositionX < BoundaryMinX)
                SetPositionStatus(DropFromLeftEdge);
        if(PositionX > BoundaryMaxX)
                SetPositionStatus(DropFromRightEdge);
        if(PositionZ < BoundaryMinZ)
                SetPositionStatus(DropFromBackEdge);
        if(PositionZ > BoundaryMaxZ)
                SetPositionStatus(DropFromFrontEdge);
}
```

Figure 3-8 An implementation of boundary check

3.3.6 Collision Detection

To avoid entities from occupying the same space, we must perform collision detection. Collision detections are complex checking algorithms that will vary when the types of entities change. In this section we will discuss two algorithms and their implementations, which show how collisions are detected for two cubes and two spheres.

3.3.6.1 Collision Detection of Two Cubes

If there are two or more cubes on a table, before a cube moves, collision detection must be performed. We should verify if two cubes occupy the same space.

Consider that there are two cubes on a table, one is apart from another for the initial positions. When cubes move, collision may happen. The collision detection should be done in both X and Z directions if we assume that cubes do not move along the Y direction.

Suppose that there are two cubes on a table – *Cube1* and *Cube2*. The *Cube1* has a position of *(X1, Y1, Z1)* with side length of *SideLength1*, and *Cube2* has a position of *(X2, Y2, Z2)* with side length of *SideLength2*. Let rotation angles of the cubes around Y axis be zero, if a force is applied along X-direction (or –X direction), a collision happens only when following condition is true:

((abs(X1 – X2) < SideLength1 / 2.0 + SideLength2 / 2.0) && ((Z2 – SideLength2 / 2.0 <= Z1 – SideLength1 /2.0 <= Z2 + SideLength2 / 2.0) || (Z2 – SideLength2 / 2.0 <= Z1 + SideLength1 /2.0 <= Z2 + SideLength2 / 2.0)))

||

((abs(X1 – X2) < SideLength1 / 2.0 + SideLength2 /2.0) &&((Z1 – SideLength1 / 2.0 <= Z2 – SideLength2 / 2.0 <= Z1 + SideLength1 / 2.0) || (Z1 – SideLength1 / 2.0 <= Z2 + SideLength2 / 2.0 <= Z1 + SideLength1 / 2.0)))

where "abs" is used to calculate absolute value of an expression,

"&&" stands for logic and,

"||" stands for logic or, and

"sqrt" is used to calculate square root of an expression.

If a force is applied along Z-direction (or –Z direction), the collision detection algorithm is the same. The figure 3-9 shows part of the code which implement collision detection.

```
GLfloat Cube1MinX=itsCube1.GetPositionX() - itsCube1.GetSideLength()/2.0;
GLfloat Cube1MaxX=itsCube1.GetPositionX() + itsCube1.GetSideLength()/2.0;
GLfloat Cube1MinZ=itsCube1.GetPositionZ() - itsCube1.GetSideLength()/2.0;
GLfloat Cube1MaxZ=itsCube1.GetPositionZ() + itsCube1.GetSideLength()/2.0;
GLfloat Cube2MinX=itsCube2.GetPositionX() - itsCube2.GetSideLength()/2.0;
GLfloat Cube2MaxX=itsCube2.GetPositionX() + itsCube2.GetSideLength()/2.0;
GLfloat Cube2MinZ=itsCube2.GetPositionZ() - itsCube2.GetSideLength()/2.0;
GLfloat Cube2MaxZ=itsCube2.GetPositionZ() + itsCube2.GetSideLength()/2.0;

if(itsCube1.GetPushDirectionValue() == Left) {
        if((Cube1MaxX >= Cube2MinX) && (Cube1MaxX <= Cube2MaxX) &&
        ( ((Cube1MinZ <= Cube2MaxZ) && (Cube1MinZ >= Cube2MinZ))
        || ((Cube1MaxZ <= Cube2MaxZ) && (Cube1MaxZ >= Cube2MinZ)))) {
                itsCube1.SetCollisionFalg(TRUE);
                itsCube2.SetCollisionFalg(TRUE);
        }
}
```

Figure 3-9 Implementation of collision detection of cubes

Where *GetPositionX()* is a member function, which returns object's X-coordinate;

GetPositionY() is a member function, which returns object's Y-coordinate;

GetPositionZ() is a member function, which returns object's Z-coordinate;

GetLength() is a member function, which returns cube's side length;

GetPushDirectionValue() is a member function, which returns force's direction;

SetCollisionFalg() is a member function, which sets collision flag for the object.

3.3.6.2 Collision Detection of Two Spheres

The principle is the same for checking if two spheres collide. Instead of using the side length of a cube, we use the radius of the sphere object. Consider that there are two spheres on a table, and one is apart from another for the initial positions. When spheres move, the collision may happen. It is necessary to check overlapping in both X and Z directions.

Suppose that there are two spheres on a table – *Sphere1* and *Sphere2*. *Sphere1* has a position of *(X1, Y1, Z1)* with radius of *Radius1*, and *Sphere2* has a position of *(X2, Y2, Z2)* with radius of *Radius2*. A collision happens only when following condition is true:

$$sqrt((X1 - X2)^2 + (Y1 - Y2)^2 + (Z1 - Z2)^2) < Radius1 + Radius2$$

where *sqrt* is used to calculate the square root, while $sqrt((X1 - X2)^2 + (Y1 - Y2)^2 + (Z1 - Z2)^2)$ calculates the distance between *Sphere1* and *Sphere2*.

Figure 3-10 shows an implementation of checking collision for spheres. Function *fabs()*

is used to calculate the absolute value of an expression; *sqrt*() is used to calculate the square root of an expression; the member functions *GetPositionX(), GetPositionY()* and *GetPositionZ()* are used to return coordinate values; member function *GetRadius()* returns sphere's radius; member function *SetCollisionFlag()* is used to set the collision flag for a sphere object.

```
GLfloat SpheresDistanceInX = fabs((double)(itsSphere1.GetPositionX() -
                    itsSphere2.GetPositionX()));
GLfloat SpheresDistanceInY = fabs((double)(itsSphere1.GetPositionY() -
                    itsSphere2.GetPositionY()));
GLfloat SpheresDistanceInZ = fabs((double)(itsSphere1.GetPositionZ() -
                    itsSphere2.GetPositionZ()));
GLfloat SpheresDistance = sqrt((double)(SpheresDistanceInX *
                    SpheresDistanceInX + SpheresDistanceInY * SpheresDistanceInY
                    + SpheresDistanceInZ * SpheresDistanceInZ));
GLfloat SumRadiuses  = itsSphere1.GetRadius() + itsSphere2.GetRadius();
if(SpheresDistance <= (SumRadiuses)) {
        itsSphere1.SetCollisionFalg(TRUE);
        itsSphere2.SetCollisionFalg(TRUE);
} else {
```

```
        itsSphere1.SetCollisionFalg(FALSE);

        itsSphere2.SetCollisionFalg(FALSE);

}
```

Figure 3-10 Implementation of collision detection of spheres

3.3.7 Simulation of Dropping Down Process

When an entity on the table moves because of a force, or it drops down from an edge of the table, the trajectories are different. The moving trajectories of an entity on a table are straight lines; however, when an entity reaches to an edge of the table, the dropping trajectory of the entity is a curve (parabola), which can be described by these equations:

$Y = a * X^2 + c$, if force's direction is parallel with X-axis

$Y = a * Z^2 + c$, if force's direction is parallel with Z-axis

Figure 3-11 shows a curve, which demonstrates the dropping trajectory of a cube object. Where *P0* is the start point of the curve, from *P0* the cube object begins to drop down; while *P4* is the end point of the curve, the cube object drops down and stops at this point.

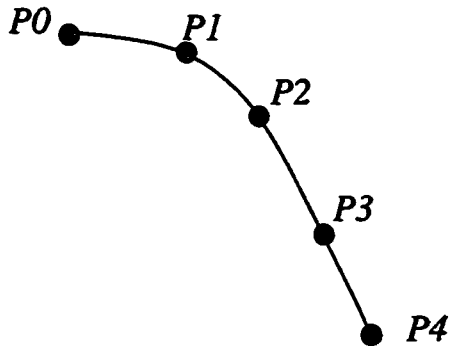

Figure 3-11 Dropping trajectory of a cube object

Figure 3-12 shows a curve, which demonstrates the dropping track of a sphere object. Where P0 is the start point of the curve, from P0 the sphere object begins to drop down; while P4 is the end point of the curve, the sphere object drops down to this point of ground, and than keeps rolling a small distance (from P4 to P5) on the ground.

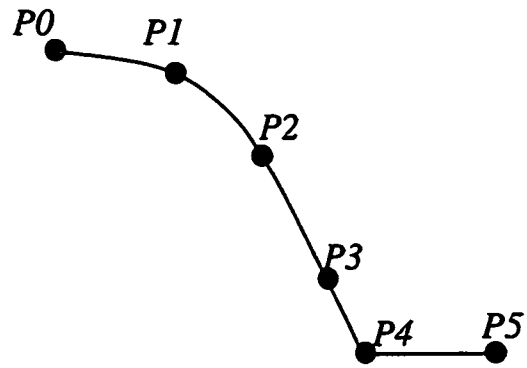

Figure 3-12 Dropping trajectory of sphere object

Suppose that in X-direction the distance between *p0* and *p4* is *L*, (*L* equals to the side length multiplied by 2 for a cube object, or equals to radius multiplied by 2 for a sphere object). Since the coordinate (X_0, Y_0, Z_0) of *P0* is known, and the coordinate (X_4, Y_4, Z_4) of *P4* can be derived from *P0* according to our assumption, therefore the factors '*a*' and '*c*' of the equation $Y = a * X^2 + c$ (or $Y = a * Z^2 + c$) can be gotten by

$$\begin{cases} Y_0 = a * X_0^2 + c \\ Y_4 = a * X_4^2 + c \end{cases} \Rightarrow \begin{array}{l} a = (Y_0 - Y_4) / (X_0^2 - X_4^2) \\ c = Y_0 - X_0^2 * ((Y_0 - Y_4) / X_0^2 - X_4^2) \end{array}$$

We can use this formula to calculate the coordinate of any point.

4. Simulation Results

In this chapter, we use four scenarios to show 3D objects worlds. Each simulation is completed based on some initial conditions. A real 3D object simulation should not only be concerned with geometry properties, but should also appear to model physical properties, such as mass, center of gravity, velocity, acceleration, etc. This means that behaviors will follow directly from these properties. In our project, we start from object's geometry properties; other physical properties will be added to framework in the future.

4.1 Example One

The first example presents a simulation of a single cube object. It will show a series of activities including object rotation, movement, zoom in, zoom out, and dropping from table.

The initial positions are shown in Figure 4-1.

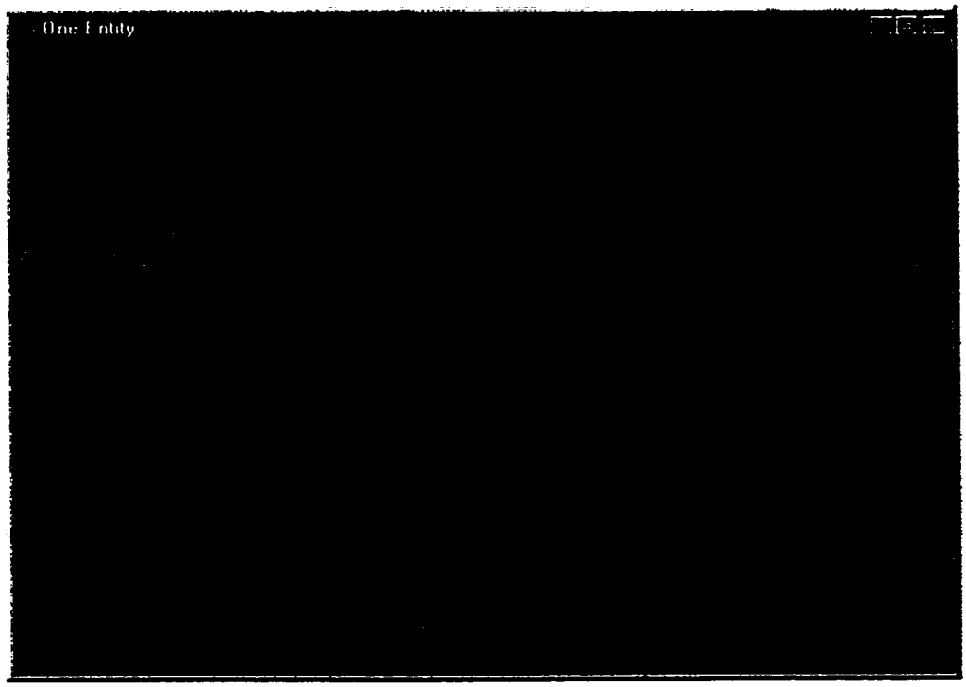

Figure 4-1. Initial position of objects

Step 1. When you start the program

When you start the program, you will see two objects on the screen: a cube and a table. The cube is positioned at the center of the table.

Step 2. Set the position and direction of a force

Click on menu 'Push Direction', an arrow will appear on the cube; press Space key to adjust arrow direction and position.

Step 3. Start simulation

- Select 'Table and Entities' from menu 'Set Rotation Entities', the cube and table will rotate around the center point of table, the rotation angle is depended on the variable Table :: itsRotationAngle. See figure 4-2.
- Select menu 'Entities on the table" from 'Set Rotation Entities', the cube will rotate around the center point of cube; the rotation angle depends on the variable Cube :: itsRotationAngle. See figure 4-3.
- Select 'Zoom In' from zoom menu, objects will be magnified; 'Zoom Out' will minify the objects.
- Select 'Rotate clockwise' or 'Rotate counterclockwise' from 'Rotation' menu, we can rotate objects in clockwise or counterclockwise direction.
- Select 'Reset', objects would return to initial positions.
- Select 'Move entity', the cube will move a distance toward the arrow direction. See figure 4-4.
- Whenever center point of cube passed over the border of table, the cube will drop down from the table. See figure 4-5.

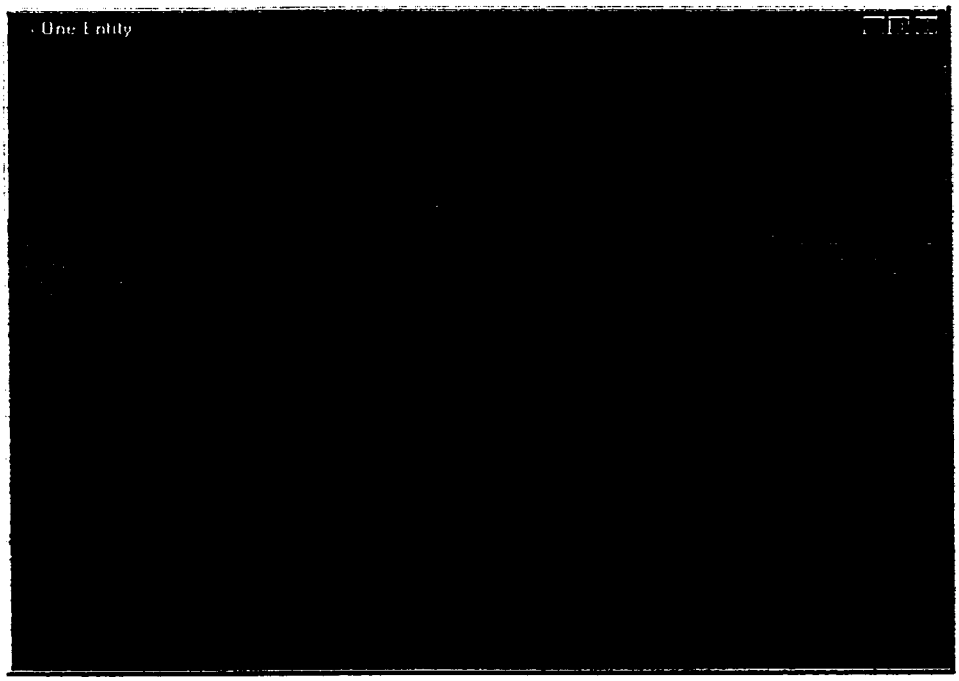

Figure 4-2 Cube and table rotate around the center point of table

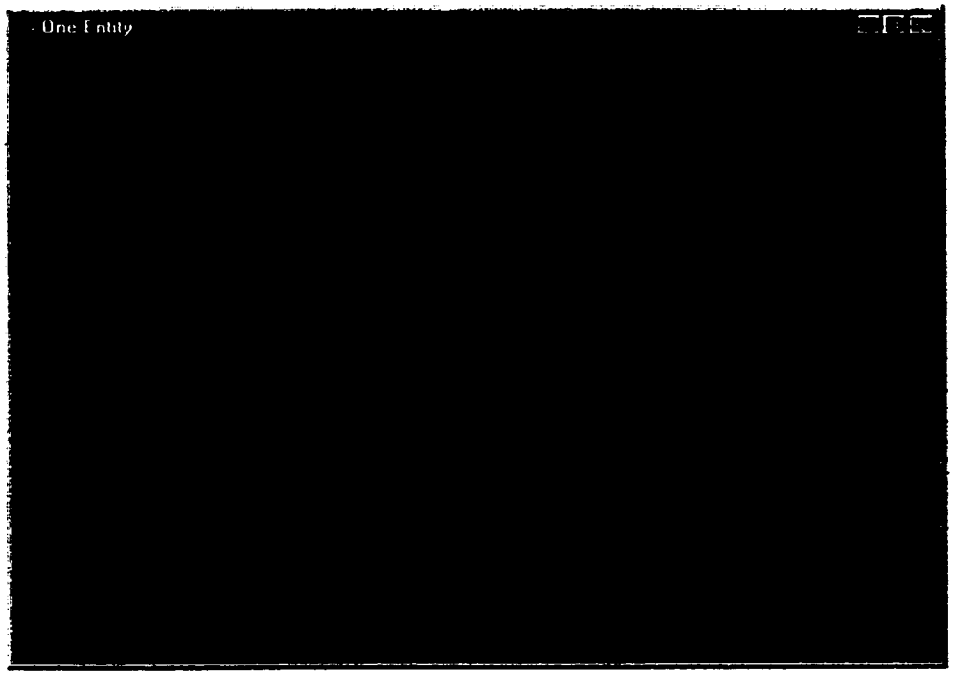

Figure 4-3 Cube rotates around its center

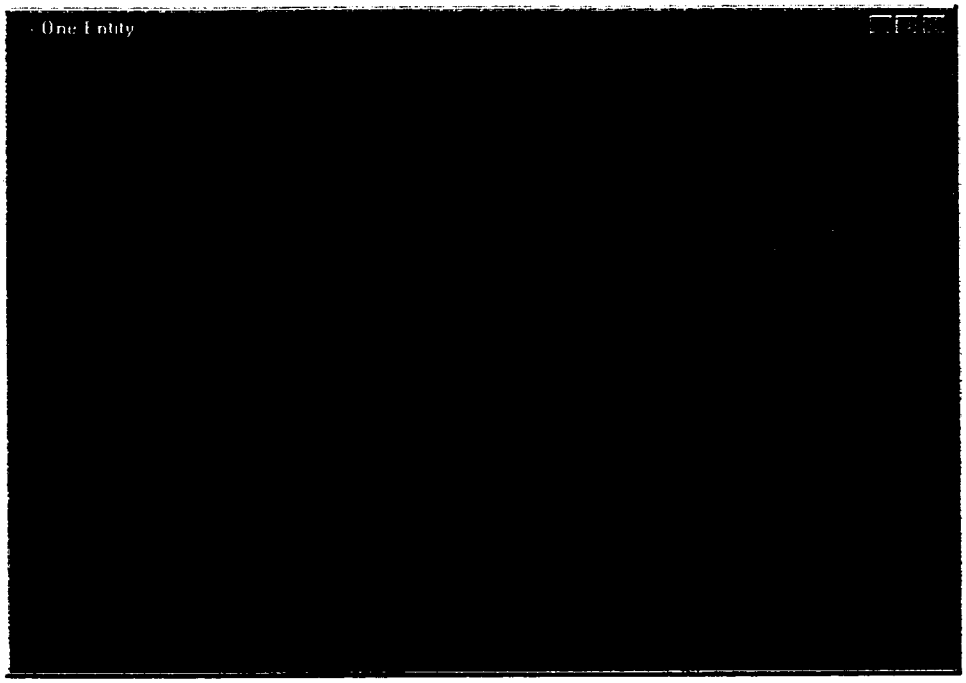

Figure 4-4 Cube moves a distance toward the arrow direction

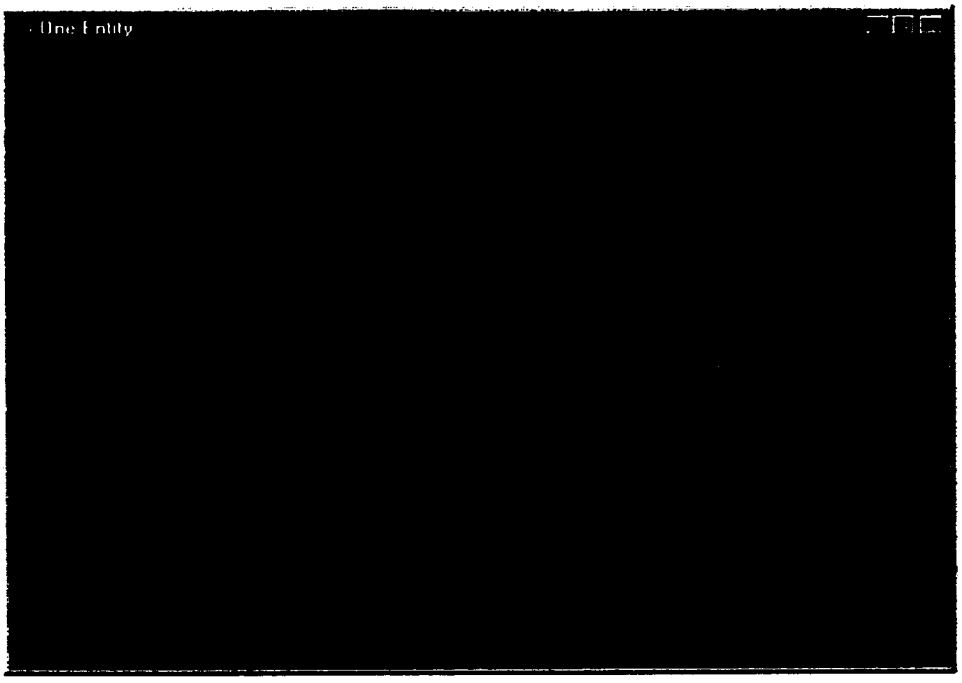

Figure 4-5 Cube drops down from the table

4.2 Example Two

This example will present a simulation of two cube objects. It will show a series of activities including object rotation, movement, zoom in, zoom out, and drop from table. The initial positions are shown in table 4-6.

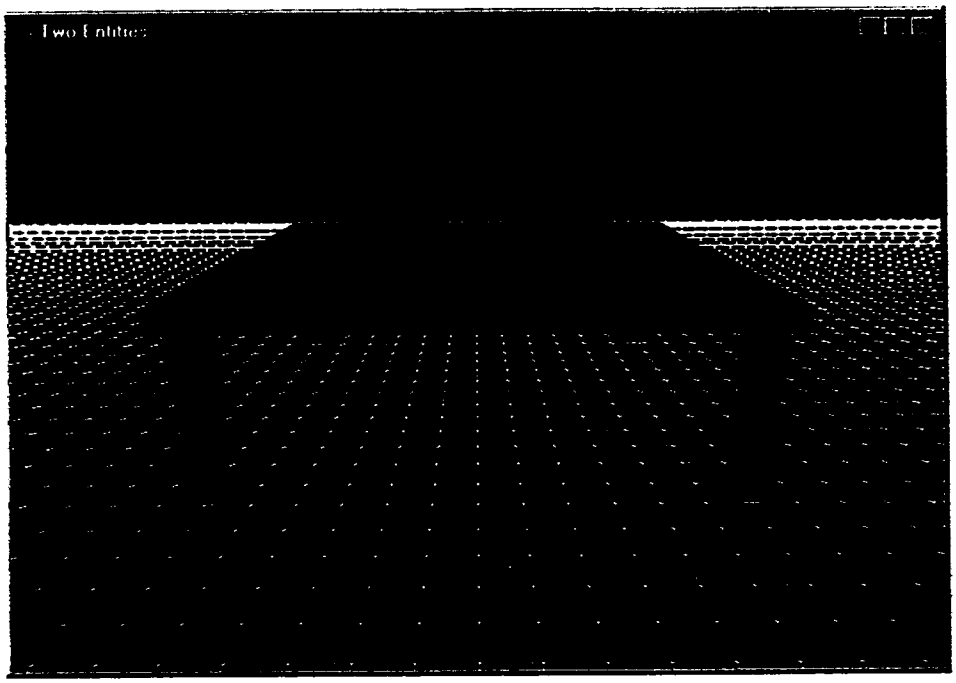

Figure 4-6 Initial positions

Step 1. Start the program

The user interface starts as shown in figure 4-6. Before simulation starts, it displays three objects: two cubes and a table. The cubes are put on the table.

Step 2. Set position and direction of a force

Select 'On' from the menu 'Push Direction", an arrow will appear onto a cube; press Space key to adjust arrow direction and position; select 'Off' from the menu 'Push Direction", the arrow will disappear.

Step 3. Start simulation

- Select menu 'Rotation', the cubes and table will rotate a specified angle around the center of table. See figure 4-7.

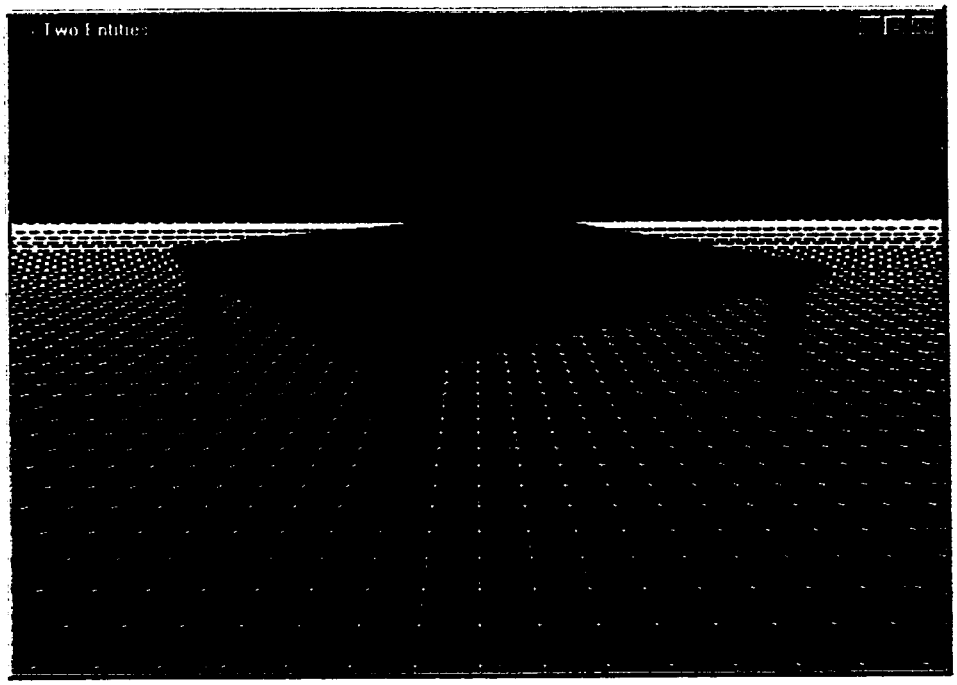

Figure 4-7 Table rotates a specified angle around the center of table

- Select menu 'Zoom In / Zoom Out', objects will be magnified or reduced. See figure 4-8.
- Press '+' or '-' key, objects will rotate a specified angle.
- Click 'Reset', objects return to initial positions.
- Press 'G' key, the cube with force acting on it will move a distance toward the arrow direction. When it meets another cube, both of them will move together in the direction of the arrow. See figure 4-9.
- Whenever the center point of a cube passes over the border of the table, the cube will drop down from the table. See figure 4-10.

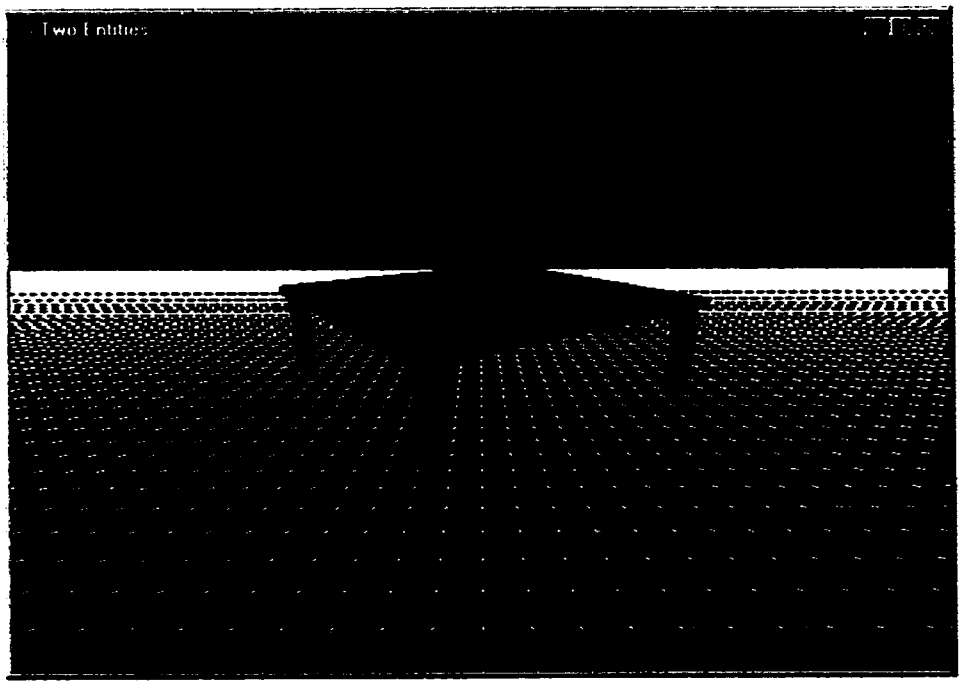

Figure 4-8 Objects are minified

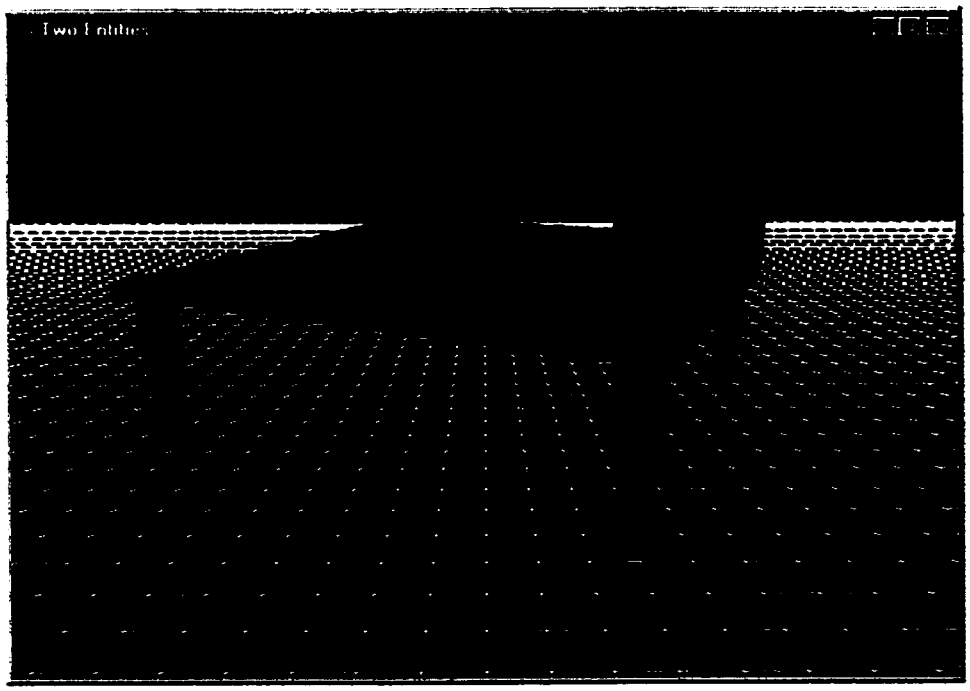

Figure 4-9 Two cubes move together toward the arrow direction

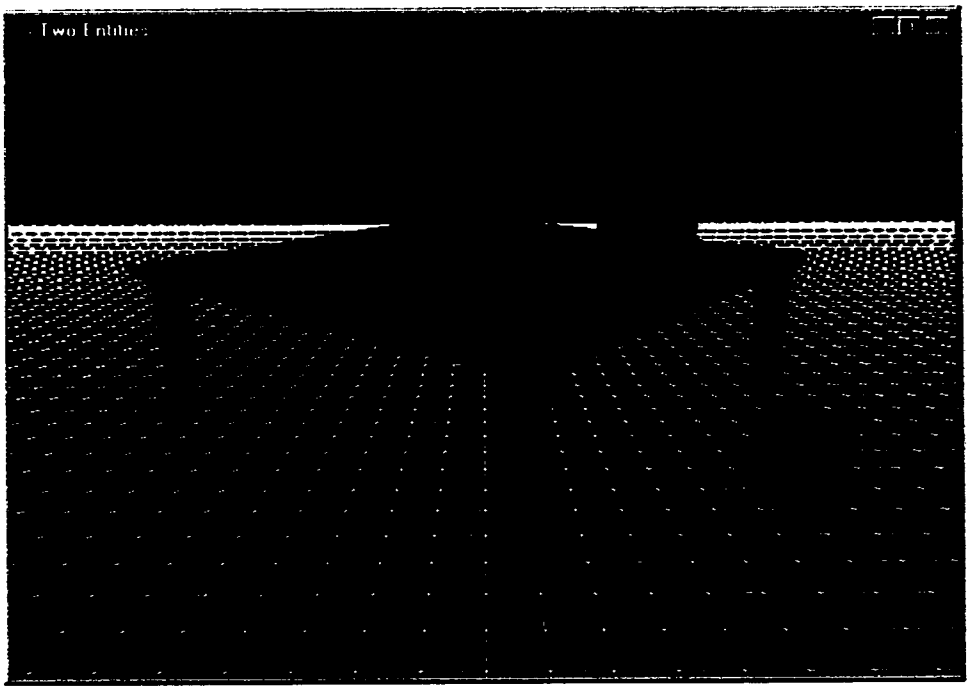

Figure 4-10 A cube will drop down from the table

4.3 Example Three

This example will present a simulation of two cube objects (one is on another one). It will show a series of activities, including object rotation, movement, zoom in, zoom out, and drop from table.

The initial positions are shown in table 4-11.

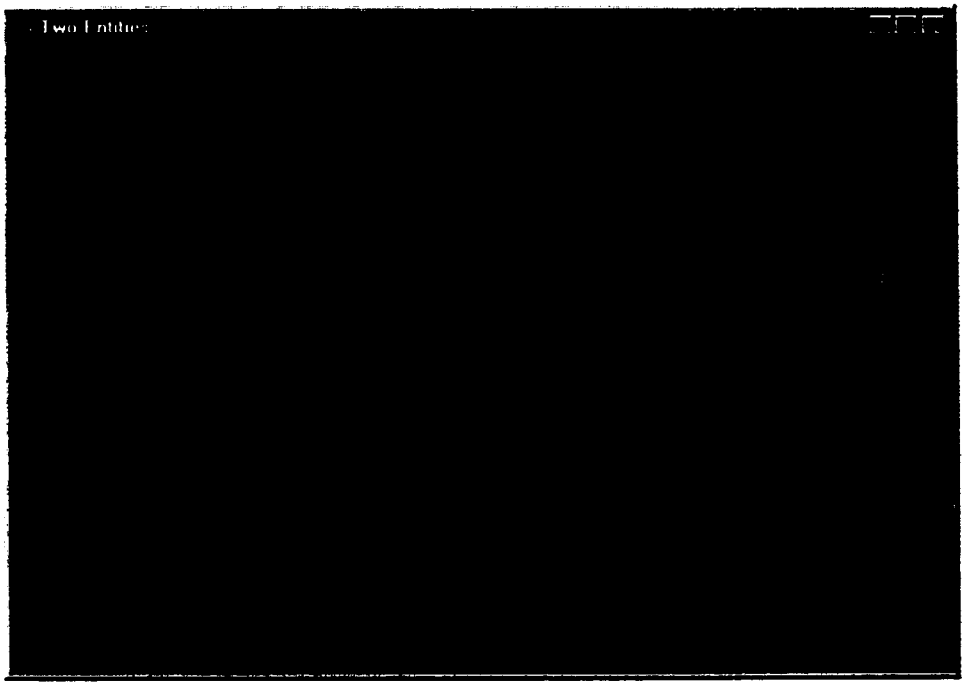

Figure 4-11 Initial positions

Step 1. Start the program:

The user interface starts as shown in figure 4-11. Before simulation, one cube is on the top of another one.

Step 2. Set position and direction of a power

Select menu 'Push Direction', an arrow will appear onto the cube; press Space key to adjust the arrow direction and position.

Step 3. Start simulation

- Select menu 'Rotation', the cubes and table will rotate an angle. See figure 4-12.
- Select menu 'Zoom In / Zoom Out", objects will enlarge or diminish. See figure 4-13.
- Select 'Reset' menu, objects will be reset to their initial positions.
- Press 'G' key, the cube that is pushed will move a step toward the arrow direction. If the upper one reaches the border of another one, it will drop down to the table or ground. See figure 4-14.

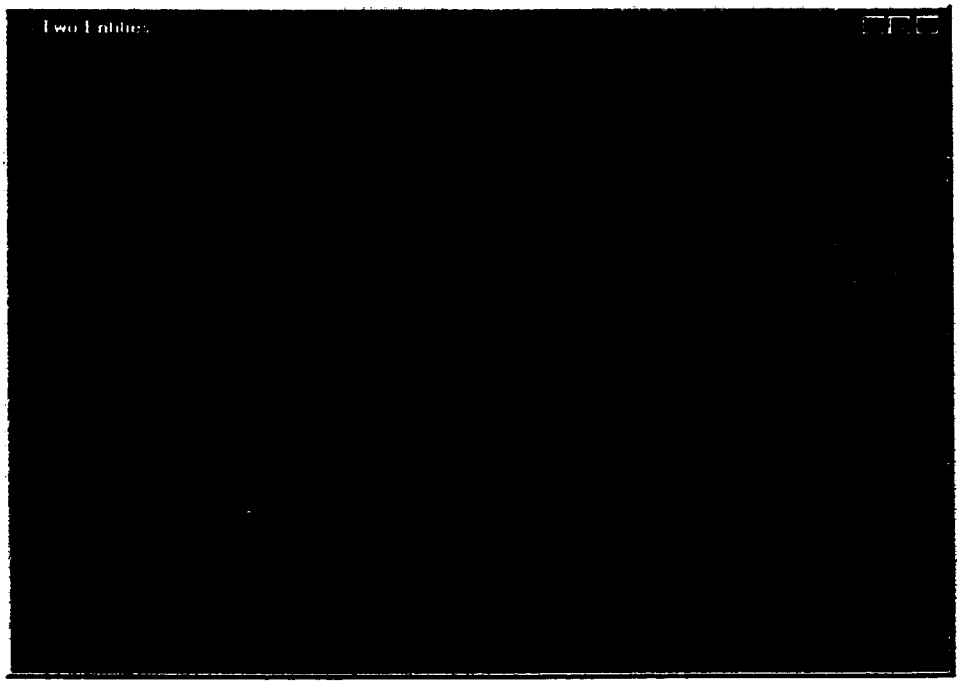

Figure 4-12 Cubes and table rotate an angle

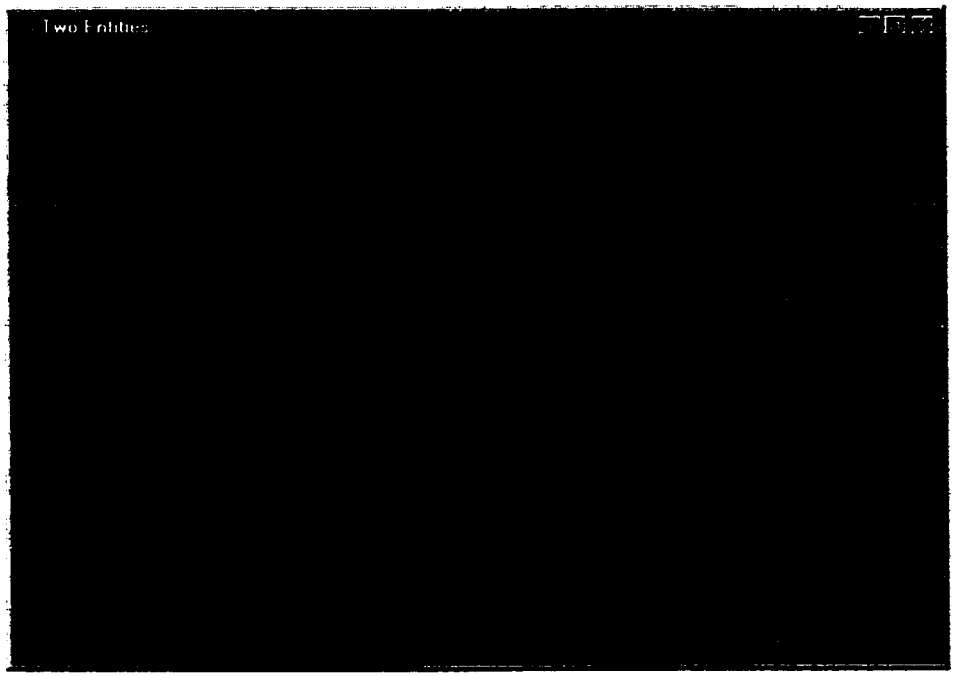

Figure 4-13 Objects enlarge

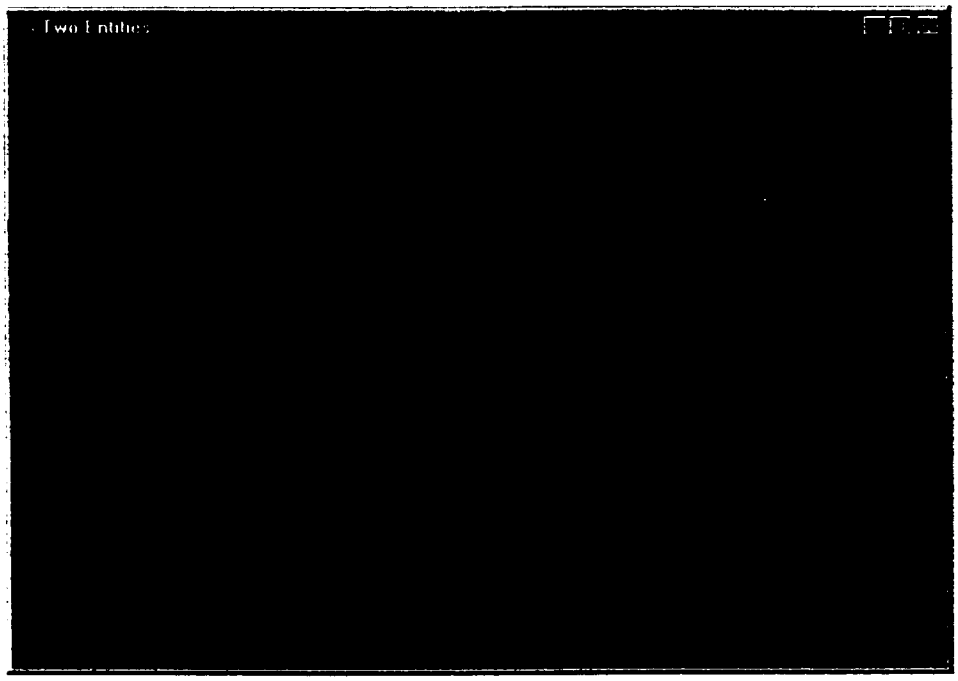

Figure 5-14 Small cube drop down to the table

4.4 Example Four

This example will present a simulation of two sphere objects. It will show a series of activities including objects rotation, movement, zoom in, zoom out, and dropping from the table. The initial simulation parameters and values are shown in table 4-15.

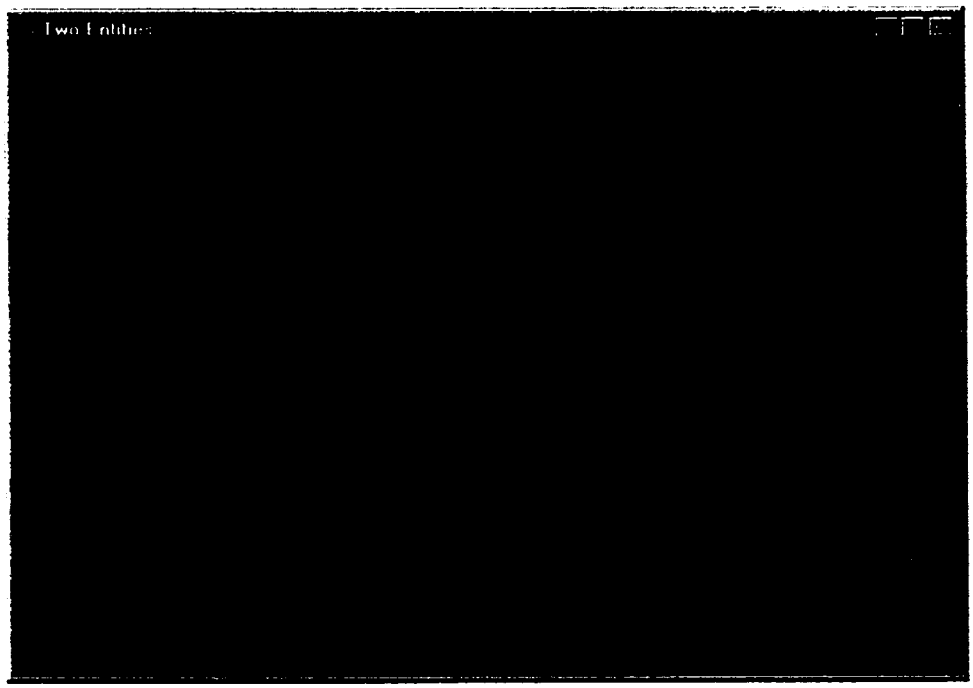

Figure 4-15 Initial positions

Step 1. Start the program

Before simulation, it shows three objects: two spheres and a table. The two spheres are on the table.

Step 2. Set the position and direction of a force

Select menu 'Push Direction', an arrow will appear onto a sphere; press Space key to adjust the arrow direction and position.

Step 3. Start simulation

- Select menu 'Rotation', the spheres and table will rotate a specified angle. See figure 4-16.
- Click on menu 'Zoom In / Zoom Out'', objects will enlarge or decrease. See figure 4-17.
- Click 'Reset', all objects will return to initial positions.
- Press 'G' key, the sphere with force will scroll a distance toward the arrow direction. See figure 4-18.
- Whenever the collision happens, both spheres will move toward the different trajectories. Once a sphere passes over the border of table, it will drop down to ground. See figure 4-19.

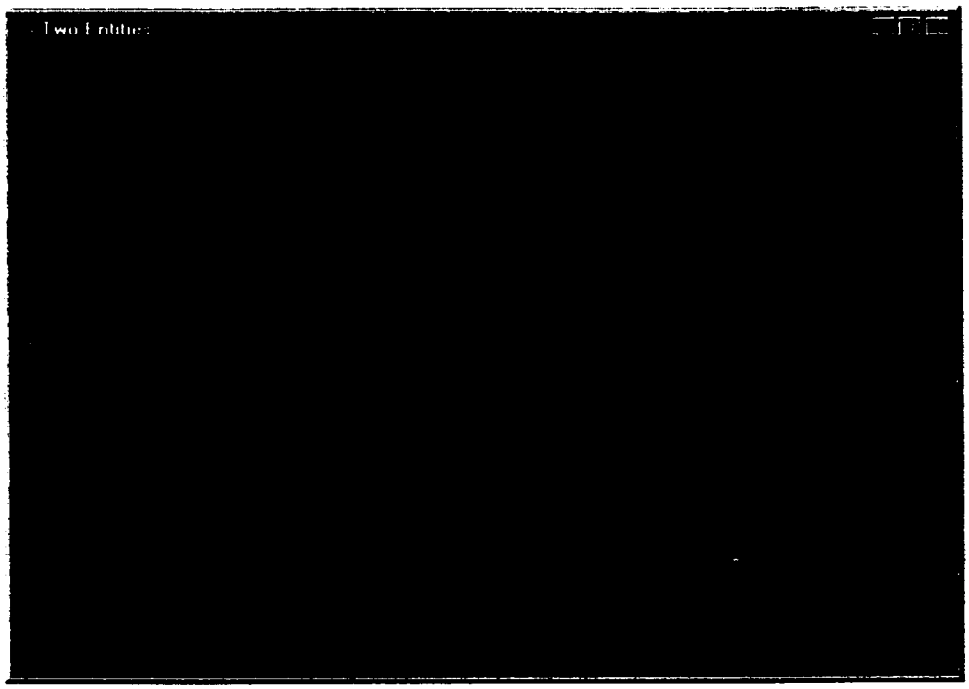

Figure 4-16 Spheres and table rotate a specified angle

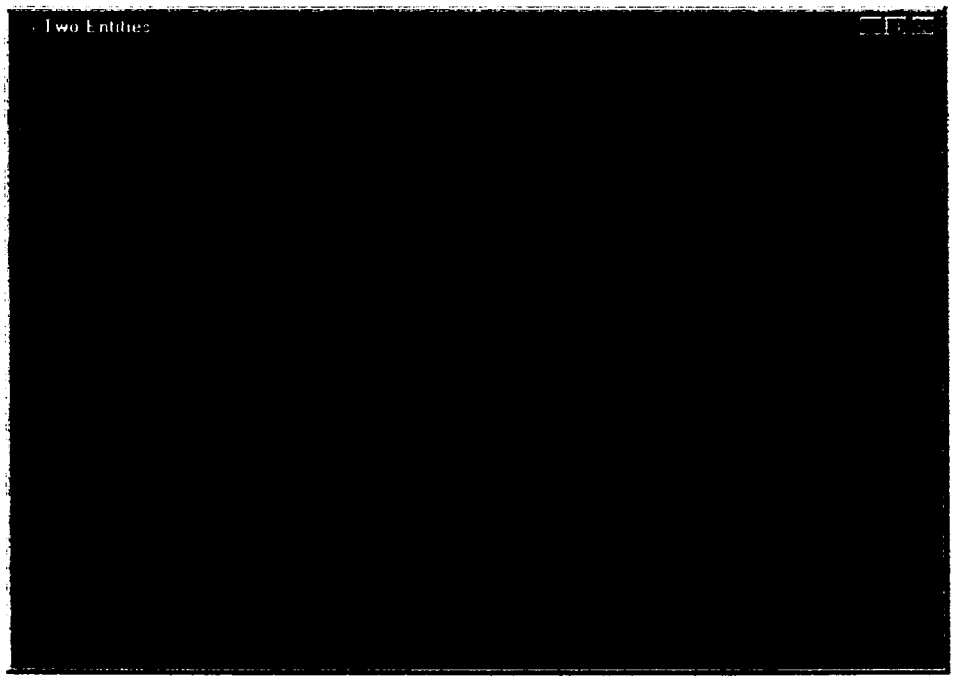

Figure 4-17 Objects enlarge

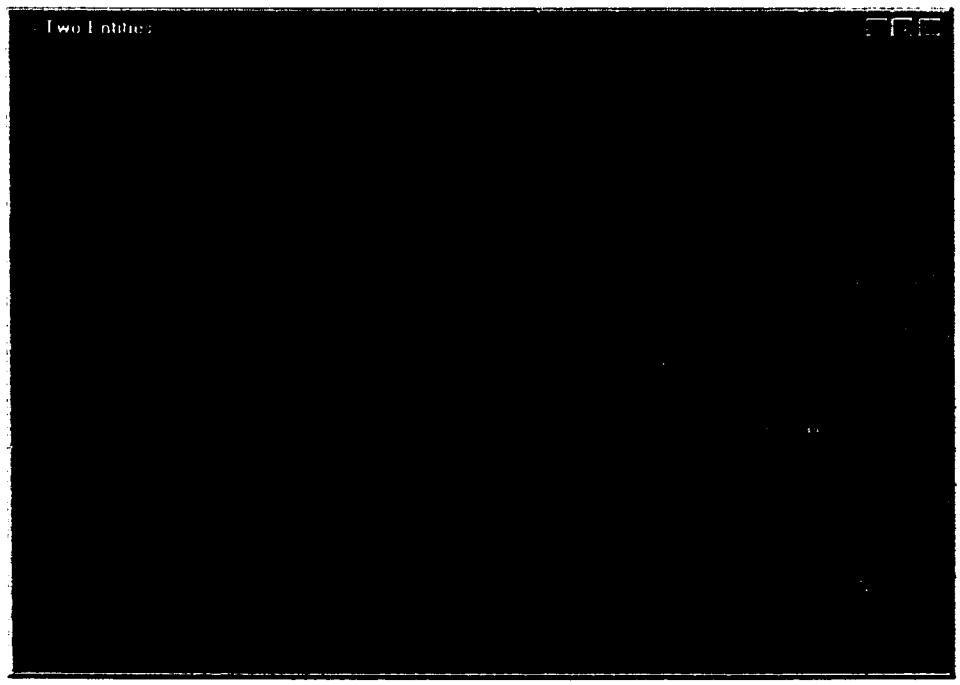

Figure 4-18 sphere scrolls a distance toward the arrow direction

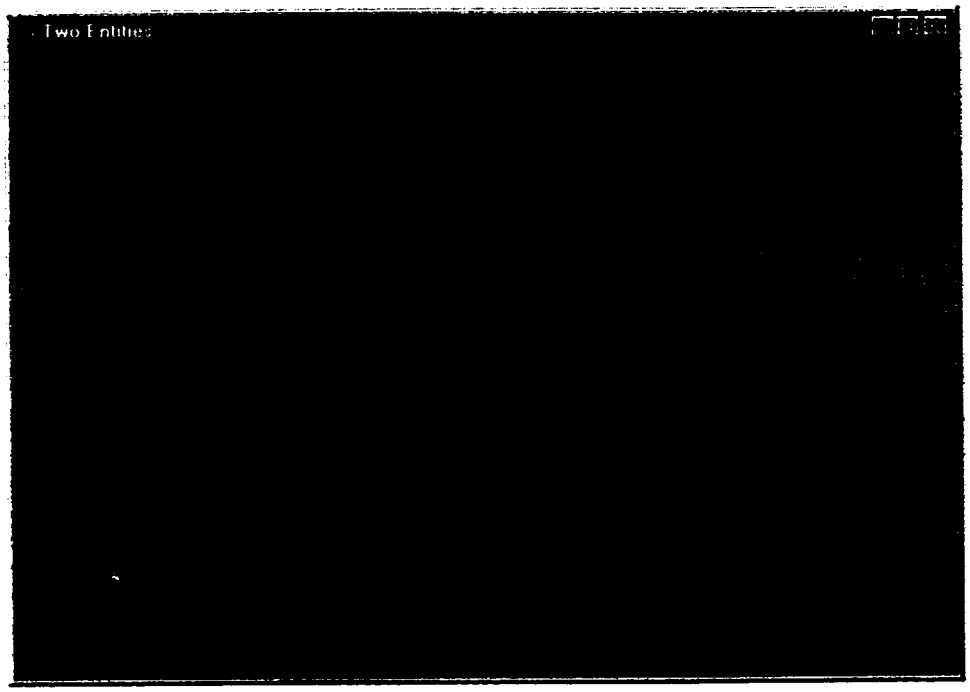

Figure 4-19 Sphere drops down from the table

5. Conclusion

The four examples above demonstrate 3D object simulations. They offer a convenient and efficient way to observe the interactions among objects. The results are correct and interesting.

5.1 Experiences on Object-Oriented Programming

Traditional programming languages separated data from functionality. Typically, data was aggregated into structures that then were passed among various functions that created, read, altered, and otherwise managed that data.

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

Object-oriented programming offers a new and powerful model for writing computer software. It speeds the development of new programs, improves the maintenance, reusability, and modifiability of software. Object-oriented programming focuses on the creation and manipulation of objects, such as cubes, spheres. This type of programming gives us a greater level of abstraction; we can concentrate on how the objects interact without having to focus on the details of the implementation of the object.

5.2 Further Work

One of the limitations of the system is that it does not add enough physical performance in class abstraction. Therefore we suggest the following future work:

- Adding physics modeling, as shown in figure 5-1.
- More powerful games framework.

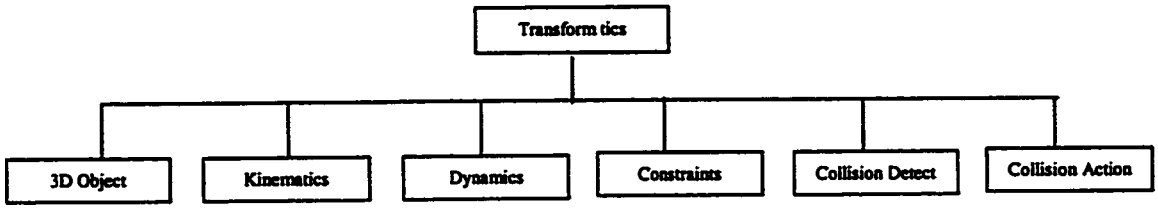

Figure 5-1. Future physics modeling

Bibliography

[Pg98] Peter Grogono. Getting Started with OpenGL. Concordia University, 1998

[JMWFW97] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. Object-Oriented Modeling and Design. General Electric Research and Development Center Schenectady, New York, 1997

[Rsw96] Richard S. Wright JR. OpenGL Super Bible, 1996

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

[Hs98] Herbert Schildt. C++: The Complete Reference. Third Edition., 1998

[Kg97] Kate Gregory. Special Edition Using Visual C++ 5, 1997.

[Web3D] 3D world simulation. http://www.martinb.com/