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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

NOTE TO USERS

The original manuscript received by UMI contains broken, slanted and or light print. All efforts were made to acquire the highest quality manuscript from the author or school. Pages were microfilmed as received.

This reproduction is the best copy available

UMI

Computer Animation of Particles and Rigid Bodies

by

Wendy He, B. Sc./M.Eng.

↗

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Engineering

Ottawa-Carleton Institute of Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
November, 1997

© copyright

1997, Wendy He

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

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

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

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

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

0-612-32399-4

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

“Computer Animation of
Particles and Rigid Bodies”

submitted by Wendy He, B. Sc./M.Eng.
in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair

Thesis Supervisor

Carleton University
November, 1997

Abstract

This thesis describes the methodology for developing advanced animation programs, and examines the current capabilities of graphics software tools used to create animation. Dynamic analysis is used in the thesis for motion control. The use of dynamic analysis involves physical laws, analysis algorithms, collision detection, and environmental effects such as friction, gravity, energy loss and other forces. As well, three dimensional linear and rotation animation are explored by C programs. Finally, OpenGL library function calls are used to show motions.

Acknowledgments

I would like to thank my thesis supervisor, Professor Samy Mahmoud . Without his guidance, comments, ideas, discussions and support, this thesis would not have been possible.

I would also like to express my appreciation to Rosemary Daniels for her patience and help in proof-reading.

Many people helped with various aspects of the thesis. I would like to express my gratitude to Narendra Mehta, Janet Scharf and Darlene Hebert.

Finally, I thank my parents for their support and concern.

Table of Contents

Abstract	iii
Acknowledgments.....	iv
Table of Contents.....	v
List of Figures.....	vii
List of Abbreviations and Symbols.....	viii
 Chapter 1. Introduction.....	 1
1.1 Objective of the Thesis	1
1.2 Components of Animation.....	2
1.3 Methodology	5
1.4 Structure of the Thesis	8
1.5 Summary of Results.....	9
 Chapter 2. Background	 11
2.1 Computer Graphics and OpenGL	11
2.1.1 Geometrical Transformation.....	11
2.1.2 Viewing.....	16
2.1.3 Color.....	25
2.1.4 Lighting	27
2.1.5 Bitmaps, Image and Textures	32
2.1.6 The Framebuffer and Matrix Stacks.....	33
2.2 Animation	38
 Chapter 3. Animation of Particles	 43
3.1 Introduction.....	43
3.2 Linear Motion Dynamics for Two Objects	43
3.3 Analysis of Motion	44
3.4 Algorithm	48
3.5 Analysis of Response Motion	49
3.6 Collision Detection	56
3.6.1 Flexible Surfaces Algorithm.....	56
3.6.2 Convex Polyhedra Algorithm.....	60
3.6.3 Bounding Box Algorithm	62
3.7 Coping with Friction.....	65
3.8 Rendering Objects.....	66
3.9 Implementation	67
 Chapter 4. Animation of Rigid Bodies.....	 73
4.1 Introduction.....	73
4.2 Rotating Motion Dynamics.....	73
4.3 Theory of Rigid Dynamics	74
4.4 Motion Analysis.....	77
4.4.1 Translation Analysis.....	77

4.4.2	Rotation Analysis	81
4.5	Collision Detection	83
4.6	Rendering Objects.....	83
4.6.1	Gear Rendering.....	84
4.6.2	Ball Rendering.....	92
4.6.3	The board and the floor.....	96
4.7	Implementation	97
Chapter 5. Conclusion		99
5.1	Summary	99
5.2	Animation Complexity	100
5.3	Improvement.....	101
5.4	Future of Animation.....	102
References		103
Appendix		107

List of Figures

FIGURE 1.	Keyframing.....	3
FIGURE 2.	In-betweening.....	4
FIGURE 3.	Gradually Decreased Shininess.....	31
FIGURE 4.	Sharply Decreased Shininess.....	31
FIGURE 5.	The Organizational Elements in Production of an Animated Cartoon.....	39
FIGURE 6.	Projectile Motion.....	46
FIGURE 7.	Position in Each Frame.....	48
FIGURE 8.	Jumps.....	50
FIGURE 9.	Different Velocities Before And After The Impact.....	51
FIGURE 10.	Same Height of The Jumping Point and The Falling Point (Case 1).....	52
FIGURE 11.	Lower Jumping Point and Higher Collision Point (Case 2).....	53
FIGURE 12.	Higher Jumping Point and Lower Collision Point (Case 3).....	54
FIGURE 13.	Collisions at The Front of The Truck.....	55
FIGURE 14.	Vectors Before and After The Time Step t	58
FIGURE 15.	A Convex Polygon and Its Outside Normal Vectors.....	60
FIGURE 16.	Error-Detected Collision between A and B.....	64
FIGURE 17.	Collision Moments.....	69
FIGURE 18.	The Inserted Frames.....	70
FIGURE 19.	Overview of The Animation Program.....	71
FIGURE 20.	Picture of The Experiment Two.....	74
FIGURE 21.	The Components of The Gear and The Ball.....	84
FIGURE 22.	A Polygon Mesh on The Sphere.....	93
FIGURE 23.	A Detail Chart of The Experiment Two.....	98

List of Abbreviations and Symbols

a	Acceleration
d	Distance
e	Coefficient restitution
f	Force
k_c	Constant attenuation
k_l	Linear attenuation
k_q	Quadratic attenuation
m	Mass
$O(n^2)$	n^2 comparisons or calculations
P	Position of a vector
r	Radius
t	Time
v	Velocity
w	Angular velocity

Chapter 1. Introduction

1.1 Objective of the Thesis

Animation is a simulation of motion in order to give an appearance of life to an inanimate image, or drawing. Animation can also be described as the art of movement expressed with images that are not taken directly from reality. Computer animation is a process using computer software and hardware to accomplish the animation task.

Computer animation, however, is not just concerned with the motion of objects. It involves color, lights, and the way of rendering objects which provides the issues of realism. There is nothing magical behind their creation. A logical approach of mathematical analysis is useful, but, above all, animators have to spend long sessions at workstations designing, testing and debugging software [1].

Computer animation is widely used in education in the form of graphics to describe two dimensional and three dimensional problems in physics, mathematics and chemistry: it is also used in industrial applications such as simulators for vehicle and aircraft, and visualizations of industrial products and construction projects: as well, it is important in the entertainment industry such as in the production of movies and video games, and it is useful in scientific research such as in the creation of visual models for multi-dimensional data.

The charm of computer animation is economy and safety. Computer animation provides a mechanism to build and view 2-3D objects without any physical materials, and without

any of the costs involved in implementing real systems. It can also simulate dangerous situations without endangering life or damaging instruments. For example, pilots training animation saves the cost of building a real simulator, and it is safe for pilots to practise flying.

This thesis not only shows the methodology for developing advanced animation programs, but also examines the current capabilities of software tools used to create animation.

It should be noted that this thesis deals with the animation of rigid bodies subject to dynamic laws of motion. It does not deal with the animation of complex biological objects such as people and animals.

1.2 Components of Animation

The advantages of computer animation drive people to study in this area. The three components of computer animation are: object modeling, motion control, and object rendering.

Object modeling is a simulation method to present objects by digital information. As most real-world objects are viewed through their surfaces, object modeling is accomplished by surface modeling [2]. The most common representations for 3D surfaces are:

- polygon mesh surfaces which are sets of connected, polygonally bounded planar surfaces

- parametric surfaces where curves define points on a 3D curve by using three polynomials in a parameter t , one for each of x , y , and z
- quadric surfaces which define implicitly by an equation $f(x, y, z) = 0$, where f is a quadric polynomial in x , y , and z .

Rendering is the process that makes models look real. The methods for presenting realistic models involve color, lighting, shading, and texture mapping.

Motion Control is control process to show movement. It is a rich area for computer animation research. The most commonly used motion control methods are keyframing and in-betweening, path specification, morphing and dynamic simulation [1].

1. Keyframing and in-betweening

Keyframing

FIGURE 1. Keyframing

Keyframing defines an animated sequence based on its key moments [3]. The key drawings in an animated sequence are called keyframes. Figure 1 shows the keyframes for sun rising. In-betweening is the technique to fill in the gaps between the keyframes. Please see Figure 2.

FIGURE 2. In-betweening

2. Path Specification

Path specification allows a user to define paths for moving objects in 3D space. This technique is useful in order to define with precision the motion of flying and swimming models because they both move by following a 3D path [1].

3. Control Procedure

This technique animates the objects in the scene based on a set of procedures and rules to control motion. It needs a special-purpose programming language so that the objects can be easily edited and previewed[4].

4. Morphing

Morphing is a representation of transformation in shape, form, or substance that takes place through biological change or magic. For example: plants growing and actors or actresses mutating into animals are animated by morphing technique [2].

5. Dynamic Simulation

Dynamic analysis is used to predict realistic motion. It involves physical laws, algorithms, collision detection, and environmental effects such as friction, gravity, energy loss and other forces. This method involves mathematical and physical analysis [1] [4]. Algorithms are the ways of implementing the results of analysis. Collision detection is a key issue in motion control. It will be discussed in Chapter 3.

The above techniques are widely used to simulate different types of objects: particles, rigid bodies, flexible bodies, articulated bodies and biological subjects such as animals.

1.3 Methodology

To achieve the stated objectives, dynamic analysis is used to predict motions and to create formulas for calculating positions of objects according to physical laws. Different formu-

las are created based on different motion situations such as motions without collision, motions with collision, etc.

C programming language is used to accomplish the animation task. It not only calculates the positions and orientations of objects according to those formulas, but also calls the OpenGL functions to render objects, which means that the values of the calculation are passed to the OpenGL functions to draw the objects at predicted positions. These procedures, which are calculating new positions and orientations of objects and drawing objects at the new positions and orientations, are repeated for rendering in each frame, so that a sequence of frames of objects at different positions shows the motions of the objects.

Two animation programs were developed for two experiments. The first experiment involved the linear motion dynamics of particles, the second experiment involved the rotating motion dynamics of rigid bodies. In both cases, mathematical analysis was created to form the basis for the animation dynamics. The analysis was then linked to OpenGL, a graphics software interface, to render displays of motion trajectories, rotation and collision of objects.

The first experiment describes a bouncing ball and a moving truck. The two objects have random chances to collide. The bouncing ball loses energy when it hits the floor or the truck. Both objects' motions are affected by friction.

The following formulas for calculating the ball's position is obtained based on the ball's motion without collision:

$$x_{k+1} = x_k + (v_x)_0 \Delta t$$

$$y_{k+1} = y_k + \Delta h_0 - k g (\Delta t)^2$$

With collision, the formulas will change to:

$$x_{k+1} = x_k + (v_x)_0 \Delta t$$

$$y = y_{collision} + v_y \Delta t - \frac{1}{2} g (\Delta t)^2$$

where $v_y = e \sqrt{v_{y-pre}^2 - 2gh}$ or $v_y = e \sqrt{v_{y-pre}^2 + 2gh}$ which depends on the collision situation.

C programming language is used to calculate the positions, and the results are passed to OpenGL functions to draw the ball at the position. For details, please see Chapter 3.

The second experiment describes how the rotating objects (a rotating gear, and a bouncing and rotating ball) collide and respond. A similar method is used to calculate the positions of the objects. Angular velocities of the objects are analyzed and calculated by the formula $\omega_{ball} = \frac{\omega_{gear} (y_{ball} - r_{gear})}{r_{ball}}$, which shows the orientation so that the rotating motion can be seen. The values of the calculation are used by OpenGL to render the objects. For details, please see Chapter 4.

Collision detection is one of the key issues in motion control which is the second stage of animation. The bounding box method is applied to the thesis because it is efficient and easy to use. For details, please see Section 3.6 and Section 4.5 (Collision Detection).

Drawing polygon meshes of surfaces is used to render objects which is the last stage of animation. For details, please see Section 3.8 and Section 4.6 (Rendering Objects).

1.4 Structure of the Thesis

There are five chapters in the thesis:

- Chapter 1 Introduction
- Chapter 2 Background
- Chapter 3 Animation of Particles
- Chapter 4 Animation of Rigid Bodies
- Chapter 5 Conclusion
- References
- Appendix

Chapter 1 (Introduction) describes the objectives of the thesis, components of animation, methodology of computer animation, structure of the thesis and summary of the results.

Chapter 2 (Background) discusses the background of computer graphics, OpenGL, and animation theory. The computer graphics and OpenGL section describes the background

of geometrical transformation: viewing: color: lighting. bitmap. image. textures. the framebuffer and the matrix stacks.

Chapter 3 (Animation of Particles) describes the linear motion dynamics for two objects. analysis of motion. algorithm. analysis of response motion. collision detection. coping with friction. rendering objects. and the implementation for the animation.

Chapter 4 (Animation of Rigid Bodies) discusses the rotating motion dynamics. the theory of rigid dynamics. motion analysis (translation analysis and rotating analysis). collision detection. rendering objects. and the implementation for the animation.

Chapter 5 (Conclusion) summarizes the thesis and describes the complexity of computer animation. the improvements that have been made. and the future of computer animation.

1.5 Summary of Results

The animation programs which implement the methodology for the two experiments demonstrate that the whole animation process works very well. The linear and rotating motions. the impact and the response motion can be seen to move as they would in actual movement.

However, a smoother motion effect will be obtained if the original version of OpenGL is used instead of the free equivalent version - Mesa. Furthermore, the animation program can be run faster if the Silicon Graphics Workstation is used.

OpenGL itself concentrates on precision of calculation, so it is not the fastest graphics software obtainable; however, it is very easy to use and it is the most popular graphics software in the world. OpenGL has become standard and the standardized version is faster than the original.

There are some other graphics software which are faster than OpenGL, such as Direct. But Direct is very hard to use and it is not supplied in the free version. The free version of OpenGL is the best choice for the thesis because it is suitable and economic.

Except for using faster graphics software, a simplified formula for calculation and simple algorithm for collision detection can be used to increase the speed of animation.

Chapter 2. Background

2.1 Computer Graphics and OpenGL

Computer graphics is a special field in computer science. It provides a mechanism to render objects in the real world. It is involved in 3D transformation, viewing, color, lighting, texture mapping and animation.

OpenGL is a software interface to graphics hardware. It consists of 120 commands or library functions: it is hardware-independent interface implemented on different hardware platforms. OpenGL does not provide high-level commands for describing models of three-dimensional objects such as automobiles, airplanes or parts of body. OpenGL helps to build up the desired model from a small set of geometric primitives such as points, lines, and polygons [5].

The following sections will introduce computer graphics theory by using OpenGL commands.

2.1.1 Geometrical Transformation

A. Mathematical Preliminaries

Vector is the collection of all direction line segments having a given magnitude and a given direction.

Length of a vector is the common length of all its representative directed line segments. If v is any vector which may be written in the form $v=(a, b, c)$ then the length of v , which is denoted by $|v|$, is given by $|v| = \sqrt{a^2 + b^2 + c^2}$.

Dot product: we define dot product $u \cdot v$ by the formula

$$u \cdot v = |u| |v| \cos \theta$$

where θ is the angle between the vectors. If either u or v is 0, we define $u \cdot v = 0$. Two vectors u and v are orthogonal if and only if $u \cdot v = 0$.

Let v and u be two vectors which make an angle θ . $|v| \cos \theta$ is the projection of v along u

$$|v| \cos \theta = \frac{v \cdot u}{|u|}$$

Cross product: given the vectors u and v , we define the vector or cross product $u \times v$ as follows:

1. if either u or v is 0, $u \times v = 0$
2. if u is proportional to v , $u \times v = 0$
3. if otherwise $u \times v = w$, where w is a vector perpendicular to a plane determined by u and v , the magnitude of w is $|w| = |u| |v| \sin \theta$.

Homogeneous coordinate: A point (x, y, z) is represented in its homogeneous form as (wx, wy, wz, w) . If the point is homogenized (divide by w), a point of the form $(x, y, z, 1)$ is obtained as homogeneous coordinate.

Homogeneous coordinates allow translation, scaling, and rotation to be expressed mathematically in terms of matrix operations. This is an efficient approach to graphical transformations [6] [7].

B. 3D Transformation

3D transformations are generally represented by 4x4 matrices using the homogeneous-coordinate representation of points in 3D-space. So a point (x, y, z) is represented as $(x, y, z, 1)$.

The translation of a point $P=(x, y, z)$ to position $P' = (x', y', z')$ is $P' = T \times P$ where

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The above matrix equation is equivalent to

$$x' = x + t_x$$

$$y' = y + t_y$$

$$z' = z + t_z$$

The rotation of a point $P=(x, y, z)$ along z-axis to position $P' = (x', y', z')$ is given by

$R_z(\theta)$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

so that $P' = R_z(\theta) \times P$.

The rotation matrix along x-axis is given by $R_x(\theta)$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix along y-axis is given by $R_y(\theta)$ (see [5])

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combination of 3D transformations:

Rotating an object which is at position (x_f, y_f, z_f) , not at the origin, is combination of translation and rotation [6].

1. Translate the fixed point to the origin: $T(-x_f, -y_f, -z_f)$
2. Rotate the object relative to the origin: $R(\theta)$
3. Translate back to the original position: $T(x_f, y_f, z_f)$

So, the combination of transformation is

$$T(x_f, y_f, z_f) \times R(\theta) \times T(-x_f, -y_f, -z_f)$$

OpenGL functions for 3D transformation:

- glTranslatef(x, y, z); glTranslated(x, y, z)
- glRotatef(angle, x, y, z); glRotated(angle, x, y, z)

2.1.2 Viewing

Computer graphics is designed to create a 2D image of 3D objects because 3D objects are drawn on a screen. Drawing geometric models converts an object's 3D coordinates to pixel position on the screen. It is involved in coordinates transformation, clipping and model projection.

OpenGL accomplishes this task by these steps:

1. Transformations:

Viewing transformation: combination of

- translation:
- rotation:
- scaling:
- reflecting.

Modeling transformation: combination of

- translation:
- rotation:
- scaling:
- reflecting.

Projection transformation:

- orthographic projection:
- perspective projection.

2. Clipping: Because the scene is rendered on a rectangular window, objects that lie outside the window must be thrown out by clipping.
3. Viewport transformation: Because the scene is rendered on a screen, 3D model coordinates have to be converted to screen coordinates.

Viewing transformation is analogous to positioning and aiming a camera. Modeling transformation positions and orients the model. There are alternative ways to consider transformations. You can imagine that the camera moves in one direction, or objects move in the opposite direction. This is why modeling and viewing transformations are combined into the modelview matrix before the transformations are applied [5].

Perspective Projection: Objects that are closer to the viewpoint (or camera) appear larger because they take proportionally larger amount of the viewing volume than those that are farther away.

Stages of Vertex Transformation:

For a perspective projection object, positions are transformed to the view plane along lines that converge to a point, which is termed the projection reference point or center of projection. The projected view of an object is obtained by calculating the intersection of the projection lines with the view plane.

Suppose, the projection reference point is set at position z_{prp} along z axis, and the view plane is placed at z' .

The following figure shows that the view plane of z plane is z' plane.

XY plane

The parametric equations defining the points on the plane are:

$$x' = x - xu \quad (1)$$

$$y' = y - yu \quad (2)$$

$$z' = z - (z - z_{prp})u \quad (3)$$

$$u = \frac{z - z'}{z - z_{prp}} = \frac{x - x'}{x} = \frac{y - y'}{y} \quad 0 \leq u \leq 1$$

When $u = 0$ the plane is at position $P = (x, y, z)$ and if $u = 1$, the plane is at the projection reference point. On the view plane z' , the value of u can be obtained by solving the z' equation.

$$d = z_{prp} - z' \quad (3)$$

From equation (1), (2), (3), we get the following equations:

$$x' = x \left(\frac{z_{prp} - z'}{z_{prp} - z} \right) = x \left(\frac{d}{z_{prp} - z} \right)$$

$$y' = y \left(\frac{z_{prp} - z'}{z_{prp} - z} \right) = y \left(\frac{d}{z_{prp} - z} \right) .$$

Using 3D homogeneous-coordinate representation, we can write the perspective projection transformation in matrix form as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \frac{d}{z - z_{prp}} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{z'}{d} & \frac{-z' \times z_{prp}}{d} \\ 0 & 0 & \frac{1}{d} & \frac{-z_{prp}}{d} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

OpenGL command for perspective projection is: `glFrustum(left, right, bottom, top, near, far)`.

Orthographic Projection: Orthographic projection is also parallel projection. In parallel projection, coordinate positions are transformed to the view plane along parallel lines. Paralleled projections are categorized into two types. One type is where the direction of projection and the normal to the projection plane are parallel, the other type is not. Orthographic projection is the former type.

Unlike perspective projection, the size of the viewing volume doesn't change from the near end to the farther end. Orthographic projections are normally used to generate top, front and side views as in architectural or engineering drawings. OpenGL command for orthographic projection: `glOrtho(left, right, bottom, top, near, far)`.

Viewing transformation: Transformation sequence from the physical world to device coordinate may involve translation, rotation, scaling clipping.

Window is the region of the object selected for display in world coordinate. Viewport is window's corresponding rectangular region in screen coordinates. By changing the positions of the viewport, the objects can be displayed at the different positions of the display

device. By changing the size of the viewports, the size and proportions of the displayed objects can be changed.

Window-to-viewport coordinate translation: The matrix for converting from world coordinate to viewing coordinate is obtained as combination of translation and scaling transformation.

$$M_{wv} = T(u_{min}, v_{min}) \times S\left(\frac{u_{max} - u_{min}}{x_{max} - x_{min}}, \frac{v_{max} - v_{min}}{y_{max} - y_{min}}\right) \times T(-x_{min}, -y_{min})$$

$$M_{wv} = \begin{bmatrix} 1 & 0 & u_{min} \\ 0 & 1 & v_{min} \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & 0 \\ 0 & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -x_{min} \\ 0 & 1 & -y_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

$$M_{wv} = \begin{bmatrix} \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & -x_{min} \times \frac{u_{max} - u_{min}}{x_{max} - x_{min}} + u_{min} \\ 0 & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & -y_{min} \times \frac{v_{max} - v_{min}}{y_{max} - y_{min}} + v_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

Assume (x, y) is a point in world coordinates. (x', y') is the corresponding point in view-port. So

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = M_{wp} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2.1.3 Color

Hardware causes each pixel on the screen to emit a different amount of red, green, and blue light. These are called the R, G, and B values. RGB mode: each pixel has R G B

Table 1:

R	G	B	Color	Functions
0	0	0	black	glColor(0.0,0.0 , 0.0)
0	0	1	blue	glColor(0.0, 0.0, 1.0)
0	1	0	green	glColor(0.0, 1.0, 0.0)
0	1	1	cyan	glColor(0.0, 1.0, 1.0)
1	0	0	red	glColor(1.0, 0.0, 0.0)
1	0	1	magenta	glColor(1.0, 0.0, 1.0)
1	1	0	yellow	glColor(1.0, 1.0, 0.0)
1	1	1	white	glColor(1.0, 1.0, 1.0)
1	0	0.5	dark pink	glColor(1.0, 0.0, 0.5)
0.5	1	0.5	light green	glColor(0.5, 1.0, 0.5)
0.5	0	0.5	purple	glColor(0.5, 0.0, 0.5)
0	0	0.8	dark blue	glColor(0.0, 0.0, 0.8)
0.7	0.7	0	brown	glColor(0.7, 0.7, 0.0)
...	...	...	...	...

value. It is used in texture mapping, lighting and blending. Pixels can be displayed in dif-

ferent shade of a color by assigning R, G, B values. Shading is used in a lighting model [5].

Color-Index mode: each pixel has an index number which represents a different color. For

example, assume 3 objects (ellipse, rectangle and triangle). All pixels in ellipse have index value 0, all pixels in rectangle have index value 1 and all pixels in triangle have index value 2. Based on the color map (table 2), the ellipse is red, rectangle is purple and

Table 2:

Index	R	G	B
0	1	0	0
1	0.5	1	0.5
2	0	0	1
...	...	...	...

the triangle is blue. Color-Index mode is used in drawing in layers and color map animation.

2.1.4 Lighting

Objects have different shades of colors under light. This depends upon the objects' position, orientation and characteristics of their surfaces, and the light sources illuminating them. Objects do not look three-dimensional until they are lit. For example, an unlit sphere looks no different from a two-dimensional disc.

Lighting models or shading models are used to express the factors determining a surface's color at a given vertex. The lighting process involves calculation of intensity of each vertex on an object's surfaces. Graphics researchers have often approximated the underlying rules of optics and thermal radiation to simplify computation. They create lighting equations based on the approximation. Although their equations are yielded from approximation, they work very well and can be computed relatively fast.

The OpenGL lighting model considers the lighting to be divided into 4 independent components: emitted, ambient, diffused and specular light.

Emitted light originates from an object and is not affected by any light sources. Ambient light is the product of multiple reflections of light from the many surfaces present in the environment. Diffused light scatters equally in all directions when it hits a surface. It looks equally bright no matter where the eye is located. Specular light comes from a particular direction, and tends to bounce off the surface in a direction. Shiny metal or plastic

has a high specular component and chalk or carpet has almost none. Specularity is shininess.

Example of OpenGL function `glLight*(GLenum light, GLenum pname, TYPE param)`:

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
```

```
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
```

```
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
```

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
```

```
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

```
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
```

```
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

```
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

The OpenGL lighting model also shades a material's color depends on the percentages of the incoming light it reflects. Materials have ambient, diffuse, and specular colors, which determine the ambient reflection, diffuse reflection, and specular reflection. The ambient, diffuse, and specular reflections are material properties [5].

Ambient reflection is the amount of ambient light reflected from an object's surface. It is most noticeable where an object receives no direct illumination. An object's total ambient reflection is affected by the global ambient light and one from individual light source. But

it is not affected by the position of viewpoint. Ambient reflectance is represented by $ambient_{material}$.

Diffuse reflection is affected by the color of the diffuse light and the angle of the light relative to the normal direction. Dull surfaces, such as chalk, exhibit diffuse reflection. Diffuse reflection is not affected by the position of the viewpoint. Diffuse reflectance is represented by $diffuse_{material}$.

Specular reflection is the shiny spot or highlight on a surface. Unlike ambient and diffuse reflection, the amount of specular reflection seen by a viewer does depend on the location of the viewpoint. It is brightest along the direct angle of reflection. As you change the view position is changed, the highlight moves with the viewer to some extent. Specular reflectance is represented by $specular_{material}$.

The lighting equation for OpenGL RGB mode is:

$$color(vertex) = emission_{material} + ambient_{globalLight} \cdot ambient_{material} + \sum_{i=0}^{n-1} \frac{1}{k_c + k_l d + k_q d^2} \cdot (spotlightEffect)_i \cdot [ambient_{light} \cdot ambient_{material} + (\max\{l \cdot n, 0\}) \cdot diffuse_{light} \cdot diffuse_{material} + (\max\{s \cdot n, 0\})^{shininess} \cdot specular_{light} \cdot specular_{material}]_i$$

where l is the unit vector that points from the vertex to the light position, s is the sum of the two unit vectors that point between the vertex and the light position, and point between the vertex and the viewpoint.

$\frac{1}{k_c + k_l d + k_q d^2}$ is attenuation factor where d is distance between the light's position and the vertex, k_c is constant attenuation, k_l is linear attenuation, and k_q is quadratic attenuation.

$(\max\{s \cdot n, 0\})^{shininess}$ is specular reflection exponent where *shininess* is integer from 0 to 128. The value of *shininess* varies depending on the surface material being simulated. Assume θ is the angle between unit vector s and n , then $\cos\theta = s \cdot n = |s| \cdot |n| \cdot \cos\theta$, in other words, the specular reflection exponent is obtained by $(\max\{s \cdot n, 0\})^{shininess} = (\max\{\cos(\theta), 0\})^{shininess}$. Maximum specular reflectance occurs when θ is zero and falls off sharply as θ increases. The rapid falloff is approximated by $\cos^{shininess}\theta$. Figure 3 and Figure 4 show the specular reflection for matte materials and glossy materials respectively.

FIGURE 3. Gradually Decreased Shininess

FIGURE 4. Sharply Decreased Shininess

A value of 1 provides a broad, gentle falloff, whereas higher values simulate a sharp, focused highlight [6].

2.1.5 Bitmaps, Image and Textures

A bitmap is a rectangular array of 0s and 1s. Assume the are drawing of a bitmap and assume the current color is blue. Everywhere there is a 1 in the bitmap, the corresponding pixel is replaced by a blue pixel. The corresponding 0s in bitmap are not affected (or in black). Bitmaps are typically used for characters in fonts. For example:

An image is a rectangular array of pixels. The difference between image and bitmap is that a bitmap consists of a single bit of information about each pixel, and image data includes

several different data such as red, green, blue and alpha. Images can come from several sources: scanner and software program generated. They can be used in texture mapping.

Textures are rectangular array of images (color data, luminance data, or color and alpha data). They can be mapped to rectangular or non-rectangular regions.

Other mapping such as bump mapping (this simulates a wrinkled or dimpled surface rather than modulation color of a flat surface by normal vector perturbation), environment mapping (reflecting the surrounding environment in a shiny object) are not described in OpenGL features.

2.1.6 The Framebuffer and Matrix Stacks

OpenGL provides color buffers, stencil buffer, depth buffer and accumulation buffer.

Open a window on the screen using library function: `auxInitWindow(x, y, width, height)`.

X, y indicate the location of the upper left corner of the window, and width and height indicate the window's size (in pixels).

For example,

```
auxInitWindow(0, 0, 300, 300); ( see window A in above figure )
```

```
auxInitWindow(300, 300, 300, 300); ( see window B in above figure )
```

The screen is composed of a rectangular array of pixels. Each pixel contains its color information, a depth value, mask flag and some other information. Different kind of information are stored in different buffers.

The color buffers store color information for each pixel on the screen. Assume that the screen is 1280 pixels wide and 1024 pixels high (there are totally $1024 \times 1280 = 1,310,720$ pixels on the screen). Assume that 2^{24} different colors can be displayed.

$$\begin{aligned} 24 \text{ bits} &= 8 \text{ bits} + 8 \text{ bits} + 8 \text{ bits} \\ &= 1 \text{ byte} + 1 \text{ byte} + 1 \text{ byte} \end{aligned}$$

So, the color buffer in this example has to store at least 3 bytes data for each of 1,310,720 pixels on the screen.

The stencil buffer keeps the mask flags of each pixel on the screen. It is like a cardboard stencil which restrict drawing to certain parts of the screen. For example, assume that you want to simulate driving a car. You can store an image of the windshield shape in the sten-

cil buffer and draw the outside scene on the entire screen. The final result is that the only scene through the windshield can be displayed on the screen.

The depth buffer stores the depth value of each pixel. Some of 3D objects might obscure all or parts of others. Changing viewpoints can change the obscuring relationship. If the scene is viewed from the opposite direction, any object that was previously in front of another is now behind it. To draw a realistic scene, these obscuring relationships must be maintained.

The depth buffer works by associating a depth, or distance from the viewpoint, with each pixel on the window. With depth buffering enabled, before each pixel is drawn, a computation is done with the depth value already stored at the pixel. If the new pixel's depth is greater than what is currently there, the new pixel would be obscured, and the color and depth information for the incoming pixel is discarded. Since information is discarded rather than used for drawing, hidden-surface removal can increase performance.

To use depth buffering, it is necessary to enable depth buffering. Each time the scene is drawn, it is necessary to previously clear the depth buffer and then draw the objects in the scene in any order.

The following figures show the difference between blending and depth test.

If blending is used instead of depth test, you will get the above picture. As there is no depth test involved, the common parts of triangle and square will be blended together. So the color of the common part is cyan.

Please compare the following picture which has depth test and without blending.

Without blending and depth test, the picture will look like above.
The second drawn object - triangle will overwrite the first
drawn object - square.

Matrix stacks - OpenGL provides matrix stack for user to construct hierarchical models.
in which complicated objects are constructed from simpler ones.

For example, assume that you are drawing a car that has four wheels, each of which is attached to the car with five bolts. It is possible to write a single routine to draw a wheel and another to draw a bolt, since all the wheels and all the bolts look the same. These routines draw a wheel or a bolt in different positions. The program can be written like this:

- Draw the car body.

- Record the position where the car body is (push matrix into stack), and translate to the right front wheel.
- Draw the wheel.
- Forget about the last translation so your current position is back at the origin of the car body (pop matrix from stack).
- Record where you are, and translate to the left front wheel and forget about the last translation.

2.2 Animation

Animation is a technique to create moving pictures in a series of individual drawings on successive frames of film. Conventional animation methods consist of photographing a series of progressive drawings [8]. The organizational elements in production of an animated cartoon is shown in Figure 5.

FIGURE 5. The Organizational Elements in Production of an Animated Cartoon

Synopsis: is a summary of the story.

Scenario: describes the story, the characters and the scenery.

Storyboard: is a sequence of drawings which show the key moments in the film.

Sequence: defines specific action.

Scene: associates with a particular location and sets of characters.

Shot: is a picture unit to compose each scene.

Computer animation follows a similar route. But it refers to the process of generating a series of frames by computer graphics software [2][8][9]. Except for the storyboard it involves three major steps: modeling, motion control, and rendering. The methods have already been introduced in chapter 1.

The following pseudo code shows how to make a million-frame movie by software: (For movies, the frame rate is 24 frames each second.)

```
open window:

for each frame (loop)
{
    clear the window:

    draw the frame:

    wait until a 24th of a second is over:
}
```

The process of generating moving pictures by computer software is different from conventional animation's. Computer animation draws a series of pictures instantly, while

movies project pictures which are drawn or taken ahead of time. This creates a disturbing screen effect.

Rendering a picture or a frame takes time. The screen is drawn from the first line of pixels to the end. Assume that drawing a full screen takes nearly $1/24$ second. When the full frame is visible, time has already passed in almost $1/24$ second, and in the next $1/24$ second a new screen will be drawn. So, items drawn toward the end are instantly cleared as the software starts drawing the next frame. Since for most of the $1/24$ second, eyes are viewing incomplete pictures or cleared backgrounds instead of the full picture, disturbing results or a flashing screen will be noticed.

A solution to the problem is to enable off-scene rendering, which means one frame is displayed while the other is drawn. This requires double buffering in hardware or software. One buffer is used to display a frame while the other is used to draw the next frame. When the drawing of a frame is complete, the two buffers are swapped, so the one that was being viewed is now used for drawing. The animation program is like this:

```
open_window_in_double_buffer_mode();

for (i = 0; i < 1000000; i++)
{
    clear_the_window();

    draw_frame(i);

    swap_the_buffers();
}
```

OpenGL itself doesn't have a `swap_the_buffer()` command because the feature might not be available on all hardware and, in any case, it is highly dependent on the window system. However, GLX (OpenGL extension to X window system) and TK extension provide such a command, for use on machines that use the X window System:

```
void glXSwapBuffers(Display *display, Window window);  
  
void tkSwapBuffers(void);
```

For example, to display a ball tracing a sine curve, it is necessary to continuously display the different position of the ball in every particular moment. If you use angle α is used to describe the moment, let's say α equals to 0.1, 0.2, 0.3 then the ball's position will be $x=\cos(0.1)$, $y=\sin(0.1)$; $x=\cos(0.2)$, $y=\sin(0.2)$... and so on. The animation program is like the following:

```
void movingDisplay(void)  
{  
    alpha += 0.1;      /* alpha increase 0.1 every time */  
    x = cos(alpha);  
    y = sin(alpha);  
    display_the_ball_in_(x,y)();  
    tkSwapBuffers();  
}
```

Chapter 3. Animation of Particles

3.1 Introduction

Motion control in animation involves dynamic simulation, which takes into account variables such as object's mass, flexibility, collision detection, as well as the environment's friction, gravity, energy loss and other forces that may influence the motion of objects [10].

This chapter will describe the analysis and the solution to these kinds of problems, and demonstrate how to use OpenGL to implement the animation program.

3.2 Linear Motion Dynamics for Two Objects

A bouncing ball with a horizontal speed has a chance to collide with a sliding truck on the way. The ball will lose energy when it hits the truck or the floor. In addition, it will be affected by friction when it stops bouncing and slides on the floor.

The ball's motion direction varies after colliding with the truck. It depends on the part of the truck it hits and its direction before it hits the truck.

3.3 Analysis of Motion

A particle can be described as a point (x, y, z) in space. The location and motion of such a point is designated by three variables. So particles have three degrees of freedom of motion.

A small bouncing ball, which is considered as a particle, is projectile motion in kinematic theory. We neglect aero dynamic drag and the earth's curvature and rotation, and we assume that the altitude range is small enough so that the acceleration due to gravity can be considered constant. The equations of motion are:

$$\frac{d^2x}{dt^2} = 0$$

$$\frac{d^2y}{dt^2} = -g$$

$$\frac{d^2z}{dt^2} = 0$$

Integrating with respect to t ,

$$\frac{dx}{dt} = (v_x)_0$$

$$\frac{dy}{dt} = -gt + (v_y)_0$$

$$\frac{dz}{dt} = (v_z)_0$$

where $(v_x)_0$, $(v_y)_0$, $(v_z)_0$ are velocities at the time $t = 0$. Integrating again,

$$x = (v_x)_0 t + x_0$$

$$y = (v_y)_0 t - \frac{1}{2}gt^2 + y_0$$

$$z = (v_z)_0 t + z_0$$

where x_0 , y_0 , z_0 are the coordinates of the point at the time $t = 0$. Eliminating t between the two equations $x = (v_x)_0 t + x_0$ and $z = (v_z)_0 t + z_0$, we get $\frac{x - x_0}{(v_x)_0} = \frac{z - z_0}{(v_z)_0}$, which

shows that the motion is in a vertical plane. If we choose this vertical plane for the plane of XY, we will get $z = 0$, $v_z = 0$, and $v_x = (v_x)_0$. Please see Figure 6.

FIGURE 6. Projectile Motion

If t is considered as a very short period time Δt , equation $y = (v_y)_0 t - \frac{1}{2} g t^2 + y_0$ can be changed to

$$y = (v_y)_0 \Delta t - \frac{1}{2} g (\Delta t)^2 + y_0 \quad (\text{EQ 1})$$

Keyframing method is widely used in computer animation. It positions the objects at specified times, and motion is interpolated between these position. The animator displays a certain number of frames in a second. The frame rate depends on the hardware and the complexity of the scene which is to be displayed. For this particular animation, the rate is ten frames per second. The normal rate for movies is 24 frames per second.

Assume the time between frames is Δt , $\Delta h_i = y_{i+1} - y_i$, ($i \geq 0$). Figure 7 shows the ball's position in each frame. Based on EQ 1,

$$\Delta h_0 = y_1 - y_0 = (v_y)_0 \Delta t - \frac{1}{2} g (\Delta t)^2 \quad (\text{EQ 2})$$

$$\Delta h_1 = y_2 - y_1 = (v_y)_1 \Delta t - \frac{1}{2} g (\Delta t)^2 \quad (\text{EQ 3})$$

EQ 3 and equation $\frac{dv}{dt} = -g$ + $(v_y)_0 = v_y$ yield

$$\Delta h_1 = ((v_y)_0 - g \Delta t) \Delta t - \frac{1}{2} g (\Delta t)^2 \quad (\text{EQ 4})$$

$$\Delta h_1 = \Delta h_0 - g (\Delta t)^2 \quad (\text{EQ 5})$$

Similarly,

$$\Delta h_k = \Delta h_{k-1} - g (\Delta t)^2 \quad (k \geq 1)$$

$$\Delta h_k = \Delta h_{k-2} - 2g (\Delta t)^2$$

$$\Delta h_k = \Delta h_0 - kg (\Delta t)^2$$

$$y_{k+1} = y_k + \Delta h_0 - kg (\Delta t)^2 \quad (\text{EQ 6})$$

As there is no acceleration in horizontal motion,

$$x_{k+1} = x_k + (v_x)_0 \Delta t \quad (\text{EQ 7})$$

FIGURE 7. Position in Each Frame

3.4 Algorithm

Assume variable GTT represents $g(\Delta t)^2$, xTranslate and yTranslate represent coordinates of the ball at position (x, y) , delta_h0 represents Δh_0 , delta_s represents $(v_x)_0 \Delta t$, mg_displacement is the accumulator of GTT.

As animation's key idea is to keep displaying serial pictures, the animation program is to run display commands in a loop. The following pseudo code is the algorithm for animation of an object which is tossed in the air and falls by gravity.

```
for ( : : )
{
    mg_displacement += GTT;

    yTranslate += delta_h0 - mg_displacement; /* based on EQ 6 */

    xTranslate += delta_s; /* based on EQ 7 */

    display_ball_at(xTranslation, yTranslation);
}
```

This code only simulates the motion of the ball without collision. Unfortunately, the real world is not that simple. The ball bounces up when it hits floor or an object (in this case, the object is a truck on the floor) and it will stop bouncing when it loses its energy.

3.5 Analysis of Response Motion

The ball will lose energy when it hits the floor or the truck. To calculate the ball's position after the collision, the initial velocity of the current jump will determine the displacement of the ball based on EQ 2: $\Delta h_0 = (v_y)_0 \Delta t - \frac{1}{2} g (\Delta t)^2$.

The current jump means the current period that the ball jumps from one point to another. Figure 8 shows the ball's motion path. If the animator is displaying the motion of the second jump, then the current jump is the second jump.

FIGURE 8. Jumps

According to classical theory of impact, coefficient restitution

$$e = \frac{v'_2 - v'_1}{v_1 - v_2}$$

v_1 , v_2 are the object1's and the object2's velocity before the impact. v'_1 , v'_2 are the object1's and the object2's velocity after the impact. The value $e = 1$ means one of the elastic impact with no energy loss. The value $e = 0$ means non-elastic impact with maximum energy loss. All impact conditions lie $(0, 1)$.

Assume the object1 is the ball and the object2 is the truck. mass of the truck is much greater than mass of the ball. So $v_2 = v'_2 = 0$, $e = -\frac{v'_1}{v_1}$.

$$v'_1 = -ev_1 \quad (\text{EQ 8})$$

EQ 8 can be applied to the ball and floor collision. Assume e of ball-truck collision and e of ball-floor collision are the same.

FIGURE 9. Different Velocities Before And After The Impact

Figure 9 shows that the speed of the ball before the impact is greater than the speed after the impact because of the energy loss.

The following analysis will discuss the value of the velocity instead of the direction of the velocity after the collision. The value of the velocity will determine the displacement of the ball so that the motion of response can be simulated.

Assume v_{y-pre} is the value of initial velocity of the previous jump. v_y is the value of initial velocity of the next jump.

Case 1 (Figure 10): The jumping point and the falling point are the same height.

$$v_y = e v_{y-pre}$$

FIGURE 10. Same Height of The Jumping Point and The Falling Point (Case 1)

Case 2 (Figure 11): The jumping point is lower than the collision point.

FIGURE 11. Lower Jumping Point and Higher Collision Point (Case 2)

Kinetic energy: $T = \frac{1}{2}mv^2$

Gravitational potential energy: $V_g = mgh$

Based on energy conservation, $T + V_g = T' + V'_g$. T and V_g are vertical kinetic energy and gravitational potential energy of the ball at position jumping point respectively. T' and V'_g are vertical kinetic energy and gravitational potential energy of the ball at position collision point respectively. Please see Figure 5.

$$T = \frac{1}{2}mv_{y-pre}^2$$

$$V_g = 0$$

$$V'_g = mgh$$

Assume v is the vertical velocity before the impact, we obtain

$$\frac{1}{2}mv_{y-pre}^2 = \frac{1}{2}mv^2 + mgh$$

As $v = \frac{v_y}{e}$, we get

$$\frac{1}{2}mv_{y-pre}^2 - mgh = \frac{1}{2}m\left(\frac{v_y}{e}\right)^2$$

$$v_y = e\sqrt{v_{y-pre}^2 - 2gh}$$

Case 3 (Figure 12): The jumping point is higher than the collision point. Similarly, we get

$$v_y = e\sqrt{v_{y-pre}^2 + 2gh} \quad (\text{EQ 9})$$

FIGURE 12. Higher Jumping Point and Lower Collision Point (Case 3)

Case 4 (Figure 13): The ball hits the front of the truck. Assume v_{jy} does not change and value of horizontal velocity does not change, its direction changes to negative x-axis.

FIGURE 13. Collisions at The Front of The Truck

As long as v_{jy} is known, the ball's coordinates (x, y) after collision can be easily obtained based on EQ 1 (assume collision happens before frame k+1). For case 1,2 and 3:

$$y = y_{collision} + v_{jy} \Delta t - \frac{1}{2} g (\Delta t)^2$$

$$x_{k+1} = x_k + v_{kx} \Delta t$$

For case 4: Δh_0 is the vertical displacement in the first frame after jumping.

$$y_{k+1} = y_k + \Delta h_0 - kg (\Delta t)^2$$

$$x_{k+1} = x_k - v_x \Delta t$$

3.6 Collision Detection

Collision detection is one of the important issues in computer animation. The method to detect whether collision occurs or not, is to use an algorithm to test if objects penetrate each other. Collision happens when two objects attempt to interpenetrate each other. Without collision detection, objects will penetrate each other on the screen, which is not the situation in the real world. If graphics software does not provide it, the animator has to implement the algorithm for detection and response.

There are a lot of different algorithms for collision detection. They detect collision in the next frame time based on the position and velocity of objects in the current frame time.

Three algorithms will be introduced in the following sections.

3.6.1 Flexible Surfaces Algorithm

Surfaces are modeled as many triangle meshes. Each triangle mesh is represented by three vertices and three edges. Collisions between surfaces are detected by testing for penetration of each vertex of one surface through any triangles of another surface. In addition, collisions are detected by testing for penetration of each edge of one surface through any triangles of other surface because collision can happen without vertex penetration [11].

Assume the surfaces are never touched initially. For each time step of animation, the positions of vertices at the beginning and the end of the time step must be compared to see if any vertex has gone through a triangle during the time step. If so, a collision is detected. Similarly, if an edge of one surface has been detected going through a triangle of another, a collision has occurred.

Let us think about a vertex penetrating through a triangle. There are two cases: moving vertex penetrating a fix triangle, and moving vertex penetrating a moving triangle. For the first case, the following parametric vector equation is set up.

$$P + Vt = P_0 + (P_1 - P_0)u + (P_2 - P_0)w$$

where P is the beginning positions of the vertex, P_i is point of the triangle, u and w are parametric variables for plane defined by the triangle. $u \geq 0$ $w \geq 0$ $u + w \leq 1$, t is a time variable which is 0 at the beginning of the time step and 1 at the end. $P + Vt$ is the parametric equation for the path of the vertex and $P_0 + (P_1 - P_0)u + (P_2 - P_0)w$ is the parametric equation for any points on the triangle.

If the triangle is moving, the following parametric vector equation is set up

$$P' = P'_0 + (P'_1 - P'_0)u + (P'_2 - P'_0)w \quad (\text{EQ 10})$$

where P' is the position of the vertex at time step t . P'_i is the position of three points on the triangle at time step t . Please see Figure 14.

FIGURE 14. Vectors Before and After The Time Step t

The triangle $P_0P_1P_2$ moves to the position $P'_0P'_1P'_2$ with velocity of each point V_0 , V_1 , and V_2 . According to vector arithmetic, we get

$$\begin{aligned} P_0P'_1 - P_0P_1 &= P_1P'_1 \\ P_0P'_1 - P_0P'_0 &= P'_0P'_1 \end{aligned}$$

where $P_0P'_0 = V_0t$, $P_1P'_1 = V_1t$, $P_0P_1 = P_1 - P_0$, $P'_0P'_1 = P'_1 - P'_0$. Plug in these equations into the previous two, we get

$$\begin{aligned} P_0P'_1 - (P_1 - P_0) &= V_1t \\ P_0P'_1 - V_0t &= P'_1 - P'_0 \end{aligned}$$

Eliminate $P_0P'_1$, we get

$$P'_1 - P'_0 = (P_1 - P_0) + (V_1 - V_0) t \quad (\text{EQ 11})$$

Similarly, we get

$$P'_2 - P'_0 = (P_2 - P_0) + (V_2 - V_0) t \quad (\text{EQ 12})$$

Plug EQ11 and EQ12 into EQ10, we get

$$P + Vt = P_0 + V_0 t + ((P_1 - P_0) + (V_1 - V_0) t) u + ((P_2 - P_0) + (V_2 - V_0) t) w$$

This equation is used to test one vertex with one triangle. To detect collision, all vertices from one object and all triangles from another have to be tested and vice versa. This involves a large amount of computation. To minimize the cost of the calculation, a pre-testing method is used. If the distance of the vertex to the triangle at the beginning time step does not change its sign, it is at the ending time step (according to triangle's normal vector), the penetration is assumed not to have occurred. So the calculation based on the above equation can be saved. If the sign does change, the calculation will be executed.

The above algorithm is to test vertex-triangle penetration. The edge-triangle penetration testing is similar. It tests if a point penetrates the triangle when it walks along the edge from one end to the other end.

3.6.2 Convex Polyhedra Algorithm

Any concave polyhedra can be decomposed into convex ones. Convex polyhedra collision detection is to test if a vertex of one polyhedron is inside of the other polyhedron or an edge of one penetrates through the other polyhedron.

This method is based on the Cyrus - Beck clipping algorithm. The two-dimensional Cyrus - Beck algorithm tests if a point is inside of a convex polygon. Please see Figure 15.

FIGURE 15. A Convex Polygon and Its Outside Normal Vectors

To test if the vertex P is inside of each side of the polygon, create a vector from any point on the edge to the vertex P, and calculate the dot product of the vector with the outside normal vector of the edge. If the result is negative, the vertex P is inside of the edge. Use the method to test all other edges of the polygon. If the vertex P is inside of all the edges, it is inside of the polygon [11][12].

To test if a vertex is inside of a polyhedron, a similar method is used - calculate the dot product of the vector, which is a point of a face on the polyhedron to the vertex P , with the outside normal of the face. If the result is negative, the vertex P is inside of the face. If the vertex P is inside of all the faces, the vertex P is inside of the polyhedron.

To test if the two polyhedra are intersected, not only each vertex of one polyhedron has to be tested with the other polyhedron, but also the edge of one polyhedron has to be tested with the faces of the other polyhedron. Please see the following figure.

Calculate the distance from the two end point of the edge to the face. If the signs are different, the edge penetrates through the plane defining the face. But it does not mean that

the edge penetrates through the face. Calculate the intersection of the edge with the plane by the following equations.

$$t = \frac{d_i}{d_i + d_j}$$

$$P = V_i + t(V_j - V_i)$$

Test if the vertex P is inside of the face by using the two-dimensional Cyrus - Beck algorithm. If P is inside of the face, the edge intersects the face.

3.6.3 Bounding Box Algorithm

The above two algorithms involve a lot of calculation, especially when surfaces or polyhedra have very complicated shapes. This section will describe the simplest algorithm for collision detection - Bounding Box Algorithm.

Any object occupies a space. A bounding box whose sides are parallel to the coordinate axes, represent its space an object takes. Please see the following figure.

Object A

Object B

Bounding box algorithm is to test if the bounding boxes of objects overlap each other.

Overlapping does not occur if the following condition is satisfied. Otherwise, collision

detected. Please see the above figure.

$$x_{min} - a > x_{max} - b \quad \text{or}$$

$$x_{min} - b > x_{max} - a \quad \text{or}$$

$$y_{min} - a > y_{max} - b \quad \text{or}$$

$$y_{min} - b > y_{max} - a \quad \text{or}$$

$$x_{min} - a > x_{max} - b \quad \text{or}$$

$$x_{min} - b > x_{max} - a$$

Bounding box overlapping detection is easy to implement. It does not cost much calculation, so it is very efficient. Bounding box algorithm will be used for collision detection in the thesis.

FIGURE 16. Error-Detected Collision between A and B

Every algorithm has its advantages and disadvantages. The first two algorithms are very accurate or reliable. They need to test vertex penetration and edge intersection so that they cost a lot in computation. Bounding box algorithm is very simple and efficient; however, it is not as accurate as other algorithms. Figure 16 shows why this algorithm is not accurate in some circumstances. There is a trade-off between precision and time. It really depends on what the animation application requires. Some applications combine bounding box algorithm with other algorithms for collision detection.

3.7 Coping with Friction

To achieve realism, simulations must incorporate the effects of friction between contact bodies. The ball which is bouncing on the floor, will move with horizontal speed. It will stop moving by the negative acceleration which is caused by friction [13].

The following pseudo code will show the motion affected by friction:

```
while ( the ball is still moving )
{
    x += v0_x * delta_time:

    y = 0:

    x = x - friction_displacement:

    display( x, y ):

    friction_displacement += fric: /* friction causes acceleration */
}
```

This code is valid under the condition that the ball is still moving. How to detect if the ball satisfies the condition, please see “stop_moving” detection in section 3.9.

If the friction does not affect the motion very much, it can be ignored. For this case, the friction which is caused by the impact is ignored. The friction generated after the ball

stops bouncing and slides on the floor or the truck cannot be ignored. Otherwise, the ball will keep moving for ever, which never happens in the real world.

3.8 Rendering Objects

There are two objects in this animation: a ball and a truck. The ball is represented by a disc and the truck is represented by two discs and a rectangle. As OpenGL does not have the command for drawing a disc, the animator program has to create a disc by geometric primitives such as triangles.

Subdivide a disc into small pies by drawing lines from center to circumference of a circle. When the number of subdivisions is very big, the pies will look like triangles. Please see the following figure. Discs with a small number subdivisions can be rendered quicker than discs with many subdivisions. But discs with more subdivisions look smoother.

The truck consists of a rectangle and two discs. OpenGL has function for drawing rectangles. Please see the following figure.

The rendering speed depends on the amount of pixels which an object covers and the number of subdivisions. The bigger an object, the slower the rendering speed. The higher amount of subdivision a disc has, the more time-consuming rendering is, but the disc looks smoother or more realistic.

3.9 Implementation

Although the common Newtonian equation of motion $F=ma$ solves the kinematic problem, it is difficult to write simulation codes to show the motion. There is a lot of detection involved: for example, ball and truck collision detection, ball and floor collision detection, ball's "stop-bouncing" detection, and ball's "stop-moving" detection.

“Stop-bouncing” detection tests if the impact is big enough to move the ball. If it is not big enough, then the ball will stop bouncing. As the ball will lose energy each time it collides with other objects, the value of velocity will decrease after collision: based on EQ6, $\Delta h_0 = (v_y)_0 \Delta t - \frac{1}{2} g \Delta t^2$. The ball will stop bouncing when $(v_y)_0$ is not big enough so that $\Delta h_0 \leq 0$. This is the point that the ball stops bouncing.

“Stop-moving” detection tests if the horizontal speed changes to 0 by negative acceleration. If so, the ball will stop moving.

In addition, as the frame rate is ten frames per second, some collision moments cannot be shown at the right frame time. Please see moment 1, moment 2, moment 4 and moment 6 in Figure 17. The ball seems to be repelled by the floor or the truck when it is going to hit them. It looks as if the ball is flying in the air.

To avoid this problem, a frame, which shows that the ball touches the floor or the truck, will be inserted after the collision. The frame is only inserted if the ball’s distance from the floor or the truck, in the frame just before the collision happens, is greater than ϵ (ϵ is tolerance error). Figure 18 shows the situation when collision frames are inserted. The flying ball problem is avoided. The motion looks more realistic.

FIGURE 17. Collision Moments

FIGURE 18. The Inserted Frames

FIGURE 19. Overview of The Animation Program

Figure 19 shows how the animation program works. The detail of Block A is shown in the following figure.

Chapter 4. Animation of Rigid Bodies

4.1 Introduction

A rigid body is defined by some number of points that must move together. They may not move relative to each other, though they may move as a whole relative to the world space. The motion of a rigid body is specified by six degrees of freedom: x, y, z translation and x, y, z rotation.

Dynamics analysis for motion control was discussed to solve particle problems, where there is no rotation involved was discussed in the last chapter. The same method will be described in this chapter in order to simulate rigid bodies.

4.2 Rotating Motion Dynamics

A big rotating ball is bouncing between a rotating gear and a static board. There are two kinds of motion involved: translation and rotation. Both kinds of motion will be changed when it hits the gear and the board. Please see Figure 20.

Assume the height of the ball bounced up from the floor is less than the diameter of the gear, and the distance between the axis of the gear and the floor is equivalent to the radius of the gear.

FIGURE 20. Picture of The Experiment Two

4.3 Theory of Rigid Dynamics

A rigid body is treated as a system of material particles. The principal characteristic of a rigid body is that its size and shape do not change. so a rigid body is one in which the distance of each point of the system from every other is fixed. Its position is known if the positions of any three of its points are known.

We call a motion **translation** if a rigid body is displaced in such a manner that the displacement of every point is represented by equal vectors, that is equal in length and parallel.

When a rigid body moves in such a way that two points in it are fixed, this is equivalent to fixing a line of particles, and the motion can only be about a fixed axis. This motion is **rotation about a fixed axis** [14].

When a rigid body moves in such a way that each point in it moves along a circular path and the center of these circular paths lies along a line passing through the fixed point, this is **rotation about a fixed point**.

The basic quantities in linear motion are displacement, velocity, and acceleration. The corresponding quantities in rotational motion are angular displacement, angular velocity, and angular acceleration. Angular displacement is the angle traversed by a point moving in a plane, a circular path about an axis that is perpendicular to the instantaneous direction of motion.

Angular velocity ω is measured by the angle described in a unit of time if the motion is uniform, or by $\frac{d\theta}{dt}$ if the motion is not uniform. The direction of angular velocity is based on the right hand rule. Please see the following figure. Particle P has an angular velocity ω , the distance between P and its rotation axis is r , the velocity of particle P is v , $v = \omega r$.

Proposition 1: For angular velocities about the same rotation axis, the resultant is the algebraic sum.

Proposition 2: If a body has at any instant two angular velocities about two axes drawn from a point, and if lengths OA , OB are taken on the axis to represent the angular velocities in direction and in magnitude, then the resultant angular velocity will be the diagonal OC of the parallelogram of which OA , OB are adjacent sides [15] - [19].

Please see the above figure.

4.4 Motion Analysis

The bouncing ball's motion is a combination of translation and rotation. The ball's translation is a projectile motion under gravity. The ball's rotation is relative to the ball's projectile motion and about a fixed point which is the ball's center.

4.4.1 Translation Analysis

As we discussed in last chapter, projectiles move in a vertical plane. Assume the ball moves in the XY plane, and the gear rotates in a plane which is parallel with the YZ plane. The axis of the gear is in the XY plane and its distance from the floor equals to the radius

of the gear. Assume the floor is the XZ plane, the board is the YZ plane and the gear rotates clockwise if it is viewed along the X axis. Please see the following figure.

According to projectile motion equations we analyzed in last chapter,

$$x_{k+1} = x_k + (v_x)_0 \Delta t$$

$$y_{k+1} = y_k + \Delta h_0 - k g (\Delta t)^2$$

$$\Delta h_0 = (v_y)_0 \Delta t - \frac{1}{2} g (\Delta t)^2$$

we get the animation code like this:

```
for ( : : )
```

```

{
    mg_displacement += GTT;
    yTranslate += delta_h0 - mg_displacement;
    xTranslate += delta_s;
    displace_ball_at(xTranslate, yTranslate);
}

```

If the ball hits the floor, which means $yTranslate \leq 0$, the horizontal velocity does not change, the vertical speed does not change, but the vertical velocity's direction change is 180 degrees. Please see the following figure (a). At this time, reset the variable `mg_displacement` to 0, or the ball will fall through the floor. The animation code is like this:

```

for ( : : )
{
    if (yTranslate <= 0)    mg_displacement = 0;
    mg_displacement += GTT;
    yTranslate += delta_h0 - mg_displacement;
    xTranslate += delta_s;
    display_ball_at(xTranslate, yTranslate);
}

```

If the ball hits the gear or the board, it bounces off it. Assume the impact is perfect elastic case. Thus, there is no energy loss and no friction involved during the impact. The ball's speed will not be changed; however, its direction will be changed and the rebound is a

mirror image of the approach [20]. Please see the following figure (b). The vertical veloc-

ity is not affected, and the horizontal speed is not changed, but the direction of the horizontal velocity is changed to the opposite. The animation code is like this after considering the collision between the ball and the board or the gear:

```
for ( : : )
{
    if ( yTranslate <= 0 )      mg_displacement = 0;

    if ( xTranslate > gear's x_coordinate ) ||
        ( xTranslate < board's x_coordinate )

        delta_s = - delta_s;

    xTranslate += delta_s;

    yTranslate += delta_h0 - mg_displacement;

    display_ball_at(xTranslate, yTranslate);
}
```

4.4.2 Rotation Analysis

As the ball's motion is in the XY plane, the ball-gear impact points are along AB. Assume O is center of the gear. The velocity of the ball depends on which part of the gear it hits.

There are two options: either the collision points fall on OA or fall on OB. If the ball falls on OA, it will get a downward angular velocity. If the ball falls on OB, it will be given an upward angular velocity. Please see the above figure.

Assume P is a point where the ball and the gear collide, and the velocity of the point P on the ball is the same as the velocity of the point P on the gear after they collide. Ignore the ball's moving motion, assume v_{P-ball} is the velocity of the impact point on the ball's

equator, ω_{ball} is the angular velocity of the ball, r_{ball} is the radius of the ball, v_{P-gear} is the velocity of the point P on the gear. OP is the distance from P to the center of the gear. Please see the following figure.

According to rigid dynamics, the following equations are obtained:

$$v_{P-ball} = \omega_{ball} \cdot r_{ball}$$

$$v_{P-gear} = \omega_{gear} \cdot OP$$

As $v_{P-ball} = v_{P-gear}$, we get $\omega_{ball} \cdot r_{ball} = \omega_{gear} \cdot OP$. So the ball's angular velocity

$$\omega_{ball} = \frac{\omega_{gear} \cdot OP}{r_{ball}}$$

As $OP = y_{ball} - r_{gear}$, the above equation is

$$\omega_{ball} = \frac{\omega_{gear} (y_{ball} - r_{gear})}{r_{ball}}$$

where y_{ball} is the y-coordinate of the ball.

Assume there is no friction on the board or on the floor so that the ball will keep the angular velocity until it hits the gear again. The angular velocity which the ball is given depends on where the impact point is according to the above equation.

4.5 Collision Detection

The bounding box method is used in this animation. If $ball_x > gear_x - \text{half width of gear} - \text{ball's radius}$, then collision occurs. If $ball_x < board_x + \text{ball's radius}$, then collision occurs. If $ball_y < floor_y$, then collision occurs.

4.6 Rendering Objects

Objects are normally represented by their surfaces in computer graphics. Polygon mesh is a common method used in surface modeling. The following sections will describe how to use polygon meshes to render the gear, the ball, the board and the floor.

A polygon mesh is a collection of edges, vertices, and polygons connected in such a way that each edge is shared by, at most, two polygons. A gear's surfaces consist of two faces, inner cylinder and outward faces of teeth. A ball's surfaces consist of polygons divided by

longitudes and latitudes. Figure 21 shows the components of the gear and the ball. The board and the floor are represented by parallel and perpendicular lines.

FIGURE 21. The Components of The Gear and The Ball

4.6.1 Gear Rendering

The gear face consists of teeth and a disk. Assume the radius of the hole is r_0 , the outer radius is r_2 , the teeth depth is $r_2 - r_1$. da is a quarter of angle that a tooth takes.

$da = \frac{2\pi}{teethNumber} / 4$. Please see the following figure.

Polygon $p_1 p_2 p_3 p_4$ constructs a tooth of the gear. The coordinates of the four vertices are:

$$p_1 = (r_1 \cos a, r_1 \sin a)$$

$$p_2 = (r_2 \cos (a + da), r_2 \sin (a + da))$$

$$p_3 = (r_2 \cos (a + 2da), r_2 \sin (a + 2da))$$

$$p_4 = (r_1 \cos (a + 3da), r_1 \sin (a + 3da))$$

Assume variable width is the width of the gear. variable teeth is the number of the teeth.

To draw the teeth of the gear's front face, the following code is used:

```
glBegin( GL_QUADS );

    da = 2.0*PI / teeth / 4.0;

    for (i=0; i<teeth; i++)
    {
        a = i * 2.0*PI / teeth;

        glVertex3f( r1*cos(a), r1*sin(a), width*0.5 );

        glVertex3f( r2*cos(a+da), r2*sin(a+da), width*0.5 );

        glVertex3f( r2*cos(a+2*da), r2*sin(a+2*da), width*0.5 );

        glVertex3f( r1*cos(a+3*da), r1*sin(a+3*da), width*0.5 );
    }

glEnd();
```

Similarly, to draw the teeth of the gear's back face, the z coordinates will be changed from "width*0.5" to "-width*0.5". To draw the outward faces of the teeth, the following code is used:

```
glBegin( GL_QUAD_STRIP );

    for (i=0; i<teeth; i++)
    {
        a = i * 2.0*PI / teeth;

        glVertex3f( r1*cos(a), r1*sin(a), width*0.5 );
```

```

    glVertex3f( r1*cos(a), r1*sin(a), -width*0.5 );

    glVertex3f( r2*cos(a+da), r2*sin(a+da), width*0.5 );

    glVertex3f( r2*cos(a+da), r2*sin(a+da), -width*0.5 );

    glVertex3f( r2*cos(a+2*da), r2*sin(a+2*da), width*0.5 );

    glVertex3f( r2*cos(a+2*da), r2*sin(a+2*da), -width*0.5 );

    glVertex3f( r1*cos(a+3*da), r1*sin(a+3*da), width*0.5 );

    glVertex3f( r1*cos(a+3*da), r1*sin(a+3*da), -width*0.5 );
}

glVertex3f( r1*cos(0), r1*sin(0), width*0.5 );

glVertex3f( r1*cos(0), r1*sin(0), -width*0.5 );

glEnd();

```

To draw the disk of the front face, the following code is used:

```

glBegin( GL_QUAD_STRIP );

    for (i=0; i<=teeth; i++)

    {

        a = i * 2.0*PI / teeth;

        glVertex3f( r1*cos(a), r1*sin(a), width*0.5 );

        glVertex3f( r0*cos(a), r0*sin(a), width*0.5 );

        glVertex3f( r1*cos(a+3*da), r1*sin(a+3*da), width*0.5 );

        glVertex3f( r0*cos(a), r0*sin(a), width*0.5 );

    }

```

glEnd();

To draw inside cylinder, the following code is used:

```
glBegin( GL_QUAD_STRIP );  
    for (i=0; i<=teeth; i++)  
  
    {  
  
        angle = i * 2.0*PI / teeth;  
  
        glVertex3f( r0*cos(angle), r0*sin(angle), -width*0.5 );  
  
        glVertex3f( r0*cos(angle), r0*sin(angle), width*0.5 );  
    }  
glEnd();
```

Three dimensional objects are modeled and rendered by polygon meshes of objects' surfaces. However, these objects cannot be seen without lighting. Lighting reveals a three dimensional world. It shades surfaces based on the position, orientation and characteristics of materials. Without lighting, three dimensional objects look like polygons, and objects at different distances look like at the same distance.

Lighting involves a calculation of intensity of each vertex. Object's normals play an important role. The normals determine the orientation relative to the light source. For each vertex, OpenGL uses the assigned normal to determine how much light that particular ver-

tex receives from each light source. Thus, to make 3D objects visible, not only the lighting library calls have to be used, but also the normal of each vertex has to be set properly.

Please see the above figure. Vector 1, vector 2 and vector 3 are normals of tooth surfaces.

$vector1 = \{x1, y1\}$. Assume u, v are vectors where

$$u = r1 \cdot \cos(a + da) - r1 \cdot \cos a$$

$$v = r1 \cdot \sin(a + da) - r1 \cdot \sin a$$

As vector $\mathbf{l}$ is perpendicular to surface $p_1 p_2$, it yields

$$\frac{u}{y_1} = \frac{v}{x_1} = k$$

$$y_1 = -ku$$

$$x_1 = kv$$

As normal vectors present orientation of surface, the most important aspect is their direction instead of magnitude. So assume $k = 1$, then we get $y_1 = -u$ and $x_1 = v$. Similarly, see

$$y_3 = -r_1 \cdot \cos(a + 3da) + r_1 \cdot \cos(a + 2da)$$

$$x_3 = r_1 \cdot \sin(a + 3da) - r_1 \cdot \sin(a + 2da)$$

$$y_2 = \cos(a + 1.5da)$$

$$x_2 = \sin(a + 1.5da)$$

$$y_4 = \cos(a + 3.5da)$$

$$x_4 = \sin(a + 3.5da)$$

the following code.

```
glBegin( GL_QUAD_STRIP );
```

```
    for (i=0; i<teeth; i++)
```

```
    {
```

```
        a = i * 2.0*PI / teeth;
```

```
        glVertex3f( r1*cos(a), r1*sin(a), width*0.5 );
```

```
        glVertex3f( r1*cos(a), r1*sin(a), -width*0.5 );
```

```
        u = r2*cos(a+da) - r1*cos(a);
```

```

v = r2*sin(a+da) - r1*sin(a);

glNormal3f( v, -u, 0.0 )                                /* vector1 */

glVertex3f( r2*cos(a+da), r2*sin(a+da), width*0.5 );

glVertex3f( r2*cos(a+da), r2*sin(a+da), -width*0.5 );

glNormal3f( cos(a+1.5*da), sin(a+1.5*da), 0.0 );        /* vector2 */

glVertex3f( r2*cos(a+2*da), r2*sin(a+2*da), width*0.5 );

glVertex3f( r2*cos(a+2*da), r2*sin(a+2*da), -width*0.5 );

u = r1 * cos(a+3*da) - r2*cos(a+2*da);

v = r1 * sin(a+3*da) - r2*sin(a+2*da);

glNormal3f( v, -u, 0.0 );                                /* vector3 */

glVertex3f( r1*cos(a+3*da), r1*sin(a+3*da), width*0.5 );

glVertex3f( r1*cos(a+3*da), r1*sin(a+3*da), -width*0.5 );

glNormal3f( cos(a+3.5*da), sin(a+3.5*da), 0.0 );        /* vector4 */
}

glVertex3f( r1*cos(0), r1*sin(0), width*0.5 );

glVertex3f( r1*cos(0), r1*sin(0), -width*0.5 );

glEnd();

```

Thus, in order to draw 3D objects, not only their surfaces have to be rendered, but also the normals of the surfaces have to be set up.

4.6.2 Ball Rendering

Assume the radius of the ball is 1 and the center is origin. The ball's surface is divided by longitudes and latitudes. The longitudes are obtained by cutting the ball by planes $y = \cos(a + i \cdot da)$, where $a + i \cdot da$ is the angle between the Y axis and the line from origin to a point on the longitudes, $a = -90$, i is integer from 0 to $\frac{180}{da}$. The latitudes are the intersections between the ball surface and the planes which are vertical to the plane XZ, along the Y axis and have angle $j \cdot db$ with the XY plane, j is integer from 0 to $\frac{360}{db}$. Please see Figure 22.

These longitudes and latitudes form the polygon meshes. For example, longitude1, longitude2, latitude1 and latitude2 construct polygon mesh v1 v2 v3 v4. Assume (x1, y1, z1) is v1, (x2, y2, z2) is v2, (x3, y3, z3) is v3 and (x4, y4, z4) is v4. The angle between the Y axis and the line from the origin to a point on longitude1 is a. The angle between the Y axis and the line from the origin to a point on longitude2 is a + da. The angle between the XY plane and the latitude1's plane is b. The angle between the XY plane and the latitude2's plane is b + db. So the coordinates of v1 v2 v3 and v4 are:

$$x1 = \cos(b) * \cos(a)$$

$$z1 = \sin(b) * \cos(a)$$

$$y1 = \sin(a)$$

$$x2 = \cos(b) * \cos(a+da)$$

$$z2 = \sin(b) * \cos(a+da)$$

$$y2 = \sin(a+da)$$

$$x3 = \cos(b+db) * \cos(a)$$

$$z3 = \sin(b+db) * \cos(a)$$

$$y3 = \sin(a)$$

$$x4 = \cos(b+db) * \cos(a+da)$$

$$z4 = \sin(b+db) * \cos(a+da)$$

$$y4 = \sin(a+da)$$

FIGURE 22. A Polygon Mesh on The Sphere

To draw the sphere it is necessary to draw all the polygon meshes. The polygon meshes between two longitudes are drawn first, then the ones between the next longitude and the current longitude, and so on. The code for drawing the sphere is similar to this:

```
for ( a = -90; a+da<= 90; a=a+da )

{
    for (b=0.0;b<=360.0;b+=db) /* draw meshes between two longitudes */
    {
        x1 = cos(b) * cos(a); /* ball's radius is 1 */
        z1 = sin(b) * cos(a);
        y1 = sin(a);

        x2 = cos(b) * cos(a+da);
        z2 = sin(b) * cos(a+da);
        y2 = sin(a+da);

        x3 = cos(b+db) * cos(a);
        z3 = sin(b+db) * cos(a);
        y3 = sin(a);

        x4 = cos(b+db) * cos(a+da);
        z4 = sin(b+db) * cos(a+da);
        y4 = sin(a+da);

        draw_polygon( (x1, y1, z1), (x2, y2, z2), (x3, y3, z3), (x4, y4, z4) );
    }
}
```

It is difficult for people to see rotating motion if the ball has only one color. There are different ways to color the ball. One is texture mapping, the other is drawing different colors on different polygon meshes. The latter method is used in this animation. Assume the polygon meshes between the first two latitudes are colored white, the ones between the second and the third latitude are colored as blue, and so on. The boolean variable “color” is used to keep track of the color for the previous polygon mesh. The code expressed like this:

```
for ( a = -90; a+da<= 90; a=a+da )
{
    color = 0;
    for (b=0.0;b<=360.0;b+=db) /* draw meshes between two longitudes */
    {
        if ( color )    set_color(blue);
        else           set_color(white);

        x1 = cos(b) * cos(a); /* ball's radius is 1 */
        z1 = sin(b) * cos(a);
        y1 = sin(a);

        x2 = cos(b) * cos(a+da);
        z2 = sin(b) * cos(a+da);
        y2 = sin(a+da);

        x3 = cos(b+db) * cos(a);
        z3 = sin(b+db) * cos(a);
        y3 = sin(a);

        x4 = cos(b+db) * cos(a+da);
```

```

        z4 = sin(b+db) * cos(a+da);
        y4 = sin(a+da);

        draw_polygon( (x1, y1, z1), (x2, y2, z2), (x3, y3, z3), (x4, y4, z4) );
        color = 1 - color;
    }

}

```

4.6.3 The board and the floor

The board or the floor are represented by planes which are drawn by parallel lines and perpendicular lines.

4.7 Implementation

The following figure shows how the animation program works. Figure 23 shows the ani-

mation procedure in detail.

FIGURE 23. A Detail Chart of The Experiment Two

Chapter 5. Conclusion

5.1 Summary

The previous chapters have introduced and described the background knowledge, analysis, simulation and implementation methods in computer animation domain. As 4D graphics (the addition of the time dimension), computer animation involves three stages: modeling, motion control and rendering.

Modeling is an analysis process which provides methods to present objects on a computer screen. Motion control is the technique to specify motions of moving and rotating objects. Dynamic analysis is a popular techniques which can predict motion based on physical laws so that the simulated motion is very realistic. Rendering is to draw objects using graphics software based on the modeling so that objects can be seen on a computer screen.

Dynamic analysis method is used to create formulas which calculate the position and orientation of objects [29]. C programming language is used to calculate and pass the results to OpenGL functions which is used to draw objects.

The C animation programs demonstrate that the implementations of the methodology for the two cases work very well.

5.2 Animation Complexity

Animation involves modeling, motion control and rendering. Except for modeling and dynamic analysis which creates formulae, the rest of the stages are all on-line processing.

The on-line motion control involves calculation according to formulae and collision detection. The rendering process involves drawing polygon meshes and lighting.

Calculation complexity depends on how complicated the environment is. It involves gravity, friction, energy loss, elastic or non-elastic impact, coefficient restitution and other forces which act on the object.

The complexity of collision detection depends on the number of objects and the shape of each object. The number of comparison is $O(n^2)$, where n is the number of objects. The complicated shape will increase the complexity in collision detection if the Flexible Surface Algorithm or the Convex Polyhedra Algorithm is used.

Drawing polygon meshes of an object's surfaces is very complicated process. The complexity depends on how complicated the shape is. The number of meshes, the shape of meshes, and the size of the object will decide how long it will take to draw the object.

Lighting model in object rendering is time-consuming. It involves a large amount of computation. Lighting calculation is built into the graphics software. To solve the time-consuming problem, the relatively uncomplicated approximation equations are used for the lighting model.

This thesis has focused on the methodology used for creating animation programs through rigid bodies that moves according to mathematical models. It is not meant to address the

general problem of creating animated objects such as people or animals which are common in the entertainment industry.

5.3 Improvement

As we can see, the whole procedure of computer animation from motion control to object rendering, is time consuming. Therefore, the speed for rendering one frame, or the frame rate, is one of the problems in computer animation. If the frame rate is low, the jaggy motion, which is called temporal aliasing, will be seen. Just as the aliasing problems in 2D and 3D graphics are partially solved by increasing the screen resolution, the temporal aliasing problem in animation can be partially by increasing temporal resolution.

To increase temporal resolution, the speed of rendering each frame must be increased. This can be solved by using a fast graphics software and special computer architecture to speed up time-consuming processing of calculation or comparison instructions.

OpenGL is not the fastest graphics software obtainable, however, it is easy to use, and it is the most popular. It has become a standard and it can be run on both Unix and PC. The equivalent free version of OpenGL, which is called Mesa, is used in the thesis. The performance is good in the two simulation cases. However, if the animation case is more complicated (more objects involved, and more complicated surfaces), Mesa will not meet the requirement. This problem can be solved by running animation programs on Silicon Graphics Workstation and using real OpenGL version.

To decrease the time-consuming in calculation and comparison, accurate formulae for calculation can be approximated to a reasonable level, and a fast algorithm for collision detection can be used to minimize the comparison times.

There is a trade-off between time and precision, the more precise the calculation or comparison is, the more time the animation process takes. As long as the precision meets our requirement, the approximation in calculation and comparison can be used to increase the speed of animation.

5.4 Future of Animation

Computer animation is a mixture of traditional animation with computer graphics. It has not yet developed into a discipline where a universal language is used by animators to manipulate moving objects on computer screen. However, the relative research has already begun, such as a visual reality system [30] which allows users to interact with an animated environment through different process-bound objects. Each process-bound object is controlled by a process so that objects in a scene can interact and be navigated by concurrent processes. Furthermore, users or participants connected by a network can meet and collaborate [30].

To conclude: research in computer animation has great potential. It is widely used today and will become an indispensable part of our life in future.

References

- [1] Jane Welhelms. "Toward Automatic Motion Control". *IEEE Computer Graphics and Applications*, vol. 7, no. 4, pp. 11-22, April 1987.
- [2] Nadia Magnemat-Thalmamn & Danial Thalmamn. *Computer Animation Theory and Practice*. Springer-Verlag 1985. Tokyo.
- [3] Julio Sanchez & Maria P. Canton. *Computer Animation Programming Methods & Techniques*. McGraw-Hill 1995. New York.
- [4] Harvey Gould & Jan Tobochnik. *An Introduction to Computer Simulation Methods*. Addison-Wesley Publishing Company 1996. New York.
- [5] Jackie Neider, Tom Davis and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company 1993. New York.
- [6] James Foley, Andries Dam, Steven Feiner and John Hughes. *Introduction to Computer Graphics*. Addison-Wesley Publishing Company 1994. New York.
- [7] John Vince . *3-D Computer Animation*. Addison-Wesley 1992. New York.
- [8] John Lasseter. "Principles of Traditional Animation Applied to 3D Computer Animation". *Computer Graphics*, vol. 21, no. 4, pp. 35-44, July 1987.

- [9] Isaac Victor Kerlow, *The Art of 3-D Computer Animation and Imaging*, Van Nostrand Reinhold, New York.
- [10] Roger McHaney, *Computer Simulation - A Practical Perspective*, Academic 1991, San Diego, Calif.
- [11] Tetsuya Uchiki, "Collision Detection in Motion Simulation", *Computers & Graphics*, vol. 7, no. 3-4, pp285-293, 1983
- [12] John W. Boyse, "Interference Detection Among Solids and Surfaces, *Communication of the ACM*, vol. 22, no. 1, pp. 3-9, Jan 1979.
- [13] David Baraff, "Coping with Friction for Non-penetrating Rigid Body Simulation", *Computer Graphics*, July 1991.
- [14] Arthur Gordon Webster, *The Dynamics of Particles and Rigid, Elastic, and Fluid Bodies*, Hafner Publishing Company Inc. 1949, New York.
- [15] W. J. Loudon, *An Elementary Treatise on Rigid Dynamics*, MacMillan and Co. 1896, New York.
- [16] TWB Kibble, *Classical Mechanics*, Longman 1985, London and New York.
- [17] Tai L. Chow, *Classical Mechanics*, John Wiley & Sons. 1995, New York.
- [18] Raymond M. Brach, *Mechanical Impact Dynamics Rigid Body Collisions*, John Wiley & Sons. 1991, New York.

- [19] J. L. Meriam & L. G. Kraige, *Dynamics*. John Wiley & Sons. 1992, New York.
- [20] Matthew Moore & Jame Wilhelms, "Collision Detection and Response for Computer Animation", *Computer Graphics*, vol. 22, no. 4, pp. 289-298, Aug 1988.
- [21] Jame K. Halm, "Realistic Animation of Rigid Bodies", *Computer Graphics*, vol. 22, no. 4, Aug 1988, pp. 299-308.
- [22] Andrew Witkin, "Fast Animation and Control of Nonrigid Structures", *Computer Graphics*, vol. 24, no. 4, pp. 243-252, Aug 1990.
- [23] Michael F. Cohen, "Interactive Spacetime Control for Animation", *Computer Graphics*, vol. 26, no. 2, pp. 293-302, July 1992.
- [24] David Baraff, "Interactive Simulation of Solid Rigid Bodies", *IEEE Computer Graphics and Applications*, 1995, pp. 63-75.
- [25] Armin Bruderlin, "Procedural Movement For Articulated Figure Animation", *Computers & Graphics*, vol. 18, no. 4, pp.453-461, 1994.
- [26] David Baraff, "Analytical Method for Dynamic Simulation of Non-penetrating Rigid Bodies", *Computer Graphics*, vol. 23, no. 3, pp. 223-231, July 1989.
- [27] Wen Bang Liu, "An Interactive Approach to Planning Snake Motion", *Computers & Graphics*, vol. 18, no. 4, pp. 537-542, 1994.

- [28] Ronen Barzel. "A Modeling System Based on Dynamic Constraints". *Computer Graphics*. vol. 22, no. 4, pp. 179-188, Aug 1988.
- [29] Jane Wilhelms. "Using Dynamic Analysis for Realistic Animation of Articulated Body". *IEEE Computer Graphics and Applications*. vol. 7, no. 6, pp. 12-27, June 1987.
- [30] Nadia Magnemat Thalmamn & Danial Thalmamn. *New Trends in Animation and Visualization*. John Wiley & Sons 1991, London.

Appendix

```

.....
* particle.c - the experiment one
*
* This program shows a ball bouncing on a floor and a truck
* on the floor
*
.....

#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "glaux.h"

#define WIDTH      1000          * window width                *
#define HEIGHT    500           * window height             *
#define MAX_SLICES 500          * no. of slices in a pre    *
#define PI        3.14159
#define RADIUS_BALL 2.0
#define RADIUS_WHEEL 2.0
#define FLOOR_H    -42.0
#define FLOOR_W    3.0
#define FLOOR_L    100.0
#define TRUCK_BODY_H 10.0
#define TRUCK_L    12.0
#define K          0.2          * percentage of energy loss *
#define G          50.0         * G = 50 units sec^2sec, 1 unit = 1.5 meter *
#define GTT        0.5         * G * sqr delta t , G = 50 units sec^2sec *
                                * 1 unit = 1.5 meter , delta t = 0.1 *

time_t   time1, time2;
double   frame_time;

enum      status(DOWN_SIDE, DOWN_TOP, UP, EDGE);
enum      status collision_status;

int       count      = 0;
int       stop_bounce = 0;

```



```

int    stop_truck = 0;
float  v_x0;          * ball's initial speed x-axis (units/frame) *
                        * v_y0 is ball's vertical initial speed per jump *
float  v_y0          = 32.5;      * v_y0 = (delta_h0 - GTT 2 * delta_t) *
float  old_v_y0      = 0.0;
float  old_jmp_h     = 0.0;      * height the ball jumps from *
float  delta_h0      = 8.0;      * ball's vertical displacement per time step *
float  delta_t       = 0.1;      * per time step = 0.1 sec *
float  delta_s;
float  delta_s_t     = 0.9;

float  TRUCK_H       = TRUCK_BODY_H + 2 * RADIUS_BALL;

float  truck_bound_l;
float  truck_bound_r;
float  ball_bound_l;
float  ball_bound_r;

float  x_collision;
float  y_collision;

float  xTranslate = 0.0;
float  xTranslate_t = 0.0;
float  yTranslate = 0.0;
float  xOldTranslate = -1.0;
float  yOldTranslate = -1.0;
float  ball_bound_l;
float  ball_bound_r;

float  x_collision;
float  y_collision;

float  xTranslate    = 0.0;
float  xTranslate_t  = 0.0;
float  yTranslate    = 0.0;
float  xOldTranslate  = -1.0;
float  yOldTranslate  = -1.0;
float  xOldTranslate_t = -1.0;
float  mg_displacement = 0.0;    * displacement caused by gravity acceleration *
*
float  loss          = 1.0;      * ball loses energy when it is bounced up *
float  fric          = 0.01;
float  friction;
float  fric_t        = 0.01;
float  friction_t    = 0.0;
float  ball_x0       = -80.0;    * ball's initial position (x-coordinate) *
float  truck_x0      = -70.0;    * truck's initial position (x-coordinate) *
*
float  truck_x;

void drawDisk(int r, float x, float y)
{
    int i;
    float delta = 2.0 * PI / MAX_SLICES;

```

```

glBegin(GL_POLYGON);

    for(i=0; i<= MAX_SLICES ; i++)
        glVertex2f((float)r*cos(delta * i) - x, (float)r*sin(delta * i) - y);

glEnd();

glFlush();
}

void drawTruck(float x, float y) /* x,y is center coordinate of truck's body */
{
    float wheel_x_l = x - TRUCK_L/4;
    float wheel_x_r = x + TRUCK_L/4;
    float wheel_y = y - TRUCK_BODY_H/2.0 - RADIUS_WHEEL;

    glColor3f(0.0, 0.6, 0.6);

    glRectf (x-TRUCK_L/2, y-TRUCK_BODY_H/2, x+TRUCK_L/2, y+TRUCK_BODY_H/2); /*
body of truck */
    drawDisk(RADIUS_WHEEL, wheel_x_l, wheel_y); /* left wheel */
    drawDisk(RADIUS_WHEEL, wheel_x_r, wheel_y); /* right wheel */
}

void drawBall(float x, float y)
{
    glColor3f(1.0, 0.0, 0.0);
    drawDisk(RADIUS_BALL, x, y);
}

.....
draw ball and truck
.....

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);

    glPushMatrix();

    /* draw floor */
    glColor3f(0.5, 0.7, 0.9);
    glRectf (-FLOOR_L, FLOOR_H - FLOOR_W, FLOOR_L, FLOOR_H);

    /* draw moving little ball */
    glPushMatrix();
    glTranslatef(xTranslate, yTranslate, 0.0);
    drawBall(ball_x0, FLOOR_H - RADIUS_BALL);
    glPopMatrix();
}

```

```

    * draw little truck *
    glPushMatrix ();
    glTranslatef(xTranslate_t, 0.0, 0.0);
    drawTruck(truck_x0, FLOOR_H - 2*RADIUS_WHEEL + TRUCK_BODY_H/2);
    glPopMatrix ();

    glPopMatrix ();
    glFlush();
    auxSwapBuffers();
}

.....

Collision detection

Did the ball collide with the truck either its front or top
between the last frame & the current frame
.....

int collision
{
    * truck's left boundary *
    truck_bound_l = truck_x0 - xTranslate_t - TRUCK_L/2;

    * truck's right boundary *
    truck_bound_r = truck_x0 - xTranslate_t + TRUCK_L/2;

    * ball's left boundary *
    ball_bound_l = ball_x0 - xTranslate - RADIUS_BALL;

    * ball's right boundary *
    ball_bound_r = ball_x0 - xTranslate + RADIUS_BALL;

    *-----*
    If the ball enters the border of truck then collides,
    the following logic can be written in cleaner way.
    *-----*

    if (yTranslate >= TRUCK_H)
        return 0;
    else if ((ball_bound_r < truck_bound_l)
            || (ball_bound_l > truck_bound_r))
        return 0;
    else
        return 1;

    *-----*

    The above logic can be written in cleaner way.
    It is the same as the following.
    *-----*

    if (yTranslate >= TRUCK_H)
        return 0;
    else return ! ((ball_bound_r < truck_bound_l)
                  || (ball_bound_l > truck_bound_r));
}

```

```

.....
When ball stops bouncing, it will slide on the floor or on the
truck. The ball speed will be affected by the friction.
.....
void stop_bouncing void
{
    stop_bounce = 1;
    yTranslate = 0.0;          * stop bouncing on the floor *

    if (v_x0 > 0)              * ball's direction is along x-axis *
    {
        if (friction > v_x0)    * stop moving *
            xTranslate = friction < xOldTranslate * prevent ball from
moving back *
            xTranslate = xOldTranslate;
        else
            xTranslate -= friction;
    }
    else                        * ball's direction is along -x axis *
    {
        * when the ball touches floor, its motion is affected by friction *
        if (-friction > -v_x0)  * stop moving *
            -xTranslate = friction < -xOldTranslate * prevent ball from moving
back *
            xTranslate = xOldTranslate;
        else
            xTranslate -= friction;
    }
}

```

```

.....
After collision happens, the ball is bounced off with a speed.
Collision scene cannot be shown if the collision time is
different from the next frame time. In this case, you will
see the ball never touches the truck and flies away. To avoid
this problem and make the simulation more realistic, collision
scene is forced to display before the next frame is displayed.
.....
void force_display(float x_forced, float y_forced)
{
    float xTemp, yTemp;

    xTemp = xTranslate;
    yTemp = yTranslate;
}

```

```

xTranslate = x_forced;
yTranslate = y_forced;

display();

xTranslate = xTemp;
yTranslate = yTemp;
if (! stop_truck )
{
    xTranslate_t -= delta_s_t;
    if (friction_t > delta_s_t
        && (xTranslate_t - friction_t < xOldTranslate_t))
    {
        xTranslate_t = xOldTranslate_t;
    }
    else
        xTranslate_t -= friction_t;
}

}

*****
When the ball is bounced up from the floor or the truck, the
speed will depend on the location of previous jump.
*****

void bounce_up_from_float_height()
{
    float temp;

    if (yOldTranslate > 0.5 - height) /* avoid flying ball */
    {
        force_display(xOldTranslate-v_x0.2, height);
    }

    old_v_y0 = v_y0;

    /*-----
    calculate yTranslate
    -----*/

    if (height == 0.0) /* jump from floor */
    {
        if (old_imp_h == 0.0) /* pre jump from floor */
            v_y0 = old_v_y0 * K;
        if (old_imp_h == TRUCK_H) /* pre jump from truck */
        {
            v_y0 = K * sqrt(old_v_y0 * old_v_y0 - 2 * G * TRUCK_H);
            puts(" jump from table ***** on ");
        }
    }

    mg_displacement = GTT; /* reset otherwise the ball will drop */
    delta_h0 = v_y0 * delta_t - GTT/2;
    if (delta_h0 <= 0.0)
    {
        delta_h0 = 0.0;
    }
}

```

```

        stop_bouncing();
    }
else
    * jump from truck *
    {
        if old_jump_h == TRUCK_H * pre jump from truck *
            v_y0 = old_v_y0 * K;

        if old_jump_h == 1.0 * pre jump from floor *
        {
            temp = old_v_y0 * old_v_y0 - 2 * G * TRUCK_H ;
            if temp > 0
                v_y0 = K * sqrt(temp);
            else
                v_y0 = 1.0;
        }

        mg_displacement = GTT; * reset otherwise the ball will drop *
        delta_h0 = v_y0 * delta_t - GTT/2;
        if delta_h0 <= 1.0
            delta_h0 = 1.0; * moving (stop bouncing) on the truck *
    }

yTranslate = delta_h0 - height;

-----
calculate xTranslate
-----

if v_x0 > 0 * ball's direction is along x-axis *
{
    * when the ball touches the floor, its motion is affected by friction *
    if friction > v_x0 * stop moving *
        xTranslate = friction < xOldTranslate * prevent ball from moving
back *
    {
        xTranslate = xOldTranslate;
    }
else
    xTranslate -= friction;
}
else * ball's direction is along -x axis *
{
    * when the ball touches the floor, its motion is affected by friction *
    if -friction > -v_x0 * stop moving *
        -xTranslate = friction < -xOldTranslate * prevent ball from moving
back *
    {
        xTranslate = xOldTranslate;
    }
else
    xTranslate += friction;
}

```

```

    }

    old_jump_h = height;
}

void bounce_off_down()
{
    v_x0 = -v_x0;
    fric = -fric;
    xTranslate -= v_x0;

    if (stop_bounce)
    {
        friction = -friction;
        xTranslate -= friction;
        yTranslate = 0.0;
    }
    else if (yTranslate < 0.0) * ball touches the floor *
    {
        bounce_up_from 0.0;
    }
}

void bounce_off_up()
{
    v_x0 = -v_x0;
    fric = -fric;
    xTranslate -= v_x0;
}

enum status detection
{
    float x0 = xOldTranslate - ball_x0;
    float y0 = yOldTranslate;
    float x1 = xTranslate - ball_x0;
    float y1 = yTranslate;

    * calculate y-coordinate of collision spot *
    y_collision = y0 - truck_bound_l - RADIUS_BALL - x0 * y1 - y0 * x1 - x0 ;

    if (y1-y0 > 0) * ball's direction is up *
        return UP;
    else
    {
        if (y_collision < TRUCK_H
            return DOWN_SIDE;
        else
            return DOWN_TOP;
    }
}

```

```

    }
}

void moveDisplay void
{
    truck_x = truck_x0;
    xOldTranslate = xTranslate;
    yOldTranslate = yTranslate;
    xOldTranslate_t = xTranslate_t;
    display();

    * truck motion *
    xTranslate_t -= delta_s_t;
    if (friction_t > delta_s_t)
        if (xTranslate_t - friction_t < xOldTranslate_t)
        {
            xTranslate_t = xOldTranslate_t;
        }
    else
        xTranslate_t -= friction_t;

    * ball motion *
    mg_displacement -= GTT;

    if (friction < v_x0 && v_x0 > 0)
        if (-friction < -v_x0 && v_x0 < 0) * still moving on the floor *
            xTranslate -= v_x0;

    yTranslate -= delta_h0 - mg_displacement; * see formula *

    if (collision)
    {
        * puts "colide .. n" ; *
        collision_status = detection;
        switch (collision_status)
        {
            case DOWN_TOP:
                bounce_up_from TRUCK_BODY_H-2*RADIUS_WHEEL.;
                break;

            case DOWN_SIDE:
                bounce_off_down.;
                break;

            case UP:
                bounce_off_up.;
                break;

            default:
                puts "edge -----n";
                bounce_off_edge.;
                break;
        }
    }
}

```



```

else      * no collision between ball and truck *
{
    if (stop_bounce)
    {
        stop_bouncing();
        friction += fric;      * ball's friction *
    }
    else if (yTranslate < 0.0      * ball touches the floor *
        bounce_up_from(0.0);

    friction_t -= fric_t;      * truck's friction *
}

void myinit(void)
{
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
    glShadeModel(GL_FLAT);
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (w <= h)
        glOrtho(-50.0, 50.0, -50.0*GLfloat h/GLfloat w,
            50.0*GLfloat h/GLfloat w, -1.0, 1.0);
    else
        glOrtho(-50.0*GLfloat w/GLfloat h,
            50.0*GLfloat w/GLfloat h, -50.0, 50.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void doNothing(void)
{
}

* Main Loop
* Open window with initial window size, title bar,
* RGBA display mode, and handle input events.
*

int main(int argc, char** argv)
{
    puts("Please enter the initial x-axis speed of the ball (from 0 to 3): ";
    scanf("%f", &v_x0);
    printf("v_x0 = %f\n", v_x0);
}

```

```

puts("Please enter the truck's position   -100, 100  : ");
scanf("%f", &truck_x0);
printf(" truck_x0 = %f\n", truck_x0);

puts("Please enter the ball's position   -100, 100  : ");
scanf("%f", &ball_x0);
printf(" ball_x0 = %f\n", ball_x0);
puts("----- n" );

auxInitDisplayMode (AUX_DOUBLE   AUX_RGB);
auxInitPosition (0, 0, WIDTH, HEIGHT);
auxInitWindow (argv[0] );
myinit ( );
myReshape(WIDTH, HEIGHT);
auxReshapeFunc myReshape;
auxIdleFunc (moveDisplay);
auxMainLoop(doNothing);

```

```

}

```

```

*****
* rigids.c - the experiment two
*
* This program animates a moving and rotating ball bouncing
* between a rotating gear and a static broad
*
* Up, Down, Left and Right Arrow keys can control the viewing
* angles
*
*****

```

```

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

#define PI 3.14159265
#define COS X    cos (X) * 3.14159 180.00
#define SIN X    sin (X) * 3.14159 180.00
#define xMin     -4.0
#define yMin     -3.0
#define xMax      4.0
#define yMax      4.0
#define delta_h0  1.6
#define w         10.0

Gluint  Ball;
Glenum  Mode;
Glfloat GTR      = 0.0;
Glfloat mg_displacement = 0.0;
Glfloat xTranslate = 0.0;
Glfloat yTranslate = yMin - delta_h0;

```

```

GLfloat yRotate      = 0.0;
GLfloat delta_w      = 0.0;
GLfloat delta_s      = 0.2, delta_h      = 0.0;

static GLfloat light_pos[4] = {-55.0, 5.0, 100.0, 0.0 };
static GLfloat ambient[]   = {0.3, 0.3, 0.3, 1.0 };
static GLfloat red[4]      = {0.7, 0.0, 0.0, 1.0 };
static GLfloat white[4]    = {0.8, 0.8, 0.9, 1.0 };
static GLfloat blue[4]     = {0.2, 0.2, 1.0, 1.0 };
static GLfloat yellow[4]   = {0.9, 0.9, 0.0, 1.0 };
static GLfloat view_rotx   = 20.0;
static GLfloat view_roty   = 30.0;
static GLfloat view_rots   = 0.0;
static GLfloat motion_angle = 0.0;
static GLfloat inner_r     = 0.7;
static GLfloat outer_r     = 4.0;
static GLfloat gear_width  = 1.0;
static GLfloat teeth_no    = 20.0;
static GLfloat tooth_depth = 0.7;

```

```

static GLint ball, lines, gear1, gear2, gear3;
static GLuint limit;
static GLuint count = 1;

```

```

static void make_lines void
( int i;

```

```

    glBegin( GL_LINES );
    for i=xMin-1;i<=xMax-1;i++      * vertical *
    {
        glVertex2i(i, yMin-1);
        glVertex2i(i, yMax );
    }
    for i=yMin-1;i<=yMax;i--      * horizontal *
    {
        glVertex2i(xMin-1,i );
        glVertex2i(xMax-1,i );
    }
    glEnd ;
}

```

```

static void make_ball void
{
    GLfloat a, b;
    GLfloat da = 18.0, db = 18.0;
    GLfloat x, y, c;
    GLuint color;

```

```

    glShadeModel GL_FLAT ;
    color = 0;
    for a=-90.0;a-da<=90.0;a+=da:
    {
        color = 0;

```

```

glBegin( GL_QUAD_STRIP );
for (b=0.0;b<=360.0;b+=db)
{
    if color
        glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, blue );
    else
        glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, white );

    x = COS(b) * COS(a);      * ball radius is 1 *
    z = SIN(b) * COS(a);
    y = SIN(a);
    glVertex3f( x, y, z );

    * This is for lighting, (x,y,z) is normal for ball *
    glNormal3f( x, y, z );

    x = COS(b) * COS(a-da);  * ball radius is 1 *
    z = SIN(b) * COS(a-da);
    y = SIN(a-da);
    glVertex3f( x, y, z );

    color = 1-color;
}
glEnd();

```

```

.....
* Draw a gear wheel
.....

```

```

static void gear( GLfloat inner_radius, GLfloat outer_radius, GLfloat width,
                 GLint teeth, GLfloat tooth_depth )

```

```

{
    GLint i;
    GLfloat angle, delta_angle;
    GLfloat r0, r1, r2;
    GLfloat u, v, len;

    r0 = inner_radius;
    r1 = outer_radius - tooth_depth;
    r2 = outer_radius;

    delta_angle = 2.0*PI / teeth * 4.0;

    glShadeModel( GL_FLAT );

    glNormal3f( 0.0, 0.0, 1.0 );

    * draw front face *

```

```

glBegin( GL_QUAD_STRIP );
for ( i=0; i<teeth; i++) {
    angle = i * 2.0*PI / teeth;
    glVertex3f( r0*cos( angle ), r0*sin( angle ), width*0.5 );
    glVertex3f( r1*cos( angle ), r1*sin( angle ), width*0.5 );
    glVertex3f( r0*cos( angle ), r0*sin( angle ), width*0.5 );
    glVertex3f( r1*cos( angle-3*delta_angle ),
                r1*sin( angle-3*delta_angle ), width*0.5 );
}
glEnd();

/* draw front sides of teeth */
glBegin( GL_QUADS );
delta_angle = 2.0*PI / teeth * 4.0;
for ( i=0; i<teeth; i++) {
    angle = i * 2.0*PI / teeth;

    glVertex3f( r1*cos( angle ), r1*sin( angle ), width*0.5 );
    glVertex3f( r2*cos( angle+delta_angle ),
                r2*sin( angle+delta_angle ), width*0.5 );
    glVertex3f( r2*cos( angle+2*delta_angle ),
                r2*sin( angle+2*delta_angle ), width*0.5 );
    glVertex3f( r1*cos( angle-3*delta_angle ),
                r1*sin( angle-3*delta_angle ), width*0.5 );
}
glEnd();

glNormal3f( 0.0, 0.0, -1.0 );

/* draw back face */
glBegin( GL_QUAD_STRIP );
for ( i=0; i<teeth; i++) {
    angle = i * 2.0*PI / teeth;
    glVertex3f( r1*cos( angle ), r1*sin( angle ), -width*0.5 );
    glVertex3f( r0*cos( angle ), r0*sin( angle ), -width*0.5 );
    glVertex3f( r1*cos( angle+3*delta_angle ),
                r1*sin( angle+3*delta_angle ), -width*0.5 );
    glVertex3f( r0*cos( angle ), r0*sin( angle ), -width*0.5 );
}
glEnd();

/* draw back sides of teeth */
glBegin( GL_QUADS );
delta_angle = 2.0*PI / teeth * 4.0;
for ( i=0; i<teeth; i++) {
    angle = i * 2.0*PI / teeth;

    glVertex3f( r1*cos( angle-3*delta_angle ),
                r1*sin( angle-3*delta_angle ), -width*0.5 );
    glVertex3f( r2*cos( angle-2*delta_angle ),
                r2*sin( angle-2*delta_angle ), -width*0.5 );
    glVertex3f( r2*cos( angle-delta_angle ),
                r2*sin( angle-delta_angle ), -width*0.5 );
}

```

```

    glVertex3f( r1*cos(angle) , r1*sin(angle) , -width*0.5 );
}
glEnd();

/* draw outward faces of teeth */
glBegin( GL_QUAD_STRIP );
for( i=0;i<teeth;i++ ) {
    angle = i * 2.0*PI / teeth;

    glVertex3f( r1*cos(angle) , r1*sin(angle) , width*0.5 );
    glVertex3f( r1*cos(angle) , r1*sin(angle) , -width*0.5 );

    u = r2*cos(angle+delta_angle) - r1*cos(angle);
    v = r2*sin(angle+delta_angle) - r1*sin(angle);
    glNormal3f( v, -u, 0.0 );
    glVertex3f( r2*cos(angle-delta_angle) ,
                r2*sin(angle-delta_angle) , width*0.5 );
    glVertex3f( r2*cos(angle+delta_angle) ,
                r2*sin(angle-delta_angle) , -width*0.5 );
    glNormal3f( cos(angle) , sin(angle) , 0.0 );
    glVertex3f( r2*cos(angle+2*delta_angle) ,
                r2*sin(angle+2*delta_angle) , width*0.5 );
    glVertex3f( r2*cos(angle-2*delta_angle) ,
                r2*sin(angle-2*delta_angle) , -width*0.5 );

    u = r1*cos(angle+3*delta_angle) - r2*cos(angle+2*delta_angle);
    v = r1*sin(angle+3*delta_angle) - r2*sin(angle+2*delta_angle);
    glNormal3f( v, -u, 0.0 );
    glVertex3f( r1*cos(angle-3*delta_angle) ,
                r1*sin(angle-3*delta_angle) , width*0.5 );
    glVertex3f( r1*cos(angle-3*delta_angle) ,
                r1*sin(angle-3*delta_angle) , -width*0.5 );
    glNormal3f( cos(angle) , sin(angle) , 0.0 );
}

glVertex3f( r1*cos(0) , r1*sin(0) , width*0.5 );
glVertex3f( r1*cos(0) , r1*sin(0) , -width*0.5 );

glEnd();

glShadeModel( GL_SMOOTH );

/* draw inside radius cylinder */
glBegin( GL_QUAD_STRIP );
for( i=0;i<teeth;i++ ) {
    angle = i * 2.0*PI / teeth;
    glNormal3f( -cos(angle) , -sin(angle) , 0.0 );
    glVertex3f( r0*cos(angle) , r0*sin(angle) , -width*0.5 );
    glVertex3f( r0*cos(angle) , r0*sin(angle) , width*0.5 );
}
glEnd();
}

```

```

static void draw(void)
{
    GLint i, j;

    count--;
    if (count==limit)
        tkQuit();

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(view_rotx, 1.0, 0.0, 0.0);
    glRotatef(view_roty, 0.0, 1.0, 0.0);
    glRotatef(view_rotn, 0.0, 0.0, 1.0);

    glPushMatrix(); /* red gear */
    glTranslatef(xMax - 1.5 * gear_width, 0.0, 0.0);
    glRotatef(90, 0.0, 1.0, 0.0);
    glRotatef(motion_angle, 0.0, 0.0, 1.0);
    glCallList(gear1);
    glPopMatrix();

    glPushMatrix(); /* lines (bottom) */
    glTranslatef(0.0, yMin-1, 0.0);
    glRotatef(90, 1.0, 0.0, 0.0);
    glCallList(lines);
    glPopMatrix();

    glPushMatrix(); /* lines (left) */
    glTranslatef(xMin-1, 0.0, 0.0);
    glRotatef(90, 0.0, 1.0, 0.0);
    glCallList(lines);
    glPopMatrix();

    glPushMatrix(); /* ball */
    glTranslatef(xTranslate, yTranslate, 0.0);
    glRotatef(yRotate, 0.0, 1.0, 0.0);
    glRotatef(60.0, 1.0, 0.0, 0.0);
    glCallList(ball);
    glPopMatrix();

    glPopMatrix();

    glFlush();
    tkSwapBuffers();
}

static void idle(void)
{

```

```

motion_angle += 3.0;

yTranslate -= delta_h0 - mg_displacement;
mg_displacement -= GTT;
if (yTranslate < yMin) * touch the floor *
{
    mg_displacement = GTT;
    yTranslate = yMin;
}

xTranslate -= delta_s;
if (xTranslate >= xMax) * touch the gear *
{
    xTranslate = xMax;
    delta_s = -delta_s;
    delta_w = w * yTranslate - outer_r - yMin;
}
if (xTranslate <= xMin) * touch the broad *
{
    xTranslate = xMin;
    delta_s = -delta_s;
}
yRotate -= delta_w;

* change view angle *
static GLenum key int k, GLenum mask;

switch (k) {
    case TK_UP:
        view_rotx -= 5.0;
        return GL_TRUE;
    case TK_DOWN:
        view_rotx += 5.0;
        return GL_TRUE;
    case TK_LEFT:
        view_roty -= 5.0;
        return GL_TRUE;
    case TK_RIGHT:
        view_roty += 5.0;
        return GL_TRUE;
    case TK_C:
        view_rotc += 5.0;
        return GL_TRUE;
    case TK_I:
        view_rotc -= 5.0;
        return GL_TRUE;
    case TK_ESCAPE:
        tkQuit();
}
return GL_FALSE;

```



```

    * new window size or exposure *
static void reshape (int width, int height)
{
    GLfloat h = (GLfloat) height / (GLfloat) width;

    glViewport (0, 0, (GLint) width, (GLint) height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-8.0, 8.0, -8.0, 8.0, -8.0, 8.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

static void init(void)
{
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);
    glShadeModel(GL_FLAT);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);

    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
    glEnable(GL_CULL_FACE);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_NORMALIZE);

    * make the lines *
    lines = glGenLists(1);
    glNewList(lines, GL_COMPILE);
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, yellow);
    make_lines();
    glEndList();

    * make the gears *
    gear1 = glGenLists(1);
    glNewList(gear1, GL_COMPILE);
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, red);
    gear(inner_r, outer_r, gear_width, teeth_no, tooth_depth);
    glEndList();

    glShadeModel(GL_FLAT);

    * make the ball *
    ball = glGenLists(1);
    glNewList(ball, GL_COMPILE);

```

```

    make_ball();
    glEndList();
};

main int argc, char *argv[]
{
    tkInitPosition 400, 0, 300, 300 ;
    tkInitDisplayMode TK_RGB TK_DEPTH TK_DOUBLE ;

    if tkInitWindow "Rigid Bodies" == GL_FALSE {
        tkQuit ;

        init();
        tkExposeFunc reshape ;
        tkReshapeFunc reshape ;
        tkKeyDownFunc key ;
        tkIdleFunc idle ;
        tkDisplayFunc draw ;
        tkExec ;
    }
};

```

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved