CMSC 23700 **Introduction to Computer Graphics** **Project 0**
Fall 2005 **September 28**

A simple OpenGL viewer

Due: Friday, October 14 at 9pm

1 Summary

For this project, you will have to implement a simple viewer in **C** using OpenGL and the GLUT library. Chapters 1–6 of the "*OpenGL Primer*" cover the aspects of the OpenGL and GLUT that are relevant to this project.

The goals of this project are to get your feet wet with simple graphics programming and to give you some quick feedback on the course submission and grading policies.

2 Description

Your task in this project is to implement an interactive viewer of a simple 3D animation. The animation is of a $2 \times 2 \times 2$ box centered at the origin, which contains one or more bouncing spheres.

The viewer is located at $(0, 0, -3)$ (this is called the *camera* or *eye* position) and is looking in the direction of the positive Z-axis. We recommend a field of view of around 65 degrees. Your program is responsible for displaying the animation and for allowing the user to vary change its state using the keyboard.

2.1 The graphics

There are three components to the rendering:

The box. The box is $2 \times 2 \times 2$ units in size and is centered at $(0, 0, 0)$. You should render five sides: back ($z = 1$), left ($x = -1$), right ($x = 1$), bottom ($y = -1$), and top ($y = 1$). You can render the sides as quads, but you may want to tessellate them to get better lighting effects. We recommend giving the walls different colors.

The balls. The balls have radius $r = 0.05$. Use the function `glutSolidSphere` to render them. When adding a ball to the mix, pick a random location, initial velocity, and color. The initial speed (length the the velocity vector) should be in the range 0.5 to 2.5. Use smooth shading to render the balls.

The lights. You should implement multiple lighting modes (ambient only, directional light, positional light, and a spot light). The space key is used to switch between lighting modes. Experiment with the placement of the lights.

2.2 User interface

Your viewer should support the following keyboard commands:

`l`	toggle the lighting mode between none, directional, point, and spotlight.
`+`	add a ball to the simulation
`-`	remove a ball from the simulation
`q`	quit the viewer

You can limit the number of balls to 100. Your application should also handle resizing the viewport.

3 Hints

We recommend the staging your implementation effort in the following steps:

1. Start by getting the box to display and getting features such as resizing and quiting to work.
2. Then animate a single bouncing ball and get the lighting modes to work.
3. Finally, add multiple balls and ball-to-ball collisions.

3.1 Representing the balls

The position of a ball can be represented by three pieces of information: a time t_0, an initial position $\mathbf{q}$, and a velocity vector $\mathbf{v}$. Given these vlaues, the ball's position can be defined as a function of time

$$\mathbf{p}(t) = \mathbf{q} + (t - t_0)\mathbf{v} \quad \text{for } t \geq t_0$$

In addition, you will want to record the ball's color information and the time of its next collision.

3.2 Event queue

To manage the simulation, you will need to maintain a priority queue of events ordered by time. There are two types of events: ball-wall collisions and ball-ball collisions. A given ball will continue moving in its current direction until it hits something, so the only time you need to update the state of the system is when an event occurs. Because the objects and their motions are simple, it is possible to accurately predict what the next event to affect a given ball is likely to be.

If we are going to render the scene at time t, then we should first process all of the events that occur at or before t. To process a wall collision, we compute the ball's new velocity ($\mathbf{v}$), set its initial time value (t_0) to the collision time, and set its initial position ($\mathbf{q}$) to the collision position. We then compute the ball's next collision by testing it against the walls and other balls. Note that if the ball is going to collide with some other ball, you may have to recompute other collisions. For example, say that ball B was scheduled to hit ball C, but we now discover that ball A will hit B first. This means that we must recompute C's next collision (and so on).

3.3 Ball-wall collisions

Given a ball with position

$$\mathbf{p}(t) = \mathbf{q} + t\mathbf{v}$$

the time of intersection with the plane defining a given wall can be determined by projecting out the appropriate component of the velocity vector $\mathbf{v}$. For example, if $\mathbf{v}_x > 0$,[1] then the ball will hit the right side of the box ($x = 1$) at time t satisfying

$$\begin{aligned}(1 - r) &= \mathbf{p}(t)_x \\ &= \mathbf{q}_x + t\mathbf{v}_x\end{aligned}$$

thus the time of intersection with the right wall is

$$t = \frac{1 - r - \mathbf{q}_x}{\mathbf{v}_x}$$

If $\mathbf{v}_x < 0$, then we check it against the left wall ($x = -1$ plane). For each ball, we must test it against at most three planes and then take the minimum time as the time of collision.

When a ball hits a wall, its new velocity vector will be a reflection of its current vector off the plane of the wall.[2] Figure 1 illustrates this situation.

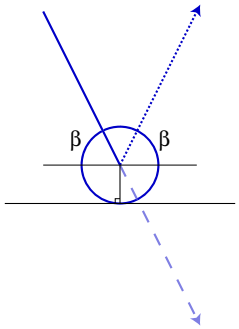

Figure 1: Ball-to-wall collision

3.4 Ball-ball collisions

Consider two balls, whose positions are specified as

$$\begin{aligned}\mathbf{p}_1(t) &= \mathbf{q}_1 + t\mathbf{v}_1 \\ \mathbf{p}_2(t) &= \mathbf{q}_2 + t\mathbf{v}_2\end{aligned}$$

The distance between the centers of these two balls is

$$d(t) = \sqrt{(\mathbf{p}_1(t) - \mathbf{p}_2(t))^2}$$

[1] In practice, we need to test for $\mathbf{v}_x > \epsilon$, where ϵ is a small positive number (*e.g.*, 10^{-6}).

[2] Question 4 of Homework 1 addresses the computation of the reflection vector.

$$
\begin{aligned}
d(t)^2 &= (\mathbf{p}_1(t) - \mathbf{p}_2(t))^2 \\
&= ((\mathbf{q}_1 + t\mathbf{v}_1) - (\mathbf{q}_2 + t\mathbf{v}_2))^2 \\
&= ((\mathbf{q}_1 - \mathbf{q}_2) + t(\mathbf{v}_1 - \mathbf{v}_2))^2 \\
&= (\mathbf{q} + t\mathbf{v})^2 \\
&= \mathbf{q}^2 + (\mathbf{q} \cdot \mathbf{v})t + \mathbf{v}^2 t^2
\end{aligned}
$$

where $\mathbf{q} = (\mathbf{q}_1 - \mathbf{q}_2)$ and $\mathbf{v} = (\mathbf{v}_1 - \mathbf{v}_2)$. When $d(t) \leq 2r$, then the balls have collided, so we can detect the point of collision by finding the roots of

$$\mathbf{v}^2 t^2 + (\mathbf{q} \cdot \mathbf{v})t + (\mathbf{q}^2 - 4r^2)$$

If there are no real roots or both roots are negative, then there is no collision, if both roots are positive, then the smaller one is the time of collision, and if one root is positive and the other negative, then the balls are currently overlapping (which should not happen in your simulation).

When two balls collide, their direction vector should change as follows. First compute the plane at the point of contact that is tangent to both balls. Then, the new velocity vector is the reflection of the old vector off of the tangent plane. Figure 2 illustrates this situation.

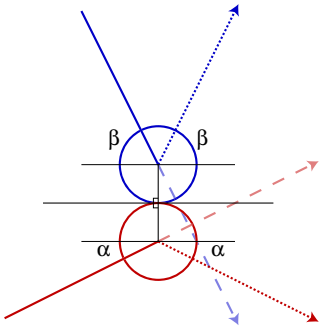

Figure 2: Ball-to-ball collision

4 Submission

We will set up a **gforge** repository for each student on the Computer Science server (see the *Lab notes* (Handout 2) for more information on **gforge**). We will collect the projects at 9pm on Friday October 14th from the repositories, so make sure that you have committed your final version before then.

You will also be expected to give a preliminary "demo" your code during Lab in Week 2 (October 5th) and to give another demo during Lab in Week 3 (October 12th).