COP290
Assignment 1 : Bubble Screensaver

Manoj Kataria Rohit Harishchandra Patidar

January 25, 2015

Contents

Chapter 1

Introduction

In this assignment, we were supposed to create a simple screen saver application. The application displays N balls on the screen. Each ball starts from a random position on the screen, and starts moving in a random direction. The collision of a ball with another ball, or the wall of the screen must be simulated using perfect reflection (no friction). Each ball on the screen is simulated by one thread. The collision between two balls must be simulated using messages between their respective threads. The speed of each ball can be increased/decreased by the user.

Chapter 2

Overall Design

Given problem is broken into following components.

- GUI
- Collisions
- User Interface Controls
- Multithreading
- Utility Functions

These things are discussed in detail in the following chapters.

Chapter 3

Design Summary

The entire application is divided into multiple files for different components and functions.

Ball.h/.cpp: Define Ball

MyDefines.h: All #define directives defined here.

MyEnum.h: Enumeration utilities defined here

global.h: Structures Color and ThreeD defined here

imageload.h/.cpp: Contains Image class for BMP image.

Particle.h/.cpp: Contains class for snow particles.

Theme.h/.cpp: Contains Theme class

themeReader.h: Contains method to read themes.txt

Wall.h/.cpp: Contains wall class

UtilityFunctions.h/.cpp: Contains methods like Add/Remove ball etc.

GUI.h/.cpp: Contains all glut methods.

SubMenu.h/.cpp: Contains UI control methods.

Collision.h/.cpp: Contains method to handle collsions.

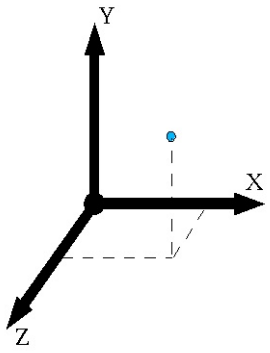
Chapter 4

GUI

4.1 Libraries included

- GL/freeglut.h
- GL/glui.h

4.2 Camera Settings

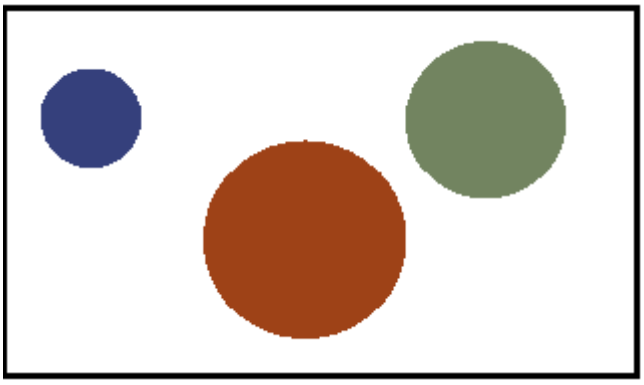


As positive z-axis points out of the screen and (0,0,0) lies in the plane of screen, therefore objects drawn on the screen are not visible. So firstly we shift the origin to (0,0,-zdistance) , where zdistance is shift in origin along negative z-axis and set camera angle to 90 degrees in y-direction.

4.3 Resizing

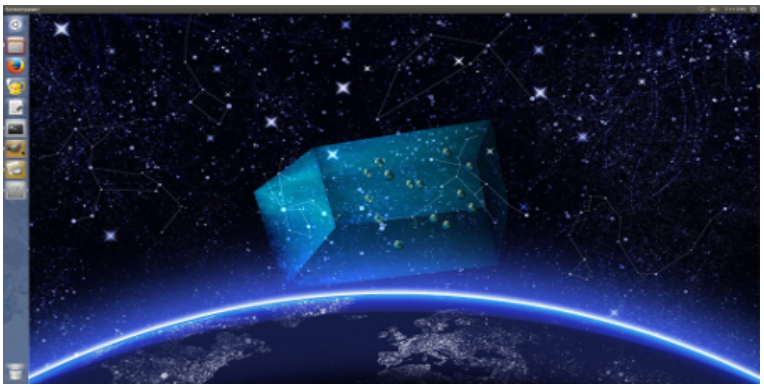
On reducing size of the window too much, at some stage the space required by balls is more than available window space. To avoid that we have set a limit on the size of the window. On reducing size below the limit, the view size does not decrease further.

4.4 Drawing Balls



We have set random radii of balls (*in some particular range*) and given initial velocities in x and y directions such that no two balls overlap each other. To avoid overlapping while entering, we have used concept of timer such that new ball enters only when previous ball is far enough to avoid overlapping with this new ball. Each ball has a mass proportional to $(radius)^2$ assuming balls as disc in 2-Dimensional case (*In case of 3d, mass is proportional to* $(radius)^3$). Also the color of balls are randomly generated. To display balls on the screen after regular interval of time *glutDisplayFunc()* and *glutTimer-Func()* are used. The number of balls are entered by the user otherwise default number of balls are displayed on the screen.

4.5 3D-View



The application allows user to switch between 2D and 3D model. 3D model is the extension of 2D model. We have use Perspective model for both 3D and 2D model. In addition to x-y, spheres have also given velocities in z-direction. Following are the methods used to draw 3D-box and Spheres:

- glutSolidSphere
- drawBox

4.6 Textures

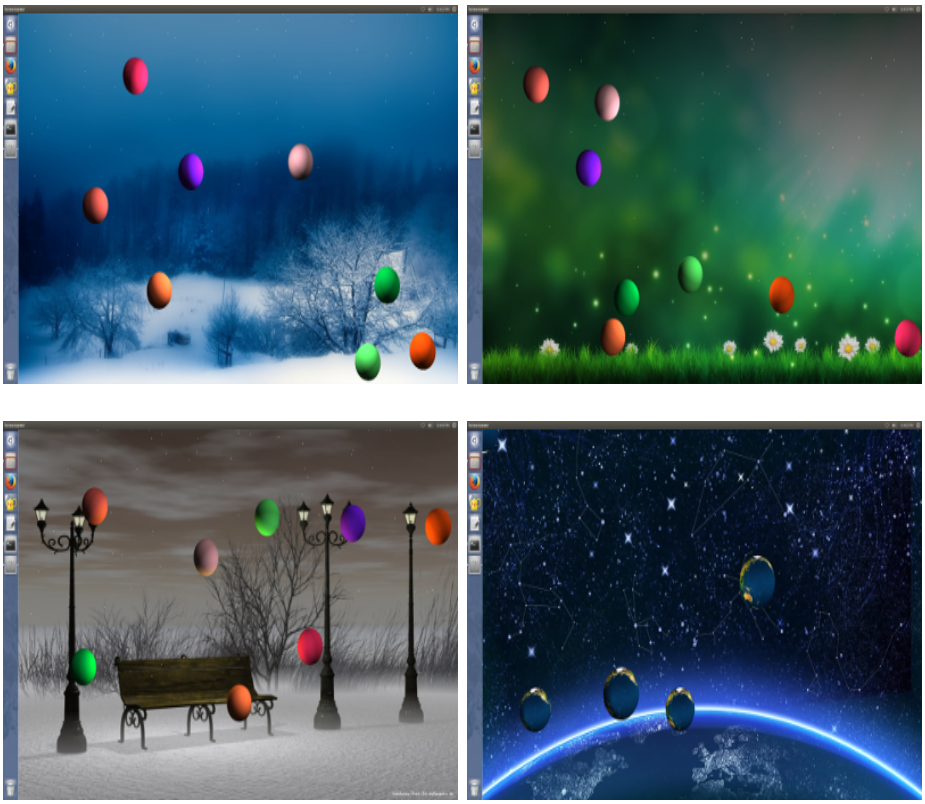
To use textures in the application , we have defined class Image as following:
Class Image:

Member Variables:

- Data, Width , Height

In order to load BMP image we build method loadImage(). This method read BMP file and load it as Image. We have used these textures on spheres, faces of cube, backgrounds in the themes.

4.7 Themes



We have made our application cooler by adding various themes to it. Each theme has different light sources and background. Light sources will be enabled/disabled depending on the theme. Theme class is defined as following:
Class Image:

Member Variables:

- Background Color
- Image
- Light Sources

The information about the colors and positions of light sources, backgrounds are stored in the text file. Theme reader method reads this information and provide it to the the theme class.

Chapter 5

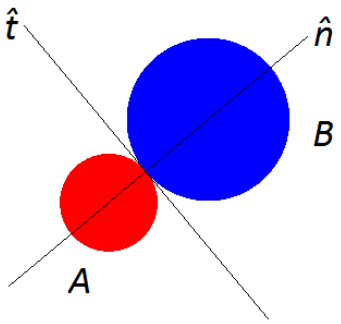
Collisions

5.1 Ball to ball collision

Condition for collision

The distance between centers of two balls should be less than or equal to sum of their radii and they should be approaching each other.

Collision Mechanism



(x_1, y_1) = Center of ball A
(x_2, y_2) = Center of ball B
m_1 = Mass of ball A
m_2 = Mass of ball B
$\vec{u_1} = u_{x_1}\hat{i} + u_{y_1}\hat{j}$, Initial velocity of ball A
$\vec{u_2} = u_{x_2}\hat{i} + u_{y_2}\hat{j}$, Initial velocity of ball B
$\vec{v_1} = v_{x_1}\hat{i} + v_{y_1}\hat{j}$, Initial velocity of ball A
$\vec{v_2} = v_{x_2}\ \hat{i} + v_{y_2}\ \hat{j}$, Initial velocity of ball B

$x = x_2 - x_1$
$y = y_2 - y_1$

$\hat{t} = \frac{y\hat{i} - x\hat{j}}{\sqrt{x^2+y^2}}$ Unit vector normal to line of Impulse

$\hat{n} = \frac{x\hat{i} + y\hat{j}}{\sqrt{x^2+y^2}}$ Unit vector along the line of impulse

When the two balls collide elastically, the change in their velocities is only due to change along $\hat{n}$ while velocities along $\hat{t}$ remain same i.e.

(1) $$v_1\hat{t} = u_1\hat{t}$$

Using conservation of momentum along the $\hat{n}$

(2) $$m_1\vec{u_1}\cdot\hat{n} + m_2\vec{u_2}\cdot\hat{n} = m_1\vec{v_1}\cdot\hat{n} + m_2\vec{v_2}\cdot\hat{n}$$

Using coefficient of restitution

(3) $$\text{e} = \frac{Velocity\ of\ approach}{Velocity\ of\ separation}$$

As the collision is elastic, therefore e = 1.

Velocity of approach = Velocity of separtion i.e. $\vec{u_1}\cdot\hat{n} - \vec{u_2}\cdot\hat{n} = -\vec{v_1}\cdot\hat{n} + \vec{v_2}\cdot\hat{n}$

Using (2) and (3)

(4) $$\vec{v_1}\cdot\hat{n} = \frac{2*m_2u_2\cdot\hat{n} + (m_1 - m_2)u_1\cdot\hat{n}}{m_1+m_2}$$

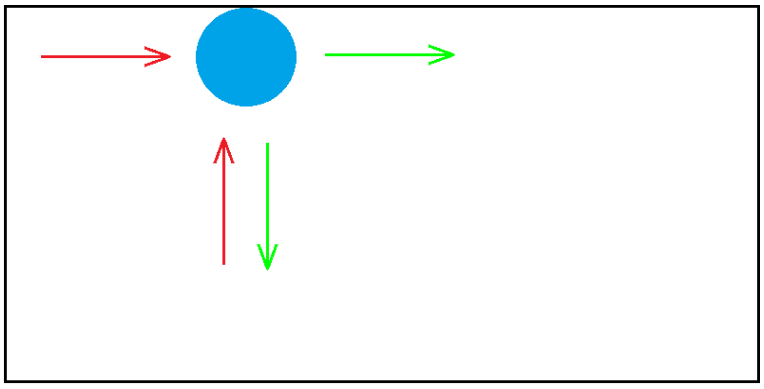
Using (1) and (4)

$$v_{x_1} = \frac{y*(u_{x_1}y - u_{y_1}x)(m_1+m_2) + x*[(m_1-m_2)(v_{x_1}x + v_{y_1}y) + 2m_2(v_{x_2}x + v_{y_2}y)]}{(x^2+y^2)(m_1+m_2)}$$

$$v_{y_1} = \frac{x*(u_{y_1}x - u_{x_1}y)(m_1+m_2) + y*[(m_1-m_2)(v_{y_1}y + v_{x_1}x) + 2m_2(v_{y_2}y + v_{x_2}x)]}{(x^2+y^2)(m_1+m_2)}$$

For ball B replace x by -x and y by -y.

5.2 Ball to wall collision



Collision with Left/Right wall:
$v_x = -u_x$

Collision with Upper/Lower wall:
$v_y = -u_y$

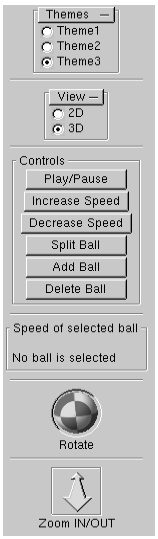
In case of 3D, the above formula can be extended accordingly.

Chapter 6

User Interface Controls

6.1 Menu controls

To use various features of the application we have created GLUI window. This window has following controls in it:



- Themes
- Views (2D/3D)
- Increase/Decrease Speed
- Display speed of the selected ball
- Add a ball

- Remove the selected ball
- Split the selected ball
- Rotate/Translate 3D box

6.2 Key Events:

User can control motion of selected ball using key buttons.

w Key : Increase speed in positive-y direction
s Key : Increase speed in negative-y direction
a Key : Increase speed in negative-x direction
d Key : Increase speed in positive-x direction
Esc Key : Exit the application

6.3 Mouse Events:

To select the particular ball user can click on it using the mouse. On clicking the border appears on the selected ball. After selection user can use features provided in the control menu.

Chapter 7

Multithreading

7.1 N threads managing N balls

Method I

Balls are maintained in a global vector. Each of the N threads checks for possible collisions of a *ball* (ID of which is passed to it as argument) with all the other balls. Each thread keeps track of change in velocity of *ball* in a variable *dv*(member variable of Ball Class). Threads do not change/set the velocities or any other attribute of *ball* passed to it. All changes are made outside the threads (in main or any other update function using dv of ball) after all threads are finished (this is accomplished by joining all threads to main thread). Since threads just check for possible collisions of balls in parallel and no changes are made to global vector of balls from inside the threads, mutexes are not required for this purpose.

Method II

In this method, we define our own thread class **MyThread**

Class **MyThread** {
    Ball **B**;
    queue<*Ball> $*q$;

}

We have a vector *threads* of MyThread objects which is in global scope and is initially empty. When a new ball is required, a MyThread object is created and pushed into the vector.

At each *update()* call , *addWorkItems()* method is called which adds to the queue of each thread,pointers to ball object of other threads.
All the threads then run in parallel, successively checking collisions with others balls in its queue and removing them after updating their velocities accordingly.
Each Ball has a mutex associated with it, which is locked whenever accessing/setting its attributes thereby in synchronization with other threads.

Chapter 8

Testing Components

8.1 Collision

- To verify that collisions between the balls are proper, we output the kinetic energy of system in a log file. Since the collisions are elastic therefore kinetic energy of the system will remain conserved at all time. (unless user increases or decreases the speed of any ball)
- To test collision of a ball with wall, set velocity only in one direction(x or y) and then check with a random velocity.
- To test collision of a ball with another ball, first test head on collision in both x and y direction and check collisions at different angles. Using GDB and setting breakpoints at the time of collision we can check whether the balls are overlapping or not.
- To test collision of multiple balls simultaneously, although a rare event in random case, take 3 balls at vertices of a equilateral triangle and give them velocities toward the centroid of triangle.

8.2 Multithreading

Whether all the threads are running or not can be tested in GDB by setting breakpoints at different function calls.