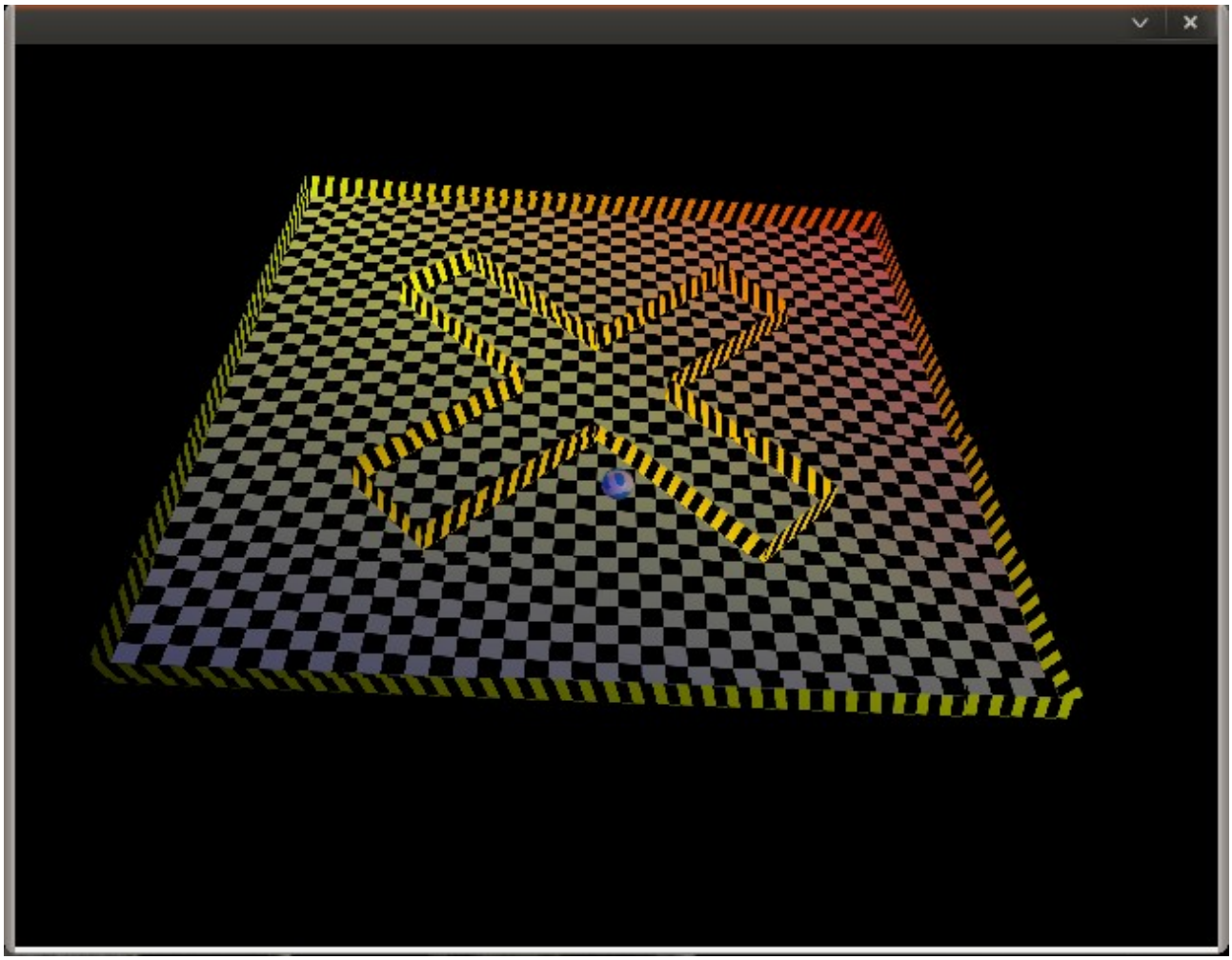
Comp323 Project:

Ball in a Maze

User Manual and Documentation

Byron Miles 220057347



User Manual

Compiling and Running:

A makefile is included for easy compiling. Simply navigate to the directory containing the project and makefile (if you are not already there) and type 'make'.

By default the makefile will compile the project to a binary called 'game', simply type './game' to run it.

Note: You will need a version of OpenGL and a graphics card that supports Vertex Buffer Objects (VBOs) as well as the SDL Library(http://www.libsdl.org/) in order to compile / run the project.

To close the program simply click the 'X' in the upper-right corner, just like closing any other window.

Controls:

The W, A, S, D keys are used to tilt the maze forward, back, left and right. The ball rolls based on the tilt of the maze and will reflect off any barriers it hits.

The camera follows the ball and zooms in and out based on the velocity of the ball.

Gameplay:

Unfortunately the project turned into more of a tech-demo, so there isn't a lot of gameplay to speak of, but have fun rolling the ball around and watching it bump into the barriers.

Design Documentation

List of Main Classes:

Ball – The ball. Holds the position, velocity, radius and frame rotation for ball. Also, since it is the only thing affected, it is responsible for collision resolution.

BarrierY – A Y-axis aligned barrier. A barrier is a height and width limited plane. Also calculates and stores the 'model' and texture co-ordinates for the barrier.

Core – Handles windowing, OpenGL and SDL settings, and contains the main game loop. Also holds the primary timer, and passes events to the Game instance.

Core_Timer – Uses SDL to keep time to the millisecond. Can be instantiated to create timers for various different things.

Game – Where all the main game logic, structures etc. are located.

- Game_onEvent – Handles events passed to the game such as key presses.
- Game_onInit – Sets the game up, creating the required objects and giving them initial values. Also sets up things like OpenGL lights.
- Game_onLoop – Handles all the loop logic such as calling the update methods of objects, held down keys (such as for tilting the maze), and the collision detection logic.
- Game_onRender – Where all the rendering is done. Uses various parameters defined for the game and within objects to render the game world.

List of Utility Classes:

Collision Detection – Really just a set of methods, it provides tests for predicting and detecting the intersection of various objects, such as the ball and a barrier.

Constants.h – Holds a bunch of constants such as PI, multipliers for conversion from degrees to radians / radians to degrees, as well as some for physics such as gravity and coefficients.

Plane – Defines a plane as a Position, Normal and D value. Calculates the D value from the cross product of the position and the normal.

Quaternion – Defines a quaternion (w, x, y, z) and methods to set it to a certain rotation as well as the quaternion cross product (*) to concatenate rotations. Used to animate the ball.

Targa – Used to read and load targa (.tga) images for textures.

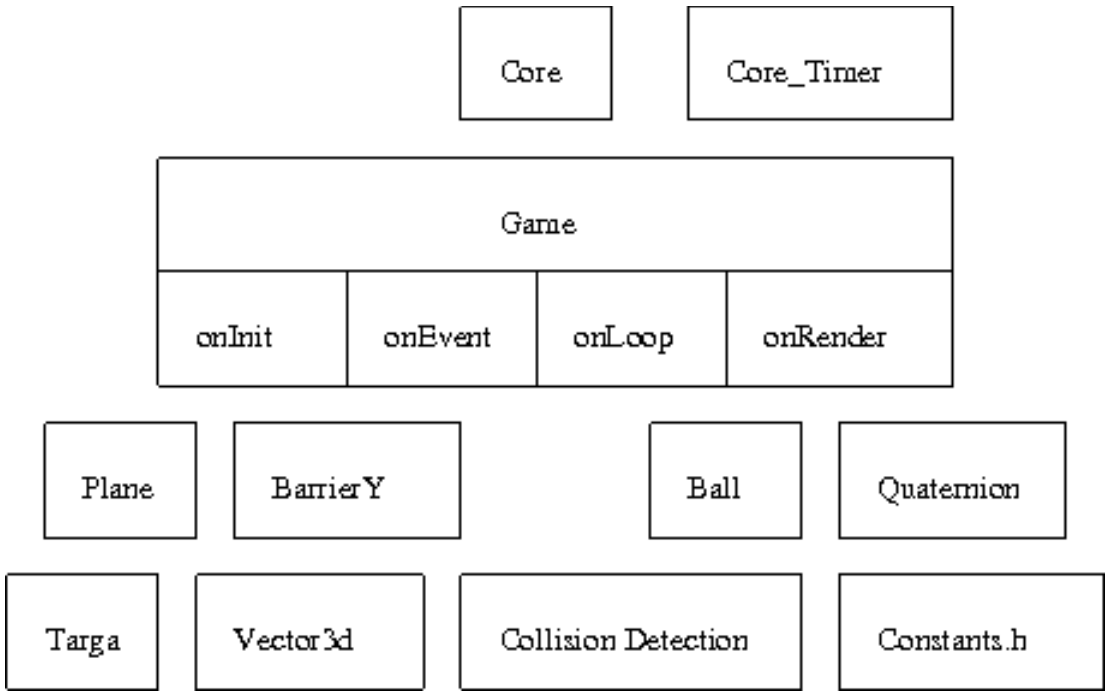
Vector3d – Defines a 3d vector (x, y, z) and provides various operations including addition, subtraction, multiplication and division by a scaler, the dot and cross products and normalisation. Also provides methods for getting information from vectors such as it's unit vector, magnitude, and the distance between two points.

Others of note:

Collision Points – While not a class in themselves (they are just a list of Vector3d's in the Game class), they are treated as 'entities' in their own right. Basically they are placed at corners to stop the ball passing through the edge of a barrier.

Architecture:

Diagram



Description

At the top level is the Core, which controls the main loop and and has an instance of the Game class. The Game class is divided into several sections which handle different aspects of the game; onInit handles setup, onEvent handles events and key presses, onLoop handles updating and collision detection logic, onRender handles rendering.

The Game object holds all the game assets, however this is not particularly effective as it gets messy very quickly and some sort of resource manager will need to be implemented if this is to be expanded beyond it's current state. The onRender method of the Game class handles all the rendering so as to keep OpenGL out of the lower classes, which makes them more focused and reusable.

At the next level you have the primary objects of the game, the BarrierYs and the Ball, Plane and Quaternion are also at this level as they are mainly used by the BarrierY and Ball classes respectively, but also used by the Game class and others.

At the lowest level are the more utility classes, Targa for loading textures, Vector3d for all sorts of vector / point related stuff, Collision Detection for providing collision prediction / detection algorithms and Constants to hold a few constants in one place.

The architecture isn't particularly neat, or efficient, but it works and separates things out enough that it's not quite one monolithic blob.

Subsystems:

Collision Detection

Collision detection is done in the onLoop part of the Game class. It uses overlap testing (though I have called the methods intersectXY..), and basically just loops through the list of barriers and collision points to see if the ball is overlapping with any of them. If it is the ball is told to resolve the collision.

The collision is resolved by moving the ball out of whatever it collided with (it is moved along the direction of the normal) and reflecting the balls velocity about normal; in the case of a collision point the normal is the vector from the point to the centre of the ball. The reflection is not perfect and the ball losses some velocity in the process, see the Physics section for more details.

I have implemented a check to insure that the ball does not move more than it's radius per collision detection step. This is to stop the ball passing through a barrier / collision point between frames.

Their are collision detection methods for prediction (intersection testing) as well, and the logic is still there (commented out) in the onLoop part of the Game class. It should still work, but I have not used it as I had issues with the ball passing through collision points by resting against them and building up velocity via acceleration due to gravity. You can uncomment it and comment out the overlap testing if you want to see it in action.

Physics

I have not used a physics subsystems as such, but rather use general physics for things such as the balls' acceleration due to gravity, friction and collision resolution.

The acceleration is simply based on a gravity constant (9.81 by default) and the angle (tilt) of the maze at the time.

Similarly friction is simply a reducing of the balls' velocity towards 0 based on a constant and a friction algorithm.

Collision resolution reflects the balls velocity about a normal, but it doesn't do it perfectly and the ball losses some velocity based on the relative direction of it's velocity and the normal. This is controlled by a constant (coefficient of restitution) in Constants.h.

Animation

Like physics I do not use an animation system as such, but rather just rotate the ball based on it's movement that frame and the accumulation of those movements over time. I use two quaternions to achieve this. The Ball class holds one, which is set to the rotation of the ball based on it's circumference and movement that frame and the Game class holds the other, which is multiplied with the balls' each frame to accumulate the rotations over time.