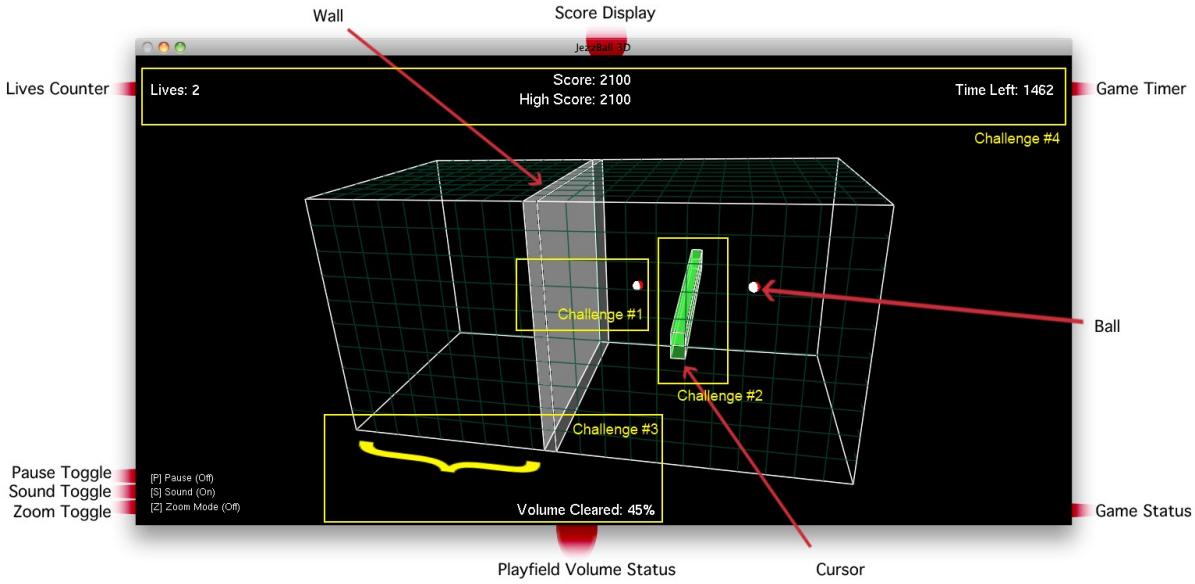
Team Members: Erik Gui (cs184-cx) and Alex Nisnevich (cs184-dp)
SID: 21178740 and 21323554
Email: erik.gui@berkeley.edu and alex.nisnevich@berkeley.edu

CS184 Final Project: JezzBall 3D

1 Project Description

In this project, we have successfully added a new dimension to the classic video game JezzBall. Instead of using a plane as the field of play, we created a rectangular prism for the player to partition by drawing prismatic walls. Each level is completed when the player partitions the playfield so that 80% of the volume is unreachable by the red-and-white colored balls. All of the features of the original JezzBall, including UI, sound effects, and control scheme, are extended in our JezzBall 3D. While we have maintained the retro feel of the game, a variety of challenges arose with adding another dimension. Below we describe the four main challenges that we faced in creating our project.

You can watch a gameplay demonstration video at http://www.youtube.com/watch?v=6CefcmQnUEo .



2 Project Challenges

1. **Realistic Collision Physics:** Erik and Alex

   There are three main types of collisions that we simulated in our project. In keeping with the physics in the original JezzBall, all collisions are perfectly elastic, and all objects have constant velocity.

   - **Ball-to-Ball:** This type of collision occurs when two balls collide while wandering about the playfield. To simulate this collision as realistically as possible, we have used a 3D collision detector written by Thomas Smid (http://bit.ly/j0ze4Q) that deals with elastic collision. When a ball-to-ball collision occurs, we modify the velocity vector of the balls in keeping with Smid's algorithm, but then normalize it to preserve constant velocity.
   - **Ball-to-Wall:** This type of collision occurs when a ball hits either a player-constructed wall or the edge of the playfield. Let p_n be a ball's position in frame n and r be the ball's radius. A collision in frame n for a given ball is detected if $p_{n+1} + r$ and $p_{n+1} - r$ fall on different sides of a wall or playfield edge. When a collision is detected, the ball's incident direction with respect to the wall's surface is reversed, maintaining the ball's current velocity.

- **Wall-to-Wall:** In many cases, a player-constructed wall would also collide with other walls that exist in the playfield. This type of collision is essential for partitioning the balls to smaller and smaller volumes. Since walls only expand in two directions at a time in Jezzball, wall-to-wall detection consists of detecting whether one of two rectangles falls within a box. We accomplished this using the `IsLineInBox` function written by *3D Programming Weekly* (`http://bit.ly/jHr08n`), thus turning wall-to-wall detection into a series of eight `IsLineInBox` tests (four for each of two rectangles being tested for each wall). Originally, we ran these these tests every frame, but we found this to be redundant: instead, when a wall is drawn, we temporarily maximize its length, find the closest collision point in each direction, and store this information, thus only having to run collision tests when a wall is instantiated.

2. **Intuitive Control Scheme:** Alex

   Preliminary research for this project showed that all previous 3D implementations of JezzBall that have been made either relied on keyboard-based controls (`http://bit.ly/mvl48O`) or were not fully 3D to begin with (`http://bit.ly/jlZUXX`). Thus, one of our primary goals for this project was to create an intuitive mouse-based control system, which carried with it its own set of challenges, as we needed to come up with a good way of converting 2D mouse coordinates to 3D playfield coordinates. After reaching a number of dead ends, we found out about the `gluUnProject` function, which retrieves the object coordinates of the point that the mouse is currently pointing towards. By creating invisible polygons along the faces of the playfield, we were able to use `gluUnProject` to map mouse coordinates to playfield coordinates, which we could use for obtaining the position of the 3D cursor.

3. **Calculating Unreachable Volume:** Alex

   The winning condition for each level is to clear 80% of the play field - that is, to partition the play field in such a way that at least 80% of it is unreachable to the balls. However, because the playfield can be partitioned into irregular shapes, determining the exact percentage of volume that is unreachable is non-trivial. Originally, our plan was to have the `Box` class subdivide itself over the course of gameplay and keep track of which balls fall within which boxes, but this proved to be too complicated.

   Our solution came when we decided to make the game operate with a discrete grid, like in the original Jezzball game. We divided the box into $40 \cdot 20 \cdot 20 = 8000$ cubic cells, and had the player-made walls snap to the grid. Now that there were only a finite number of cells to work with, it wasn't too hard to set up a simple algorithm for calculating unreachable space. Wall positions are stored in a three-dimensional boolean array, and whenever a wall is completed, a recalulation of space occurs as follows: a "virus" appears in each cell where a ball currently is, and over a series of $40+20+20 = 80$ iterations, the virus spreads between adjacent cells, stopped only by walls. After 80 iterations, whichever cells are not occupied by the "virus" are the unreachable cells.

4. **Simple and Interesting Game UI, Effects, and Sound:** Erik

   The UI of our project follows that of the original game. We displayed a text overlay with gameplay information by setting up an orthogonal projection covering the window, using the `glRasterPos2i` function to set text coordinates, and using the `glutBitmapCharacter` function to print character arrays into those coordinates. As for game effects, the ball is seen as a rotating red and white sphere, which we implemented not through textures but by drawing a white sphere infintesimally close to a red sphere and moving the white sphere in a circular path around the center of the red sphere.

   For game sound, we tried a variety of cross-platform sound libraries, such as OpenAL, GLFW, sfml, and SDL_mixer, but found that they were all rather difficult to integrate into our project and generally unfriendly with GLUT. Instead of a cross-platform solution, we settled on using the Windows `playSound` function in `mmsystem.h`, which was remarkably easy to use. We made sure that sounds would fail gracefully on MacOS X through a preprocessor switch.