Milwaukee School of Engineering
CS421 – Advanced Computer Graphics

Final Project Report

COOLAABER

Marek Handl

February 21, 2007

- ## Project Description

The goal of the project was to create a simple computer game of pool (billiard). The plan was to implement at least two balls on a pool table with pockets. Collision handling was one of the major concerns of the project. Chosen advanced computer graphics techniques were texturing and shadows. It was decided to develop the application for MS Windows environment using C++ language and OpenGL.

- ## Achievements

All planned features were implemented and others were added. A playable game of pool was realized. There is a full set of balls on the table with pockets, there are two players and basic game rules apply. Balls cast shadows on pool table walls by using shadow projection technique. Textures were applied to make the table look more realistic. Simple fog effect makes the bottom part of the table blend into background. A couple of sounds were added to provide also acoustic effects. To be able to use buttons and other widgets Microsoft Foundation Class (MFC) was used.

- ## Source Code Organization

oglMFCDialog – MFC code, that creates application window

oglMFCDialogDlg – MFC code and window events handlers (button etc.)

OpenGLControl – central logic, the game is controlled from here

Ball – class representing one ball – displaying and collision handling is implemented here

Table – class representing pool table – its sizes are defined here, drawing is handled here

Vector – simple class for a two dimensional vector with some vector math

constants.h – constant values used throughout the program

- ## Building

The project was developed using MS Visual Studio .NET 2003. It is necessary to have OpenGL installed. No other software is required.

- **Design**

- documentation generated with Doxygen is available in Documentation folder

- **Algorithms**

Ball and Wall collision

- once the center of the ball is beyond a boundary (which is exactly one radius away from the wall), there must have been a collision
- either x or y part of speed is then turned to negative value
- ball is moved to position where it would actually be if it rebounded in the real world
- pseudo code for collision with a vertical wall (x coordinate is constant for the wall):

```
IF ( position.x - wall < R ) {
        speed.x = - speed.x * wallDeceleration;
        position.x += 2 * (wall + R - position.x)
}
```

- more complicated situation occurs when a ball hits a corner of a pocket, the program handles this situation as if there was a wall under 45° angle – a ball coming in perpendicular direction will be bounced off in direction parallel to the wall

Ball and Ball collision

- once centers of two balls are closer than two radii, there must have been a collision
- balls need to be pulled back to position where they first touched
- to make the computation easier, approximation has been made and the slower ball is considered to be fixed, the faster ball is moved backwards in the direction where it came from
- this point is computed as an intersection of a circle with the center in the slower ball's center and radius equal to 2*R and a ray coming out of position of the faster ball going in opposite direction than the speed vector of the faster ball has

```
// rename variables
a = - faster->speed.x;
b = - faster->speed.y;
c = faster->position.x;
d = faster->position.y;
X2 = slower->position.x;
Y2 = slower->position.y;
// solving quadratic equation
quadratic = a*a + b*b;
linear = 2 * ( a*c - a*X2 + b*d - b*Y2 );
```

```
absolute = -2*(c*X2 + d*Y2) + c*c + d*d + X2*X2 + Y2*Y2 – D*D;
det = sqrt(linear*linear - 4*quadratic*absolute);
result = (-linear + det) / (2*quadratic);
// changing position of the faster ball
faster->position.x = result*a+c;
faster->position.y = result*b+d;
```

- impact consequences on speed vectors are computed and new speeds are assigned to both colliding balls

```
// segment connecting the two centers
X_Axis = (center2 - center1)
X_Axis.normalize();
// projections of speed vectors into connecting axis and perpendicular direction
U1x = X_Axis * (X_Axis dot U1)
U1y = U1 - U1x
U2x = -X_Axis * (-X_Axis dot U2)
U2y = U2 - U2x
// computation based upon momentum, weights of balls are considered too
V1x = ((U1x * M1) + (U2x * M2) - (U1x - U2x) * M2) / (M1 + M2)
V2x = ((U1x * M1) + (U2x * M2) - (U2x - U1x) * M1) / (M1 + M2)
V1y = U1y
V2y = U2y
// new speed vectors
V1 = V1x + V1y
V2 = V2x + V2y
```

- the faster ball is moved to position where it would be, if it actually rebounded

Speed Attenuation

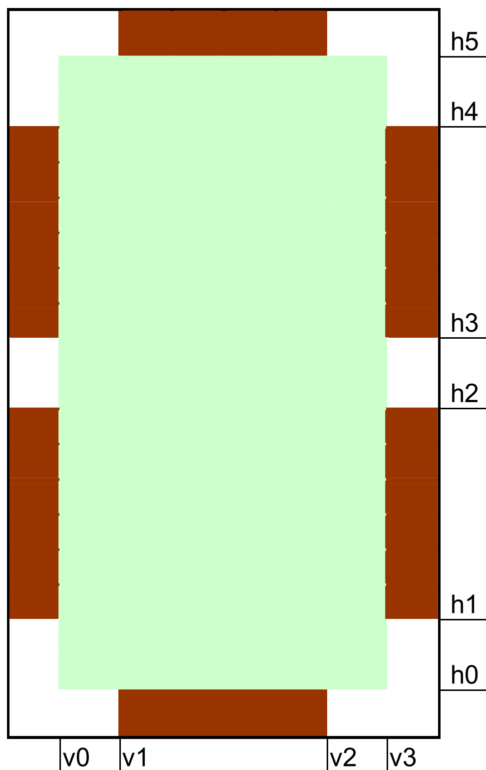
- each time move() method is called the ball loses part of its speed, furthermore speed is decreased by collisions as well
- both x and y component of the speed are multiplied by a deceleration factor
- deceleration factor is usually around 0.995 and is higher for lower speeds
- when a ball hits a wall loss of speed is determined by the wall deceleration factor
- when a ball hits another ball bounce deceleration factor is used

- ## Important notes

To change behavior of balls like their speed attenuation, go to constants.h and tweak local constants.

To determine ball’s position before impact for the Ball-Ball situation, it is possible to use trigonometry. Three goniometric function computations are necessary, therefore any efficiency increase is not granted.

Table variables set-up: (used in wall-collisions handling)



- ## Improvements

Various improvement proposals are noted in comments at the beginning of OpenGLControl.cpp file. General idea is to implement other more detailed pool rules, make computations more effective, clean the code from unnecessary statements and replace MFC code by native Win32 API.

- **User Guide**

Run the program by executing Coolaaber.exe. Texture files (leather_green.bmp and wood.bmp) need to be in the same directory as the executable. The application will run even without texture files, but the visual quality will be lower. It may be necessary to have glut32.dll in the directory to be able to run the program.

In the application there are a couple of controls on the right sight.
- A Power slider that defines how much power is put into the stroke
- Sound checkbox that enables/disables sounds in the application
- Change View button that switches views (described further)
- Restart button that ends current game and starts a new one
- Exit button that immediately closes the application

Below the controls there is an information bar that says which player's turn it is, what color of balls is his and it also informs about the final result of the game (who wins).

There are two possible views. Overhead view is a fixed view from above the table; whole table is visible at all times. The second view is bound to the cue ball, which is then always in the middle of the screen and the view is changing according to direction where the user aims.

By holding down the right mouse button and moving the mouse horizontally it is possible to aim, once the button is released, stroke is performed.
By holding down the left button and moving the mouse, it is possible to aim, without any stroking.