Masaryk University
Faculty of Informatics

3D Breakout

Game Report

Richard Trembecký

Brno, Spring 2016

Abstract

This paper introduces the game 3D Breakout, which was created as a project for PA199 – Advanced Game Design course on Faculty of Informatics, Masaryk University. It describes the goals of the game, the creation of the models, game engine functionality, the game mechanics such as collision detection and calculating reflections of the ball, user interface and explains the implementation of classes in detail.

Contents

1 Introduction **1**

2 Background **2**
2.1 *Models* . . . 2
2.2 *Game engine* . . . 3
2.3 *Collision detection and reflections* . . . 3
2.4 *User interface and gameplay* . . . 4
2.4.1 HUD . . . 4
2.4.2 Menu . . . 7

3 Implementation **8**
3.1 *Callbacks* . . . 8
3.2 *Initialization* . . . 9
3.3 *Game engine* . . . 9
3.3.1 `TVector` class . . . 9
3.3.2 `TMatrix` class . . . 10
3.3.3 `TRay` . . . 10
3.4 *Models* . . . 11
3.4.1 `TBall` class . . . 11
3.4.2 `TGround` class . . . 12
3.4.3 `TBounds` class . . . 12
3.4.4 `TBat` class . . . 13
3.4.5 `TBrick` class . . . 14
3.4.6 `TWell` class . . . 14
3.5 *Collision detection and reflection* . . . 15
3.5.1 Ball-wall . . . 15
3.5.2 Ball-bats . . . 16
3.5.3 Ball-bricks . . . 17

4 Conclusion **18**

1 Introduction

3D Breakout is an arcade game inspired by classic ball-and-paddle games such as *Breakout* and *Arkanoid*. The game area consists of a circle ground with invisible wall, three curved bats controlled by a player, a well of twelve 3D bricks in the center and most importantly, the ball. The main objective is to use the ball to destroy the well by knocking down all the bricks forming it. The ball is moving a constant speed and is reflected off the wall, bats and bricks.

2 Background

In this chapter I describe the background of the game, which means naming all the objects and mechanics of the game and also describing the gameplay and key elements of user interface.

2.1 Models

As mentioned before, there are some objects in 3D Breakout, and they have different properties. Every object has to be represented in some way, so it can update the information when it changes. This includes internal representation (properties like position, velocity, activated/deactivated brick, ...), but visual representation as well, to be able to draw it on the screen. More on implementation of different models is written in Section 3.4.

Ball The ball is a moving sphere that reflects off the bats, bounds and bricks. It has the ability to knock down the bricks.

Ground The ground is a textured circle that defines the game area. There is an invisible wall on the edge of the circle. That means the ball cannot move further than the ground radius from the center.

Bats There are three curved bats, positioned at an angle of 120 degrees from each other. They are controlled by a player – he can rotate the bats around the center of the game area by using left and right arrow keys.

Bricks There is a well of twelve colored bricks in the middle. They lay in two layers of six, where each layer has its own color scheme consisting of two alternating colors. Bricks are breakable by getting in contact with the ball. When a lower brick is knocked down, the brick above "falls" down and replaces it.

2.2 Game engine

Creating a 3D game from scratch (without game engine) effectively means to create and implement a small game engine yourself. Such game engine consists of some basic implemented classes.

A vector class The game contains several 3D objects: the ball, the bats and the well of bricks. Each object has to be placed somewhere in the world. Furthermore, the ball and the bats should be able to move. Thus, a vector class is created for representing 3D coordinates, forces and velocities. Such class has to provide functions for various vector transformations, e.g. add/subtract vector, multiply by scalar, invert, make unit vector. Secondly, a function returning magnitude of a vector is useful. To represent relationships between vectors as well, more functions have to be implemented, to compute a distance between two points, an angle between vectors, and, nevertheless, dot and cross products.

A matrix class Matrix class is important to represent transformations of coordinates in the world, such as translation, rotation and scale. Since OpenGL supports these transformations natively, the class is only used for some internal computations, when the exact position after the transformation is needed to compute further. This is the case of rotating the bats, because the positions of the bats are needed to compute collisions later.

A ray class During the game, the ball moves and collides with other objects. The ball moves on a straight line. To be able to detect a collision, a ray class is useful to represent a line on which the ball moves. It provides functions to compute perpendicular distances to points and other rays and functions to find intersection with other ray or a circle.

2.3 Collision detection and reflections

A huge topic for a game engine is a correct collision detection. The topic covers not only a detection, but finding the spot of collision and

the time when the collision happened as well. The computed data can be then used to compute the reflection.

In 3D Breakout, it is needed to constantly check for incoming collisions of ball with the bats, the bricks and the invisible wall at the outside of the circle and then make it bounce off them. The implementation using the mentioned classes is discussed in Section 3.5.

2.4 User interface and gameplay

When the game starts, it looks like this:

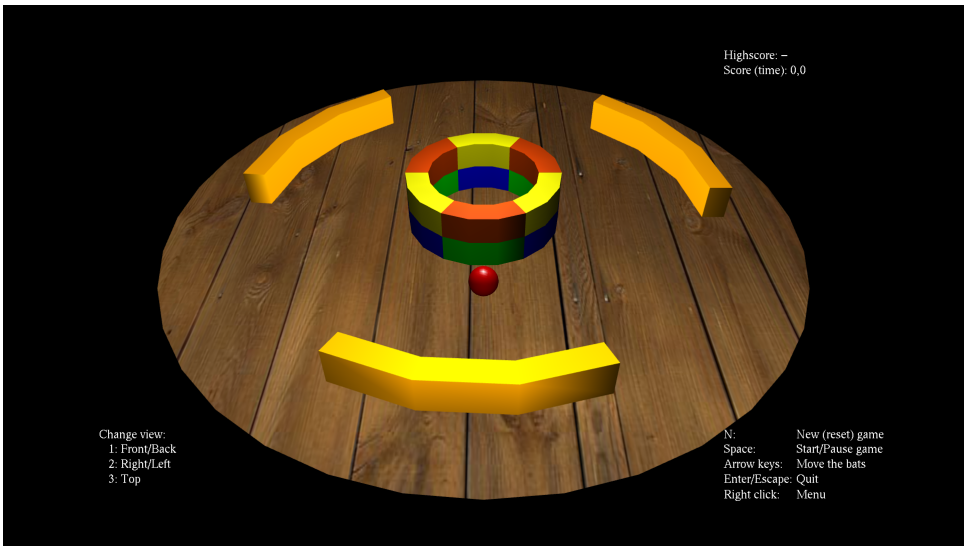

Figure 2.1: First screen

2.4.1 HUD

In the corners of the screen, there is some information about the game displayed, creating a HUD (*heads-up display*).

In the top-right corner, there is a current score and the highscore displayed (Figure 2.2). The score is counted in seconds passed since the start of the game. When all the bricks are knocked down and the score is lower than the highscore, the highscore gets overwritten. Therefore, the highscore represents the best time achieved.

Figure 2.2: Score and highscore counter

In the bottom-right corner, there is a quick help explaining the controls of the game (Figure 2.3). When a player opens up the game, he presses Space to start the game. Doing so makes the ball move in random direction and the time starts running. He then uses the arrow keys to move the bats and affect how the ball moves. Anytime he can press N to reset the game and start over (e.g. when he already accumulated time higher than the best).

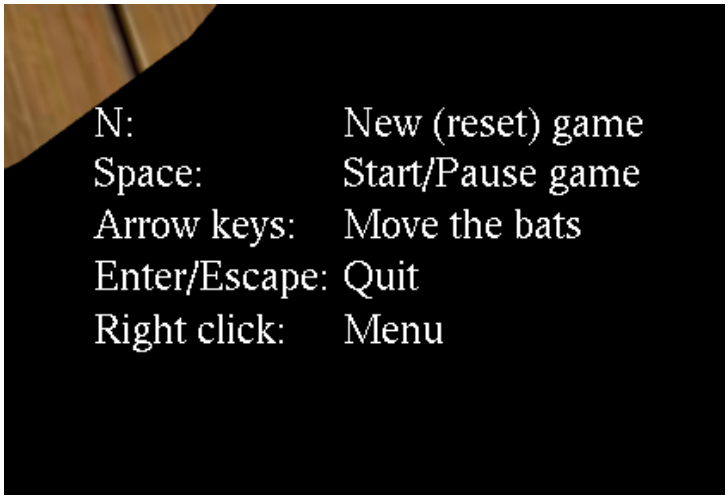

Figure 2.3: Help

In the bottom-left corner, there are another controls explained – the ones to change the view of the game:

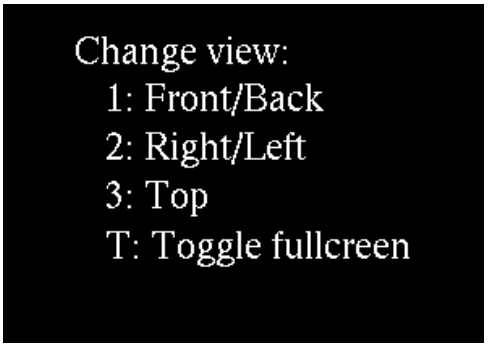

Figure 2.4: Views

Basic view is the front view (as on Figure 2.1), but the side view and the top view is also available (Figures 2.5 and 2.6, respectively).

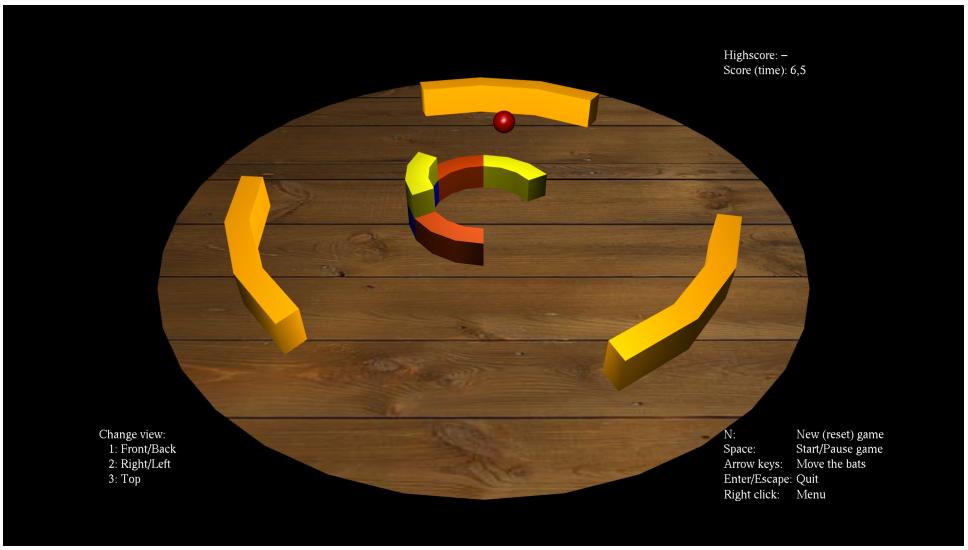

Figure 2.5: Side view of the game

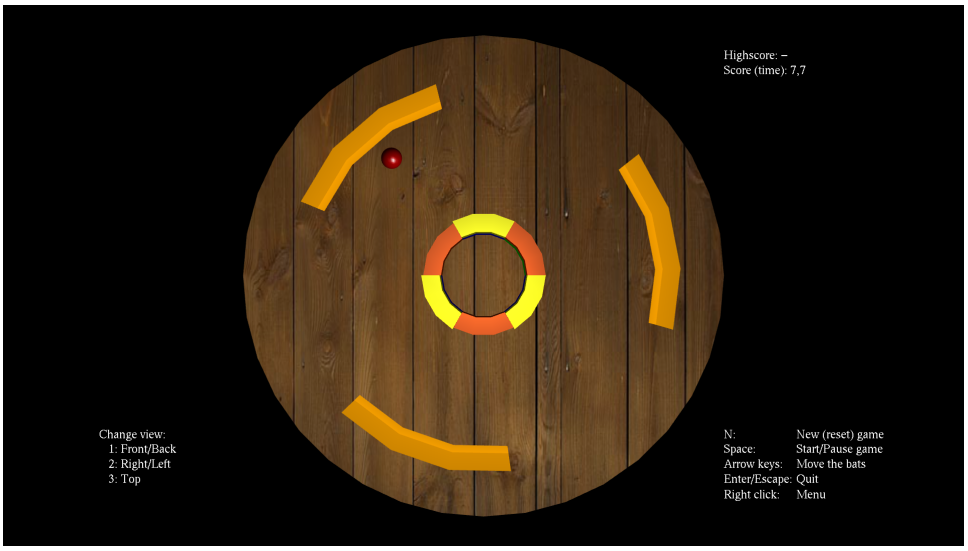

Figure 2.6: Top view of the game

2.4.2 Menu

There is yet another way to access game functions – the right-clicking on the screen opens up an expandable menu:

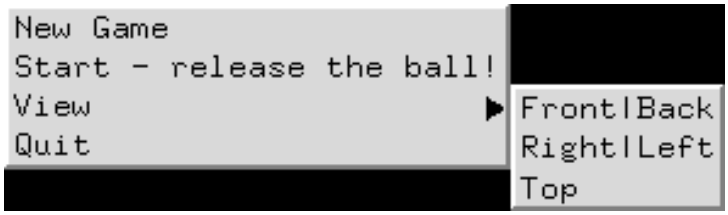

Figure 2.7: Right-click menu

3 Implementation

The game was programmed in C++ language using OpenGL (*Open Graphics Library*), GLUT (*OpenGL Utility Toolkit*) and DevIL (*Developer's Image Library*). In this chapter, I explain implementation details on using OpenGL callbacks, initialization, creating classes for game engine and models and computing collision spots, times and ball reflections.

3.1 Callbacks

The two main functions to program when creating an OpenGL application are the Display function and the Timer function.

Firstly, I created the function `render`, in which objects are drawn, texture is bound and UI is drawn on top, and assigned it by calling `glutDisplayFunc`. At the end of `render` function, `glutSwapBuffers` is called to perform a buffer swap for the current window.

For the latter, I created `timer`, which is run by OpenGL at regular intervals, I chose it to run 60 times per second. Everything related to physics or time, like computing collisions, moving objects or counting score, is done in the function. It is assigned to OpenGL by calling `glutTimerFunc`. At the end of the function, `glutPostRedisplay` is called to let OpenGL know that a frame is ready to be drawn on the screen.

There is a callback for reshaping the window, so I assigned a function `reshape` by calling `glutReshapeFunc`. The function resets the perspective view using `gluPerspective`, so the objects on the screen don't get deformed when resizing the window.

There are 4 other callbacks I used, all of them for registering the input from keyboard. The GLUT function calls:

```
glutKeyboardFunc(key_pressed);
glutKeyboardUpFunc(key_released);
glutSpecialFunc(special_pressed);
glutSpecialUpFunc(special_released);
```

In the functions something is triggered when a key is pressed, or an appropriate bool is set, when a change is reflected in `timer` or `render`.

3.2 Initialization

After callbacks are set, the `init` function is called. This is the initialization function, where object constructors are called, OpenGL variables are set (`GL_DEPTH_TEST` or `GL_LIGHTING`), lights are set and texture is loaded.

There are three lights in the game: The first is located right above the center of the game area and directed down, illuminating all the ground and having a specular component as well. The second is located above the center as well, brightening the well and only a small area around. The third is located near the camera position, illuminating the front side of the well. This light moves when the view changes and is turned off when the top view is set.

The wooden texture is loaded via DevIL `ilImage` class and `Load` function, to which a path to "wood.jpg" is passed. Then it is generated a unique ID by `glGenTextures` and a few parameters are set using `glTexParameteri` and `glTexImage2D`.

3.3 Game engine

The first thing to do was to implement the classes mentioned in Section 2.2. Here I summarize some implementation details for each of them.

3.3.1 `TVector` class

Properties:

```
double _x, _y, _z;
```

The variables `_x`, `_y` and `_z` represent the vector coordinates.

The class includes a basic constructor taking three coordinates as input.

Many functions were straightforward to implement. For example function `Invert` returns a vector, where coordinates are `-_x`, `-_y` and `-_z`.

Functions `Add`, `Subtract` and `Multiply` take other vector and add, subtract or multiply every coordinate by the respective coordinate in the vector. In relation to these, the operators `+`, `-` and `*` were overloaded.

Functions for dot and cross products are defined in keeping with their math definitions. The `Magnitude` function then returns a square root of dot product of vector with itself. The `Unit` function for creating a unit vector then divides each coordinate by magnitude and the `Distance` function for calculating distance between two points returns a magnitude of the subtraction of the points.

There is also a function `Angle` returning an angle between two vectors, based on functions `acos`, `Unit` and `Dot`.

3.3.2 `TMatrix` class

Properties:

```
std::array<std::array<double, 3>, 3> _m;
```

`_m` is a 3x3 2D array of decimal numbers representing a 3x3 matrix.

The class includes a basic constructor taking 9 values to fill the array, but also two constructors to create rotation matrices – one taking 3 angles to rotate around respective axes, other one taking a vector defining an axis and an angle to rotate around it.

Functions `Add`, `Subtract` and `Multiply` (by scalar, vector or matrix) work in the same manner as their vector counterparts – they change every value in matrix by respective binary operation. Operator `*` is overloaded in similar manner.

Function `Transpose` transposes the matrix.

Function `Det` computes the determinant of a matrix and `DetMinor` computes the determinant of 2x2 matrix (the excluded row and column are passed by arguments).

There is a function `Inverse` for computing the inverse of a matrix, based on a cofactor matrix (matrix of minors), so it uses both functions for determinants.

3.3.3 `TRay`

Properties:

```
TVector _p, _dir;
```

`_p` is a vector representing the point on the line, `_dir` represents the direction vector of the line.

The class includes two constructors, one taking 6 coordinates for the vectors `_p` and `_dir`, the other one taking the vectors directly.

Function `Distance` computes the perpendicular distance either to point defined by a vector, or a distance between two parallel rays (lines). They are based on a magnitude of a cross product divided by a magnitude of a direction vector.

Function `IntersectionXZ` returns a vector defining the intersection point of two lines in an XZ plane.

Function `IntersectCircle` returns a first intersection with a circle defined by a point and a radius. The function computes the first root of a quadratic equation.

3.4 Models

3.4.1 `TBall` class

Properties:

```
double _radius;
TVector _position;
TVector _velocity;
```

`_radius` defines a radius of the ball, `_position` defines a 3D position of the ball and `_velocity` represents a velocity of the ball.

The class provides a basic constructor assigning given values to the variables.

Function `SetPosition` sets the position of the ball.

Function `InGround` checks whether the ball is in the given ground (the distance from center is less than the ground radius).

Function `CalculateVelocity` assigns the ball a velocity. If a given velocity's magnitude is greater than 1, it makes it unit. This limits the maximum speed of the ball.

Function `CalcDistanceTravelled` calculates the distance vector the ball passes in a given time. It just multiplies the velocity by the time in seconds.

Function `CalcPosition` calculates the new ball position. It calls `CalcDistanceTravelled` and adds the result to current position.

Function `Move` calls the `CalcPosition` function and assigns it to the move, so it effectively moves the ball in a given time.

`Draw` function consists of two commands: translate to the ball's position using `glTranslated` and drawing a sphere using `glutSolidSphere` defined in GLUT.

3.4.2 `TGround` class

Properties:

```
TVector _points[37];
TVector _tex_coords[37];
TVector _normal;
double _radius;
```

`_points` and `_tex_coords` are arrays of points defining a circle ground and the appropriate texture coordinates. `_normal` is a basic up vector and `_radius` represents the radius of the circle.

The class provides a constructor based on a given radius. Points for the circle and texture coordinates are generated here in a cycle of 37 iterations. Variable equal to 0 holding current angle is defined, incremented by 10 and transformed to radians every iteration. The `x` coordinate of points is then equal to `std::cos(rad_angle) * _radius`, `z` coordinate to `std::sin(rad_angle) * _radius`. As for the texture coordinates, the code is:

```
cos = std::cos(rad_angle);
sin = std::sin(rad_angle);
_tex_coords[i] = TVector((cos + 1.0)*0.5, (sin + 1.0)*0.5, 0);
```

`Draw` function consists of setting the normal for the lighting using `glNormal3d` and drawing by using `glBegin` and `glEnd` with option `GL_TRIANGLE_FAN`. In between these commands the vertices of circle a texture coordinates are set based on values generated in constructor by `glVertex3d` and `glTexCoord3d`.

3.4.3 `TBounds` class

This is a class with only one function called `Wall_Reflection` used to calculate reflection off the invisible wall (bounds). More on reflections is described in Section 3.5.

3.4.4 TBat class

Private properties:

```
bool _lastRight;
```

_lastRight is a boolean remembering the last direction of the bat's movement. Protected properties:

```
double _radius1, _radius2, _height;
TRay _sideLeft, _sideRight;
float _color[4];
```

_radius1 and _radius2 are the inner and outer radius of the bat, _height represents the height of the bat. _sideLeft and _sideRight are used to remember the side lines (rays). They are useful for computing collisions of the ball with sides. _color defines the color of the bat. Public properties:

```
std::array<TVector, 9> _normal;
std::array<TVector, 16> _points;
```

_points and _normal represent arrays of points of bats and normals for all the visible sides.

The class provides a constructor taking two radiuses, height and color, but the altitude and rotation angle as well. The rotation angle is the angle, by which a bat curves. The constructor creates two initial construction points, then rotates them three times by multiplying by a rotation matrix (created by rotation angle). The top 8 points are created by adding a height vector. At the end, the constructor calls the CalculateNormals and CalculateSides functions.

Functions CalculateNormals and CalculateSides calculate the normals and side rays.

Functions MoveRight and MoveLeft rotate the bat by a given angle. It is done by multiplying every construction point by a rotation matrix.

Functions Collisions, CollisionsNow, Faces_Reflection, Sides _Reflection, Corners_Reflection and Reflections are used to calculate collisions and reflections. More on that is described in Section 3.5.

Draw function consists of setting the color using glColor3fv and drawing by using glBegin and glEnd with option GL_QUAD_STRIP. In between these commands the vertices and normals are set using the values generated by constructor.

3.4.5 `TBrick` class

The class for bricks inherits from `TBat` class, because they share the same model. There is one new property

```
bool state;
```

defining an activated/knocked down state.

The constructor calls the `TBat` constructor and sets the state to `true`.

Functions `Deactivate` and `Reactivate` set the state to `false` and `true` respectively.

3.4.6 `TWell` class

Properties:

```
std::vector<std::vector<TBrick> > _bricks;
```

`_bricks` is a 2D array of bricks.

The constructor fills the array with 2 layers of bricks alternating the color and changing the color scheme for the other layer. It calls the function `colorScheme` defined in `Colors.cpp` to find the right color. The constructor takes two radiuses, a height, a number of layers and a number of bricks, though max 2 layers are supported.

Function `Collisions` checks the activated bricks, finds out which brick does the ball collide with and calls `Collisions` function on it.

Function `Reflections` calls the function `Reflections` on a brick given by indexes.

Function `FallBrick` assures that the brick above the brick currently being knocked down seems to "fall down" and replace the knocked one. This is done by deactivating the top one and reactivating and changing the color of the bottom one.

Function `DeativateBrick` sets a given brick's state to `false` and calls `FallBrick`.

Functions `ResetColors` and `ReactivateAll` ale called when the new game is started. They reset the color (similarly to what constructor is doing) and sets the state to `true` for all the bricks in the well.

Function `State` returns `false` when all the bricks have been knocked down. It checks every brick's state.

`Draw` function just calls every brick's `Draw` function.

3.5 Collision detection and reflection

The game uses a continuous (*priori*) method to detect most of the collisions. That means, when running functions for collision checking, the ball's next position is considered, not the current one. The next position is computed using ball's function `CalcPosition` as described in 3.4.1. When a collision is detected next step, the exact collision spot and the time is computed and provided that the normal of the surface is known, the final velocity (reflection direction) can be calculated as well. Later, in `timer` function, the ball is moved by its original velocity for the computed collision time, so it is located in collision spot, then it is assigned the final velocity and is moved for the rest of the step time to the final position.

There is one situation, in which a discrete (*posteriori*) method is used. That is when a moving bat moves in a way that a collision with the ball happens. In that case the position of the ball is fixed by moving it out of the bat in the direction of the bat's movement (side normal) and a slight velocity is added to the ball in the direction.

In this section I explain all the physics interactions between objects in the game.

3.5.1 Ball-wall

Collision detection in this situation is easy – when the magnitude of the ball's position (the distance from center) is greater than ground radius minus ball radius, a collision is happening.

So a function `Wall_Reflection` is called. If it finds out the collision happens, it uses this code to compute the collision time:

```
TVector position = ball.GetPosition();
double absBallVelocity = ball.GetVelocity().Magnitude();
double RV = position.Dot(ball.GetVelocity()) / absBallVelocity;

double abs_RV = abs(RV);
double initial_distance = ground.GetRadius() - ball.GetRadius()
    ;
double Determinant = ((RV*RV) - position.Dot(position) +
    initial_distance*initial_distance);
```

```
collision_time = abs(-abs_RV + sqrt(Determinant)) /
    absBallVelocity;
```

From the collision time, collision spot is computed by running `CalcPosition` passing it the time. Collision position (spot) then defines the normal of the surface, it just needs to be unit. When having the normal, final velocity is calculated as follows:

```
final_velocity = ball.GetVelocity() - normal*(ball.GetVelocity
    ().Dot(normal))*2.0;
```

3.5.2 Ball-bats

A collision with bat should be detected, when the ball's position is within the ball's radius distance from any bat's point. This area can be divided into three by considering the expected result – reflection off a side, a face or a corner.

The implementation of `Collisions` function first discards any positions not in the same half-circle as the bat (when a dot product of position with the bat's middle point is less than 0), then it discards all the positions with magnitudes less than bat's inner radius minus ball's (similar to ball-wall detection) and magnitudes greater than bat's outer radius plus ball's one.

Face A face collision is detected by `Collisions` function, when dot products of the position with the bat side normals are less or equal to zero. Mathematically it means that the position is in the circle sector defined by bat's sides. If an original position was closer to the center, it is a front face collision, if not, it is a back face collision.

Either way, a function `Faces_Reflection` is called, which acts essentially the same as the reflection off the wall, the face is needed only to choose the right radius.

Side A side collision is detected, when the position's magnitude is between the bat's radiuses and the distance from side is less than the ball's radius. As the side rays are the bat's properties and they are always updated, checking the distance is as easy as calling the `Distance` function on the side rays with the ball's position argument.

If the distance from the right side is less than the radius, it is a right side collision, a left side otherwise.

Function `Sides_Reflection` is called. It creates a ray representing the ball's trajectory and calls `IntersectionXZ` on it, passing the side ray to the function. The intersection point represents the collision position. It is straightforward to calculate the collision time from it – the distance to the collision divided by the current velocity magnitude. Normal of the side is easily accessible, so the final velocity is computed as before.

Corners A corner collision is detected, if it is not a face, nor side collision, but the distance between ball and one of the 4 corner construction points of the bat is less than ball's radius.

Function `Corners_Relection` is called. It create a ray for the ball's trajectory as well, but calls `IntersectCircle` on it, with the bat's corner as center and the ball's radius. This circle is the area in which the ball positions the next step, so the first intersection is precisely the collision spot. Collision time is then trivial to calculate – again the distance to the collision divided by the current velocity magnitude. The normal is then defined as the vector from corner to collision, the final velocity is computed regularly.

3.5.3 Ball-bricks

As the bricks inherit from the bats, the code for the bats works for the bricks as well. The only difference here is, the bricks disappear after hit and so the collision detection is turned off. This is done against the state of the brick, which is set to `false` on hit.

4 Conclusion

The game was developed from scratch, new features were added in stages and there is a great space for more.

When looking at the physics, there are two possible points of view. One is looking purely at the successful implementation of the features, the other compares the game physics with reality and names more features that need to be implemented to come closer to real.

It can be said that all of the implemented physics features (the engine classes, collision detection) are working flawlessly, with the single exception of reflection when moving the bats, which is both sometimes glitchy and kind of fake. On the other hand, the game could make use of friction to bring it closer to real physics. Also, implementing gravity would make the game more interesting from the physics look.

As for the graphics side, there is still a room for improvement. Models are edgy, single colored, only the ground is textured. More higher quality textures, some of them maybe procedurally generated, would make a great enhancement to the game's look.