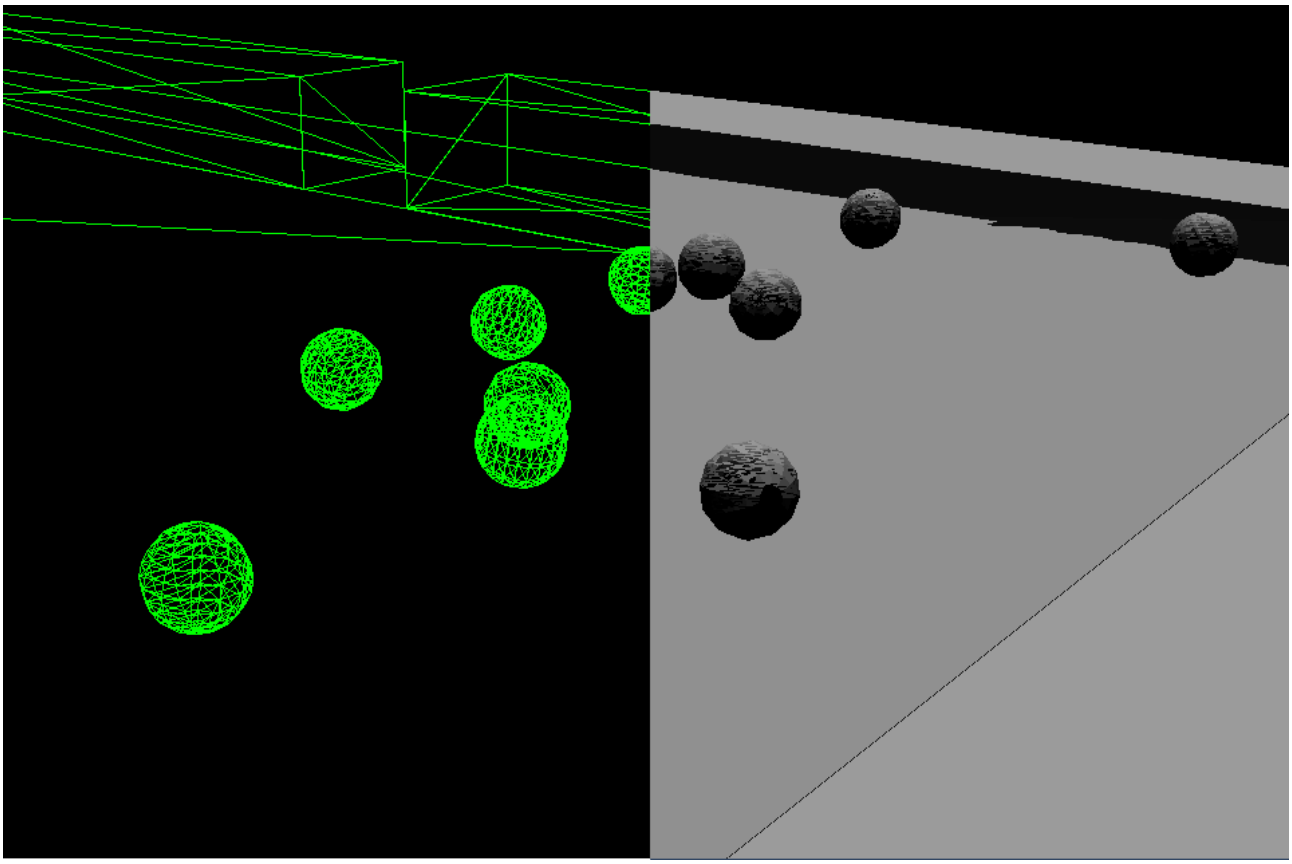
A billiard game engine



By: Niels Orsleff Justesen (noju@itu.dk) 2012

Course: Game Engines

Supervisor: Mark Nelson

Introduction

This report describes a part of a project that was created by Niels Orsleff Justesen and Jacob Grønlund Dinesen as an exam project in 2012 at the IT University of Copenhagen in the course Game Engines and it was supervised by Mark Nelson.

In this project we have created a simple 3D Game Engine for billiard games. The engine is written in Java and is build upon a 3D wireframe renderer written by Jacob in a previous project. It has been our goal to make the engine applicable for (almost) all sorts of billiard games such as Snooker, Carom billiards and the numerous versions of pool (eight-ball, nine-ball, ten-ball, straight-pool etc.). The differences is these games besides the rules, are the sizes of the pool table, the balls and the pockets (if any) and the number of balls, all of which can be setup when using our engine. The engine is however not applicable for pin billiards such as Danish pin billiards since we have not implemented the option to include pins.

My main focus in this project have been to make these various billiards games possible and simple to implement using the engine. I will therefore describe how the architecture supports this vision and also how the physics in the game works. I have also been a part of the process of implementing the Scanline and Z-buffering algorithms but that I will not describe this in the report.

Architecture

The code of this project is divided into two main packages, namely Engine and Game. The Engine package is for classes that are relevant to all billiards games while the Game package is for specific implementations of these. In this way, the engine can be used without having to change any of the core classes. I will start out this section by describing how a billiards game is implemented using our engine and thereafter how the Engine package is structured.

In the Game package we have made an implementation of an eight-ball pool game which only needs to have two classes. The first class it needs, the most important, is an implementation of the Game class. The Game class is an abstract class from the Engine package which we can extend. By doing so we have to implement our own init and update methods. To be able to show anything on the screen, we also need to add a scene. The Scene class is also an abstract class from the Engine package. We have created an EightBallScene class which extends Scene and added this to our EightBallGame. Our scene is now automatically rendered each frame without the need to call anything. In fact, all the physics are also applied to the scene each frame automatically, so the only thing we need to implement ourselves is a way to hit the balls with the cue stick and the game rules. The game rules could be something like: “If the cue ball is in a pocket, place it at its starting position”. All these rule implementations should be placed in the update loop.

The Engine package has five sub packages with classes which I will explain. Its root only have one class, the BilliardGame class. It is an abstract class that contains a game loop which begins to run when its run method is called. As mentioned earlier it has two abstract methods, init and update which needs to be implemented by the extended game class. The draw method however is is a private method. The game loop has a default frame rate of 24 frames per second and thus tries to call the update and draw method 24 times per second. If the physics, the update loop or the rendering takes too much time to complete, the frame rate naturally drops. The draw method uses an instance of the class Renderer to draw the scene and a JFrame and a JPanel to present the final image. The Renderer class is an abstract class and is extended by the WireframeRenderer, PaintersRenderer and ScanlineRenderer which our BilliardGame classes can switch between. You can actually switch renderer in run time using the 1, 2 and 3 keys. The Renderer class has an abstract method render, which takes a Scene and a Screen as parameters. A Scene class is also abstract and needs to be extended as we did with our Eight-ball implementation. The Scene class has three fields, a list of GameObject instances, a list of Light instances and a Camera instance. The Screen class is simply just a description of the screen dimensions.

The BilliardGame class also has an instance of InputManager attached. The InputManager implements the KeyListener, MouseListener and MouseMotionListener and keeps track of which keys are down as well as the mouse position. The InputManager is used to request whether a specific key is down. Additionally, it creates a small abstraction layer when requesting e.g. the arrow keys. The function getHorizontalArrows() returns either -1, 0 or 1 based on the sum of the directions. By pressing only the left key it would return -1 while pressing both left and right it would return 0. The InputManager also has similar functions for the vertical direction and for the WASD keys.

GameObject is also an abstract class. It has a list of Shape3D instances, a position and a boolean expressing its visibility. The GameObject class has an abstract method called build, which should be called when it is created. Here is an example of how a GameObject is implemented:

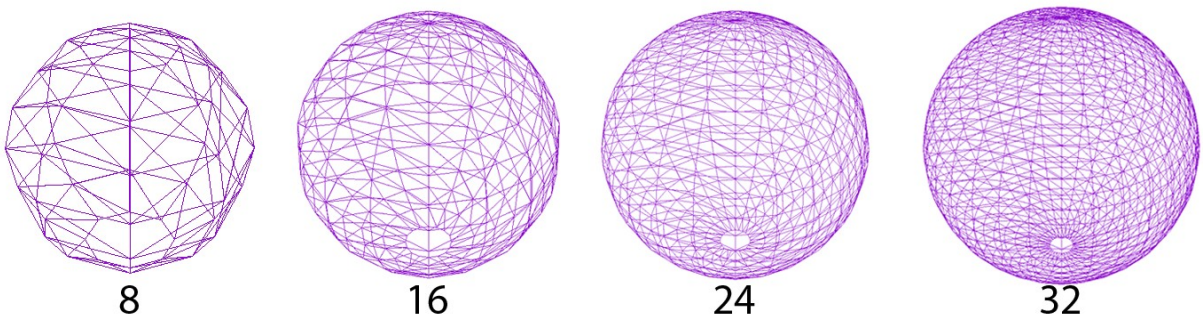
```
public class BilliardBall extends GameObject implements Movable {

	private static final int gradient = 10;

	private int number;
	private double radius;
	private Vector3D velocity;
	private boolean inPocket;

	public BilliardBall(int number, double radius, Point3D position){
		super(position);
		this.number = number;
		this.radius = radius;
		this.velocity = Vector3D.Zero;
		this.inPocket = false;
		build();
	}

	@Override
	protected void build() {

		addShape(new Sphere(Point3D.Zero, radius, gradient));

	}
```

The build method simply just creates and adds shapes to itself. A BilliardBall, not surprisingly, only has one shape which is a Sphere in contrast to a PoolTable instance which is constructed of many Shape3D instances, such as a Cloth instance and several Rail instances. You will notice that BilliardBall also implements the Movable interface and that it has a velocity field. We will explain this in a moment.

A Shape3D instance simply just consists of a list of Triangle3D instances and an anchor point that is relative to the position of its parent. One of the more complex shapes used in the project is a Sphere. The Sphere class has a field called gradient which determines the number of points along its latitude (M) and longitude (N). The sphere is build in the following way.

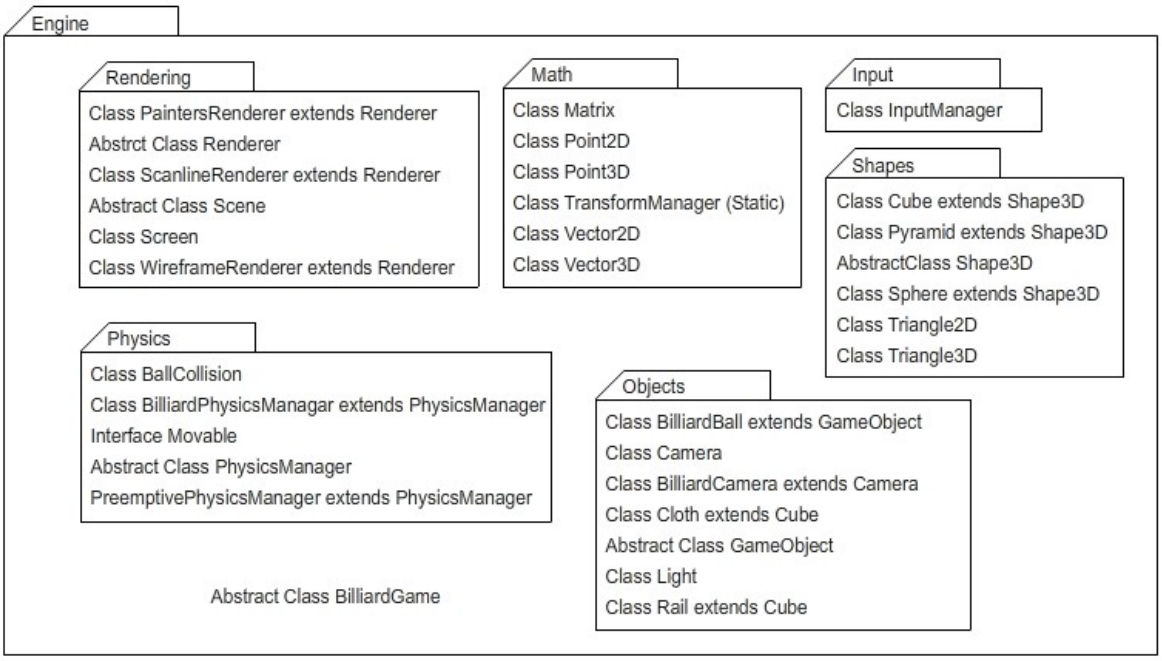
```
int N = gradient;
int M = gradient;
// M lines of latitude (horizontal)
for(int m = 0; m < M; m++){
        // N lines of longitude (vertical)
        for(int n = 0; n < N; n++){
                x = radius * Math.sin(Math.PI * m/M) * Math.cos(2*Math.PI * n/N),
                y = radius * Math.sin(Math.PI * m/M) * Math.sin(2*Math.PI * n/N),
                z = radius * Math.cos(Math.PI * m/M));
                Point3D p = new Point3D(x,y,z);
                points.add(p);
        }
}
```

When the points are created they are connected and triangles are formed. A triangle is formed by connecting an arbitrary point with index *i* in the list *points* with the point with index *i*+1 and *i*+*N*. By doing this to every point in the list half of the triangles will be created. The other half of triangles are formed by connecting a point with index *i* with the point with index *i*-1 and and *i*-*N*.



Now, as I have explained the classes needed to create a billiards game that can be rendered we need some physics so the game objects can interact. The PhysicsManager class has a method called move which is called for every frame. This method moves all the balls and calculates collisions. As we saw earlier the PoolBall object also implements the Movable interface which gives the balls a velocity that can be manipulated. The PhysicsManager uses the velocities to calculate the movement and collisions but also changes the velocities as they move and collide.

Last but not least is the sub package Math which contains classes such as Matrix, Point3D and Vector3D. These classes are used by almost all the components in the engine from rendering to physics and are packed with arithmetic and geometric methods.



A package diagram of the engine

Physics

Physics in billards games are quite complicated if we are to simulate it perfectly, but by simplifying it a bit and ignoring features such as spin it can be narrowed down to some relatively simple vector calculations. All dimensions in the scenes are described in centimeters and but the PhysicsManager uses meters and seconds as preferred units. Additionally, it uses 3D vectors as the velocities are directional. The z value for the balls in the scene will always be zero so we can use the 3D vectors as they were in 2D.

Friction

As a billiard ball moves across the cloth of the table it is slowed down due to friction. There are two types of friction involved in a billiards game. A static friction occurs when a ball is rolling on the cloth while a kinetic friction happens when a ball overcomes the static friction due to its high velocity and is sliding across the cloth[1]. When creating an instance of the PoolTable game object the constructor takes both a static friction coefficient and a kinetic friction coefficient. The coefficient for the kinetic friction is usually higher[2], which means that balls at high and medium velocity are deaccelerating faster than slow moving balls. When requesting the table for its friction it returns either the static or the kinetic friction coefficient based on the velocity of the moving ball. I have not investigated the actual velocity needed to overcome the static friction but in the EightBallScene I have set it to 1.5 m/s (3.36 mph) which is considered to be a medium to slow moving billiard ball[3].

In the EightBallScene the static friction coefficient is set to 0.01 which gives a so called *table speed* of:

$$\frac{1}{\mu_{rolling}} = 100$$

A *table speed* of 100 is in the high end of normal pool tables[2]. The kinetic friction is set to 0.2[3].

After the PhysicsManager moves a ball it slows it down due to the friction. The change in velocity of a ball rolling can be found by using this formula[2]:

$$V_0 - V = \mu_{rolling} \times g \times t$$

Where V_0 and V is the initial and final velocity of the ball, the gravitational acceleration is $g = 9.81 \frac{m}{s^2}$ and *t* is the time in seconds. Using this formula and the static friction used in our EightBallScene we can calculate that a ball with $V_0 = 1 \frac{m}{s}$ comes to rest in *t* seconds:

$$V_0 = \mu_{rolling} \times g \times t$$

$$t = \frac{V_0}{\mu_{rolling} \times g}$$

$$t = \frac{1\frac{m}{s}}{0.1 \times 9.81\frac{m}{s^2}} = 10.19s$$

10.19 seconds seems to be quite a long time for a slow moving billiard ball to come to rest but this example is not considering loss of velocity when the ball hits the rails and this have not been implemented in the engine, but it should be set to have a coefficient of restitution of 0.6-0.9[3].

Collision detection

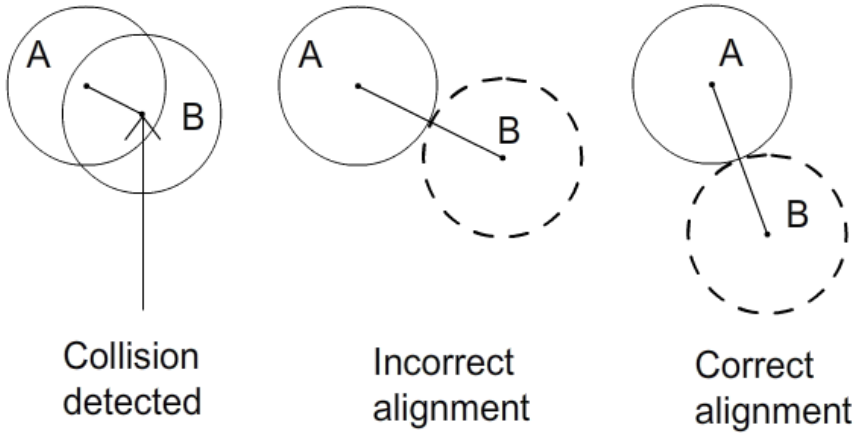
Detection collision in a billiards games is relatively simple since we only have spheres and cubes in the scene. I have implemented two different physics managers that handles collisions in different ways. The first I implemented is the BilliardPhysicsManager which uses a corrective method of handling collisions. This means that it first moves all balls and then corrects the positions and velocities based on the collisions found[1]. This method works well when the delta time is low, but horribly when it is high. The first problem that can happen is that balls can pass through each other as their position can vary a lot between the updates. This can also mean that balls can fly off the table if they are moving with a high velocity as they never collides with a rail. The second problem is actually the same as the first but with a smaller impact. Since balls have to overlap to collide they sometimes overlap a lot. While it is possible to correct the positions precisely at the collision point by analysing the last move of one of the balls it is not that simple to implement. The solution I implemented a much easier solution which actually would work very well if the delta time is low but if not it results in wrong angles and weird position corrections. I will first explain how this solution is implemented and thereafter how the preemptive method works.

Corrective physics handling the easy way

To check if a collision happens between a sphere and a cube we first calculate the distances between the center of the sphere and the center of the cube. If the distance in the x-axis is lower than the sum of half the width of the cube and the radius of the sphere a collision has occurred. Same goes if the distance in the y-axis is lower than the sum of the height of the cube and the radius of the sphere. The axis for which the collision is detected I call the collision axis.

A collision between two spheres happens if the distance between the two centers are lower than their two radii combined. When a collision happens the position of the movable objects needs to be corrected so they no longer intersect but instead align. For a sphere-cube collision the sphere is simply moved along the collision axis to a point where it aligns the cube. This does create a small error in my implementation since it is only corrected in one axis. This can also be corrected properly by analysing the last move of the ball.

When two spheres are colliding the positions are corrected for one of the spheres to align the other. This is done by finding the unit vector of the line between the two centers and then multiply it with the sum of the two radii. Since we are calculating the new position from a vector that only exists in this incorrect situation we also get an incorrect result. But again, for small delta times it is close to a correct result.



Billiard ball B gets assigned an incorrect position. This does not only affect the way it is position at the collision time but also the result of the force transfer. Which we will see later.

Preemptive physics handling

Instead of trying to fix the corrective way which seemed difficult I have implemented a preemptive way of handling the ball collisions. The rail collisions can also be done preemptively but this has not been implemented.

This physics manager handles collisions by foreseeing future collisions. For a ball that moves it calculates the vector it can move in the given time step and checks if this vector intersects with any other ball. The other balls are represented as circles with a double radius since it is testing for intersection with a line and not a circles. For each intersection detected a BallCollision object is instantiated. A BallCollision keeps track of the collision location and the two balls in the collision. If several ball collisions where found it it only looks at the one closest to the current position of the moving ball. The correction part is simply just to move the ball to the collision location. This method is a lot better than my corrective physics handling but still not perfect since the balls are moved one at the time which makes it incorrect and since its outcome depends on the delta time it is also non-deterministic.

Stepwise preemptive physics handling

Luke Anderson [1] suggests an algorithm that also is preemptive but in a stepwise and deterministic way. The algorithm calculates at what time t the first collision in the current time step will occur, if any. It then moves all the balls as if the delta time were equal to t, changes the velocity of the balls inflicted in the collision and then starts over until no more collisions are detected in the time step. This method is completely deterministic since it is completely independent of the delta time and it is correct since it simulates the way balls move at the same time.

Ball/ball collision

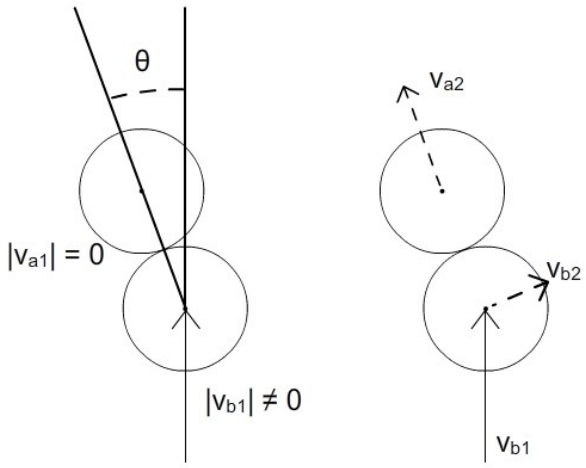
When two billiard balls collide it is close to a complete elastic collision where all the energy is conserved the system after collision. While the rotations of the balls are affecting the resulting collision I have chosen to ignore this for my physics implementation.

To explain how I calculate the resulting angles of a ball/ball collision I will start with a simple example where ball B hits ball A with a velocity V_{b1} while ball A is at rest. First, a vector *v* is found which goes from the center of ball B to the center of ball A. Then, angle θ between vector V_{b1} and *v* is calculated. The length of the resulting velocity of ball A $|V_{a2}|$ will then be:

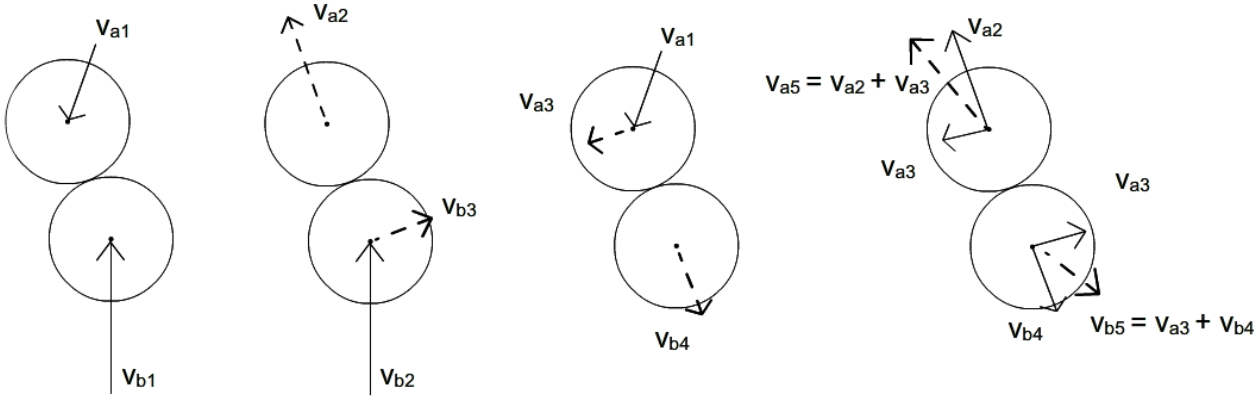
$$|V_{a2}| = \frac{90}{(90-\theta)} \times |V_{b1}|$$

V_{a2} is then found by multiplying the unit vector of *v* with $|V_{a2}|$.
What happens to ball B is easy as the the velocity it has lost is known.

$$V_{b2} = V_{b1} - V_{a2}$$



Collisions where none of the balls are at rest is a bit more complicated but by looking at it as two separate collisions it is just as the previous example. First ball A transfers some of its velocity to ball B and then ball B transfers some velocity to ball A. The final velocity are then the sum of the two resulting vectors.



Conclusion

In this project we have made a simple 3D game engine for billiards games. It is possible to write an implementation of a billiards game using our engine without changing any of the files in the engine and by only writing an extended BilliardGame class and Scene class. The engine has semi-correct preemptive physics handling for ball collisions and uses kinetic friction for sliding balls and static friction for rolling balls. The engine ignores the coefficient of restitution of ball/rail collision as this has not been implemented and the collisions for rails are not preemptive as the rest of the physics. To get a correct and deterministic physics simulation an algorithm is needed that does not move one ball at the time but instead moves all balls simultaneously to the next point in time where a collision happens. To get totally realistic simulation spin should also be implemented.
The engine has three renderers implemented which can be changed at runtime. Unfortunately, it is only possible to play the game with the wireframe renderer due to the low frame rate with the other renderers. The renderer using Painters Algorithm looks quite funny because many of the triangles in a cube actually shares the same depth in this implementation and of course Painters Algorithm is not supposed to look perfect. The renderer using the Scanline Algorithm and a Z-buffer works somehow but has some small errors which could look like a rounding error somewhere but it looks like the algorithm is implemented correctly.

References

[1]. Anderson Luke. Rensselaer Polytechnic Institute, Troy, NY. “Realistic Billiards Simulation with Variable Time-Step” <http://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S09/final_projects/anderson.pdf>

[2]. Shepard Ron, Argonne Pool League, Argonne National Laboratory. “Amateur Physics for the Amateur Pool Player” <ftp://146.137.86.42/pub/shepard/pool/old_versions/physics.PDF>

[3]. ?. “Physics of Pool and Billiards” <http://billiards.colostate.edu/threads/physics.html#properties>