Report: Advanced Game- and Simulator Programming

Shvetsov Nikita

March 1, 2016

Abstract

In this report the simulation of rolling balls in the box program is presented. The main aim is to simulate the rolling balls on the surface and collisions between balls and walls. One ball is controlled by a viewer. Using directional keys player can change the velocity value and direction and create a ball collison situation. All balls are moving and colliding according to the laws of physics.

1 Introduction

This project is about simulation of the laws of physics in a closed environment with balls. Balls are randomly rolling on a curved Bezier surface. A viewer can manipulate with one of the balls. Different types of collision are implemented in this project, according to laws of physics. This simulation is using graphics engine, scene graph and hierarchy from GMlib and Qt Framework API. The project was programmed with C++ using Qt Creator. This report contains information about used technologies, collision environment and code structure of the program.

2 GMlib

Geometric Modeling library, or GMlib written in C++, is an open source project developed and maintained by the professors, teachers and students at Narvik University College [**?**]. There is a set of important modules, such as core, opengl, parametrics, scene, which are handling the graphics and mathematics. GMlib core module contains the structure of points and vectors which is used for simulating. In core module, there are numerous functions for calculating the mathematics between vectors and points. The parametrics module contains multiple classes like class $PSphere$, $PSurf$, $PPlane$, $PBezierSurface$ which we are using for simulating balls, walls and surface. The other used class is the scene class. We have to insert all created objects into the scene graph and then visualize them.

3 Qt Framework

Qt is a cross-platform application and UI framework for developers using C++ or QML, a CSS JavaScript like language. Qt Creator is the supporting Qt IDE. Qt, Qt Quick and the supporting tools are developed as an open source project governed by an inclusive meritocratic model. Qt can be used under open source (GPL v3 and LGPL v2.1) or commercial terms [?]. QtCore is the core module of the Qt-framework. It contains various features, including a meta-object system, property system, object models, object trees and ownerships and signals and slots. The module also provides for platform independent threading classes, a way to safely post events through threads and signal-slot connections across different threads. In our project we directly use different modules of Qt. For example: QTimerEvent, QObject, QKeyEvent.

4 Concept

The initial idea was to create a simulation of rolling balls inside a box to show functioning of the physics laws, without energy loss within enclosed system. Also the goal was to implement collision detection and handling and rolling of the ball on Bezier surface. After giving a viewer ability to manipulate with ball energy gain-loss was implemented for the controlled ball. The viewer can manipulate the selected ball with arrow keys. The initial scheme of the simulation is shown in fig.1.

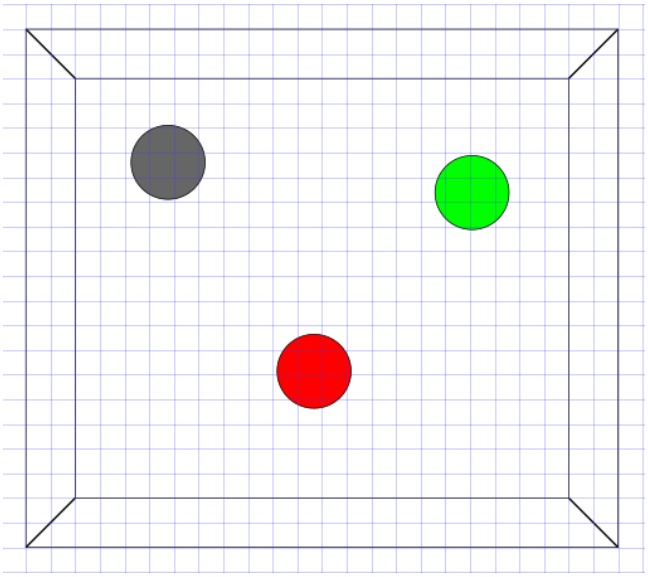

Figure 1: Concept of the program

5 Class structure of the application

Main designed classes for the project are:

- *Ball*

  In this class we set all parameters for balls (mass, velocity, radius), have methods for controlling a ball and have *localSimulate* function for translation and rotation of the ball.

- *Collision*

  This class is designed to store objects which are colliding, including the collision time.

- *Controller*

  Controller class is used for storing objects, finding and handling collisions and computing the next ball movement coordinates.

Wall class was inherited from GMlib $PPlane$ class and Bezier surface from GMlib $PBezierSurface$. In *gmlibwrapper* class we create all objects, define construct parameters, add objects to controller for handling. Class scheme is shown in fig.2.

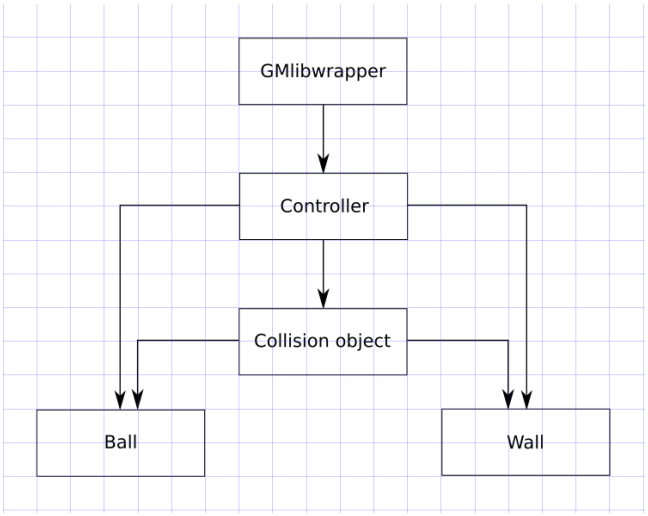

Figure 2: Class scheme of the project

6 Collision detection and handling system

6.1 Ball-Ball collisions

First, consider ball-ball collision detection. Collision is happened when the distance between two balls is equal to the sum of radiuses of these balls. So we need to multiply dS vector by part of dt time - value x - time of the collision, add it to the previous position of the ball and get the difference. This value should be equal to sum of radiuses.

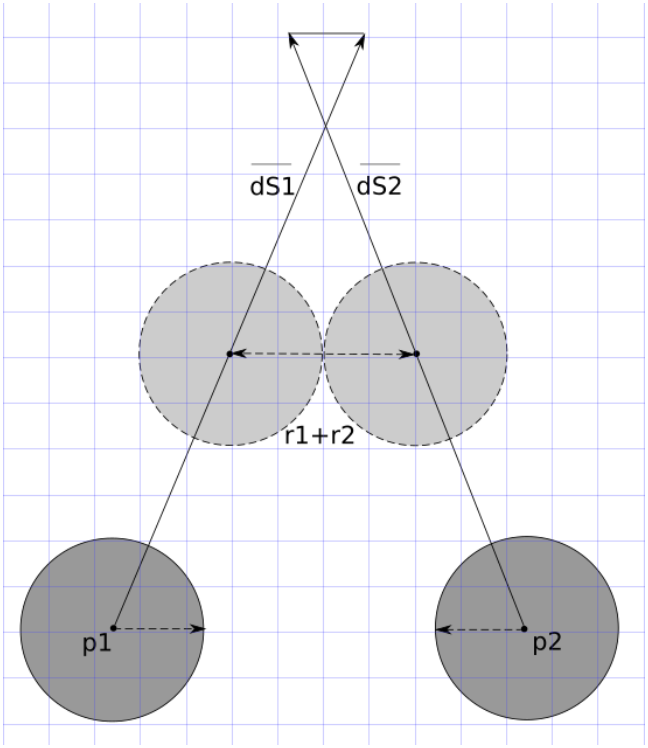

Figure 3: Ball-Ball collision illustration

Thus, we get

$$| (p_1 - xdS_1) - (p_2 - xdS_2) |^2 = | r_1 + r_2 |^2$$

After transformations we get a quadratic equation of type $ax^2 + bx + c = 0$:

$$< divdS, divdS > x^2 + 2 < divPos, divdS > x + < divPos, divPos > -r^2 = 0$$

where $divPos = p_1 - p_2$, $divdS = dS_1 - dS_2$.
For the collision time we take the smallest x.

If $c < 0$,then balls are intersecting. Sometimes this happens when balls have small velocity values. In order to fix it we need to translate them on required distance and redefine coefficients in the equation.

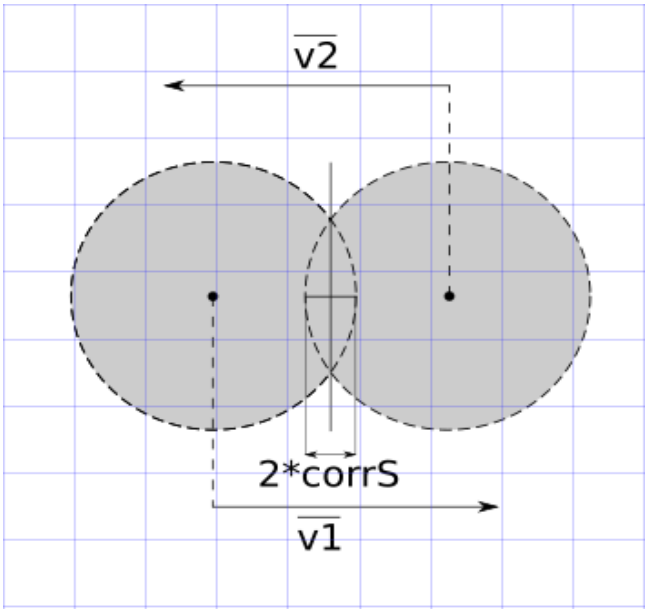

Figure 4: Intersection of balls

If $x > 1$ then balls are not colliding, in order collision to happen we need $0 < x \leq 1$.

When we find collision, we need to handle it.

U_1 and U_2 are the velocity vectors of the two spheres in a collision. Vector d is a difference between positions of the balls. Projections of U_1 and U_2 onto the axis of d vector are vectors U_{d1} and U_{d2}. U_{n1} and U_{n2} are projections on the axis perpendicular to d. m_1 and m_2 are the masses of the balls. V_1 and V_2 - new velocity vectors after the collision, V_{d1} and V_{d2} are projections on d. So we have according to [**?**]:

$$d = p_2 - p_1$$

$$U_{d1} = U_1 * d, U_{d2} = U_2 * d$$

$$V_{d1} = \frac{U_{d1} * m_1 + U_{d2} * m_2 - (U_{d1} - U_{d2}) * m2}{m_1 + m_2}$$

,

$$V_{d2} = \frac{U_{d1} * m_1 + U_{d2} * m_2 - (U_{d2} - U_{d1}) * m1}{m_1 + m_2}$$

$$V_1 = U_1 + V_{d1} * d - U_{d1} * d, V_2 = U_2 + V_{d2} * d - U_{d2} * d$$

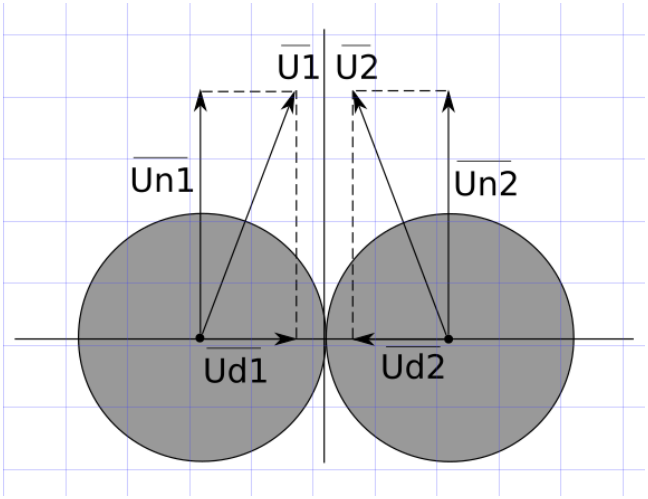

Figure 5: Ball-Ball impact situation

After we acquired new velocity vectors for each ball, we should *computeStep* for each ball.

6.2 Ball-Wall collisions

Next we deal with Ball-Wall collisions.

Assume we have ball with center in point p, then vector $d = q - p$, where q is acquired by evaluation matrix as before. xdS - vector connecting point q and position of the ball, when it collides with wall. So we have

$$< d + xdS, n >= r$$

where r is radius of the ball, n - normal to the wall.

From this equation we can get x value:

$$x = \frac{r + < d, n >}{< dS, n >}$$

If $(< dS, n > + r) \geq 0$,it means that ball and wall intersecting. Then we need to translate the ball by a respective value and redefine $< d, n >$.

Collision with the wall of the ball occurs when the angle between the vector dS and normal to the wall is obtuse. So, we need to find scalar product of these vectors.

If $(< dS, n >) = 0$, then the ball is moving parallel to the wall.

If $(< dS, n >) > 0$, then angle is acute and the ball moves away from the wall.

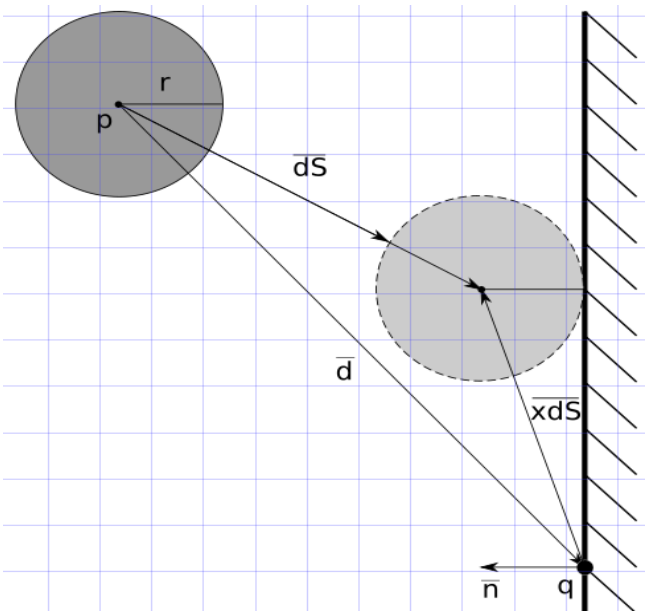

Figure 6: Ball-Wall collision illustration

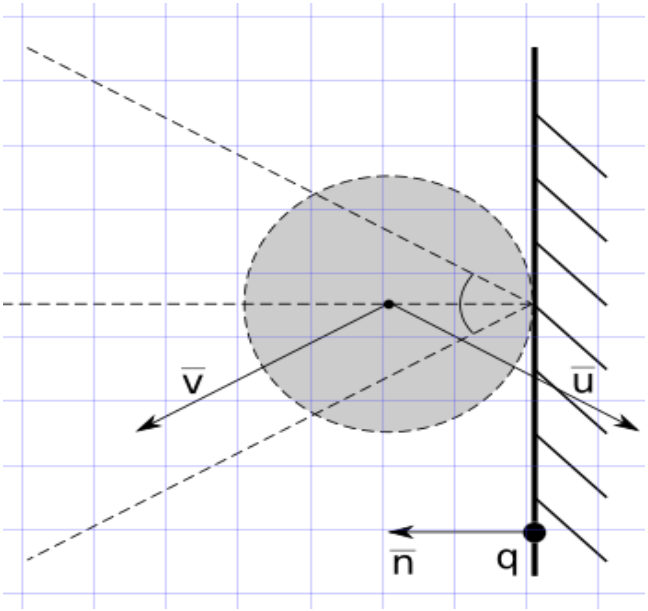

Figure 7: Ball-Wall impact situation

If $(< dS, n >) < 0$, then the angle is obtuse and the ball is moving toward the wall.

When the ball collides with the plane, it changes the direction of the velocity vector. Angle of the new direction from the normal point of the impact is the same.

If we take the scalar product of velocity vector and a normal, we get the angle between them. If we want no energy loss, then we need to multiply angle by 2. So the result formula will be as

$$velocity = velocity - 2* < velocity * normal > *normal$$

7 Simulation process

For updating properties of the ball, methods *computeStep* and *localSimulate* of the ball class are used.

In *computeStep* function we use methods for redefining the displacement of the ball and velocity vector.

First we need to find dS value:

$$dS = dt * velocity + \frac{dt^2 * g}{2}$$

After we get new position as $pos = getPos + dS$, where $getPos$ is the previous position. Next step is to fix the obtained position according to the surface bending. For this purpose we find closest point to the surface using function *getClosestPoint*. Then we create a matrix of type

$$sMatrix = \begin{pmatrix} p & p_u \\ p_v & p_{uv} \end{pmatrix}$$

using function *evaluate*. From this matrix we can get a normal to the surface as $normal = sMatrix[0][1]\hat{}sMatrix[1][0]$ and final displacement (new postion - old position)

$$dS = (sMatrix[0][0] + r * normal) - getPos$$

Then we update velocity of the ball. So $velocity = velocity + g * dt$. Then we need to correct it to the tangent plane of the surface at the new position:

$$velocity = velocity - (velocity * normal) * normal$$

For each dt ball have own function *localSimulate*in which it translates by calculated dS and rotate by angle $\frac{|dS|}{r}$ about axis $(normal\hat{}dS)$.

Controller object is responsible for all interactions between objects. This object is inserted directly into the scene in *gmlibwrapper*. In *Controller* class we store all interacting objects for simulation. It has its own *localSimulate* function, which deal with finding and handling collisions. The algorithm of *localSimulate* function for each part of time dt:

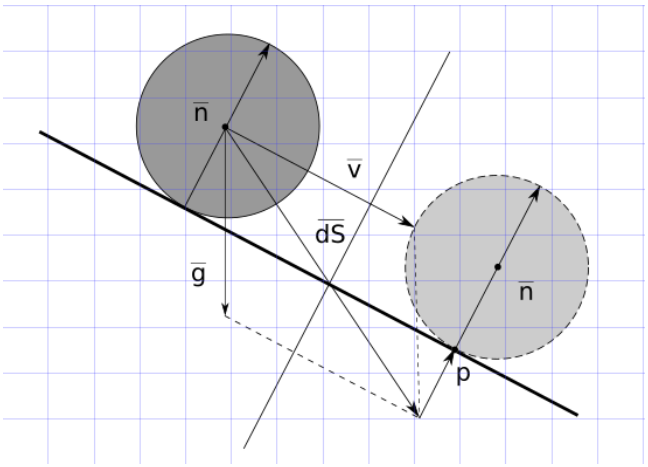

Figure 8: Rolling of the ball by *computeStep* function

- Compute step (updating velocity vector, position and rotation) for all balls in controller set.
- Find all ball-ball collisions and add to collision set.
- Find all ball-wall collisions and add to collision set.
- While set of collisions is not empty:
  - Sort the set and make every element unique.
  - Take the first element into consideration for handling further and delete it from set.
  - If element is ball-ball collision: handle collision (update velocity, compute step for each ball) and find new collisions.
  - If element is ball-wall collision: handle collision (update velocity, compute step for a ball and check if the ball stopped) and find new collisions.

8 User interaction and interface

A viewer can manipulate one of the balls using arrow keys. It is done by changing parts of the velocity vector. In order to make it 4 functions in the *Ball* class were created: $moveUp$, $moveDown$, $moveRight$, $moveLeft$. Besides, velocity control checks were implemented, to make sure that ball is not getting extremely high values of velocity. The final interface of the simulation is shown on fig. 9.

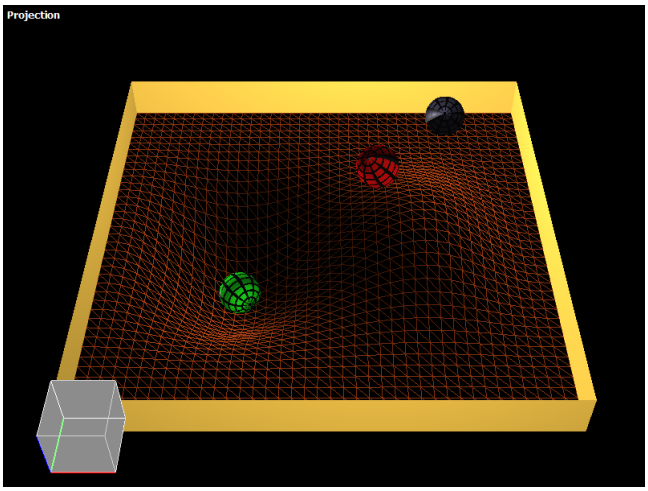

Figure 9: The simulation process

9 Conclusion

During this project a simulation of rolling balls on a Bezier surface was created. All types of collisions and collision handling were implemented and combined in a controlled simulation.

9.1 Future improvements

There are some features, that can be implemented in order to improve the simulation:

- Adding more obstructions or making different simulation playgrounds.
- Improving the system of ball controlling.
- Adding more balls to the system and transforming a simulation into a game, with the goal to avoid collision with player ball.

References

[1] Bratlie, Jostein. *Gmlib. http:://episteme.hin.no.*. Narvik University College, 2016.

[2] Qt Project. *Qt Documentation. http://doc.qt.io/qt-5/reference-overview.html.* 2016.

[3] Narvik University College. *STE6245 Advanced Game- and Simulator Programming.*. Narvik University College, 2016.

[4] Arne Laksa, B. Bang. *Simulating rolling and colliding balls on freeform surfaces with friction.*. Narvik University College, 2005.