Advanced Game-and Simulation Programming

PROJECT REPORT

hxu001 | STE6245| February 26, 2016

Abstract

This report is about a study of creating a 3D computer game. In this study, a simulation of multiple balls rolling and bouncing off a curved has been done by using graphics engine, scene graph, scene hierarchy API from GMlib and Qt's Framework API. In the report, it is including description of GMlib, Nokia's Qt Framework and the description of the project. The work was carried out by Department of Computer technology and computation oriented engineering in UiT-The Arctic University of Norway.

Key words: ball collision, C++, GMlib, Qt, OpenGL, simulation

Contents

1 Introduction

This is a project based on course STE6245 Advanced Game- and Simulator Programming. In this report, it is described a work of simulating of multiple balls rolling and bouncing off a terrain and motions based on material-properties and the laws of physics.

This report has four main parts which are introduction, description of GMlib, description of Nokia's Qt Framework, description of implementation and conclusion. This program was implemented by C++ using graphics engine, scene graph, scene hierarchy API from GMlib.

2 Geometric Modeling Library (GMlib)

The Open Source Geometric Modeling Library (GMlib) is a programming library developed in C++. It was initially started and first developed by the staff at UiT-The Arctic University of Norway, Narvik campus from 1994. The main field for the programming library is geometry, computer graphic and simulation. Thus, it is assumed that the user has prior knowledge in C++, Euclidian and affine geometry, homogenous coordinate systems and OpenGL[1].

2.1 Main structure/modules

GMlib consists of eight modules while OpenCL and Wavelet modules are not active in this project. Each of the modules has two sections which are scr folder and tests folder. Following are description of the six modules:

Module 1 - Core: is a template library for affine spaces; points, vectors, matrices, frames, simplexes etc. There are four folders and one header file inside the scr folder. In the 'containers', there are array, matrix, vector classes. In the 'static', there are two classes for 1 dimensional and 2 dimensional points and vectors making interface for the Static Meta programming to roll out the code use in: Matrix SqMatrix, HqMatrix classes. In the 'type', there are angle, matrix, point, simplex and subspace. In the 'utils', it provides a number of functions and definitions globally used or needed in GMlib.

Module 2 - OpenGL: contains the graphic interface including interfaces to GPU and shaders for graphic purposes.

Module 3 - Scene: is a module consisting of camera, event, light, render sceneobjects, selector, visualizers and utils. The scene has display, select, tracing, editing and simulation functionality. It also includes special SceneObject as different camera types, different light types etc. In the sceneobject, each object has its own local system while the scene has a global system. When the user need the object rotating or moving, by changing the local coordinate will be achieved. In the gmpathtrack class, it has localSimulate function with only argument time which is very useful to make the object moving, rotating simulation. When this object is attached to another *SceneObject, it will keep track of the path of the object. By setting the element size and the stride between points one can control the behaviour of the PathTrack object.

In the visualizer, there are 5 different way to render object from different parts to get different effects: for coordinate system respectively, selector grid, selector, surrounding sphere and etc. it also has texture visualizer and different material setting up in the utils.

Module 4 - Parametrics: has a template baseclass Parametric with the inherited PCurve, PSurf and PTriangle handling parametric curves, and surfaces as Bezier, Hermite, B-Splines, NURBS, Exporational B-Splines, etc. The pre-made geometry model can be called directly to the project. In the evaluators, it has several class for getting interface of different types of shapes. In this section, it also has visualizers. It does the function that rendering the objects on screen with requiring appearance. In the gmpsurftexvisualizer class, it has a function called setTexture () that allow to use your own picture to render the surface of the object. There is a translate () function inside the gmparametrics class that is very useful for relocating the object.

Module 5 - Trianglesystem: is a set of template classes TriangleFacets, Triangle, Edge and Vertex useful for terrain modeling, and other type of ”2D-based” (without overhang) triangle surfaces. A Delaunay triangulation also including constrains is implemented. It can be used a lot when solving FEM problem with triangle meshing.

Module 6 - Stereolithography: handling stl-objects for stereo-lithography, object scanning, 3D-printing etc.

2.2 Programming style and code convention

GMlib is obviously template based so that it allows functions and objects to operate on generic types. But in the visualizer section, it must be type of float. All types in the core moduls of GMlib are templated except the angle class.

Most of the classes have both a header file (.h) and a source file (.cpp). The definitions are in the header files while the declarations are in source files. All the code is inside the GMlib namespace which helps separating the code from other libraries.

In order to differentiate the functions and variables, protected variables and private variables are named with an underscore in front of the name. When it needs more than one words, it will have underscore to connect them. Variable’s name doesn't have capital letters. For example: _max_elements. Most of the function’s name is starting with a small letter. When the name contains more than one words, each of the following word will start with a capital letter for distinguish the words. For example: getMaxSize.

When adding the name of including classes or libraries, it always separates in different groups with the order that local classes first, GMlib second and stl in the end.

In conclusion, this GMlib is readable and the programming style is designed for user friendly.

2.3 Application in the project

In this project, such classes: gmpplane, gmpbeziersurf, gmpplane and gmpsphere in the Parametrics Module have been used for implementing the ball, wall and the surface. Classes in the OpenGL module such as gmtexture, gmprogram, gmvertexshader, shaders, gmvertexbufferobject and gmfragmentshader as well as gmtexturerendertarget in Scene module are applied for rendering the objects. When we add the ball, wall and the collision objects, we use gmarray from the Core Module. We use gmpoint class from Core Module to locate the mouse position in gmlibwrapper class.

3 Description of the Nokia's Qt Framework

Qt is a cross-platform application and UI framework for developers using C++ or QML, a CSS and JavaScript like language. Qt is available under GPL v3, LGPL v2 and a commercial license. The letter 'Q' was chosen as the class prefix because the letter looked beautiful in Haavard's Emacs font. The 't' was added to stand for "toolkit", inspired by Xt, the X Toolkit. The company was incorporated on March 4, 1994, originally as Quasar Technologies, then as Troll Tech, and today as Trolltech[2].

3.1 Main structure/modules

Starting with Qt 4.0 the framework was split into individual modules. With Qt 5.0 the architecture was modularized even further. Qt is now split into essential and add-on modules (shown in Table 1 and Table 2)[3].

Table 1 Qt essentials[3]:

Module	**Description**
Qt Core	The only required Qt module, containing classes used by other modules, including the meta-object system, concurrency and threading, containers, event system, plugins and I/O facilities.
Qt GUI	The central GUI module. In Qt 5 this module now depends on OpenGL, but no longer contains any widget classes.
Qt Widgets	Contains classes for classic widget based GUI applications and the QSceneGraph classes. Was split off from **QtGui** in Qt 5.
Qt QML	Module for QML and JavaScript languages.
Qt Quick	The module for GUI application written using QML2.

Qt Quick Controls	Widget like controls for **Qt Quick** intended mainly for desktop applications.
Qt Quick Layouts	Layouts for arranging items in **Qt Quick**.
Qt Network	Network abstraction layer. Complete with TCP, UDP, HTTP, SSL and since Qt 5.3 SPDY support.
Qt Multimedia	Classes for audio, video, radio and camera functionality.
Qt Multimedia Widgets	The widgets from **Qt Multimedia**.
Qt SQL	Contains classes for database integration using SQL.
Qt WebKit	Qt's WebKit implementation and API.
Qt WebKit Widgets	The widget API for **Qt WebKit**
Qt Test	Classes for unit testing Qt applications and libraries.

Table 2 Qt add-ons[3]

Module	**Description**
Active Qt	Classes for applications which use ActiveX.
Qt Bluetooth	Classes accessing Bluetooth hardware.
Qt D-Bus	Classes for IPC using the D-Bus protocol.
Qt NFC	Classes accessing NFC hardware. Only officially supported on BlackBerry hardware so far (or N9 in the MeeGo port).
Qt OpenGL	Legacy module containing the OpenGL classes from Qt 4. In Qt 5 the similar functionality in Qt GUI is recommended.
Qt Positioning	Classes for accessing GPS and other location services. Split off from the Qt 4 Mobile module of Qt Location. Supported on Android, BlackBerry, iOS and Linux (using GeoClue).
Qt Script	Legacy module for scripting Qt application using ECMAScript/JavaScript. In Qt 5, using similar classes in Qt QML is recommended.
Qt Sensors	Classes for accessing various mobile hardware sensors. Used to be part of Qt Mobile in Qt 4. Supported on Android, BlackBerry, iOS, WinRT, Mer and Linux.
Qt Serial Port	Classes for access to hardware and virtual serial ports. Supported on Windows, Linux and Mac OS X.
Qt WebChannel	Provides access to Qt objects to HTML/Js over WebSockets.
Qt WebEngine	A new set of Qt Widget and QML webview APIs based on Chromium.
Qt WebSockets	Provides a WebSocket implementation.
Qt XML	Legacy module containing classes for SAX and DOM style XML APIs. Replaced with QXmlStreamReader and QXmlStreamWriter classes in Qt Core.
Qt XML Patterns	Support for XPath, XQuery, XSLT and XML Schema validation.

3.2 Programming style and code convention

In Qt, there is an alternative to the callback technique: use signals and slots[4]. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but we can always subclass widgets to add our own signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to subclass widgets and add your own slots so that you can handle the signals that you are interested in. As shown in Figure 1 [4], The signature of a signal must match the signature of the receiving slot.

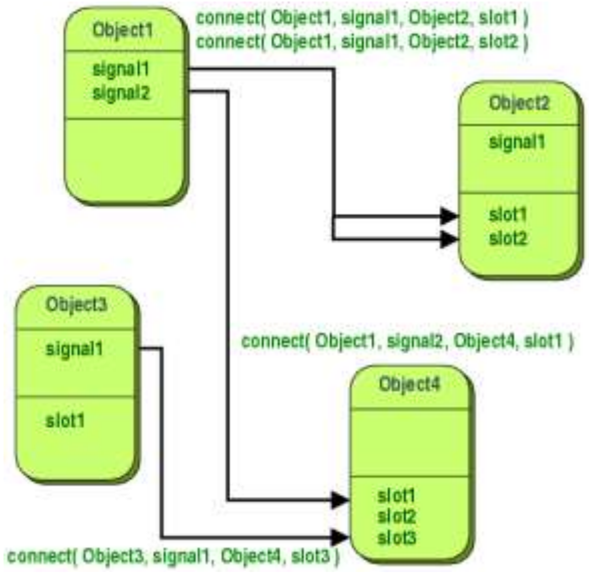
Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Figure 1 Illustration of signal and slot in Qt

It can be easily found that all the class prefix is Q. All classes that inherit from QObject or one of its subclasses (e.g., QWidget) can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to other objects. This is all the object does to communicate.

Qt uses "/" as a universal directory separator in the same way that "/" is used as a path separator in URLs. If you always use "/" as a directory separator, Qt will translate your paths to conform to the underlying operating system

3.3 Application in the project

In this project, many classes from Qt have been used for simulating. Table 3 described the classes which are applied in the whole program.

Table 3 Qt classes have been used in this project

Module	Description
QTimerEvent	Contains parameters that describe a timer event.
QRectF	Defines a rectangle in the plane using floating point precision

QMouseEvent	Contains parameters that describe a mouse event.
QKeyEvent	Describes a key event.
QWheelEvent	Contains parameters that describe a wheel event.
QDebug	Provides an output stream for debugging information.
QObject	It is the base class of all Qt objects.
QSize	Defines the size of a two-dimensional object using integer point precision.
QOpenGLContext	Represents a native OpenGL context, enabling OpenGL rendering on a QSurface.
QOpenGLFramebuffer Object	Encapsulates an OpenGL framebuffer object.
QOpenGLShaderProgra m	Allows OpenGL shader programs to be linked and used.
QQuickView	Provides a window for displaying a Qt Quick user interface
QQuickWindow	Provides the window for displaying a graphical QML scene
QQuickItem	Provides the most basic of all visual items in Qt Quick.
QOffscreenSurface	Represents an offscreen surface in the underlying platform.
QImage	Provides a hardware-independent image representation that allows direct access to the pixel data, and can be used as a paint device.
QImageReader	Provides a format independent interface for reading images from files or other devices.
QString	Provides a Unicode character string
QDirIterator	Provides an iterator for directory entry lists.
QGuiApplication	Manages the GUI application's control flow and main settings.

4 Description of the program and the implementation

In this section, we are going to illustrate more details about what I implemented in this project. It includes the concept of the game, structure of the application, description of the classes and collision system and user interface.

4.1 Concept/Gameplay

In this project, I simulated a ball and ball, ball and wall collision on a Bezier surface. Collisions are based on material-properties and the laws of physics. In the simulation, all the motions are designed without energy loss. Several cases of collision was considered and detected.

In order to make the game playable, I implemented that one of the balls can move up and down, left and right under the control of typing corresponding keys. There are also some other functions when clicking the keyboard, for example: R can stop the ball running, Alt can select the ball, Shift can move the ball, Control can rotate the ball. It also has mouse handling function. By pressing the mouse, can rotate the camera to look at the object in different angle.

4.2 Structure of the resulting application

In this project, the ball, wall and surface were referenced with specified precision in gmlibwrapper class. The functions which are generating for the simulation are called in the physical controller. Figure 2 shows the structure of the resulting application.

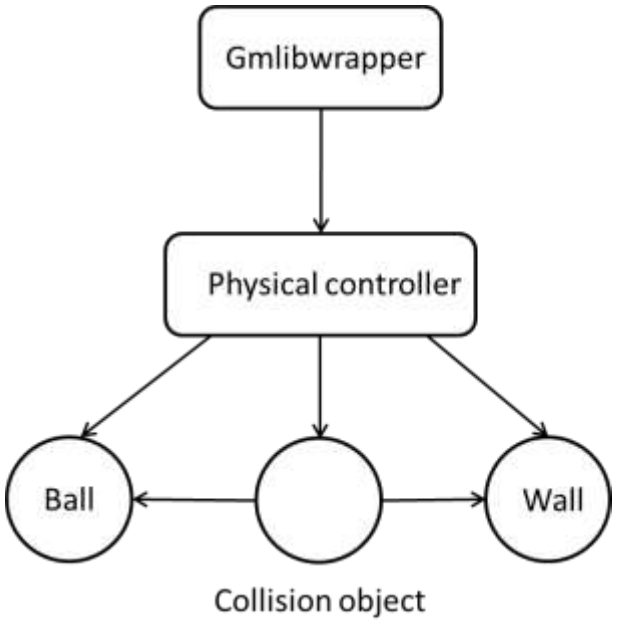

Figure 2 Illustration of the structure

4.3 Description of the classes

In this project, I implement five classes which are plane, physicalcontroller, gmlibwrapper ball and collisionobject.

- *Ball*

The ball class was inherited from gmpsphere class in GMlib library. In this class, all the parameters for describing a ball has been defined the ball. It is including:

_mass: the mass of the ball

_x: the value to store the time

_Surf: the plane object for defining the ground plane _p: the position of the ball

_q: the point of the plane

_velocity: the velocity of the ball

_u: the u coordinate

_v: the v coordinate

_normal: the normal of the wall

_ds: the step of the ball

In this class, it has updateStep() function to compute the ball moving steps. It also has four functions for controlling the ball moving up and down, left and right four directions. The localSimulate() function makes the ball moving by using rotateGlobal() and translateGlobal().

- *Plane*

This class inherited from the PSurf from the GMlib library. Wall class was inherited from GMlib PPlane class and it can be defined by one point and two vectors. The floor was computed by 4th order Bezier basis of 3rd degree Beinstein polynomials. Equation 4.1 shows the 3rd degree surface with 16 points.

$$S(u,v) = \begin{pmatrix}(1-u)^3 & 3u(1-u)^2 & 3u^2(1-u) & u^3\end{pmatrix}\begin{pmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \\ p_{13} & p_{14} & p_{15} & p_{16}\end{pmatrix}\begin{pmatrix}(1-v)^3 \\ 3v(1-v)^2 \\ 3v^2(1-v) \\ v^3\end{pmatrix} \quad (4.1)$$

- *collisionobject*

In this class, I defined a ball array, and a variable of wall which is PPlane inherited from GMlib. Variable x was defined as time. The Boolean variable bw is for checking the ball and the wall collided or not.

In this class, it has three construction functions which are preparing the ball collision , ball wall collision. Operator <, operator = are for operating the collisions. Use getPlane(), getBall(), getX() and isBW() functions for returning the values.

- *Physicalcontroller*

This class is used for storing objects, finding and handling collisions and computing the next ball movement coordinates. It is including PhysicallController(), insertObject(), findBallWallCollision(), findBallBallCollision(), collisionBallWall() and collisionBallBall() to bring out the collisions. The localSimulate() is for calling the functions of updateStep(dt) and coll_dectect(dt). In the coll_dectect() function, it puts all the collisions into arrays.

- *gmlibwrapper*

This class creates the scene objects and handles all the events in the scene, like mouse events and key events. It also creates the camera used in the scene. This class inserts the physicalcontroller class in the scene. Balls, floor and walls are given the values here.

4.4 Collision detection system

In this paragraph, we are going to discuss how the collision happens and how it achieved.

4.4.1 Ball and wall collision

In the physicalcontroller class, I have findBallWallCollision() to detect the ball and wall collision. When the ball collided with the wall, the distance between the center point of the ball to the wall is the radius of the ball *r*. From Figure 3, we can get the relationship:

$$< d + xds, n >= r \tag{4.2}$$

Where p_0 is the previous position of the ball center, d is the distance between p_0 and q, x is time, ds is the step of the ball, r is the radius of the ball, n is the normal vector. In additional, q is the (0, 0) point of the matrix when we define the wall surface.

From Equation 4.2, we can compute the *x to* update the steps. See Equation 4.3.

$$x = \frac{r - < d, n >}{< ds, n >} \tag{4.3}$$

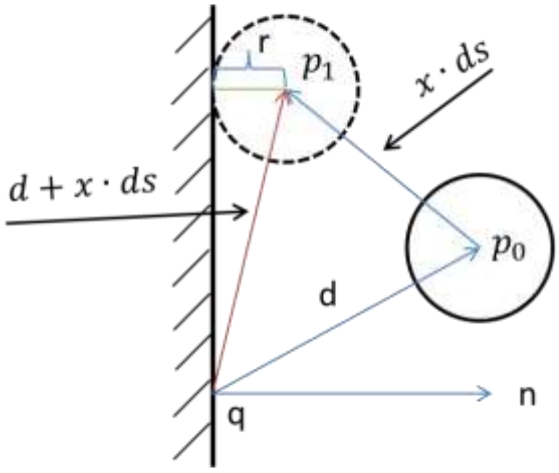

Figure 3 Geometry illustration of ball and wall collision

From Equation 4.3, we can see that if $< ds, n >$ is zero, the ball will slide parallel to the wall. If $< ds, n >$ is less than zero, the ball will move oppsite way from the wall. If $< ds, n >$ is more than zero, the ball will move towards to the wall. We also need to check the ball if it is intersected with the wall. If $< ds, n >$ is less than

r, it means that ball and wall intersecting. Then we need to translate the ball by a respective value and redefine $< d, n >$. In the programming, I defined $d =$ *mat[0][0]-pointBall* so that d has minus value. When I apply the Equation 4.3, I need to change the symbol – to +.

In the physicalcontroller class, I also have collisionBallWall() function to make the ball update the step after detecting the collision. When the ball collides with the wall, the direction of the new velocity can be found by adding normal direction and previous velocity direction (Figure 4). It can get the project of v on normal of the wall direction by doing inner product$< v, n >$. Then we need to scaler with n which is the normal. Since it is no energy loss, we can get the velocity by following formula (Equation 4.4):

$$v = v - 2 * < v, n > * n \tag{4.4}$$

We use minus symbol because the really direction of the velocity is opposite to the direction before collide.

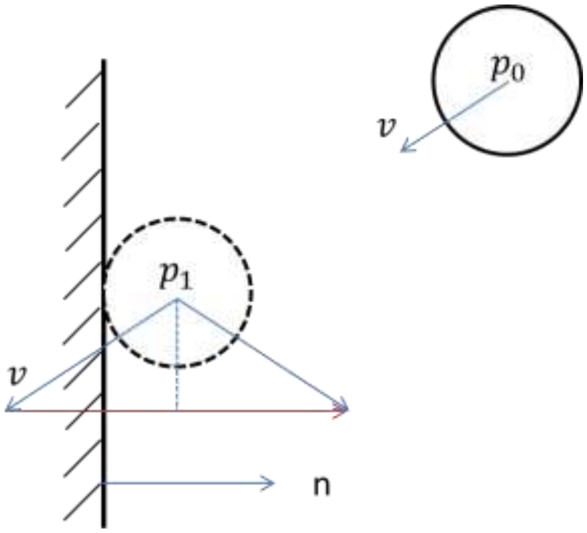

Figure 4 Illustration of velocity after ball colliding with the wall

4.4.2 Ball and ball collision

In the phsicalcontroller class, I make a findBallBallCollision() function to detect the ball and ball collision. When the two balls collide, it means that the two balls are moving to the position where the distance between the two ball s from the center point is the sum of the radius (See Figure 5). It can be expressed as following equation (Equation 4.5):

$$|p_1 + xds_1 - (p_2 + xds_2)| = |r_1 + r_2| \tag{4.5}$$

Let $dp = p_1 - p_2$, $ds = ds_1 - ds_2$ and $r = r_1 + r_2$, I multiply both side of the equation with itself, we get a new formula as following (Equation 4.6):

$$< dp + xds, dp + xds >= r^2 \tag{4.5}$$

After transformations we get a quadratic equation with unknown variable x (Equation 4.6).

$$< ds, ds > x^2 + 2 < ds, dp > x + < dp, dp > -r^2 = 0 \tag{4.6}$$

In this function, when the variable x has only one solution, it means that the two balls have no collision. If the x has two solutions, the collision happens. We choose the smaller value since we only need the first impact positon. If the function has no solution, it means the shortest distance between the two balls is too long. The collision cannot be happened. Also x should be more than zero and less or equal to one.

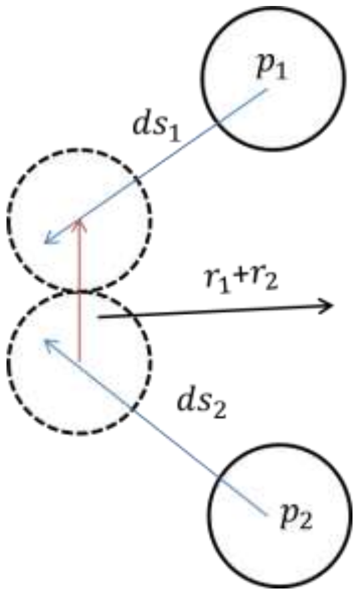

Figure 5 Ball and ball collision detect

Inside the findBallBallCollision() function, there is another checking for avoiding the two balls intersecting each other when the velocities are very slow. In order to fix it we need to compute the intersection distance and let each of the ball translate half of the distance from each other them (see Figure 6). Afterwards, we need to redefine the coefficients from the detecting function.

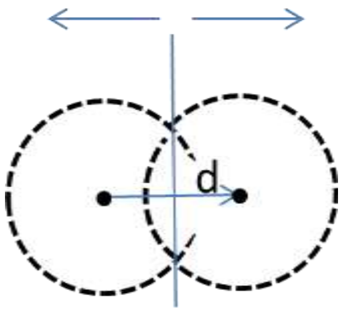

Figure 6 Two balls are intersecting

The translate distance can be compute as following way (Equation 4.7):

$$s = \frac{dp.getLength() + r_1 + r_2}{2} \cdot 1.02 \tag{4.7}$$

Beware of that, in order to make the translation happen before the two balls are collided, we need to scalar the distance slightly bigger than the exactly distance.

After the detect collision detecting, I need to deal with the collision. In the physicalcontroller class, it has collisionBallBall() function to make the ball update the step after collision. From Figure 7, we can see that the velocity of the balls can be decomposed by two directions: towards to the center point of the ball, and horizontal direction. Since the horizontal velocity will not change after impact, the problem can be simplified with computing the velocity towards to each ball center first. The new velocity will be composed with the two different direction velocities.

We assumed that the mass of the two ball are m_1 and m_2 respectively. The initial velocity for the two balls in the center line is v_{d1} and v_{d2}, after impact, the velocity are $v_{d1}{'}$ and $v_{d2}{'}$. According to the physic laws that conservation of energy and conservation of momentum, we got following relations (Equation 4.7):

$$\begin{cases} m_1(v_{d1} - v_{d1}{'}) = m_2(v_{d2} - v_{d2}{'}) \\ m_1(v_{d1}{}^2 - v_{d1}{'}^2) = m_2(v_{d2}{'}^2 - v_{d2}{}^2) \end{cases} \tag{4.7}$$

By solving this equation, we got the velocity, as a scalar, of two balls in the direction after the collision $v_{d1}{'}$ and $v_{d2}{'}$ respectively. The resulting equations are (Equation 4.8 and equation 4.9):

$$v'_{d1} = \frac{m_1 - m_2}{m_1 + m_2} v_{d1} + \frac{2m_2}{m_1 + m_2} v_{d2} \tag{4.8}$$

and

$$v'_{d2} = \frac{m_2 - m_1}{m_1 + m_2} v_{d1} + \frac{2m_1}{m_1 + m_2} v_{d2} \tag{4.9}$$

and the velocity for balls after impact is given by the compose of velocity vector in two directions(Equation 4.10 and equation 4.11)[5] :

$$v'_1 = v_{n1} + \frac{v'_{d1}}{|d|} d \tag{4.10}$$

$$v'_2 = v_{n1} - \frac{v'_{d2}}{|d|} d \tag{4.11}$$

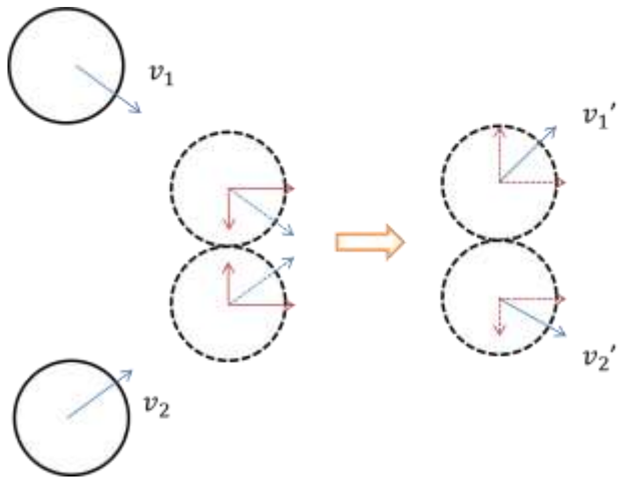

Figure 7 Ball and ball collision update

4.5 ODE-solver/simulation (used)

4.5.1 Theory of plan

In this project, we only use the polygons which are complex. It is illustrated in Figure 7.

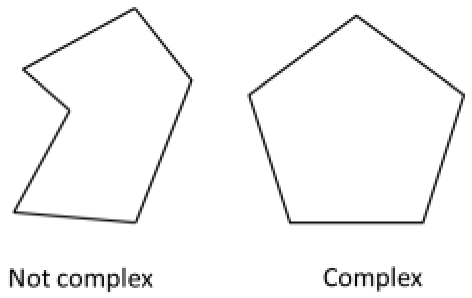

Figure 8 Polygons

When we define a flat plan, we need one point and two vector in the coordinate. The plane can be defined in following form(Equation 4.12):

$$S(u,v) = p + \binom{u}{v}(v_1, v_2) = p + uv_1 + vv_2\ , u \in [0,1], v \in [0,1] \qquad (4.12)$$

In this project, it was using 16 points smooth surface as the floor. And the wall was a plane inherited from GMlib.

Figure 9 shows the second degree surface with 9 points.

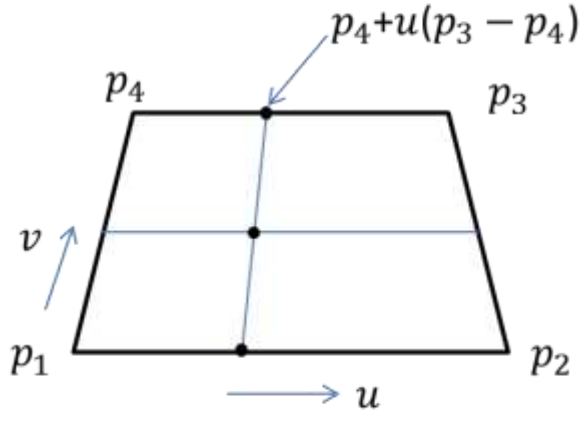

Figure 9 Define a smooth surface with 9 points

4.5.2 Simulation steps

In the ball class, function updateStep() and localSimulate() are using for the simulation. The algorithm is based on an iterative process finding the next local time step, and based on the process described as follows.

When the ball is moving on the surface, the distance can be described as following formula (Equation 4.13):

$$dS = p_0 + uv_1 + vv_2 + rn - p \tag{4.13}$$

Where $p_0 + uv_1 + vv_2$ means the point on the surface, *n* is the normal, *p* is the old position. When the closest point plus *r*n*, it will get a new position.

Therefore we can make the ball moving step in two steps. First we assume that it moves because of the gravity. The moving distance can be found in following formula (Equation 4.14):

$$ds = dt * velocity + 0.5 * dt * dt * g \tag{4.14}$$

As shown in Figure 10, the ball first arrived in the position under the surface. The new position will be *p = getPos() + ds*, where getPos is the initial position.

Secondly, we need to rotate the ball on the surface. So I find the closest point to the surface using function getClosestPoint() in GMlib. In order to move the ball on the surface, we use evaluate() to get a DMatrix from GMlib, and get the normal of the surface by using $normal\ =\ m[0][1] \times \mathrm{m}[1][0]$.

So the moving step will be shown in Equation 4.15:

$$dS = m[0][0] + r * n - getPos() \tag{4.15}$$

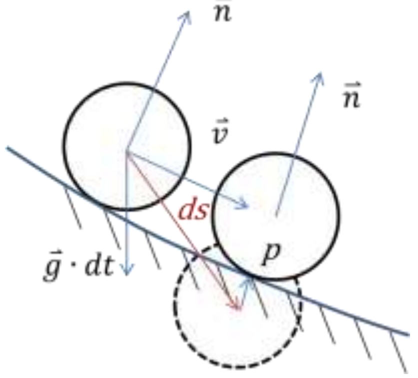

Figure 10 Diagrammatic drawing of ball moving

The velocities of the balls are changing when moving on the surface. As shown in Figure 11, we can compute the velocity based on the gravity, so we have *velocity+=dt*g*. Then we got the velocity which is according to the surface by using *velocity -= (normal*velocity)*normal*.

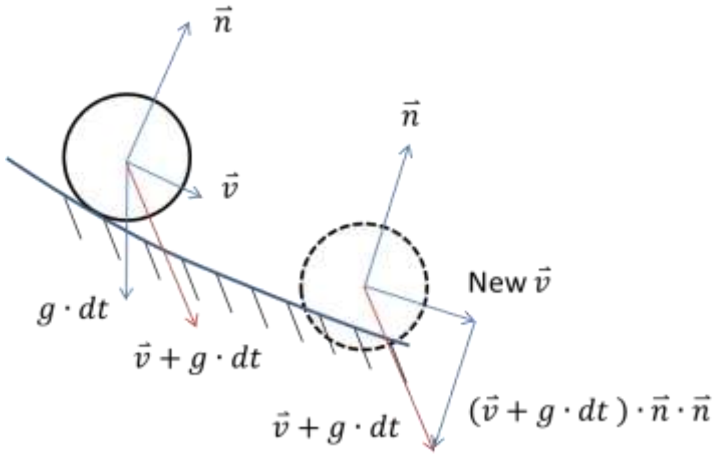

Figure 11 Diagrammatic drawing of velocity updating

In the ball class, it has localSimulate for the ball moving on the surface when the time being. In the physicalcontroller class, it has interacting objects for simulation. The localSimulate function deals with update steps of the ball and finding , handling collisions.

4.6 User interface

In the gmlibwrapper class, it is already exist some mouse handling for the user interface.

- R: Pause simulation/program. Press r again to resume simulation/program.
- Mouse wheel: Zoom in and out.

- Right mouse button (click): Select object (terrain, wall) and follow (selected ball).
- Right and left mouse button (hold): Move camera around scene.
- Alt + Right mouse button: Select object(s) in the scene
- Alt + Right mouse button, Ctrl + Right/Left mouse button: Select object(s). Rotate the object(s).
- Alt + Right mouse button, Shift + Right/Left mouse button: Select object(s). Translate them in response to the camera angle.

In this project, I implement four buttons to control the ball to move in four directions. These are located in the keyPressed in gmlibwrapper class. Then I add one of the balls can be controlled. In the ball class, I add four functions to bring out the ball moving. The main idea is when pressing the key; the controlled ball will change velocity accordingly.

The final appearance is shown in Figure 12. In order to make it better looking, I change the visualizer of controlled ball from other balls. I add picture on the wall so that the wall has brick appearance. Be aware of that each texture function only can be used by one object even it is adding the same picture. So I make a container for putting the same picture. Use push_back from stl for adding the same pictures. I also set the project camera color to be black.

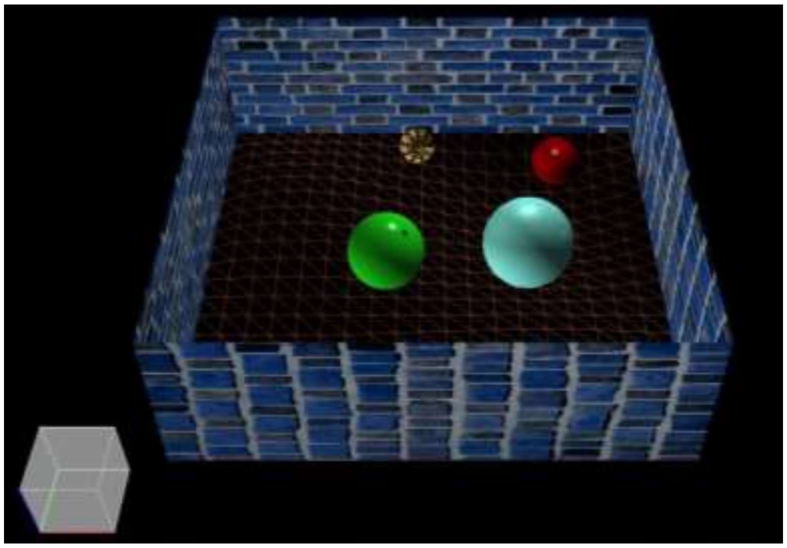

Figure 12 Demo

5 Conclusion and future work

In this project, ball-ball and ball-wall collisions are simulated. A 16 point Bezier surface was created for the balls rolling on. Some situation was considered when the collision failed. For example, I make the ball and wall intersection checks to avoid the ball go through the wall when the velocity is really slow. Also I checked ball and ball intersection when the velocity is very slow, and make them translate in required distance. But there are still some problem didn't fix yet. For example, the ball might make a weird step go outside the wall and come back or go through the wall disappeared when the ball is having ball-wall and ball-ball collision at the same time.

For future work, I am going to find the solution to solve the ball disappear problems and doing following implementation:

- Make more complicate terrain by defining the surface with more points.
- Implement friction on the floor instead of using a friction free surface.
- Make a dome for this box, add camber for the wall and floor so that the balls can go up to the dome.
- Make the ball can jump on the floor.
- Control the ball walking up through the wall to the dome.

REFERENCES

1. Laksâ, A., Blending technics for Curve and Surface constructions. 2012.

2. Summerfield, J.B.M., C++ GUI Programming with Qt 4. 2006: Prentice Hall.

3. Qt Documentation. Available from: https://doc.qt.io/qt-5/signalsandslots.html.

4. "Qt - About Us". Available from: https://doc.qt.io/qt-5/signalsandslots.html.

5. Arne Laks°a, B.B., Simulating rolling and colliding balls on freeform surfaces with friction.