Министерство образования и науки РФ

Нижегородский государственный университет им. Н.И. Лобачевского

Центр дополнительного профессионального образования

Программа дополнительного профессионального образования

«Профессиональное программирование»

Выпускная работа
«Моделирование Игры Бильярд «Восьмерка» в среде Qt Creator»

Выполнил:

Родин Е.В.

Руководитель:

к.ф.-м.н., доцент

Городецкий С.Ю.

Нижний Новгород

2021

Оглавление

1 Введение

В рамках курса профессионального программирования были получены достаточные знания в алгоритмизации и структурах данных для попытки реализовать симуляцию игры Бильярд «Восьмерка». В качестве среды разработки был Qt creator обладающий достаточными графическими возможностями и средствами самого языка, в качестве которого был выбран C++. Для составления физико-математической модели достаточными были материалы из «Справочника по Физике Для Студентов Высших Учебных Заведений» и «Справочника по Высшей математике и аналитической геометрии»

2 Постановка задачи

Перед началом работы ставилась задача написания 2D приложения с простым интерфейсом для симуляции игры в Пул. В приложении не планировалось использование сетевых протоколов и поддержки многопользовательской игры.

Основной задачей был написание производительного кода, способного обсчитывать передвижение 16 объектов на плоскости с высокой точностью и отрисовка графической сцены с частотой 60 кадров/сек.

3 Общие принципы работы приложение

Перед пользователем на экране пристает пустой стол, при нажатии на кнопку «New Game» на столе появляется набор шаров и предлагается выполнить первый удар. Далее, согласно правилам классической «Восьмерки», после попадания в лузу первого игрового шара, назначается тип шаров для каждого игрока, и игра идет до того момента, пока в лузу на будет забит шар 8 (Черный шар).

В любой момент игры пользователь может очистить игровое поле и начать новую игру.

4 Физико-математическая модель

4.1 Модель стола

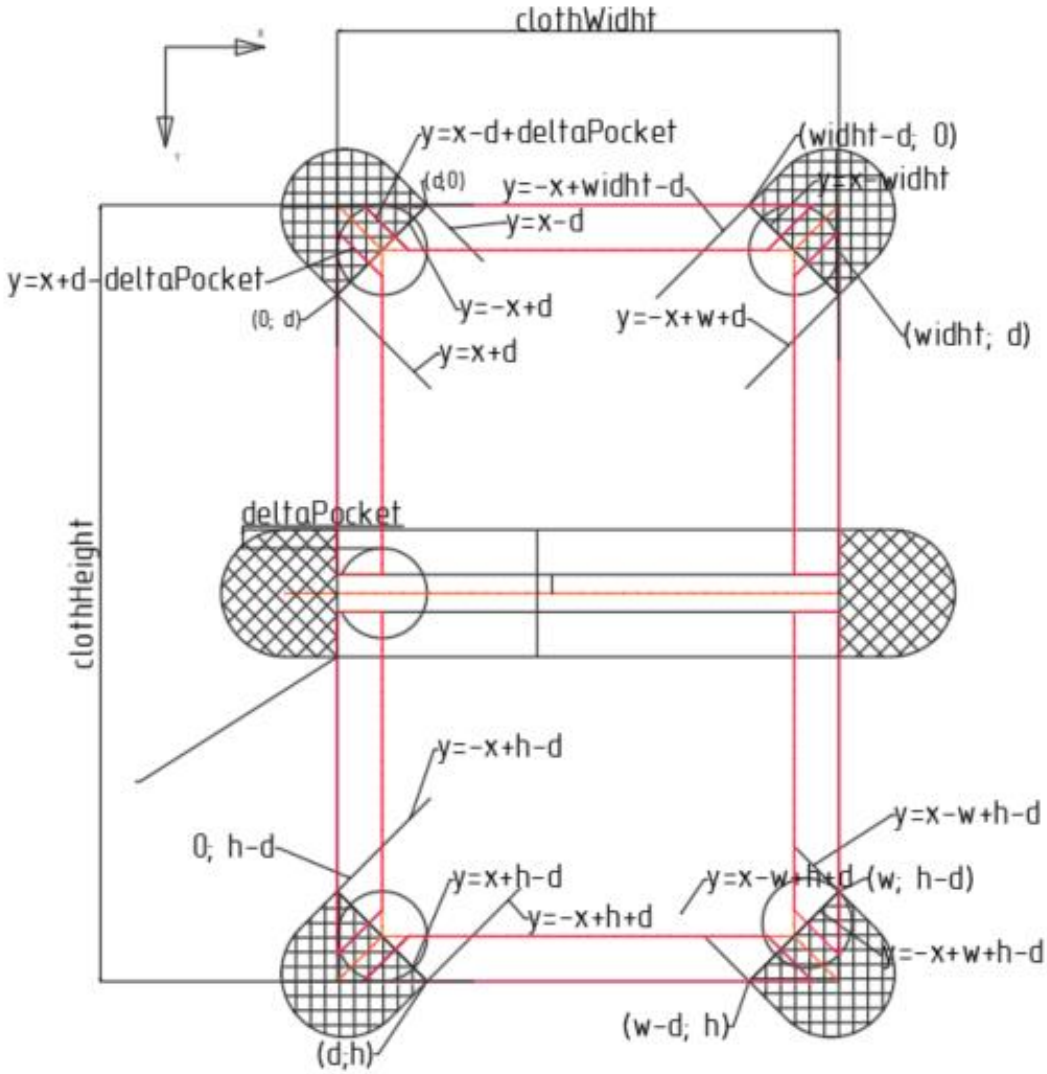
Стол описан множеством прямых, при пересечении который для шара вызываются соответствующие события:

-отражение от борта;

-попадание в лузу.

Прямые описаны уравнениями, коэффициенты и слагаемые которых определяются в зависимости от размера шара и размеры лузы. Данные параметры автоматически пересчитывают при изменении размеров шаров и луз.

Уравнения соответствующих бортов и луз показаны на рисунке.



4.2 Модель движения шара на столе

В качестве физической модели движения шара использовано уравнение движение, согласно которому за каждый промежуток deltaT координата меняется на V0.x*deltaT и V0.y*delatT при этом V0*=deltaV, а deltaV*=deltaForce. В в итоге имеем:

x=x+V0.x*deltaT;

y=y+V0.y*deltaT;

V0*=deltaV;

deltaV*=deltaForce;

Параметры изменения значение скоростей и ускорений подбирались опытным путем.

Движение шара происходит до момента, пока модуль изменения скорости по обеим координатам не станет меньше значения deltaVmin. После чего шару присваивается статус неподвижного. Это сделано для остановки шара и предотвращения его движения в обратном направлении, что неминуемо бы произошла из использования полинома в качестве функции, описывающей движение.

Движение шара за одну итерацию описывает метод move() класса Ball:

```
void Ball::move()
{
    moveBy(this->v0.x * MainWindow::deltaT, this->v0.y * MainWindow::deltaT);

    V vPrev = this->v0;
    this->v0 *= MainWindow::deltaV;
    MainWindow::deltaV*=MainWindow::deltaForce;
    if ((fabs(vPrev.x - this->v0.x) < MainWindow::deltaVmin) &&
            (fabs(vPrev.y - this->v0.y) < MainWindow::deltaVmin))
    {
        this->setMovingStatus(false);
    }
}
```

4.3 Модель соударения двух шаров

Модель соударения двух шаров строится на законе сохранения импульса и энергии при абсолютно упругом ударе:

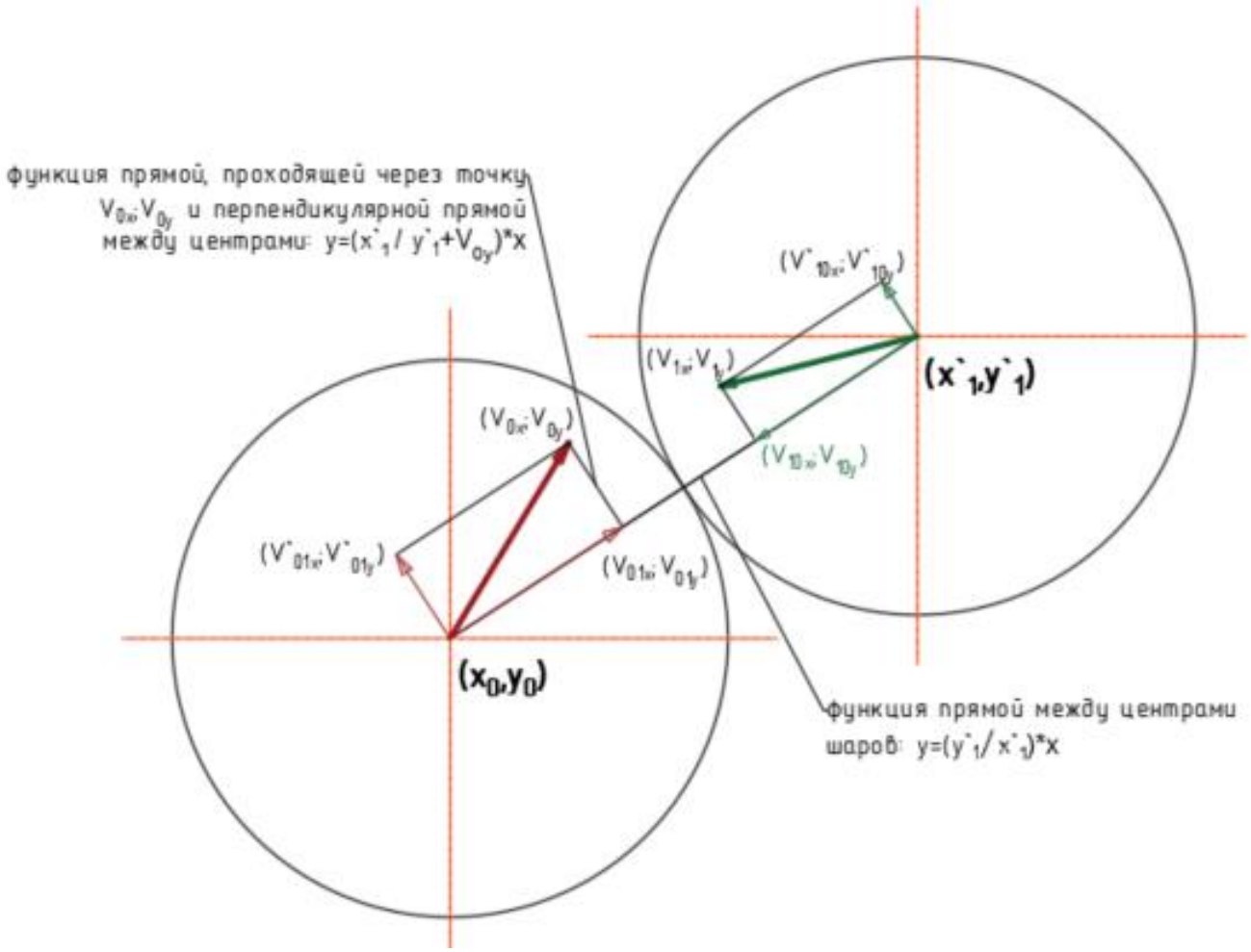
Закон сохранения импульса:

$$m_1 v_1 + m_2 v_2 = m_1 u_1 + m_2 u_2$$

Закон сохранения энергии:

$$\frac{m_1 v_1^2}{2} + \frac{m_2 v_2^2}{2} = \frac{m_1 u_1^2}{2} + \frac{m_2 u_2^2}{2}.$$

Исходя из этого, при ударе двух шаров одинаковой массы они обмениваются проекциями вектора скорости на прямую, соединяющую центры этих шаров:



Алгоритм вычисления конечных векторов скорости представлен ниже.

1. Принимаем за точку начала координат центр шара 1 (x_0,y_0).
2. Приводим глобальные координаты шара 2 к первому шару ($x`_1$,$y`_1$)

3. Получаем уравнение прямой $y = \frac{y`_1}{x`_1} * x$
4. Ищем уравнение прямой, перпендикулярной к данной, проходящей через точку:

$$y - y_1 = \frac{1}{k} \cdot (x - x_1)$$

5. Подставляем значение из функции и точку (V_{0x};V_{0y}) и получаем уравнение прямой:

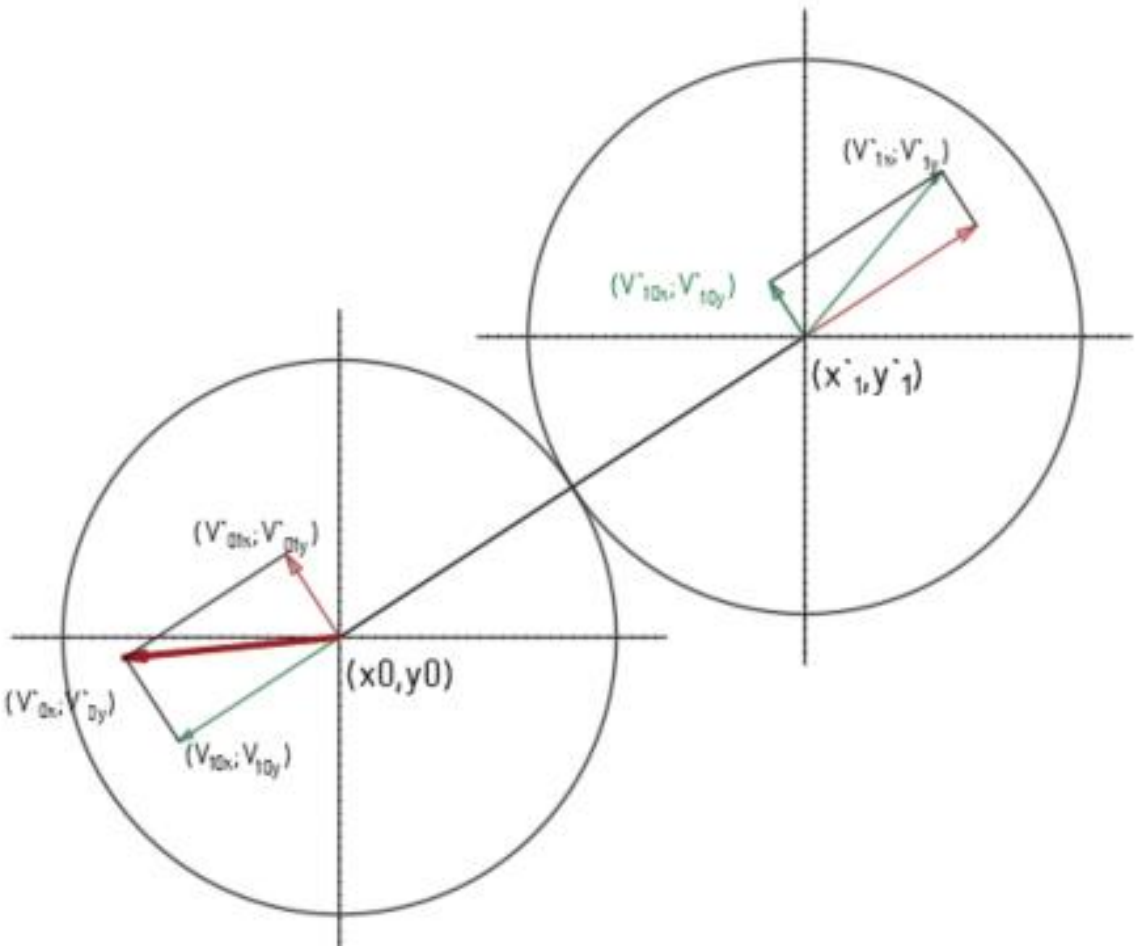
$$y = \left(\frac{x`_1}{y`_1} + V_{0y}\right) * x$$

6. Далее приравниваем данной уравнение к уравнению из п.3 для поиска точки пересечения этих прямых. И получаем формулу для вычисления координат проекции вектора скорости первого шара на прямую между центрами:

$$V_{01x} = \frac{\left(\frac{x`_1}{y`_1} * V_{0x} + V_{0y}\right)}{\left(\frac{x`_1}{y`_1} + \frac{y`_1}{x`_1}\right)}$$

$$V_{01y} = \frac{y`_1}{x`_1} * V_{01x}$$

7. Получив вектор скорости V_{01} мы находим перпендикулярный ему вектор $V`_{01}$, которые в сумме дают итоговый вектор V_0, путем вычитания V_{01} из V_0.
8. Аналогичные действия проводим для второго шара и ищем у него векторы V_{10} и $V`_{10}$
9. Обмениваем векторы V_{01} и V_{10} между собой.
10. Собираем векторы скорости для каждого шара. В итоге получаем следующую картину:



За реализацию этого алгоритмы в программе отвечает метод:

void Ball::**collisionTwoBalls**(Ball &other);

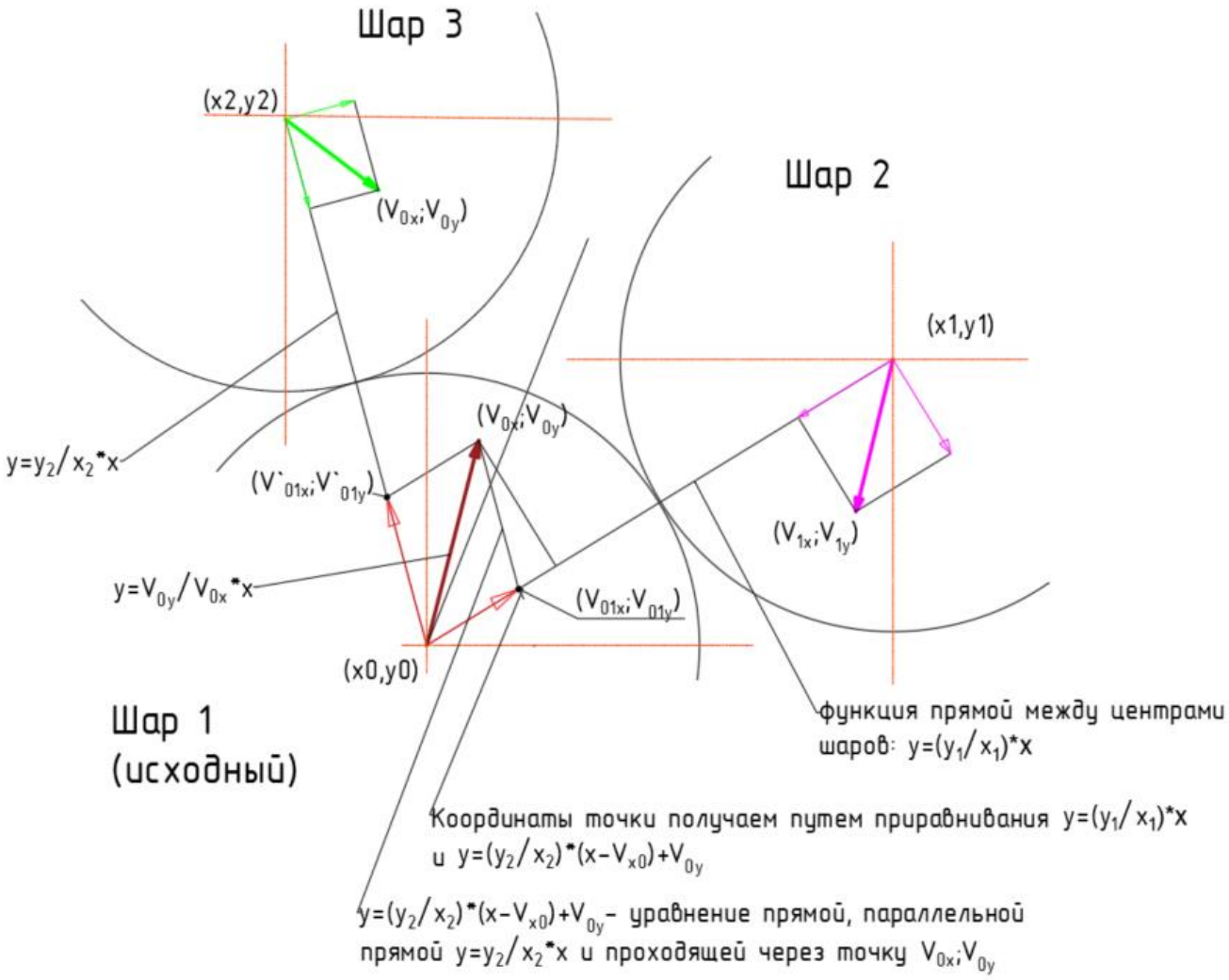
```
void Ball::collisionTwoBalls(Ball &other)
{
 //    qDebug()<<"this"<<this->x()<<this->y();
  //   qDebug()<<"other"<<other.x()<<other.y();
//
    Coord otherBallCoord(other.x() - this->x(), other.y() - this->y());
//координаты шара other относительно центр шара this
    if(abs(otherBallCoord.x)<0.001)
    {
        if(this->isTouch(this->v0.y, other.v0.y,otherBallCoord))
        {
            qSwap(this->v0.y,other.v0.y);
        }
    }
    else if(abs(otherBallCoord.y)<0.001)
    {
```

```
        if(this->isTouch(this->v0.x,other.v0.x,otherBallCoord))
        {
            qSwap(this->v0.x,other.v0.x);
        }
    }
    else{
        QPair<V,V> thisVProection=this->v0.makeProection(otherBallCoord);
        QPair<V,V> otherVProection=other.v0.makeProection(otherBallCoord);

        if (this->isTouch(thisVProection.first,otherVProection.first,otherBallCoord))
        {
            qSwap(thisVProection.first,otherVProection.first);
        }
        this->v0=this->v0.addTwoVector(thisVProection);
        other.v0=other.v0.addTwoVector(otherVProection);
    }
    this->setMovingStatus();
    other.setMovingStatus();
}
```

4.4 Модель соударения трех шаров

В процессе выполнения программы может наступить момент, когда шар одновременно сталкивается с двумя шарами. При этом последовательное применения двух методов для столкновения двух шаров дает неверный результат, так как импульс от шара будет передавать неравномерно(несправедливо). Такая ситуация наиболее вероятная при «разбитии пирамиды». В реальной жизни такая ситуация практически невозможна, так скорость передается не мгновенно, однако в компьютерной программе это имеет место быть.

Для обсчета такого случаю используем метод, который несколько отличается от вышеописанного.



Для примера рассмотрим поиск итоговых векторов скоростей для исходного шара и шара 2.

1. Принимаем за точку начала координат центр шара 1 (x_0,y_0).
2. Приводим глобальные координаты шаров 2 и 3 к первому шару (x_1,y_1) и (x_2,y_2)
3. Получаем уравнение прямых $y = \frac{y_1}{x_1} * x \quad y = \frac{y_2}{x_2} * x$
4. Далее ищем прямую, параллельную $y = \frac{y_2}{x_2} * x$ и проходящую через точку (V_{0x};V_{0y})

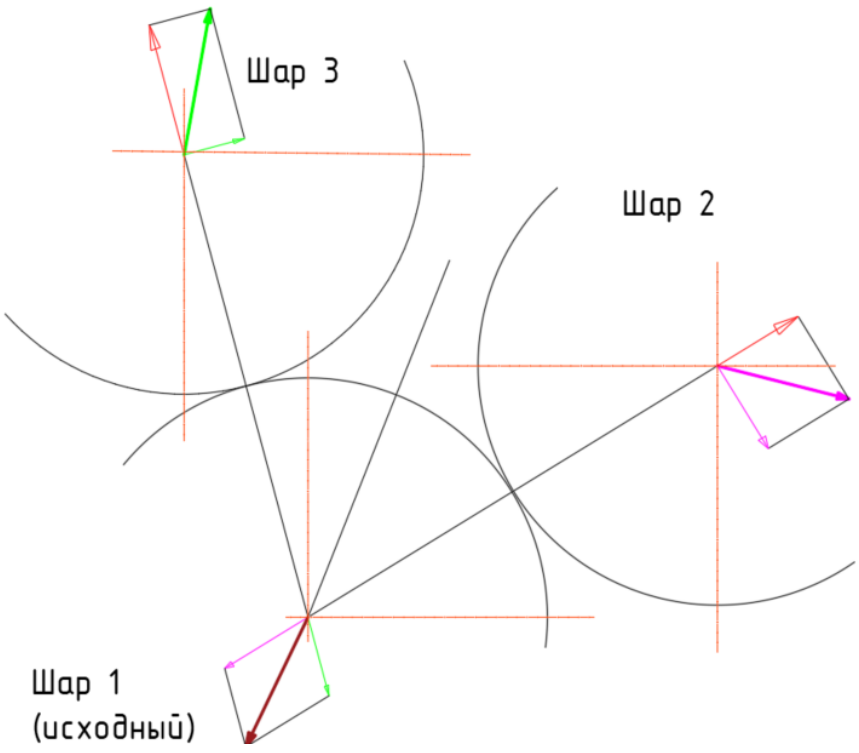
$$y = \frac{y_2}{x_2}(x - V_{x0}) + V_{0y}$$

5. Приравниваем прямые друг к другу и находим точку пересечения

$$V_{01x} = \frac{\left(V_{0y} - \frac{y_2}{x_2} * V_{0x}\right)}{\left(\frac{y_1}{x_1} - \frac{y_2}{x_{21}}\right)}$$

$$V_{01y} = \frac{y_1}{x_1} * V_{01x}$$

6. Получив вектор скорости V_{01} мы находим перпендикулярный ему вектор $V`_{01}$, которые в сумме дают итоговый вектор V_0, путем вычитания V_{01} из V_0.
7. Далее действуем аналогичным образом со случаем столкновения с двух шаров: обмениваем скорости вдоль межцентровой линии
8. Затем проделываем тоже самое с другим шаром
9. Далее так собираем вектора в один V_0, V_1 и V_2



За реализацию данного алгоритма отвечает метод

```
void Ball::collisionThreeBalls(Ball &first, Ball &second)
{
    Coord firstBallCoord(first.x() - this->x(), first.y() - this->y());
//координаты шара first относительно центр шара this
    Coord secondBallCoord(second.x() - this->x(), second.y() - this->y());
//координаты шара second относительно центр шара this


    V thisVProectionOnFirst = this->v0.makeRhombProection(firstBallCoord,
secondBallCoord);      //делаем проекцию скорости шара зис на прямую до центра
первого шара
    QPair<V,V> firstVProection = first.v0.makeProection(firstBallCoord);
//проекция скорости первого шара на пряму между центрами с ЗИС




    V thisVProectionOnSecond = this->v0.makeRhombProection(secondBallCoord,
firstBallCoord );    //тоже самое со вторым шаром
    QPair<V,V> secondVProection= second.v0.makeProection(secondBallCoord);


    qSwap(firstVProection.first,thisVProectionOnFirst);     //меняем скорости
местами
    qSwap(secondVProection.first,thisVProectionOnSecond);


    this->v0=this-
>v0.addTwoVector(qMakePair(thisVProectionOnFirst,thisVProectionOnSecond));
//собираем вектора скорости для всех шаров
    first.v0=first.v0.addTwoVector(firstVProection);
    second.v0=second.v0.addTwoVector(secondVProection);


    this->setMovingStatus();  //проверка на движения шаров
    first.setMovingStatus();
    second.setMovingStatus();
}
```

4.5 Модель удара о борт стола

При столкновении шара со бортом стола, исходя из того, что удар абсолютно упругий, умножаем соответствую из проекции скорости шара на соответствующую ось на -1 (разворачиваем шар в обратном направлении относительно это оси).

```
ball->v0.x*=-1;
```

или

ball->v0.y*=-1;

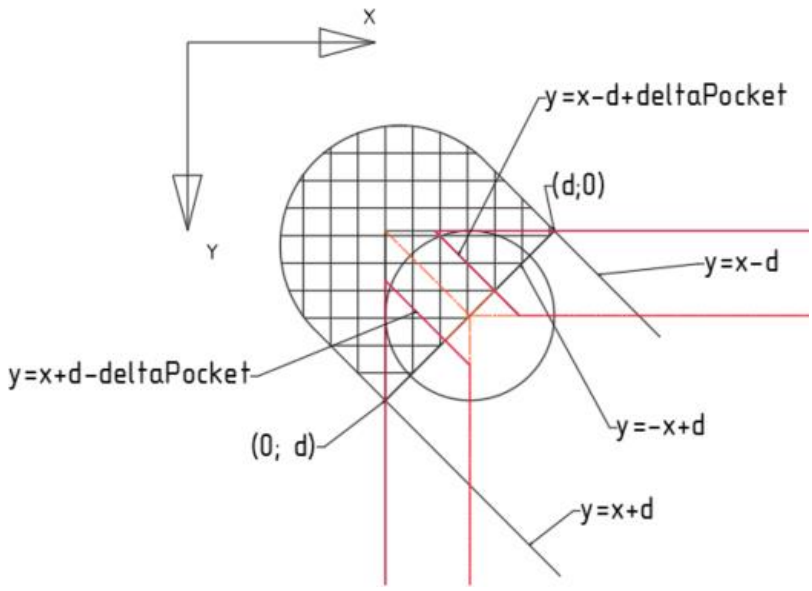
При этом учитывается находится ли шара в зоне лузы, если да, то отражение не происходит и шар продолжает движение без изменения, пока не пересече границы лузы.

За проверку столкновения с бортом в программе отвечает метод:

```
void MainWindow::boardCollision()
{
    double d=Ball::r*1*2;         //
    for(auto ball: Balls)
    {
        if (ball->isMoving())
        {
            bool midleArea = ball->y()>height/2-deltaPocket && ball-
>y()<height/2+deltaPocket;      //в зоне бортовых луз
            if(ball->x()<=Ball::r&&          //LEFT
                    ball->y() >=  ball->x()+d&&     //and top left pocket
                    ball->y() <= -ball->x()+height-d&&
                    !midleArea&&         //не в зоне луз
                    ball->v0.x<0)        //чтоб не зациклилось
            {
                ball->v0.x*=-1;
            }

            if(ball->y()<=Ball::r&&      //TOP
                (ball->y() <= ball->x()-d&&
                 ball->y() <= -ball->x()+width-d)&&
                 ball->v0.y<0)
            {
                ball->v0.y*=-1;
            }

            if(ball->x()>=width - Ball::r&&          //RIGHT
                    (ball->y() >= -ball->x()+width+d&&
                     ball->y() <=  ball->x()-width+height-d&&!midleArea)&&
                    ball->v0.x>0)
            {
                    ball->v0.x*=-1;
```

```
            }

            if(ball->y()>=height-Ball::r&&      //BOTTOM
                    (ball->y() >= -ball->x()+height+d&&
                     ball->y() >=  ball->x()-width+height+d)
                    &&ball->v0.y>0)
            {
                ball->v0.y*=-1;
            }
        }
    }
}
```

4.6 Попадание шара в лузу



Зоны луз задаются аналитически с помощью уравнений прямых в координатах стола. Пример функций для верхней левой лузы показан на рисунке.

За реализацию в программе отвечает метод

```
void MainWindow::pocketChech();
```

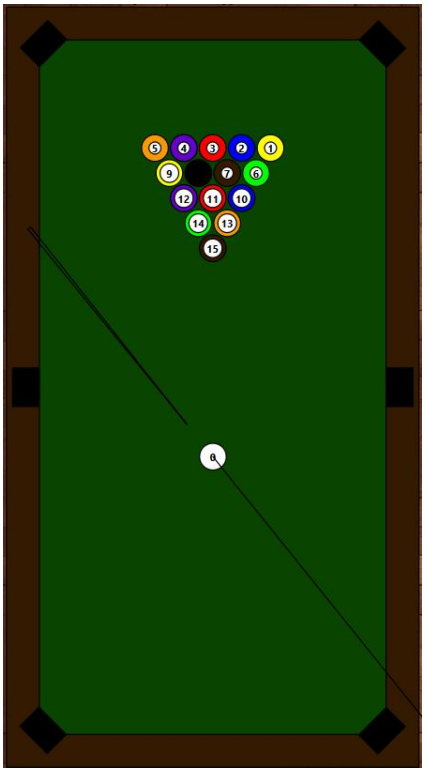
Помимо проверки попадания в лузу, метод так же определяет тип забитого шара и присваивает его соответствующему игроку или выставляет статус «Фол»

```
void MainWindow::pocketChech()
{
    double d=Ball::r*2;
    for(auto& ball:Balls)
    {
        double k=1;
```

```
        double delta = 3.;
        if(ball->isMoving())
        {
            if (
                ball->y() < (-ball->x()+d)*k +delta||
                ball->y() <  (ball->x()-width+d)*k+delta ||
                ball->y() >  (ball->x()+height-d )*k-delta||
                ball->y() > (-ball->x()+width+height-d)*k-delta
                    ||( (ball->x() < 0+delta || ball->x() > width-delta) &&
                        (ball->y() < height/2+pocketSize/2 &&
                         ball->y() > height/2-pocketSize/2))

                )
            {
                ballsInPocket.push_back(&ball);
                ball->setActiveStatus(false);
                QPointF posBall;

                if ((int)ball->getType()==(int)getActivePlayer().getBallType())
                {
                    getActivePlayer().raiseBallCount(1);
                    posBall=this->getActivePlayer().getNextBallShelfPoint();
                    ball-
>setPos(posBall+getActivePlayer().mapToItem(table,QPointF(0,0)));
                }
                else if ((int)ball->getType()==
(int)getWaitingPlayer().getBallType())
                {
                    getWaitingPlayer().raiseBallCount(1);
                    posBall=this->getWaitingPlayer().getNextBallShelfPoint();
                    ball-
>setPos(posBall+getWaitingPlayer().mapToItem(table,QPointF(0,0)));
                }
                else if(ball->getType()==Ball::Type::BLACK)
                {
                    if (getActivePlayer().isLastBall()){
                        getActivePlayer().raiseBallCount(1);
                        posBall=this->getActivePlayer().getNextBallShelfPoint();
                        ball->setPos(posBall+
                               getActivePlayer().mapToItem(table,QPointF(0,0)));
                    }
                    else{
                        posBall=table->getBlackBallWaitingPos();
```

```
                    ball->setPos(posBall);
                }
            }
            if(ball->getType()==Ball::Type::WHITE)
            {

                posBall=table->getWhiteBallWaitingPos();
                ball->setPos(posBall);

            }
            if (getActivePlayer().getBallType()==Player::BallType::NO_TYPE)
            {
                getActivePlayer().raiseBallCount(1);
                posBall=this->getActivePlayer().getNextBallShelfPoint();
                ball->setPos(posBall+
                   getActivePlayer().mapToItem(table,QPointF(0,0)));
            }
        }
    }
}
```

4.7 Удар кием



При получением всеми шарами статус isMoving()==false метод

void MainWindow::***mouseMoveEvent***(QMouseEvent *event) перехватывает координаты курсора мыши и вращает кий относительно центра белого шара. При зажатии левой кнопки мыши активизируется метод накопления силы удара кия

```
void Cue::gainPower()
{
    power+=(power<=maxPower)? deltaPower:0;
}
```

и при отпускании левой кнопки в метод

```
void MainWindow::setWhiteBallSpeed(double alpha, double impulse)
{
```

```
    whiteBall().setV(-impulse*cos(alpha),-impulse*sin(alpha));
}
```

Передаются проекции начальной скорости кия на соответствующие оси.

```
void MainWindow::mouseReleaseEvent(QMouseEvent *)
{
    if(isGameActive()){
        if(!ballsInMoving())
        {
            qDebug()<<"mouseReleaseEvent";
            cue->cueTimer->stop();
            setWhiteBallSpeed(cue->alpha,cue->power);
            whiteBall().setActiveStatus(true);
            qDebug()<<cue->alpha<<cue->power;
            whiteBall().setMovingStatus(true);
            setDeltaVDefault();
            timer->start(1000/60);
            getActivePlayer().setPreparingKickStatus(false);
        }
    }
}
```

5 Структура программы

5.1 Классы и типы данных

5.1.1 class Coord

Класс является вспомогательным классом для работы с координатами на плоскости. В нем для удобства перегружены нужные арифметические и логичечские операторы.

```
class Coord {
public:
    Coord(double x = 0., double y = 0.)
    {
        this->x = x;
        this->y = y;
    }
    double x;
    double y;
public:
    Coord operator+ (const Coord& other) const;
    Coord operator*(const int val)const;
    bool operator==(const Coord& other);
    bool operator<(const Coord& other);
    bool operator>(const Coord& other);

};
```

5.1.2 class Point

Класс является базовым классом для объектов шаров на столе

```
class Point {
public:
    Point() :x0(0), y0(0), v0(0, 0) {}
    Point(Coord coord);
protected:
    double t = 0.02;            //приращение времени
    double x0, y0;                    //начальные координаты
    V v0;              //начальная скорость
public:
    void moving(double deltaT);               //приращение координат
    virtual double distance(Point& other);    //измерение дистанции между шарами
    void setV(double Vx, double Vy);          //устанавливает начальную скорость
    void setV(V v);                   //устанавливает начальную скорость
```

```
    void setAlpha(double alpha);       //устанавливает угол

    friend class MainWindow;
};
```

5.1.3 class Ball

В классе прописан объект шара. Основные метод взаимодействи между шара, такие как столкновения с один шаром и с двумя шарами, проверка на касание с другим шаром, расчет квадрата дистанции и тд.

```
class Ball : public QObject, public QGraphicsItem, virtual public Point
{
    Q_OBJECT
public:

    enum class Type { PAINTED=1, STRIPPED,WHITE,BLACK };                    //тип
шаров:  сплошные, полосатые, белый и черный
    Ball();
    Ball(Coord coord=Coord(0,0) , int number=-1 , QColor color=Qt::yellow
            , Type type=Type::PAINTED, double scale=1, QGraphicsItem *
parent=nullptr);
    Ball(const Ball& other);
    Ball& operator=(const Ball& other);
    ~Ball();
    static void setBallRadius(double scale); //устанавливает радиус всех шаров
согласно сквозному масштабу
protected:

    QRectF boundingRect() const;
    void paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
QWidget * widget);
    QPainterPath shape() const;

protected:
    int number; //номер шара

    static double r;                                          //радиус

    QColor color;   //цвет
    Type type;      //тип шара: полосатый, сплошной , белый или черный
    bool active;              //шар на столе
    bool moving;        //шар в движении
```

```
    bool calculated;           //шар не просчитан за текущий цикл
    double scale;       //масштаб
    QFont * _font;      //шрифт шадписи

public:

    double distance(Ball& other);       //возвращает КВАДРАТ расстояние между
шарами
    bool isTouch(V currV, V otherV, Coord &otherCoord);      //проверка на касание
шаров. см аннотацию
    void collisionTwoBalls(Ball &other);      //обработка столкновения двух шаров
    void collisionThreeBalls(Ball &first, Ball& second);
        //обработка столкновения трех шаров

    bool isMoving();
    bool isActive();
    bool isCalculated();
    bool isGameBall();      //возвращает истину, если шар полосаты или сплошной

    QColor getColor();
    Type getType();
    QPointF centerPos();        //для отладки
    void setActiveStatus(bool active);
    void setMovingStatus(bool moving);      //установка статуса движущегося шара
вручную
    void setMovingStatus();
    void setCalculatedStatus(bool calc);
    void setSpeed(double, double);      //установка начальной скорости шара

    void setCoords(Coord const &coord);     //установка начальных координат шара
    void setCoords(double,double);

    void move();        //продвижения шара за один цикл

friend class MainWindow;
friend class Table;
};
```

5.1.4 class Player

Класс содержит в себе описания игрока. В то числе его имя, статус активности, тип шаров тд. В игре присутвуют два игрока.

```
class Player : public QObject, public QGraphicsItem
{
```

```
public:
    enum class BallType{ PAINTED=1, STRIPPED,WHITE,BLACK, NO_TYPE };
//    class Ball;
    Q_OBJECT
public:
    Player(QString name="Player", double ballRadius=57, QGraphicsItem *parent =
nullptr);
    bool isLastBall();      //возвращает истину, если игроку осталось забить
только 8 шар
    bool isActive();        //истина если игрок владеет ходом
    bool isPreparingKick();     //истина если сработал обработчик нажатия кнопки
мыши
    void addBallToShelf();      //добавляет шар на полку игрока
    void setBallType(BallType);     //закрепляет за игроком тип шара в текущей
игре
    void setPreparingKickStatus(bool);  //устанавливает статус подготовки к удару
    QString getName();
    BallType getBallType();
    int getBallCount();
    void raiseBallCount(int);   //увеличивает число забитых шаров
    QPointF getNextBallShelfPoint();        //возвращает координаты для следующего
шара на полке
    void setActiveStatus(bool);
protected:
    QRectF boundingRect() const;
    void paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
QWidget * widget);
 ////////VARIABLES
 protected:
    QFont *_font;
    QString name{};
    int ballCount{0};       //колчиество забитых шаров в игре
    QVector<QPointF> ballsPos;      //вектор с координатами шаров на полке
    QPointF firstBallPos{};
    BallType ballType{BallType::NO_TYPE};
    bool lastBallStatus{false};
    bool active{false};
    bool preparingKick{false};
    double ballRadius;

    friend class MainWindow;
    friend class Cue;
};
```

5.1.5 class Cue

Класс описывает объект кия

```
class Cue:public QObject,public QGraphicsItem{
    Q_OBJECT
public:
    Cue(QGraphicsItem *parent =nullptr);
    void setCuePower(double);   //устанавливает мощность удара кия
    double getPower();  //возвращает
    static double getMaxPower();
protected:
    QRectF boundingRect() const;
    void paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
QWidget * widget);
    ////////////////////////////////VARIABLE//////////////
protected:
    QVector<QPointF> vector;    //вектор контура кия
    double scale;   //сквозной масштаб
    double alpha;   //угол поворота
    QTimer  * cueTimer; //таймер для увеличения силы удара вовремя зажатой кноки
мыши
    double power{0.};   //сила удара
    static double maxPower;     //максимальная сила удара
    double deltaPower{50.};     //шаг приращения силы удара
public slots:
    void gainPower();       //увеличивает силу удара каждый тик таймера
    ///////////FRIENDS///////////////
    friend class MainWindow;
    friend class MyGraphicsView;

};
```

5.1.6 class Table

Класс объекта стола. Класс является родителем(владельцем) все отрисованных объектов на сцене. Все координаты объектов приводятся к началу координат игрового поля.

```
class Table : public QObject,public QGraphicsItem
{
    Q_OBJECT
public:

    Table(double maxHeight = 800, QObject *parent = nullptr);
    double getScale(); //установка масштаба
    double getClothWidht();     //возвращает ширину поля стола
    double getClothHeight();    // длину
    QPointF getWhiteBallWaitingPos();   //поле белого шара для отрисовки при
попадании в лузу
    QPointF getBlackBallWaitingPos();   //то же для черного шара
protected:
    QRectF boundingRect() const;
    void paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
QWidget * widget);

////////////////////////////////////VARIABLE///////////////

    QList<QPointF> vecInside;       //контур стола внешний
    QList<QPointF> vecOutside;      //внутренний
    QColor insideColor;
    QColor outsideColor;
    double tableHeight;         //размеры
    double clothWidht;
    double clothHeight;

    const double static defaultTableHeight;     //размеры по умолчнанию
    const double static defaultClothWidht;
    const double static defaultClothHeight;
    const double static defaultBallRadius;
    const double static defaultBoardSize;
    double  boardSize;

    double ballRadius;      //радиус шара
```

```
    double scale;       //сквозной масштаб

    QPointF blackBallWaitingPos=QPointF(350,100);
    QPointF whiteBallWaitingPos=QPointF(390,100);

    QPointF firstPlayerPos{};       //позиция для отрисовки игроков
    QPointF secondPlayerPos{};
};
```

5.1.7 Класс class MainWindow

В класс описаны основные методы и поля, отвечающие за отрисовку и логику игры.

```
class MainWindow : public QMainWindow
{
Q_OBJECT
public:
    MainWindow(QWidget *parent = nullptr, bool ii=false);
    ~MainWindow();

private:

    Ui::MainWindow *ui{nullptr};
    QGraphicsScene * scene{nullptr};  //main scene
    QGraphicsScene * sceneSub{nullptr};

    QList<Ball*> Balls;                                 //вектор всех шаров
    QTimer * timer{nullptr};        //main timer
    Cue *cue{nullptr};          //кий
    Table * table{nullptr};     //стол
    QVector<Pocket*> pockets;   //массив луз
    Player * player1{nullptr};      //игрок 1
    Player * player2{nullptr};      //игрок 2
    QVector<Ball**> ballsInPocket;      //вектор шаров забитых в лузу на этом ходу
    Player **activePlayer{&player1};    //указатель на активного игрока


    QVector<Coord> defaultBallsCoords;     //Вектор координат шаров по умолчанию в
треугольнике при расстановке
    static Coord firstBallPosition;                                     //координаты
первого шара


    double windowHeight{620};       //размер окна приложения для расчета общего
масштаба
```

```
    double height{0};       //высота стола
    double width {0};       //ширина стола
    double scale{1};        //текущий масштаб всех элементов
   Coord angleBallCoords{Coord(0,0)}; //координата первого шара с пирамиде

   static double deltaForce;        //изменение static double deltaV*=deltaForce;
   const static double deltaVDefault;         //коэффициент замедления скорости с
каждой итерацией
   static double deltaV;        //v*=deltaV
   static double deltaT;              //шаг времени для каждой итерации
   static double deltaVmin;     //минимальное изменение скорости для статуса шара
isMoving()
   double deltaPocket;          //зазор в лузе
   double pocketSize;           //ширина лузы



   bool ballsMoving{false};   //дивгаются ли шары
   bool gameActive{false};      //активна ли игра
   bool foul{false}; //статус фола
   bool whiteBallInPocket{false}; //забит белый шар
   bool blackBallInPocket{false};       //забит черный шар
   bool noBallsInPocket{true};  //не забито шаров
   bool firstHit{true};    //первый удар кием за партию
   bool ballsTouch{false};     //касание шара
   bool sameBallInPocket{false};  //в лузах шары одного типа
   bool noBallTouch{true};      //не было касания шаров
   Ball** firstTouchedBall{nullptr}; //Первый коснувшийся шар


   bool ii{false};  //
   double iiScale{0.3}; //
public:
   QPointF firstPlayerPos=QPointF(330,20);  //позиция для отрисовки объекта
первого игрока
   QPointF secondPlayerPos=QPointF(330,200);
   int TABLE_OFFSET_X=50;       //смещения координат
   int TABLE_OFFSET_Y=50;


   int SCENE_OFFSET_X=-120;
   int SCENE_OFFSET_Y=-10;


   int SCENE_SIZE_X =702;
   int SCENE_SIZE_Y =852;
```

```
public:


    bool isGameActive();
    bool isFoul();
    bool isTouched();
    bool iSii();
    bool isFirstHit();
    bool isWhiteInPocket();
    bool isBlackInPocket();
    bool isNoBallsInPocked();
    bool isSameBallInPocket();
    bool isNoBallTouched();
public:
    void setII(bool);
    void gameOver();
    void loseGame(Player*);
    void winGame(Player*);



    void setWhiteBallSpeed(double alpha, double impulse);
        //устанавливает скорость для битка и добавляет его первым в список
движущихся шаров
    void setWhiteBallPosition();              //установка позиции белого шара
    void setWhiteFoulPosition();         //установка белого шара в начальную
позицию
    void setBallsDefaultPosition();           //устанавливаем всем шарам
координаты из списка дэфолтных координат defaultBallsCoords
    void setCalculetedStatusFalse();    //устанавливает всем активным шарам статус
"не просчитаны"
    void calcBallsDistances();         //расчет текущих дистанци между шарами
    void boardCollision();       //проверка столкновения с бортом
    void moveBalls();        //просчет движения всех активных шаров
    void pocketChech();      //проверка попадания в лузу


    void checkPocketedBalls();  //проверка забиты в лузу шаров


    //////////////////
    void setNewGame();
    //////////////////
    void changeMove();       //меняет ход игрока при фоле
    void goOn();         //продолжение хода
    //////////////////
```

```
    void generateBallDefaultPosition();     //генерирует координаты шаров в
пирамиде
    void setGameScene();    //установка сцены
    void setTable();    //установка стола
    void setScale();    //установка сквозного масштаба
    void setWidht();    //установка размеров
    void setHeight();
    void setBallsScale();
    void setAngleBallCoord();      //уставнока координат углового шара
    void setStartView();    //установка вида пр запускек приложения
    void setBalls();        //добавления шаров на стол
    void setCue();      //добавление кия
    void setCuePosition();      //установка кия к белому шару
    void setCueRotation(QPointF point); // установка угла вращения кия
    void setPockets();  //установка луз
    void setPlayers();  //установка игроков на сцену
    void setPlayersActiveStatus();  //установка статуса игрокам

    void setGameActiveStatus(bool);
    void setFoulStatus(bool);
    void setWhiteBallInPocket(bool);
    void setBlackBallInPocket(bool);
    void setNoBallsInPocket(bool);
    void setTouchStatus(bool);
    void setFirstHitStatus(bool);
    void setSameBallInPocketStatus(bool);
    void setDeltaVDefault();
    void setFirstTouchedBall(Ball**);
    void setNoBallTouchStatus(bool);
    void setBallsInMoving(bool);
    bool ballsInMoving();


    /////////////////GET MAIN OBJECTS////////////
    Ball& getFirstTouchedBall();
    Ball &whiteBall();
    Ball &blackBall();
    Player& getActivePlayer();
    Player& getWaitingPlayer();

public slots:
```

```
    void pitch();        //просчет кадра

private slots:
    void on_NewGameBtn_clicked();
    void on_ClearBtn_clicked();
    void on_stopBalls_clicked();
    void on_IIMoveBtn_clicked();

protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent * );

friend class Ball;
friend class V;
friend class Player;
};
```

5.1.8 class Pocket

Класс лузы описывает геометрические размеры луз на столе.

```
class Pocket:public QObject, public QGraphicsItem
{
Q_OBJECT
public:
    Pocket(QPointF pocketPoint=QPointF(0,0), double alpha=0, double
ballRadius=57., double delta=1.44, QGraphicsItem *parent = nullptr);
protected:
    QRectF boundingRect() const;
    void paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
QWidget * widget);
protected:
    //////VARIABLES///////
    double deltaPocket;
    double alpha;
    double ballRadius;
    QPolygonF rect;
    QPointF pocketPoint;
    QVector<QPointF> vec;
};
```

5.2 Насчет кадра

За расчет кадра отвечает метод класса MainWindow void MainWindow::**pitch**(), связанный с таймером QTimer * timer

Реализация:

```
void MainWindow::pitch()
{
    for(int i =0;i<40;++i)
    {
        setCalculetedStatusFalse();     //выставляем статус "просчитан" в ложь для
всех шаров
        pocketChech();      //проверяем падение в лузу
        boardCollision();       //проверяем столкновение с бортом
        calcBallsDistances();       //проверяем дистанцию между шарами
        moveBalls();    //продвигаем все находящиеся в движении шары на одну
итерацию
    }
    scene->update();    //отрисовываем сцену

    if (!ballsInMoving())       //Обработка завершения движения шаров
    {
        timer->stop();      //останавливаем таймер
        checkPocketedBalls();   //проверяем забитые шары
        setCuePosition();   //устанавливаем кий на белый шар
    }
}
```

На каждый тик таймера (60 раз в секунду) происходит дополнительно определенное количество итераций за счет использования цикла for. Это необходимо для обеспечения нужной точности движения и экономит ресурсы системы на излишнюю отрисовку сцены с ее объектами.

В каждом цикле вызова метода **pitch**() проверяется падение шара в лузу, столкновение с бортом, высчитывается дистанция для всех шаров и происходит продвижение всех шаров на один шаг deltaT.

Далее проверяет двигается ли хотя бы один шар на столе, если движение прекращается, таймер останавливается и запускается метод проверки упавших шаров в лузу checkPocketedBalls() в котором выставляются соответствующие флаги, влияющие на дальнейшее течение игры (фол, продолжение хода и завершение игры).

5.3 Метод расчета дистанции до шаров void MainWindow::calcBallsDistances()

В методе последовательно проверяются все движущиеся активные шары на квадрат дистанции между другими шара. При уменьшении этого параметра меньше квадрата диаметра шаров, вызывается метод столкновения двух или трех шаров в зависимости от того, сколько шаров находится в диапазоне досягаемости шара. При просчете шара ему присваивается статус “calculated”, чтобы не просчитывать его внутри вложенного цикла.

```
void MainWindow::calcBallsDistances()
{
    bool firstTouch=false;  //было ли касание с шаром
    QVector<Ball *> tmp;    //вектор шаров с которыми столкнулся this
    for(auto& currBall:Balls)   //для всех шаров
    {
        if(!currBall->isCalculated() && currBall->isMoving())   //если шар еще не просчитан и движется
        {
            currBall->setCalculatedStatus(true);    //ставим флаг о просчете, что бы не считать второй раз
            for(auto& otherBall:Balls)      //для всех непросчитанных шаров
            {
                if(!otherBall->isCalculated()   //ставим флаг о просчете
                    && currBall->distance(*otherBall) <
                        1.01 * 4 * Ball::r * Ball::r )//сравниваем квадрат дистанции с квадратом диаметра
                {
                    tmp.push_back(otherBall);  //если меньше, кидаем шар в вектор столнувшихся
                }
            }
            if(tmp.size()==1){      //если коснувшихся шаров 1
                if (!firstTouch)        //ставим метку касания первого шара
                {
                    firstTouch=true;
                    setFirstTouchedBall(&Balls[tmp[0]->number]);
//устанавливаем первый коснувшийся шар
                                //это необходимо для логики игра и продолжения хода
                }
                currBall->collisionTwoBalls(*tmp[0]);       //вызываем метод столкновения двух шаров
                if (!isTouched()){
```

```
                    setFirstTouchedBall(&Balls[tmp[0]->number]); //Устанавливаем
первый коснувшийся шар
                    setTouchStatus(true);
                }


            }
            if(tmp.size()==2){  ////если коснувшихся шаров 2
                if (!firstTouch)
                {
                    firstTouch=true;
                    setFirstTouchedBall(&tmp[0]);
                }
                currBall->collisionThreeBalls(*tmp[0],*tmp[1]);
                if (!isTouched()){
                    setFirstTouchedBall(&tmp[0]); //Устанавливаем первый
коснувшийся шар
                    setTouchStatus(true);
                }
            }
            tmp.clear();        //чистим ветор
        }
    }
     }
```

5.4 Логика игры

Логика игра строится на правилах классической игры «Восьмера» доступных по ссылке

https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D0%BB-8#%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5_%D0%BF%D1%80%D0%B0%D0%B2%D0%B8%D0%BB%D0%B0

За логику в приложении отвечает метод void MainWindow::**checkPocketedBalls**()

На основе флагом и их сочитаний вызывается либо метод смены хода void MainWindow::**changeMove**() или метод продолжения хода игрока void MainWindow::**goOn**().

Так же попадании в лузу шара 8 вызывается либо метод, устанавливающий выигрыш текущим игроком winGame((**activePlayer*));

либо его проигрыш loseGame(**activePlayer*);

Реализация:

```
void MainWindow::checkPocketedBalls()
{
    bool paintedBall=false;
    bool strippedBall=false;
    ///////////////FLAGS//////////////////////
    if (!ballsInPocket.isEmpty()) //выставляем флаги если есть забитые шары
    {

        setNoBallsInPocket(false);
        for(auto ball:ballsInPocket)
        {
            if (*ball==&whiteBall()){
                setWhiteBallInPocket(true);
            }
            if (*ball==&blackBall()){
                setBlackBallInPocket(true);
            }
            if((*ball)->getType()==Ball::Type::PAINTED){
                paintedBall=true;
            }
            if((*ball)->getType()==Ball::Type::STRIPPED){
                strippedBall=true;
            }
        }
        if(paintedBall^strippedBall){
            setSameBallInPocketStatus(true);
        }
    }
    else //ни одного шара не забито
    {
        setNoBallsInPocket(true);
    }
    ///////////GAME LOGIC///////////
    if(isFirstHit())
    {
        bool foundGameBall=false;
        if(isSameBallInPocket()&&!isFoul())
        {
            getActivePlayer().setBallType((paintedBall) ?
Player::BallType::PAINTED : Player::BallType::STRIPPED);
            getWaitingPlayer().setBallType((!paintedBall) ?
Player::BallType::PAINTED : Player::BallType::STRIPPED);
            setFirstHitStatus(false);
```

```
        }

        else if(isNoBallsInPocked())
        {
            setFirstHitStatus(true);

        }
        if (isBlackInPocket())
        {
            setFoulStatus(true);
            loseGame((*activePlayer));
        }

        if (isWhiteInPocket())
        {
            setFoulStatus(true);
            changeMove();
            setWhiteFoulPosition();
//              void goOn();
        }
        for(auto ball:ballsInPocket)
        {

            if (!foundGameBall && (*ball)->isGameBall()){
                foundGameBall=true;
                getActivePlayer().setBallType((Player::BallType)(*ball)-
>getType());
                setFirstHitStatus(false);
                break;
            }
        }
        if(firstTouchedBall==nullptr)
        {
            setNoBallTouchStatus(true);
            setFoulStatus(true);
        }
        else if(getFirstTouchedBall().getType()==Ball::Type::BLACK){
            setFoulStatus(true);
        }
        else{
            setNoBallTouchStatus(false);
```

```
        }
        //////////////LOGIC for FIRST shots
        if (isBlackInPocket()){
                loseGame((*activePlayer));
        }
        else //если черный не забит
        {
            if (isWhiteInPocket())
            {
                setFoulStatus(true);
                changeMove();
                setWhiteFoulPosition();
            }
            else{
                if(isFoul())
                {
                   changeMove();
                   setWhiteFoulPosition();
                }
                else if (isNoBallsInPocked())
                {
                    changeMove();
                }
            }
        }

    } //endif
    else
    {
        if(firstTouchedBall)
        {
            setNoBallTouchStatus(false);
            if((int)((*firstTouchedBall)-
>getType())!=(int)(getActivePlayer().getBallType()))
            {
                setFoulStatus(true);
            }
            else{
                setFoulStatus(false);
            }
        }
        else{
```

```
            setFoulStatus(true);
            setNoBallTouchStatus(true);
        }
        //////////////LOGIC for not firstd shots
        if (isBlackInPocket()){
            if((*activePlayer)->isLastBall() && !isFoul()){
             winGame((*activePlayer));  //если забит черный шар и он был последний
и нет фола, то побеждает актив плэйер
            }
            else
            {
                loseGame((*activePlayer));
            }
        }
        else //если черный не забит
        {
            if (isWhiteInPocket())
            {
                setFoulStatus(true);
                changeMove();
                setWhiteFoulPosition();
            }
            else{
                if(isFoul())
                {
                   changeMove();
                   setWhiteFoulPosition();
                }
                else if (isNoBallsInPocked())
                {
                    changeMove();
                }
            }
        }

    } //end else

    if (isBlackInPocket()&&!(*activePlayer)->isLastBall())
    {
        loseGame(*activePlayer);
    }
    if (isBlackInPocket()&&(*activePlayer)->isLastBall())
```

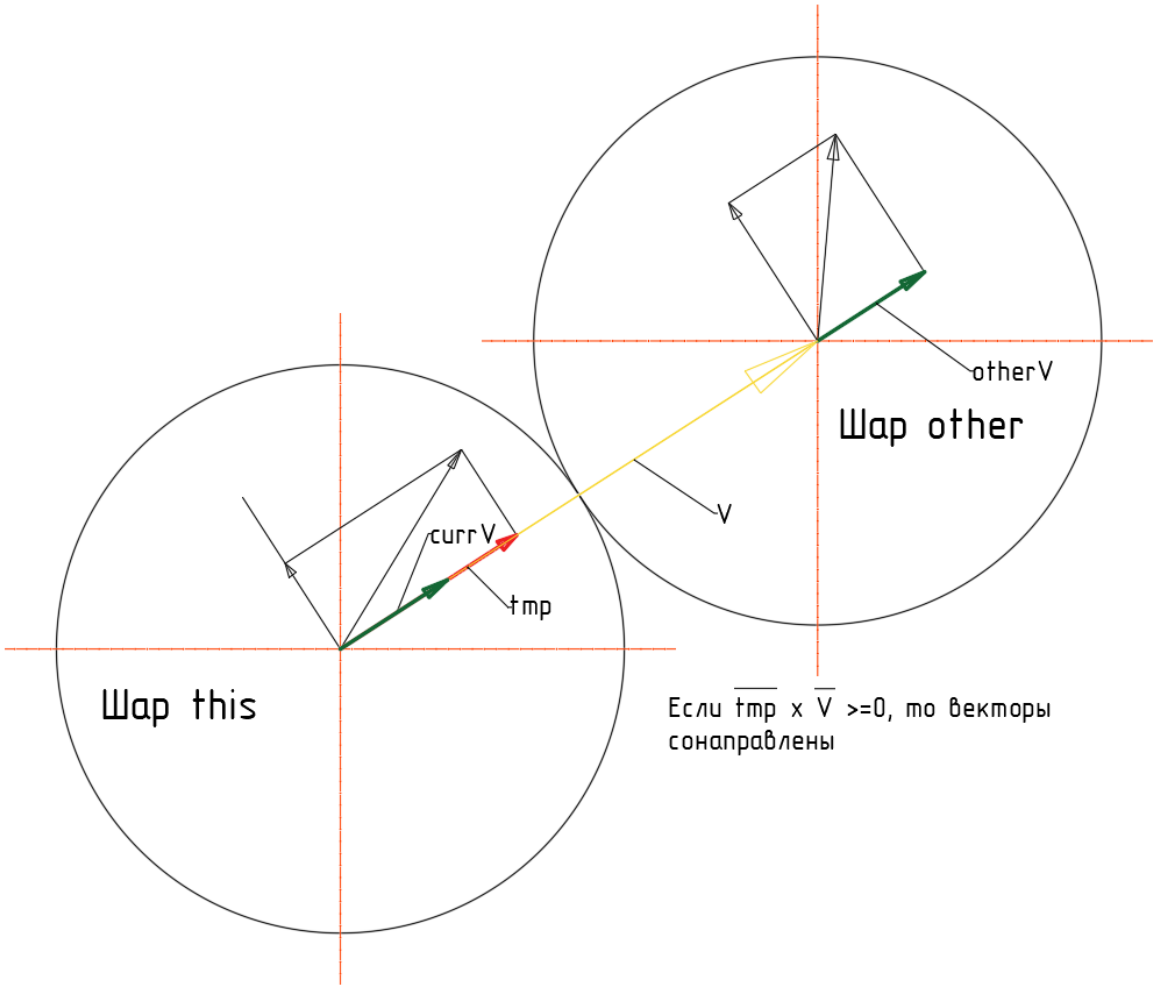
```
    {
        winGame(*activePlayer);
    }
    goOn();
}
```

6 Сложности реализации

6.1 Признак столкновения шаров

При вызове метода void MainWindow::**calcBallsDistances**() для расчет дистанции между шарами и последующего вызова функции соударения шаров может наступить момент, когда шары столкнулись на предыдущей итерации, но не успели «отскочить» на достаточное расстояние друг от друга из-за потери скорости и для них вновь будет вызвана функция соударения и шара войдут в зацикленное биение друг о друга.

Во избежание этой ситуации необходимо ввести условие, при котором шары движутся навстречу друг друга. Выяснить это можно путем оценки направленности векторов их скоростей.



В программе за реализацию проверки это условия отвечает метод

bool Ball::**isTouch**(V currV, V otherV, Coord &otherCoord)

метод встроен в обработку столкновения шаров

```
void Ball::collisionTwoBalls(Ball &other)
{
.....
        if(this->isTouch(this->v0.y, other.v0.y,otherBallCoord))
        {
            qSwap(this->v0.y,other.v0.y);
        }
.....
}
```

Суть метода сводится к вычислению скалярного произведения между вектором направленным от центра текущего шара к центру противоположного и разницей проекций векторов скорости шаров на межцентровую прямую.

```
bool Ball::isTouch(V currV, V otherV, Coord &otherCoord)
{
    V otherCenterVector(otherCoord.x,otherCoord.y);
    V tmp = (currV-otherV);
    double result = tmp<<otherCenterVector; //скалярное произведение
    return result>=0;    //если скалярное произведение больше
        //или равно 0, то шары двигаются навстречу друг другу
}
```

6.2 Признак столкновения шара с бортом

Аналогичная ситуация возможна при столкновении шара с бортом стола.

Здесь проверка проще и необходимо лишь проверить, что шар движется навстречу борту, оценив соответствующую проекцию скорости

```
void MainWindow::boardCollision()
{
.....................
            if(ball->x()<=Ball::r&&          //LEFT
                    ball->y() >=  ball->x()+d&&     //and top left pocket
                    ball->y() <= -ball->x()+height-d&&
                    !midleArea&&         //не в зоне луз
                    ball->v0.x<0)        //чтоб не зациклилось
            {
                ball->v0.x*=-1;
            }
.....................
}
```

7 Преимущества и недостатки. Пути развития

7.1 Преимущества

Как видно из методов расчете ударов шаров, при поиске проекций скорости на прямые между центрами, полностью удалось избежать вычисления

тригонометрических функций sin() и cos(). В методе используются исключительно арифметические операции. Это позволяет существенно экономить вычислительные ресурсы.

Наряду с этим, при для оценки расстояния между шарами значение дистанции заменено на ее квадрат. Если оценить, что дистанция вычисляется приблизительно n^2 раз за один цикл, где n – количество шаров, то это так же ускоряет работу приложения, так как извлечение корня значительно более «дорогая операция» нежели умножение.

7.2 Недостатки

Основным недостатком приложения можно назвать упрощенную математическую модель движения шаров, в которой отсутствует вращение относительно нескольких осей.

Так же в программе реализован упрощенный отскок шаров от бортов стола в области луз

7.3 Пути развития приложения

Помимо доработки пользовательского интерфейса, возможны следующие пути развития.

В качестве основного пути развития приложения можно выбрать устранение недостатков и предыдущего пункта.

Наряду с этим стоит рассмотреть возможность удара кием не только в центр шара, но и с некоторым смещением.

Внедрение возможности многопользовательской игры так же может быть уместно.

С глобальной точки зрения стоит пересмотреть иерархию классов. В реализации классов применяется множество статических переменных, что затрудняет реализацию игры с компьютером. Для внедрения опции компьютерного управления ходом игрока, а также возможности создания других видов бильярда, таких как «Русский» или «Снукер», следует избегать использование статических переменных классов и грамотно подойти к механизмам наследования.