Государственное бюджетное
общеобразовательное учреждение
"Президентский физико-математический
лицей №239"

State budget
general educational institution
"Presidential Physical-Mathematical
Lyceum №239"

Президентский ФМЛ № 239

Отчет по годовому проекту

The Space adventure

Ученик 10-3 класса
Козар Илья Дмитриевич

Преподаватель:
Клюнин А.О.

15 мая 2017 г.

Оглавление

1 Постановка задачи

Задача состояла в написании игры в стиле running. В процессе этой игры игроку будет дана возможность управления персонажем при помощи клавиатуры. Задача игры продержаться как можно дольше, уворачиваясь от препятствий (огненных астероидов). Количество пройденных метров будет показано после очередного проигрыша.

2 Математическая модель

Для решения поставленной задачи, необходимо было создать альтернативный мир, в котором будут выполняться примитивные законы человеческой физики:

1. *Движение персонажа*
   Для движения персонажа используется обычное равноускоренное движение тела в гравитационном поле массивного объекта: Для этого описания введем оси координат (Так как игра 2D, то работаем в плоскости) и обсудим движение по каждой из них. По вертикали координата персонажа менется по закону:
   $$y = y + V_y * \Delta t \tag{1}$$
   В то же время скорость меняется по закону:
   $$V_y = V_y + ACC * \Delta t \tag{2}$$
   где ACC - это ускорение в нашем поле.
   По горизонтали движение равномерное, т.к. не действуют силы. Значит движение описывается законом:
   $$x = x + V_x * \Delta t \tag{3}$$
2. *Движение текстур*
   Движение текстур происходит гораздо проще. Было посчитано, что они будут двигаться равномерно, несмотря на гравитационное поле, то есть их движение описывется законом:
   $$x = x + V_x * \Delta t \tag{4}$$
3. *Счетчик времени*
   Любой кинематический процесс основывается на времени. Чтобы это симулировать, нам нужно создать счетчик, который бы инициализировал малые промежутки времени. Для этого будем использовать статичный метод currentTimeMillis(), который возвращает текущее время в милисекундах, и вычитать его в каждом проходе через игровой цикл(подробнее про игровой цикл смотри в Построение алгоритма), как это представлено в коде:

```
    while(running) {
            delta = (System.currentTimeMillis() - lastTime);
            lastTime = System.currentTimeMillis();
    }
```

3 Метод решения

3.1 Игровой процесс

Итак, создаем проект, в нем класс Game(попутно создав в нем точку входа)TideSofDarK (2012). Данный класс должен наследовать класс Canvas и реализовать интерфейс Runnable:

```
    public class Game extends Canvas implements Runnable {

     private static final long serialVersionUID = 1L;
     public void run()
    }
```

Создадим переменную running типа Boolean, которая будет показывать запущена игра или нет. Создадим функцию start() и в ней будем создавать новый поток, переводя running в true:

```
    public void start() {
            running = true;
            new Thread(this).start();
    }
```

Теперь создадим в методе run() главный игровой цикл, который будет работать всегда, пока running = true, то есть здесь видно, что running показывает запущена ли игра или нет.Попутно создадим в методе run() счетчик времени. (Реализацю смотри в математических выкладках)

```
    public void run() {
        long delta;
        long lastTime = System.currentTimeMillis();
        while (running) {
            delta = (System.currentTimeMillis() - lastTime);
            lastTime = System.currentTimeMillis();
    }
```

Нужно продумать реализацию графики и физики в целом. Для этого создадим методы render() и update() соответственно.Их тоже поместим в главный игровой цикл, чтобы они обновлялись за каждое обращение в него. (Подробнее про render() и update () смотри в разделе Графика и Физика)

```
    public void run() {
            long delta;
            long lastTime = System.currentTimeMillis();
            while (running) {
                delta = (System.currentTimeMillis() - lastTime);
                lastTime = System.currentTimeMillis();
                render();
                update((double) delta / Constants.deltaConst);
            }
    }
```

Кстати, стоит сказать, что в update() обращаемся через счетчик времени. Чтобы время не было очень большим, делим значение на deltaConst (подобрано) (Подробнее про эту переменную смотри в Базовой структуре данных)

Что ж, игровой процесс почти реализован. Осталось продумать как игра будет завершаться и начинаться заново. Для этого обратимся в подраздел Физика, чтобы быть в курсе всех выкладок насчет препятствий. Если вы уже с ними ознакомились, то приступим:
Когда персонаж врезается в блок (как это реализуется смотри в разделе Физика) переменная boolean gameOver становится true, то есть мы в главном цикле можем выполнить несколько условий:

```
  while (running) {
        delta = (System.currentTimeMillis() - lastTime);
        lastTime = System.currentTimeMillis();
        render();
        update((double) delta / Constants.deltaConst);
        if (gameOver) {
            CHECK_THE_RESTART = true;
            render();
            blockQueue.clear();
        }
    }
```

Давайте разберемся, что выполняется, когда gameOver = true.Во-первых, переменная типа boolean CHECK-THE-RESTART = true, для того, чтобы при нажатии пробела:

```
    if (((e.getKeyCode() == KeyEvent.VK_SPACE)) && (CHECK_THE_RESTART)) {
        gameOver = false;
        CHECK_THE_RESTART = false;
        restart();
    }
```

Сработал метод restart():

```
     private void restart() {
        cnt1 = 1;
        Block_Speed = Constants.Vx_FOR_TEXTURE;
        MaxSpeed = false;
        Hero.CHECK_THE_OVERLAPS = false;
        FREQUENCY_FOR_BLOCK = Constants.FREQUENCY_FOR_BLOCK;
        Queue<Block> blockQueue = new PriorityQueue<Block>();

        hero = new Hero(Constants.SPAWN_FOR_HERO, Constants.minY, Constants.
    Vy_FOR_HERO, Constants.Vx_FOR_TEXTURE, Constants.Ay_FOR_HERO);
```

```
10
11          Floor floor = new Floor(Constants.SPAWNx_FOR_FLOOR, Constants.
        SPAWNy_FOR_FLOOR, Game.Block_Speed);
12      }
```

А также поле gameOver снова стало false (для того, чтобы нельзя было бесконечно вызывать restart()) и CHECK-THE-RESTART тоже стало false (по тем же причинам). Если gameOver = false, а CHECK-THE-RESTART = true, то можно постоянно вызывать restart(), что плохо. Аналогично, когда gameOver = true и CHECK-THE-RESTART = false.
Также, когда gameOver = true, в главной игровом цикле очишаем очередь препятствий:

```
blockQueue.clear();//подчищаем за оставшимися
```

и обращаемся в render(), чтобы отрисовать анимацию взрыва (подробнее смотри в разделе Графика) и усыпить поток на 4 секунды, чтобы можно было после этого нормально запустить restart():

```
1       if (gameOver) {
2           TheRecord = points_cnt;
3           points_cnt = 0;
4           if (cnt1 == 10) {
5               try {
6                   TimeUnit.SECONDS.sleep(4);
7               } catch (Exception e) {
8                   System.out.print(e);
9               }
10           }
11          hero.image.draw(g,hero.getX(),hero.getY());
12      }
```

Теперь обсудим, что происходит в методе restart(). Напомню:

```
1       private void restart() {
2           cnt1 = 1;
3           Block_Speed = Constants.Vx_FOR_TEXTURE;
4           Queue<Block> blockQueue = new PriorityQueue<Block>();
5           MaxSpeed = false;
6           Hero.CHECK_THE_OVERLAPS = false;
7           FREQUENCY_FOR_BLOCK = Constants.FREQUENCY_FOR_BLOCK;
8
9           hero = new Hero(Constants.SPAWN_FOR_HERO, Constants.minY, Constants.
        Vy_FOR_HERO, Constants.Vx_FOR_TEXTURE, Constants.Ay_FOR_HERO);
10          Floor floor = new Floor(Constants.SPAWNx_FOR_FLOOR, Constants.
        SPAWNy_FOR_FLOOR, Game.Block_Speed);
11      }
```

По сути мы возвращаем все поля и объекты к исходным значенниям, создаем новую очередь. (Подробнее про каждую переменую будет описано в разделе Физика)
Что ж, скелет игры уже готов, осталось только создать метод, который бы все время игры выполнялся параллельно главному игровому циклу. Назовем его init(), в нем можно реализовывать таймеры, как это делаю я (подробнее о таймерах будет дальше):

```
1       public void init() {}
```

3.2 Физика

Персонаж В этом разделе мы познакомимся со всеми игровыми процессами и их реализацией.Также узнаем о реализации движения персонажа, астероидов, пола и других игровых объектов.Приступим.

Начнем с реализации движения персонаж. Создадим класс Hero(), в котором будем реализовывать базовые конструкторы и писать методы, характерезующие персонажа:

```
1       public class Hero {
2
3       public Sprite image;
4       static boolean CHECK_THE_OVERLAPS = false;
5       private int x;
```

```
6       private int y;
7
8       private double Vy;
9       private double Vx;
10      private double ay;
11
12
13      public void setX(int x) {
14          this.x = x;
15      }
16
17      public void setVx(int Vx) {
18          this.Vx = Vx;
19      }
20
21      public void setY(int y) {
22          this.y = y;
23      }
24
25      public void setVy(double vy) {
26          Vy = vy;
27      }
28
29      public void setAy(double ay) {
30          this.ay = ay;
31      }
32
33
34      public int getX() {
35          return x;
36      }
37
38      public int getY() {
39          return y;
40      }
41
42
43      public double getVy() {
44          return Vy;
45      }
46
47      public double getVx() {
48          return Vx;
49      }
50
51      public double getAy() {
52          return ay;
53      }
54
55      public Hero(int x, int y, double vy, double vx,double ay) {
56          this.x = x;
57          this.y = y;
58          Vy = vy;
59          this.ay = ay;
60          Vx = vx;
61          image = Game.getSprite("pictures/sprites_8.run1.png");
62          }
63
64      public void jump(double delta) {
65          if (y > Constants.minY) {
66              Vy = 0;
67              y = Constants.minY;
68          } else {
69              Vy += Constants.ACC * delta;
70          }
71          y += (int) Vy * delta;
```

```
72
73          if ((y <= Constants.minY) && (y >= Constants.minY-10))
74              Game.CHECK_THE_JUMP = false;
75      }
76
77      public void down(double delta) {
78          if (y > Constants.minY) {
79              Vy = 0;
80              y = Constants.minY;
81          } else {
82              Vy += Constants.ACC1 * delta;
83          }
84
85          y += (int) Vy * delta;
86
87          if ((y <= Constants.minY) && (y >= Constants.minY-10))
88              Game.CHECK_THE_JUMP = false;
89      }
90
91
92      public void processUpPressed()
93      {
94          if ((Vy > 20) && (!Game.gameOver)) {
95              image = Game.getSprite("pictures/heroJump.png");
96              Game.CHECK_THE_JUMP = true;
97          }
98          if ((Vy <-20)&& (!Game.gameOver)){
99              image = Game.getSprite("pictures/heroDOWN.png");
100             Game.CHECK_THE_JUMP = true;
101         }
102     }
103
104     public boolean processOverlaps() {
105         if ((this.x <= 10) || (this.x >= Constants.WIDTH - 50))
106             Hero.CHECK_THE_OVERLAPS = true;
107
108         return Hero.CHECK_THE_OVERLAPS;
109     }
110
111     public void processJump() {
112         if (y <=0) {
113             this.Vy = 10;
114         }
115     }
116
117     static void runningLeft(double delta) {
118         Game.hero.setX(Game.hero.getX()-(int)(Game.hero.Vx * delta));
119     }
120
121     static void runningRight(double delta) {
122         Game.hero.setX(Game.hero.getX()+(int)(Game.hero.Vx * delta));
123     }
```

Давайте разберемся что здесь ,и как это работает. Во-первых, мы наблюдаем базовые поля и конструкторы, которые я позволю себе не объяснять. Это задание координат, скорости по двум осям и ускорения персонажа:

```
1       public class Hero {
2
3       public Sprite image ;
4       static boolean CHECK_THE_OVERLAPS = false;
5       private int x;
6       private int y;
7
8       private double Vy;
9       private double Vx;
10      private double ay;
```

```
11
12
13      public void setX(int x) {
14          this.x = x;
15      }
16
17      public void setVx(int Vx) {
18          this.Vx = Vx;
19      }
20
21      public void setY(int y) {
22          this.y = y;
23      }
24
25      public void setVy(double vy) {
26          Vy = vy;
27      }
28
29      public void setAy(double ay) {
30          this.ay = ay;
31      }
32
33
34      public int getX() {
35          return x;
36      }
37
38      public int getY() {
39          return y;
40      }
41
42
43      public double getVy() {
44          return Vy;
45      }
46
47      public double getVx() {
48          return Vx;
49      }
50
51      public double getAy() {
52          return ay;
53      }
```

Также, как это оговаривалось раньше, в этом классе реализуется движение персонажа:

```
1       public void jump(double delta) {
2           if (y > Constants.minY) {
3               Vy = 0;
4               y = Constants.minY;
5           } else {
6               Vy += Constants.ACC * delta;
7           }
8           y += (int) Vy * delta;
9
10          if ((y <= Constants.minY) && (y >= Constants.minY-10))
11              Game.CHECK_THE_JUMP = false;
12      }
13
14      public void down(double delta) {
15          if (y > Constants.minY) {
16              Vy = 0;
17              y = Constants.minY;
18          } else {
19              Vy += Constants.ACC1 * delta;
20          }
21
```

```
22          y += (int) Vy * delta;
23
24          if ((y <= Constants.minY) && (y >= Constants.minY-10))
25              Game.CHECK_THE_JUMP = false;
26
27      public void processJump() {
28          if (y <=0) {
29              this.Vy = 10;
30          }
31      }
32
33       public boolean processOverlaps() {
34          if (this.x <= 10)
35              Hero.CHECK_THE_OVERLAPS = true;
36
37          return Hero.CHECK_THE_OVERLAPS;
38      }
39       public void processLeft() {
40          if (x >=Constants.WIDTH-50) {
41              this.x = Constants.WIDTH-50 ;
42          }
43      }
44
45
46      static void runningLeft(double delta) {
47          Game.hero.setX(Game.hero.getX()-(int)(Game.hero.Vx * delta));
48      }
49
50      static void runningRight(double delta) {
51          Game.hero.setX(Game.hero.getX()+(int)(Game.hero.Vx * delta));
52      }
53
```

Для реализации движения по оси Ox используются законы, описанные в Математической модели.Стоит здесь добавить лишь то, что оба метода: runningRight и runningLeft задаются от переменной delta (счетчик времени), о которой оговаривалось тоже в Математической модели.

Что касается прыжка, то здесь все обстоит не так очевидно, давайте разбираться. За прыжок отвечают методы jump и down. В jump говорится, что если координата персонажа меньше какой-то константы (Constans.minY) (Стоит сказать, что координаты в Java считаются от левого верхнего угла вниз, поэтому написано:y > Constants . minY. Но об этом будет оговорено подробнее в Графике), то скорость персонажа по Oy становится равной нулю, а его координата тем минимальным значением. Это сделано для того, чтобы у персонажа была опора, и он не мог вечно падать. Ну, и если персонаж где-то над землей, то выполняются законы из Математической модели, то есть реализуется движение по Oy. Про CHECK-THE-JUMP будет сказано в разделе Графика.

Аналогичные рассуждения с методом down. Вы спросите, зачем он. Отвечу, что мой персонаж во время игры может летать и, чтобы этот полет был более управляем (персонаж взлетал медленней) я делаю ускорение вниз ACC1 > ACC. Вот и все различия.

Обсудим метод processJump: Этот метод нужен для того, чтобы у персонажа было ограничение по взлету: когда он достигает верхней границы экрана, его скрорсть резко меняется на противоположную, и он не может преодолеть этот барьер. Аналогично метод processLeft

Аналогичным по своей сути является метод processOverlaps. Он не дает возможности персонажу вылететь за границы Экрана по горизонтальной составляющей. Отличается от processJump лишь тем, что возвращает значение true для переменной, характерезующей процесс соударения с препятсвиями. Подробнее о ней в раздел, описывающий препятствия.

Подытожим. Мы описали в классе Hero(), отвечаеющим за героя, базовые поля, конструкторы (координаты, скорости, setter, getter итп.). Также реализовали все движение персонажа по двус осям Ox и Oy.Сделали ограничения по полету.

Осталось реализовать Графику и процесс соударения персонажа с препятствиями. Реализацию

графики можно будет посмотреть в соответствующем разделе.А процесс соударения далее в этом разделе в описании препятствий.

Заметим, что processJump,processLeft постоянно вызываются в update():

```
update() {
hero.processJump();
hero.processLeft();
}

```

Препятствия Все что связано с препятствиями описывается в классе Block():

```
public class Block implements Comparable<Block> {


int x;
int y;

double Vx;

public void setX(int x) {
    this.x = x;
}

public void setY(int y)
{
    this.y = y;
}

public void setVx(int Vx) {

    this.Vx = Vx;
}


 Sprite image;

public int getX() {
    return x;
}

public int getY() {

    return y;
}

public double getVx() {
    return Vx;
}


public Block(int x, int y, double Vx) {
    this.x = x;
    this.y = y;
    this.Vx = Vx;
    }


public void BlockMoving(double delta)
{
    this.x = (int) (this.x - this.Vx*delta);
}


@Override
```

```
    public int compareTo(Block o) {
        if(o.getX() < this.getX()) return 1;
        else return -1;
    }

    public boolean overlaps(Hero hero){
        boolean b;
         if (this.x < hero.getX())
        b =(this.x+50 < hero.getX() + hero.image.getWidth()) && (this.x + this.image.getWidth())  > hero.getX() &&
                this.y +35 < (hero.getY() + hero.image.getHeight()) && (this.y + this.image.getHeight()-20 > hero.getY()+hero.image.getHeight());
        else
            b =  (this.x+27 < hero.getX() + hero.image.getWidth()) && (this.x + this.image.getWidth())  > hero.getX() &&
                    this.y+35 < (hero.getY() + hero.image.getHeight()) && (this.y + this.image.getHeight()-20 > hero.getY());
         return b;

    }
}


```

Давайте разберемся, что здесь происходит.

Во-первых, как и в классе Hero() создаются базовые поля и конструкторы. На них заострять внимание я не буду. Но помимо них, есть много интересных вещей.

Движение препятствий описываютя в методе BlockMoving от delta (смотри зачем delta в описании движения героя).Закон, по которому происходит движение можно посмотреть в Математической модели.

В Block() описывается процесс соударения героя и препятствия:

```
         public boolean overlaps(Hero hero){
        boolean b;
         if (this.x < hero.getX())
        b =(this.x+50 < hero.getX() + hero.image.getWidth()) && (this.x + this.image.getWidth())  > hero.getX() &&
                this.y +35 < (hero.getY() + hero.image.getHeight()) && (this.y + this.image.getHeight()-20 > hero.getY()+hero.image.getHeight());
        else
            b =  (this.x+27 < hero.getX() + hero.image.getWidth()) && (this.x + this.image.getWidth())  > hero.getX() &&
                    this.y+35 < (hero.getY() + hero.image.getHeight()) && (this.y + this.image.getHeight()-20 > hero.getY());
         return b;

    }

```

Здесь все предельно просто: если герой правее на сколько-то координат препятствия и при этом не выходит своей задней частью за его границы, то overlaps true. Заметим, что я разделил анализ на две составляющие: когда героя за половиной препятствия и когда до нее. Это сделано для того, чтобы соударения были примерно реальными (героя касался препятствия), ведь координаты препятствия не точны из-за того что они вырезаны в виде прямоугольников с прозрачным фоном(подробнее в разделе графика). Также скажу, что все числа и смещения в неравенствах были подобраны вручную.

Также в этом классе происходт сортировка очереди из препятствий (блоков):

```
     @Override
    public int compareTo(Block o) {
        if(o.getX() < this.getX()) return 1;
        else return -1;
    }
```

```

```

Классическая сортировка объектов очереди. Если метод возвращает -1, то текущий объект будет располагаться перед тем, который передается через параметр и наоборот с 1. Подробнее описание очереди будет далее.

Пол/Поверхность Все, что связано с поверхностями описывается в классе Floor():

```
    public class Floor {

        private int x;
        private int y;

        private double Vx;
        Sprite image;

        public void setX(int x) {
            this.x = x;
        }

        public void setY(int y) {
            this.y = y;
        }

        public void setVx(double Vx) {
            this.Vx = Vx;
        }




        public int getX() {
            return x;
        }

        public int getY() {
            return y;
        }

        public double getVx() {
            return Vx;
        }


        public Floor(int x, int y, double Vx) {
            this.x = x;
            this.y = y;
            this.Vx = Vx;
            image = Game.getSprite("pictures/floor1.png");
        }

        public void Floor_Moving(double delta) {
            this.setVx(Game.Block_Speed);
            this.x = (int) (this.x - this.Vx*delta);
        }
    }
```

Здесь все довольно очевидно. В нем реализуется движение пола и базовые конструкторы. Подробничать я здесь не стану. Закон для движения можно посмотреть в Математической модели.

Стоит сказать, что Floor-Moving() вызывается постоянно в update():

```
update() {
floor1.Floor_Moving(delta);
floor2.Floor_Moving(delta);
}
```

Препятствия: реализация количества, частоты и скорости Если зайти в главный игровой класс Game(), то можно заметить как реализуется там создание очереди из Block() (препятствий)Создаем очередь из объектов препятствий.

```
    private Queue<Block> blockQueue = new PriorityQueue<Block>();

```

Введем поле, характерезущее частоту появления препятствий:

```
    private int Speed_cnt;

```

Будем ее увеличивать на 1 каждое обращение в главный игровой цикл (то есть очень часто) и, когда она достигнет своего максимального значения FREQUENCY-FOR-BLOCK, будем создавать новый элемент в очереди с рандомный появлением по Oy, скоростью, которая меняется в init() (смотри подробнее далее) и постоянной координатой по Ox : Constants.SPAWN-FOR-BLOCK.

```
    while(running) {
    Speed_cnt++;

      if (Speed_cnt >= FREQUENCY_FOR_BLOCK) {
                blockQueue.add(new Block((int) Constants.SPAWN_FOR_BLOCK,
    Constants.IMIN + RG.nextInt((Constants.IMAX - Constants.IMIN) / Constants.w) *
     Constants.w, Game.Block_Speed));
                Speed_cnt = 0;
            }
        }

```

Также частота появления уменьшается с течением времени до момента, пока не станет равной максимальной частоте: Constants.MAX-FREQUENCY-FOR-BLOCK. Делается это для того, чтобы было сложнее играть с течением времени.

```
    if (!MaxSpeed_Frequency)
                FREQUENCY_FOR_BLOCK -= 0.05;
            if (FREQUENCY_FOR_BLOCK <= Constants.MAX_FREQUENCY_FOR_BLOCK)
                MaxSpeed_Frequency = true;

```

Теперь непосредственно про изменение скорости препятствий. Реализуется изменение в init() таким образом:

```
    timer_for_speed.schedule(new TimerTask() {
            @Override
            public void run() {
                if (!MaxSpeed) {
                    Game.Block_Speed += Constants.Ax_FOR_BLOCK;
                    if (Game.Block_Speed >= Constants.MAX_SPEED_FOR_BLOCK) {
                        MaxSpeed = true;
                    }
                }
            }
        }, Constants.DIPLAY, Constants.FREQUENCY_FOR_SPEED);

```

Создаю таймер timer-for-speed. И каждые Constants.FREQUENCY-FOR-SPEED/1000 секунд (таймер считает время в 0.1 милисек) изменяю скорость на значение ускорения: Constants.Ax-FOR-BLOCK, до того момента, пока скорость не станет максимальной: Constants.MAX-SPEED-FOR-BLOCK. (Делается это тоже для усложнения игры)

Теперь обратимся в update() и там найдем реализацию очищения объектов очереди:

```
    if (blockQueue != null && !blockQueue.isEmpty() && blockQueue.peek().getX()
  < -500)
                blockQueue.poll();
            if (blockQueue != null && !blockQueue.isEmpty()) {
                for (Block block : blockQueue) {
                    if (block.overlaps(hero)) gameOver = true;
                }
            }
```

```
            for (Block block : blockQueue) block.BlockMoving(delta);

        }

```

Коментировать здесь особо нечего. Все операции стандартные. Заметим, что, если объект не достиг граничной координаты (-500), то Мы проверяем для каждого объекта из очереди является ли overlaps для героя true (подробнее про эту переменную было описано в разделе Препятствия). Таким образом мы всегда проверяем не произошло ли удара. Также для каждого объетка релизуем метода движения BlockMoving(delta);

Управление В классе Game() реализуется игровой интерфейс:

```
    private class KeyInputHandler extends KeyAdapter {

        public void keyPressed(KeyEvent e) {

            if (e.getKeyCode() == KeyEvent.VK_UP) {
                upPressed = true;
            }

            if (e.getKeyCode() == KeyEvent.VK_LEFT) {
                leftPressed = true;
            }

            if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
                rightPressed = true;
            }

            if (((e.getKeyCode() == KeyEvent.VK_SPACE)) && (CHECK_THE_RESTART)) {
                gameOver = false;
                CHECK_THE_RESTART = false;
                restart();
            }
        }

        public void keyReleased(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_UP) {
                upPressed = false;
            }

            if (e.getKeyCode() == KeyEvent.VK_LEFT) {
                leftPressed = false;
            }

            if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
                rightPressed = false;
            }
        }

```

Здесь реализутся слушатели клавиатуры. Принцип их работы следующий. Если нажата какая-то клавиша, то поле типа boolean становится true, и его можно потом использовать как условие выполнннения метода. Ксати, таким образом у меня реализуется движение персонажа:

```
       if (upPressed)
                hero.jump(delta);
            if (!upPressed)
                hero.down(delta);
            hero.processUpPressed();

              if (leftPressed) {
                Hero.runningLeft(delta);
            }

            if (rightPressed) {
                Hero.runningRight(delta);
```

```
            }


```

Или также запускается рестарт:

```
if (((e.getKeyCode() == KeyEvent.VK_SPACE)) && (CHECK_THE_RESTART)) {
                gameOver = false;
                CHECK_THE_RESTART = false;
                restart();
            }

```

Но о нем мы поговорим в следующем разделе.

Рестарт Начнем с того, что причиной вызова рестарта явялется переход поля overlaps в true. Как это происходит можно посмотреть в классе Block() или в классе Hero(). Так вот, если overlaps = true, то gameOver тоже становится true. Так вот, когда gameOver = true, то в главном игровом цикле:

```
    if (gameOver) {
                CHECK_THE_RESTART = true;
                render();
                blockQueue.clear();
                floor1.setVx(0);
                floor2.setVx(0);
            }
```

Поле CHECK-THE-RESTART = true, то есть мы можем нажать пробел и вызвать restart():

```
            if (((e.getKeyCode() == KeyEvent.VK_SPACE)) && (CHECK_THE_RESTART)) {
                gameOver = false;
                CHECK_THE_RESTART = false;
                restart();
            }
```

Также, если gameOver = true, то render() вызывается, чтобы отрисовать завершающуу панель (подробнее в Графике).
Подчищаем очередь и останавливаем пол.

Теперь непосредственно про restart():

```
          private void restart() {
        cnt1 = 1;
        Block_Speed = Constants.Vx_FOR_TEXTURE;
        hero = new Hero(Constants.SPAWN_FOR_HERO, Constants.minY, Constants.
    Vy_FOR_HERO, Constants.Vx_FOR_TEXTURE, Constants.Ay_FOR_HERO);
        Floor floor = new Floor(Constants.SPAWNx_FOR_FLOOR, Constants.
    SPAWNy_FOR_FLOOR, Game.Block_Speed);
        Queue<Block> blockQueue = new PriorityQueue<Block>();
        MaxSpeed = false;
        Hero.CHECK_THE_OVERLAPS = false;
        FREQUENCY_FOR_BLOCK = Constants.FREQUENCY_FOR_BLOCK;
        points_cnt = 0;
        floor1.setVx(Constants.Vx_FOR_TEXTURE);
        floor2.setVx(Constants.Vx_FOR_TEXTURE);
    }
```

Здесь просто все счетчики и все переменные принимают исходные значения. В прямом смысле происходит рестарт игры.

3.3 Графика

Реализация графики Будем рисовать, используя Canvas. Создадим класс Sprite():

```
          public class Sprite {
    private Image image;

    public Sprite(Image image) {
        this.image = image;
```

```
    }

    void draw_with_size (Graphics g,int x,int y, int x1, int y1){
        g.drawImage(image,x,y,x1,y1,null);
    }

    public int getWidth() {

        return image.getWidth(null);
    }

    public int getHeight()
    {
        return image.getHeight(null);
    }

    public void draw(Graphics g, int x, int y)
    {
        g.drawImage(image,x,y,null);
    }
}
```

Надеюсь, читателю понятны данные поля,конструкторы и методы. Опишу один из методов. К примеру, метод draw(), при помощи которого реализуется рисование объектов. Используем уже готовый метод рисования для переменной типа Graphics.

Персонаж Начнем с того, что в классе Hero() есть соответствующее поле Sprite image, в котором хранится картинка героя. Меняя эту картинку мы можем реализовать анимации движения и.т.п. Инициализируем картинку героя и меняем ее в init() при помощи таймера:

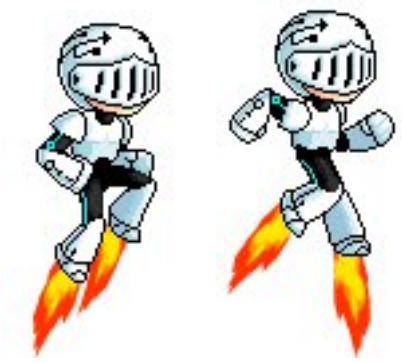
```
          timer_for_hero.schedule(new TimerTask() {
            @Override
            public void run() {
                if (pos++ > 5) {
                    pos = 1;
                }
                if ((!Game.CHECK_THE_JUMP) && (!gameOver)) {
                    Game.hero.image = Game.getSprite("pictures/sprites_8.run" +
   pos + ".png");
                }
            }
        }, Constants.DIPLAY, Constants.PERIOD);
```

Все довольно просто:картинка героя Game.hero.image (hero и image статичные поля) меняется с изменением cnt, которое в свою очередь менется каждые Constants.Period/1000 сек (Почему делить на 1000, смотри в разделе Физика). Заметим, что изменяется картинка героя до того момента, пока Game.CHECK-THE-JUMP = false и gameOver = false, потому что это соответственно полет и момент проигрыша, а там другие анимации.

Так реализуется полет героя:

```
          public void processUpPressed()
    {
        if ((Vy > 20) && (!Game.gameOver)) {
            image = Game.getSprite("pictures/heroJump.png");
            Game.CHECK_THE_JUMP = true;
        }
        if ((Vy <-20)&& (!Game.gameOver)){
            image = Game.getSprite("pictures/heroDOWN.png");
            Game.CHECK_THE_JUMP = true;
        }
    }
```

Если герой сильно ускоряется вверх (Vy > 20) или наоборот вниз, то картинки меняются. Стоит сказать, что processUpPressed постоянно анализируется в update().Прилагаю исходники картинок

полета: 

Или так реализуется анимация, когда произошло столкновение:

```
          timer_for_Boom.schedule(new TimerTask() {
            @Override
            public void run() {
                if (gameOver) {
                    if (cnt1++ > 9)
                        cnt1 = 9;
                    Game.hero.image = Game.getSprite("pictures/boom " + cnt1 + ".
    png");
                }
            }
        }, Constants.DIPLAY, Constants.PERIOD_FOR_BOOM);
```

Анимация реализуется аналогично случаю, когда герой бежит.

Пол Все реалиизуется аналогично герою. Стоит указать лишь то, что пол перерисовывается: как бы накладывается одна картинку на другую и получается так, что он бесконечный. Это реализуется в init():

```
            timer_for_floor1.schedule(new TimerTask() {
            @Override
            public void run() {
                floor1.setX(Constants.SPAWNx_FOR_FLOOR);
            }
        }, Constants.DIPLAY, Constants.PERIOD_FOR_FLOOR1);

        timer_for_floor2.schedule(new TimerTask() {
            @Override
            public void run() {
                floor2.setX(Constants.SPAWNx_FOR_FLOOR);
            }
        }, Constants.DIPLAY, Constants.PERIOD_FOR_FLOOR2);
```

Исходник ячейки пола:

Препятствия Все реалиизуется аналогично герою. Только объекты достаются из цикла:

```
    init() {
    timer_for_Boom.schedule(new TimerTask() {
            @Override
            public void run() {
```

```
                if (gameOver) {
                    if (cnt1++ > 9)
                        cnt1 = 9;
                    Game.hero.image = Game.getSprite("pictures/boom " + cnt1 + ".
    png");
                }
            }
        }, Constants.DIPLAY, Constants.PERIOD_FOR_BOOM);

    }
    render() {
             for (Block block : blockQueue) {
                    block.image = Game.getSprite("pictures/The Block" + cnt +
    ".png");
                }
}
```

Исходник препятствия:

Панель GameOver Когда gameOver = true (то есть игра завершилась) прорисовывается панель и пишется количество очков, которые игрок набрал в процессе игры. Давайте разберемся как это происходит:

```
    timer_for_Block1.schedule(new TimerTask() {
          @Override
          public void run() {
              if (!gameOver) {

                  if (cnt++ > 12)
                      cnt = 1;
                  points_cnt++;
              }
          }
      }, Constants.DIPLAY, Constants.PERIOD_FOR_BLOCK);
```

Здесь пока !gameOver увеличичется значение счетчика (игровых очков). Когда настал GameOver:

```
      if (gameOver) {
           TheRecord = points_cnt;
           Game.getSprite("pictures/Gam1.png").draw(g, 500, 200);
               Game.getSprite("pictures/Gam1.png").draw(g, 500, 200);
               String points_max = String.valueOf(TheRecord);
               g.setColor(Color.white);
               g.setFont(new Font("TimesRoman", Font.PLAIN, Constants.FONT_SIZE)
);
               g.drawString(points, 800, 498);
           hero.image.draw(g,hero.getX(),hero.getY());
       }
```

Задаем панель как картинку, на по ней рисуем текст, которому предварительно передаем значение очков. Цвет и шрифт задаются по умолчанию.

Фон В заключение к граическому разделу стоит описат, как задается задний фон. Все реализуется по аналогии к предыдущим пунктам.

```
        backgroundImg.draw(g, 0, 0);
```

3.4 Интерфейс

Описывая интерфейс, я буду предельно краток, потому что все операции здесь стандартные. Начнем с создания непосредственно фрейма:

Фрейм В классе main(), через который запускается приложение, создается фрейм

```
        public class Main {
    public static void main(String[] args) {
        JFrame frame1=new JFrame();
        frame1.setPreferredSize(new Dimension(Constants.WIDTH, Constants.HEIGHT));
        }
```

Кнопки На фрейме будет отрисовывать кнопки:

```
        Button button1 = new Button(0,0,80,80,new ButtonImage("pictures/play.png",
    "pictures/play.png","pictures/play.png"));
        button1.setBounds(Constants.WIDTH/2-15,Constants.HEIGHT/3+150,button1.
    width,button1.height);
        frame1.add(button1);

        Button button3= new Button(0,0,Constants.WIDTH,Constants.HEIGHT,new
    ButtonImage("pictures/Menu.png","pictures/Menu.png","pictures/Menu.png"));
        button3.setBounds(0,0,button3.width,button3.height);
        frame1.add(button3);
        frame1.setLayout(new BorderLayout());
        frame1.pack();
        frame1.setVisible(true);
```

Здесь создаются и отрисовываются кнопки.Их координаты задаются методом подбора. Причем button3 является как бы картинкой меню, поэтому ее координаты соответствуют размерам экрана (Constants.WIDTH/HEIGHT подробнее в разделе константы)

Также создадим слушатели для инициализации нажатия по кнопкам:

```
        button1.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                button1.k = 1;
                frame1.remove(button1);
                frame1.remove(button3);

                Game game=new Game();
                game.init();
                game.setPreferredSize(new Dimension(Constants.WIDTH, Constants.
    HEIGHT));

                frame1.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
                frame1.setLayout(new BorderLayout());
                frame1.getContentPane().add(game,BorderLayout.CENTER);
                frame1.pack();
                frame1.setVisible(true);
              }
```

Особо заострять внимания не буду. Реализация слушателей такая же как и в разделе Физика/Управление. Если нажать на кнопку старта, то создается новая игра (новый объект типа Game), запускается init(), задаются размеры игрового фрейма как размеры экрана (подробнее в разделе константы).

Запускается игра:

```
        game.start();
```

3.5 Константы

Также в процессе изучения метода решения, мы часто использовали значения, которые были подобраны вручную или как-то инициализированы. Для них я создаю класс Conastants, в котором все поля static. Вот его небольшая часть:

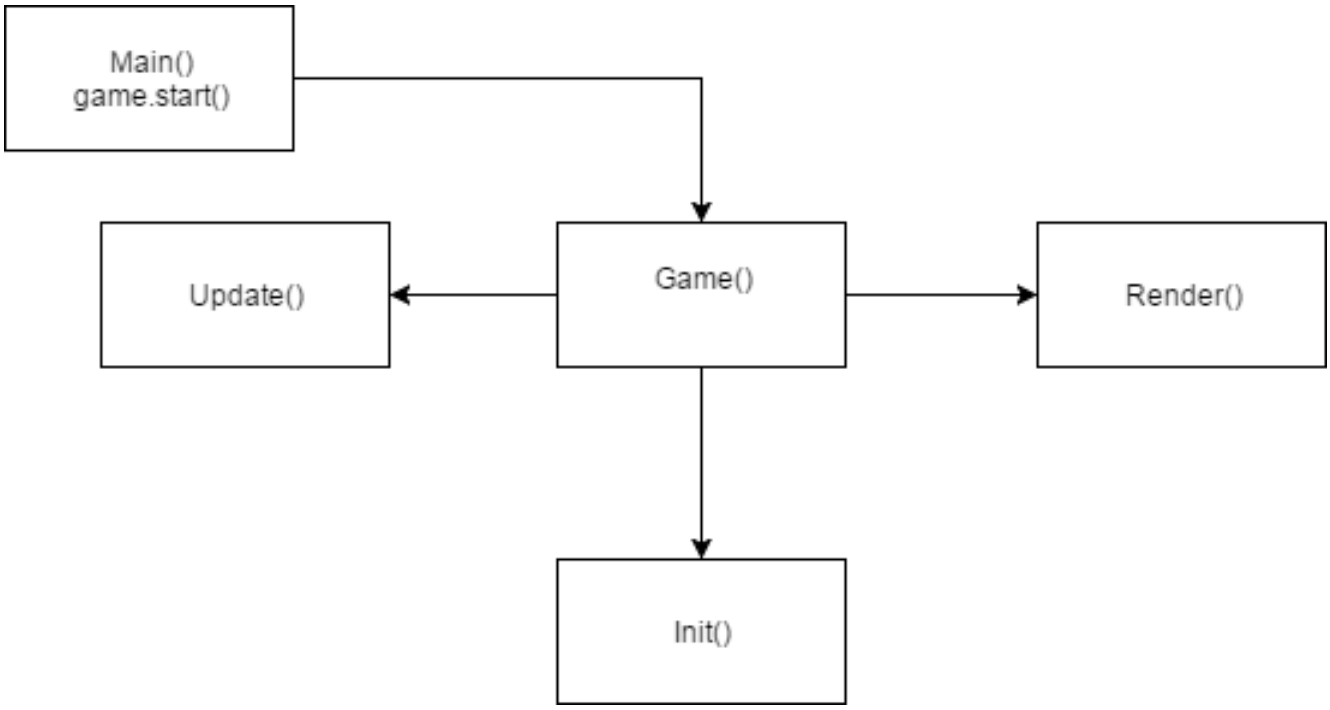
```
1        public abstract class Constants {
2     static  final int  WIDTH = (int) Toolkit.getDefaultToolkit().getScreenSize().
    getWidth();
3     static  final  int HEIGHT = (int)Toolkit.getDefaultToolkit().getScreenSize().
    getHeight();
4     static final int minY = HEIGHT - 200;
5     static final int SPAWN_FOR_HERO = 250;
6     static final int SPAWNx_FOR_FLOOR = 0;
7 }
```

4 Выходные данные

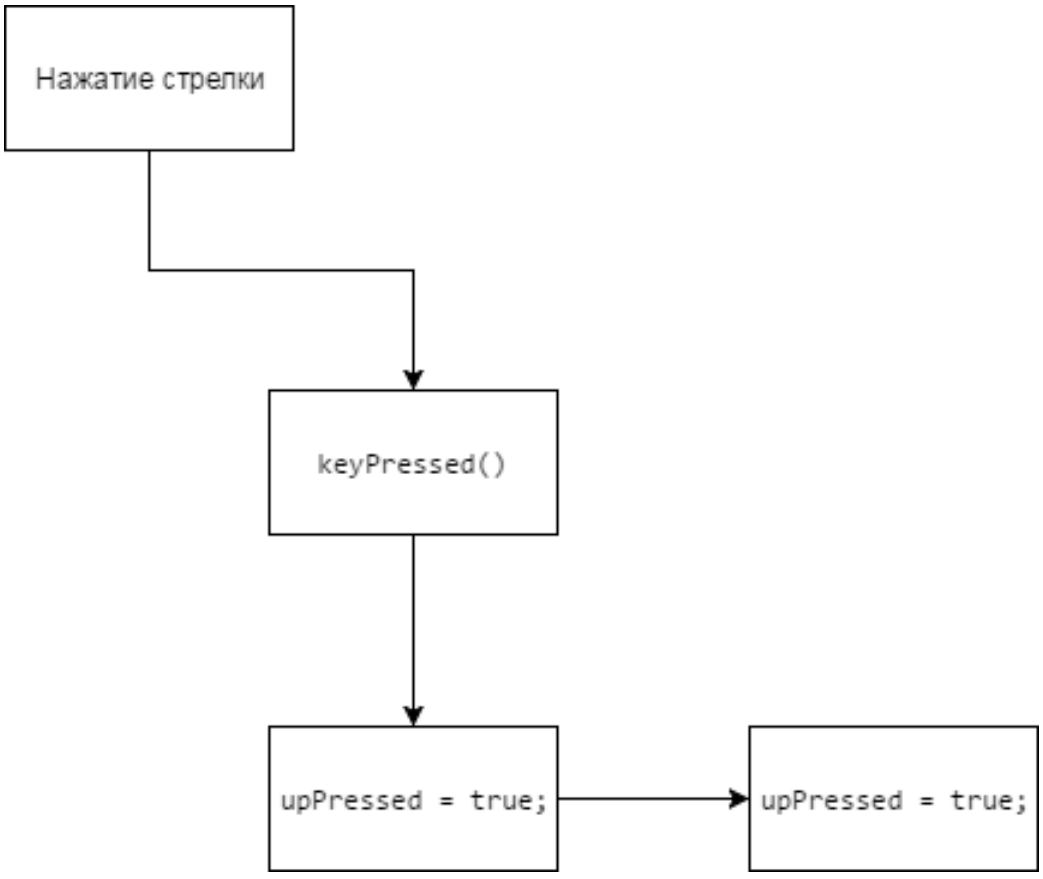
В процессе игры игрок должен как можно дольше продержаться, то есть набрать как можно больше очков. Выходными данным являются очки, набранные в этом процессе. Их вывод можно посмотреть в разделе Графика/Панель GameOver

5 Блок-Схемы

Главный алгоритм:



Алгоритм считывания клавиши:



Литература

TideSofDarK, (2012). Создание игры на java без сторонних библиотек, часть первая. Хабр.