Dokumentation

Arcade-Automat
28.06.2019

1 Teammitglieder und ihre Rollen

Simon Weck | 49,5%
Programmierung:
Pacman, Implementierung und Gestaltung sämtliche Menüs, Mergesort, Erstellung von Medien für das Programm(z.B. Rahmen und Raster die für einen Retro-look sorgen), Binary-Search

Pflichtenheft :
Funktionale Anforderungen, Qualitätsanforderungen, Übersicht Beziehungen der Klassen

Dokumentation:
Anforderungsanalyse, Algorithmen und Datenstrukturen, Klassendiagramm, Sequenzdiagramm, Evaluation, Literatur

Chris Teiwan | *49,5%*
Programmierung:
Snake, Heapsort, Laden und Speichern der Highscores, Ändern der Steuerungstasten

Pflichtenheft:
Abnahmekriterien, Anwendungsfall-Diagramm, Visionen und Ziele, Rahmenbedingungen, Kontext und Überblick

Dokumentation:
Projektbeschreibung, State of the Art und Related Work, Zusammenfassung und Ausblick, Anwendungsfalldiagramm, Video zur Produktvorstellung, Literatur

Daniel Schewe | *1%*
Konzeption Interface

2 Projektbeschreibung

Wir haben dieses Thema gewählt, da wir es interessant fanden, wie einige der ersten Spiele damals programmiert wurden.
Wir haben uns gedacht, dass wir die alten Spiele PacMan und Snake auf unseren eigenen Computern spielen wollen und nicht auf einem alten Arcade-Automaten. Nach Überlegungen über erfolgreiche Arcade-Spiele haben wir uns für das Spiel “PacMan”, welches früher auf sogenannten Arcade-Automaten spielbar war, und das Spiel ”Snake”, welches man auf dem sehr bekannten Nokia 3310 spielen konnte. Jedoch konnte man nicht beide Spiele auf einem gerät spielen, sondern musste das Spielgerät wechseln.

Dadurch hatten wir das Ziel ein solches Gerät zu simulieren. Zusätzlich wollten wir uns gegenseitig in diesen Spielen duellieren und gucken wer der Beste ist. Dieses sollte in einer Übersicht zu sehen sein.
Die Ziele haben wir dadurch gelöst, dass wir die Spiele implementiert haben und nach jedem Spiel nach den Namen des Spielers abfragen. Dieser wird dann in eine Liste eingetragen. Dort kann man sich diese Liste sortiert ansehen und weiß somit, wer der Beste Spieler in dem entsprechenden Spiel ist. Desweiteren wird automatisch die Liste mit allen Punktzahlen jeden Spielers in eine Datei gespeichert, wodurch man die Spielstände auch nach Neustart nicht verliert.

3 State of the Art und Related Work

Heutzutage gibt es sehr viele Online-Plattformen wo man Spiele dieser Art ohne eine Installation oder Download spielen kann. Zusätzlich gibt es Emulatoren die den Retro-Look sehr gut wiederherstellen können.

4 Anforderungsanalyse

Hauptmenü
/PF10/ start Screen anzeigen mit Hintergrund Video und Musik
/PF20/ Anzeigen Hauptmenü
/PF30/ Steuerung des Hauptmenüs mittels Pfeiltasten
/PF40/ Anzeigen Optionsmenü
/PF50/ Programm beenden
/PF60/ Ändern der Bildschirmgröße 720p und 1080p
/PF70/ Einstellung der Tastenbelegung für die Spiele
/PF80/ Vollbildmodus ein und aus schalten
/PF90/ Musik ein und aus schalten
/PF100/ Anzeige der 10 größten Scores
/PF110/ 2 unterschiedliche Sortier-Methoden zum sortieren der Scores
/PF120/ Wechseln zwischen pacman und snake score Listen
/PF130/ Suchen von Namen innerhalb der score Listen
/PF140/ Speichern der Scores in eine Text-Datei
/PF150/ Laden der Scores aus einer Text-Datei
/PF160/ Anzeigen eines Menüs während des Spieles um das Spiel zu verlassen, neu starten oder fortzuführen
/PF170/ Pacman Level Auswahl anzeigen

Pacman
/PF180/
/PF190/ Pac-man Zeichnen
/PF200/ Pac-man Animation(Abspielen , Starten und Stoppen)
/PF210/ Änderung der Richtung mittels Tastatur(inklusive Rotation des Pac-man)

/PF220/ Blockierung der Richtungstaste wenn Wand im Weg ist
/PF230/ Spielfeld Zeichnen inklusive Münzen PowerBalls und Früchte
/PF240/ Geister Zeichnen
/PF250/ Änderung der Richtung der Geister mittels KI (inklusive Änderung der Augen in Richtung der Bewegung)
/PF260/ A Pathfinding für die Geister KI*
/PF270/ Die Unterschiedlichen Geister Charaktere originalgetreu simulieren
/PF280/ Pac-man Kontakt mit Münze: Münze verschwinden lassen, Score +10
/PF290/ Pac-man Kontakt mit Frucht Score +500 (Frucht als Kirsche darstellen)
/PF300/ Pac-man Telepoltation wenn am Ende des Spielfeldes(Beide Richtungen)
/PF310/ Pac-man Kontakt mit Geist: Leben-1, Pac-man und Geister auf Startposition zurücksetzen,Pac-man Tod Animation abspielen
/PF320/ Pac-man Kontakt mit PowerBall: PowerBall-Bonus für einige Sekunden, Score +50
/PF330/ während des Power-Ball Bonus Animation für essbare Geister abspielen
*/PF340/ Pac-man mit PowerBall-Bonus bei Kontakt mit Geist: Geist verschwinden lassen, Score + 200*X pro Geist , Augen des Geistes in Basis bewegen*
/PF350/ Keine Münzen mehr auf dem Spielfeld: Neustart des Spiels mit altem Score, Zurücksetzen Pac-man und Geister
/PF360/ Bei 0 leben Spielende, Highscore in Liste sortieren, Spiel Beenden, Highscores anzeigen
/PF370/ Sounds abspielen bei: Kontakt mit Münzen, Kontakt mit Geister, Spiel Start, Pac-man Tod,
/PF380/ Level Auswahl
/PF390/ Steuerung nach Original Pac-man
/PF400/ Echtzeit Score
/PF410/ "Scatter-Phase" originalgetreu simulieren
/PF420/ "Frightened -Mode " originalgetreu simulieren
/PF430/ Anpassbarkeit auf jedes Level
/PF440/ ein zusätzliches Level zu dem standart Pacman Level
/PF450/ originalgetreuen Spielablauf erstellen

Snake
/PF460/ Zeichnen der Schlange
/PF 470/ Steuern der Schlange
/PF480/ Zeichnen der Frucht
/PF490/ Schlange Kontakt mit Frucht: Schlangenlänge+1, Score+1, Schnelligkeit+1
/PF500/ Schlange Kontakt mit sich selbst oder Begrenzung des Spielfelds= Spiel beenden
/PF510/ bei Spiel Ende Highscore in Liste sortieren und Highscore anzeigen
*/PF 520/ Schlangenlänge = Spielfeldlänge*Spielfeldbreite = Spiel beenden*
/PF 530/ aufsammelbare Power-Ups mit unterschiedlichen Fähigkeiten
/PF 540/ zufällige generierung der Power-Ups

5 Algorithmen und Datenstrukturen

A*-Pathfinding -Klassen: Ghost,Node (Package Pacman)

Die Bewegung der Geister bei Pacman funktioniert mittels einem A*-Pathfinding Algorithmus. Das bedeutet wir geben der Funktion einen Startpunkt A und einen Endpunkt B vor und die Funktion gibt uns den schnellsten Weg von A nach B. Die Pacman-Level sind aufgebaut wie ein Labyrinth daher ist es von Vorteil wenn wir einen schnellen Weg finden damit z.B. der Rote Geist den Spieler effizienter verfolgen kann.

```
public void pathFinding(Point end) {
    int k=0;
    Point start=new Point(getX(),getY());
    closed=new ArrayList<Node>();
    open=new ArrayList<Node>();
    open.add(new Node(start,end,start));
    while(!open.isEmpty()&&k<5000) {

        k++;
        Node currentNode=findBestNode();

        setX((int)currentNode.getPosition().getX());
        setY((int)currentNode.getPosition().getY());

        closed.add(currentNode);
        open.remove(currentNode);

        if(currentNode.equals(new Node(start,end,end))) {
            retracePath();
            break;
        }

        currentNode.setNeightbours(this);

        for (int i = 0; i < currentNode.getNeighbours().size(); i++) {
            Node neighbour=currentNode.getNeighbours().get(i);

            if(closed.contains(neighbour))
                continue;

            if(!open.contains(neighbour))
                open.add(neighbour);
            else if(currentNode.getG()+1>=neighbour.getF())
                continue;

            neighbour.setParent(currentNode);
        }
    }

    setX((int)start.getX());
    setY((int)start.getY());
}
```

Klasse: Ghost | Zeilen: 530-571

Startpunkt des pathfinding Algorithmus sind immer die X und Y Koordinaten des Geistes der den Weg berechnet. Der k wert steht für die Anzahl der Durchläufe des Algorithmus und stoppt ihn falls zu viel gesucht wird damit es im Notfall nicht zu einer endlos-Schleife kommen kann.

```
Node(Point start,Point end,Point position){
    this.start=start;
    this.end=end;
    this.position=position;
    double x=position.getX()-start.getX(); //h x cost
    if(x<0) //absolute value
        x*=-1;
    double y=position.getY()-start.getY(); //h y cost
    if(y<0) //absolute value
        y*=-1;
    h=x+y; //h cost
    x=end.getX()-position.getX(); //g x cost
    if(x<0) //absolute value
        x*=-1;
    y=end.getY()-position.getY(); //g y cost
    if(y<0) //absolute value
        y*=-1;
    g=x+y; //g cost
}
```

Klasse: Node | Zeilen: 21-39

Die Node Klasse ist wichtig um den kürzesten Weg zu finden, da sie einen vom Geist begehbaren Punkt repräsentiert. Ein Node-Objekt enthält außerdem einen G und einen H Wert. Diese beiden Werte stellen Kosten dar. Der G-Wert enthält die Kosten um von der Node-Position zum Ziel-Punkt des Pfades zu gelangen (Manhattan-Distanz). Der H-Wert enthält die Kosten um vom Startpunkt zu der Node-Position zu gelangen (Manhattan-Distanz).

Die closed Liste enthält alle Punkte die bereits während des Suchens besucht wurden.
Die open Liste enthält alle Punkte die der Geist noch besuchen kann. Der erste Punkt der besucht wird ist offensichtlich der Startpunkt. Jetzt beginnt die Suche nach dem schnellsten Weg.

Als ersten Schritt wollen wir ein Node-Objekt finden welches wir begehen können und welches die niedrigsten Kosten(H+G) hat um zum Ziel zu gelangen.

```
  public Node findBestNode() {
      Node bestNode = open.get(0);
      for (int i =1; i<open.size(); i++)
          if ( bestNode.getF() > open.get(i).getF())
              bestNode = open.get(i);

      open.remove(bestNode);
      return bestNode;
  }
```

Klasse: Ghost | Zeilen: 577-585

Die findBestNode() Methode sucht in der open Liste nach dem Node mit den niedrigsten F Kosten. Wir bewegen uns nun zu der Position von diesem Node. Da dieser Node nun besucht wurde wird er in die closed Liste hinzugefügt und aus der open List gelöscht da dieser nicht noch einmal begehbar ist. Ist die Position von dem Node unsere Ziel Position haben wir unser Ziel erreicht und können den pfad zurückverfolgen. Ist dies nicht der Fall müssen wir weiter suchen. Als nächsten Schritt betrachten wir die Nachbarn des Nodes auf dem wir uns gerade befinden.

```
public void setNeightbours(Ghost ghost) {
    neightbours.clear();
    if(ghost.isMovePossible(1, 0))
        neightbours.add(new Node(start,end,new Point(ghost.getX()+1,ghost.getY())));
    if(ghost.isMovePossible(-1, 0))
        neightbours.add(new Node(start,end,new Point(ghost.getX()-1,ghost.getY())));
    if(ghost.isMovePossible(0, 1))
        neightbours.add(new Node(start,end,new Point(ghost.getX(),ghost.getY()+1)));
    if(ghost.isMovePossible(0, -1))
        neightbours.add(new Node(start,end,new Point(ghost.getX(),ghost.getY()-1)));
}
```

Klasse: Node | Zeilen: 51 -60

Die Nachbarn des Nodes wird als Liste dargestellt und enthält alle Nodes die vom Geist in einem Schritt begehbar sind.
Für jeden dieser Nachbarn werden nun die folgenden Schritte durchgeführt:
-Wenn der Nachbar bereits in der cloed List ist wird mit dem Nächstem Nachbarn weitergemacht. Dieser Nachbar wurde bereits besucht also wird er ignoriert.
-Ist der Nachbar nicht in der closed Liste und auch nicht in der open Liste wurde er jetzt erst entdeckt und wird zu der open Liste hinzugefügt da er ja nun begehbar ist.
-Ist der Nachbar bereits in der open Liste wurde er bereits entdeckt aber wurde noch nicht begangen dann können wir die Kostendes Nodes worauf wir uns befinden mit den Kosten des Nachbarn vergleichen. Sind die G Kosten (von der Node Position zum Ziel) vom Nachbar Node größer als die vom Node auf dem wir uns befinden ist der Nachbar Node kein besserer Weg also fahren wir mit den anderen Nachbarn fort.
-Nun wird der Eltern Node gesetzt auf dem Node auf dem wir uns gerade befinden. Von diesem Node sind wir ja auch gekommen und haben von hier aus auch diese Nodes entdeckt. Wenn der Nachbar Node aber einen geringeren G wert hat als der Node auf dem wir uns befinden setzen wir von diesem Node den Eltern Node um auf den Node auf dem wir uns befinden, da dieser Weg besser ist.

Der Suchalgorithmus bricht nach 10.000 Durchläufen ab oder wenn die open Liste leer ist, es also keinen node mehr gibt zu dem wir uns bewegen können. Dies wird aber in der Umsetzung von pacman (hoffentlich) nicht der Fall sein.

Ist das Ziel gefunden müssen wir den Weg noch zurückverfolgen um eine Liste mit Punkten zu bekommen an dem sich der Geist orientieren kann. Wir müssen quasi die Brotkrumen für den Geist legen.

```
public void retracePath() {
    path=new ArrayList<Node>();
    Node goal=closed.get(closed.size()-1);
    while(goal.getParent()!=null) {
        path.add(goal);
        goal=goal.getParent();
    }
}
```

Klasse: Ghost | Zeilen: 590 -597

Zum zurückverfolgen des schnellsten Weges nehmen wir uns als erstes den zuletzt hinzugefügten Node der closed Liste. Die Position des Nodes ist die Ziel Position die wir am Anfang gesucht haben. Nun hangeln wir unts entlang der Eltern des Nodes bis wir zurück zum Startpunkt angelangt sind. Es werden also die Eltern Nodes zur Liste hinzugefügt und deren Eltern und wiederum deren Eltern usw. . Wir müssen nun lediglich die Position der Nodes in der path Liste schritt für schritt zur Ermittelung der Richtung nutzen in die sich der Geist bewegt.

```
    Point next=getNextPoint();
    if(next!=null) {
        if(next.getX()>getX()) {
            setDirectionX(1);
            setDirectionY(0);
        }
        if(next.getX()<getX()) {
            setDirectionX(-1);
            setDirectionY(0);
        }
        if(next.getY()>getY()) {
            setDirectionX(0);
            setDirectionY(1);
        }
        if(next.getY()<getY()) {
            setDirectionX(0);
            setDirectionY(-1);
        }
    }
}
```

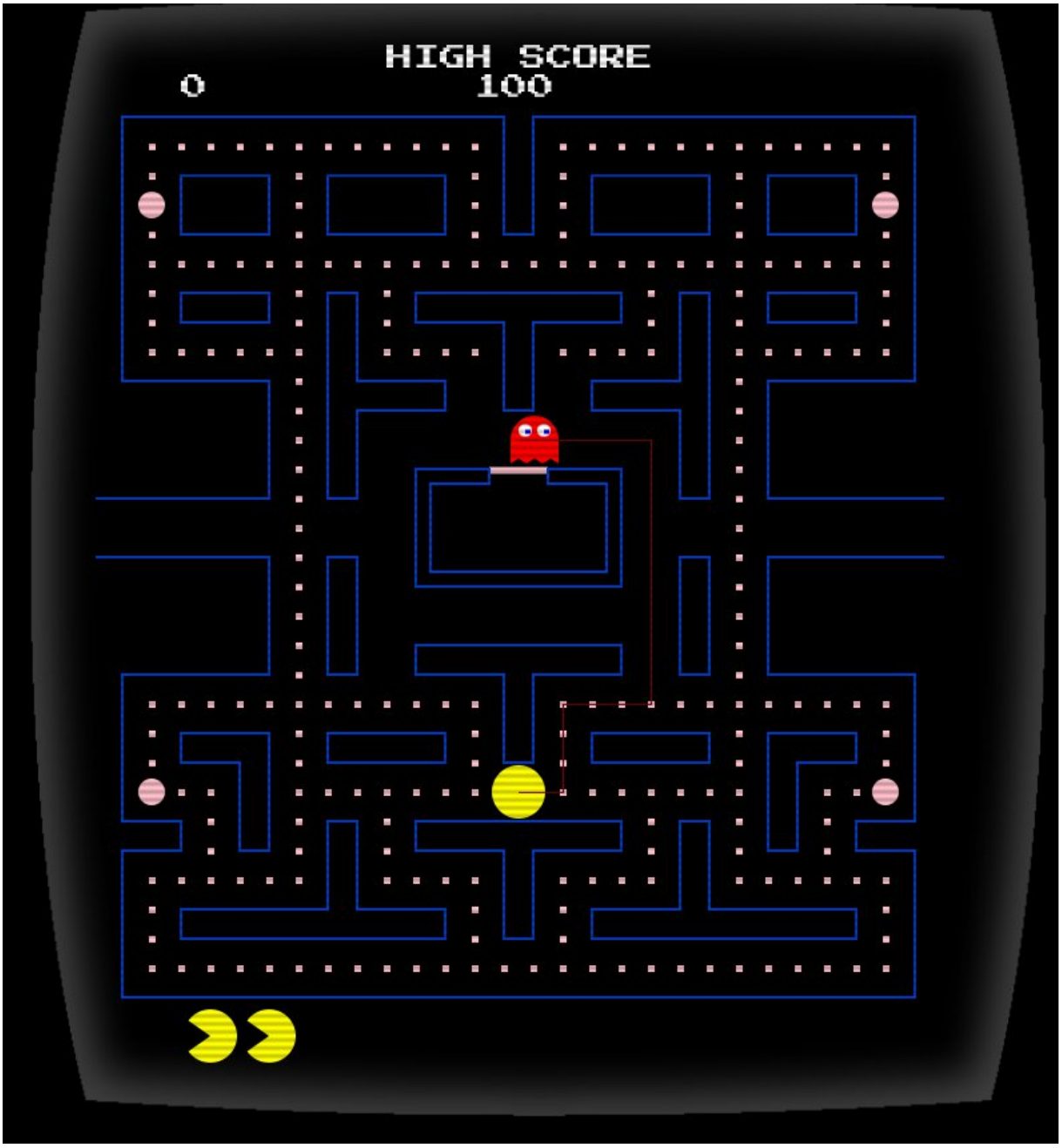
Klasse: Ghost | Zeilen: 303 -321

So verfolgt der Geist den bereits errechneten Pfad. Mit der gesetzten Richtung bewegt er sich identisch zum Pacman.

```
public Point getNextPoint() {
    if(path.size()==0)
        return null;
    Node nextNode=path.get(path.size()-1);
    path.remove(path.size()-1);
    return nextNode.getPosition();
}
```

Klasse: Ghost | Zeilen: 602 -608

Mit dieser Methode wird Stück für Stück der Weg “aufgebraucht”.
Auf eine Vereinfachung des Suchbereichs wurde vorerst verzichtet um bei der Level Gestaltung relativ frei zu sein. Dies führt aber zu vielen Berechnungen für den Weg und kann das Spiel verlangsamen.



Errechneter Pfad des Roten Geist (rote Linie bestehend aus den gefärbten Pfad-Punkten)

“Scatter-Phase” - Klasse : Ghost (Package Pacman)

Die Geister im Pacman Spiel wechseln zwischen 2 wichtigen Phasen während des Spiels. In der “Chase-Phase” versucht jeder Geist den Pacman auf seine Art und Weise zu jagen. In der “Scatter-Phase” bewegen sich die Geister in ihrer persönlichen Ecke des Spielfeldes. Der Geist verweilt immer 20 Sekunden in der “Chase-Phase” und wechselt dann für 8 Sekunden in die “Scatter-Phase” und dann wieder in die “Chase-Phase”. Um diese “Scatter-Phase” zu simulieren müssen wir für jeden Geist eine Ecke zuweisen. Während der “Scatter-Phase” wird dann für jeden unterschiedlichen Geist ein Pfad zu einem Zufälligen Punkt in dem jeweiligen Bereich des Levels errechnet und verfolgt. Daher besteht die Motivation darin jeden vom Geist begehbaren Punkt im Level zu finden und dann diese Punkte in 4 unterschiedliche ecken aufzuteilen. Eine kleine Herausforderung stellt dies dar, denn der Algorithmus muss auch dann das Level in gleich große Ecken aufteilen, wenn das Level ausgetauscht wird.

```
public void setupEveryPath(Point p) {
    if(!everyPath.contains(p)) {
        everyPath.add(p);
        setX((int)p.getX());
        setY((int)p.getY());
        if(isMovePossible(1,0))
            setupEveryPath(new Point(getX()+1,getY()));
        setX((int)p.getX());
        setY((int)p.getY());
        if(isMovePossible(-1,0))
            setupEveryPath(new Point(getX()-1,getY()));
        setX((int)p.getX());
        setY((int)p.getY());
        if(isMovePossible(0,1))
            setupEveryPath(new Point(getX(),getY()+1));
        setX((int)p.getX());
        setY((int)p.getY());
        if(isMovePossible(0,-1))
            setupEveryPath(new Point(getX(),getY()-1));
    }
}
```

Klasse: Ghost | Zeilen: 628 -648

Es ist ein rekursiver Algorithmus der jeden Weg in jeder Richtung abgeht bis er zu einer Wand gelangt. Alle begehbaren punkte werden zu einer Liste hinzugefügt. Abbruchbedingung für den Algorithmus ist das der Punkt bereits in der Liste ist und somit wurde dieser Weg bereits gefunden.

```
public Boolean isMovePossible(int x, int y) {
    moveHitBox(x,y);
    if(level.checkColissionWalls(hitBox.getBoundsInParent())) {
        moveHitBox(-x,-y);
        return false;
    }
    moveHitBox(-x,-y);
    return true;
}
```

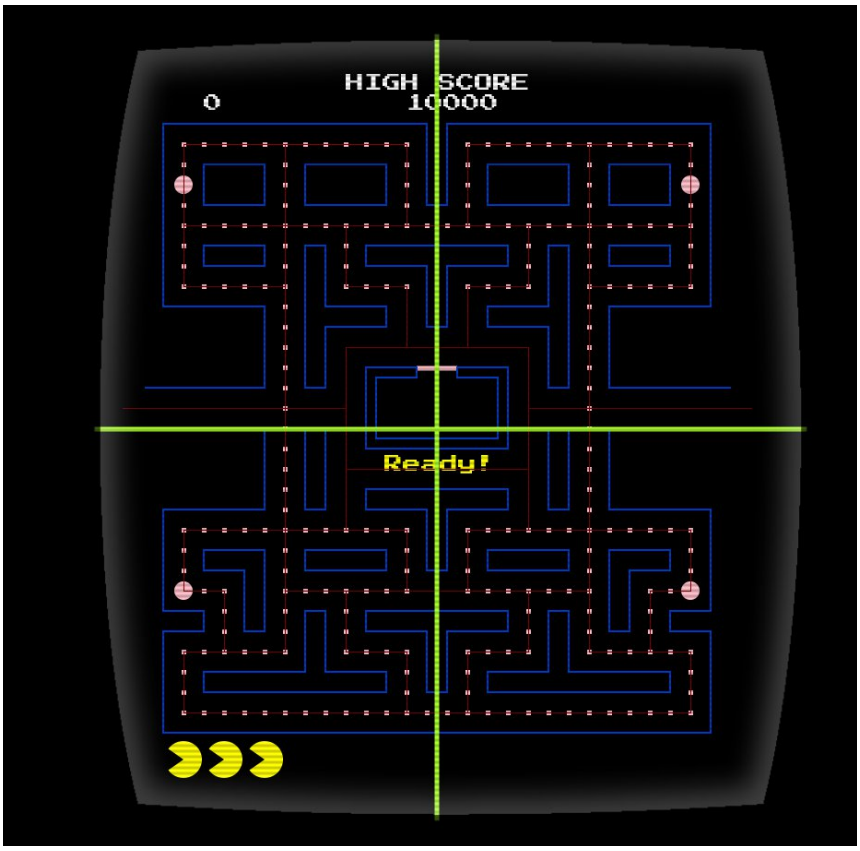
Klasse: Character | Zeilen: 321 -329

Diese Methode wird zur Hilfe genommen um jeden begehbaren Weg zu finden. Sie Liefert einen Boolean wert zurück ob der Geist mit etwas kollidiert sobald er sich um die Parameter x und y bewegt. Die Hitbox ist ein unsichtbares Rechteck welches zu Kollisionserkennung genutzt wird.

```
public void addEveryPath() {
    everyPath.clear();
    int startX=getX();
    int startY=getY();
    setupEveryPath(new Point(Ghost.getOutOfBaseX(),Ghost.getOutOfBaseY()));
    setX(startX);
    setY(startY);
    int x=0;
    int y=0;
    for(Point p:everyPath) {
        x+=p.getX();
        y+=p.getY();
    }
    x=x/everyPath.size(); //find the middle of every possible x coordinate
    y=y/everyPath.size(); //find the middle of every possible y coordinate
    //separate everyPath into 4 lists
    for(Point p:everyPath) {
        if(p.getX()<x && p.getY()<y)
            topLeft.add(p);
        if(p.getX()>x && p.getY()<y)
            topRight.add(p);
        if(p.getX()<x && p.getY()>y)
            bottomLeft.add(p);
        if(p.getX()>x && p.getY()>y)
            bottomRight.add(p);
    }
}
```

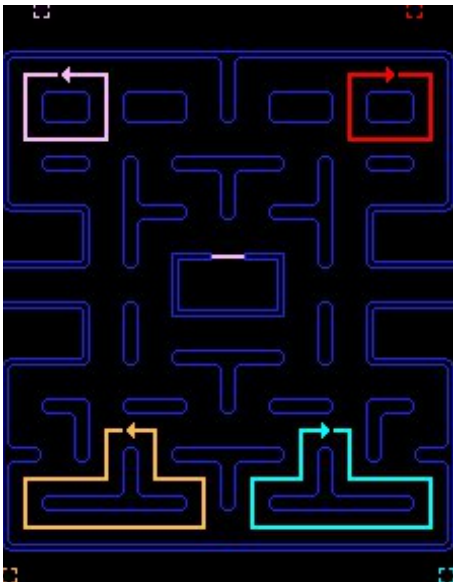
Klasse: Ghost | Zeilen: 654 -680

Diese Methode wird zu beginn des Spiels ausgeführt und teilt zudem die Liste mit allen begehbaren punkten auf in 4 gleich große Teillisten. Das Aufteilen funktioniert mittels dem Mittelwert aller X und Y Werte der begehbaren Punkte.



Alle erreichbaren Punkte in rot und Aufteilung der Ecken in grün

Alle erreichbaren Punkte und Aufteilung in Ecken für das HSHL-Level



https://dev.to/code2bits/pac-man-patterns--ghost-movement-strategy-pattern-1k1a
abgerufen: 24.06.2019

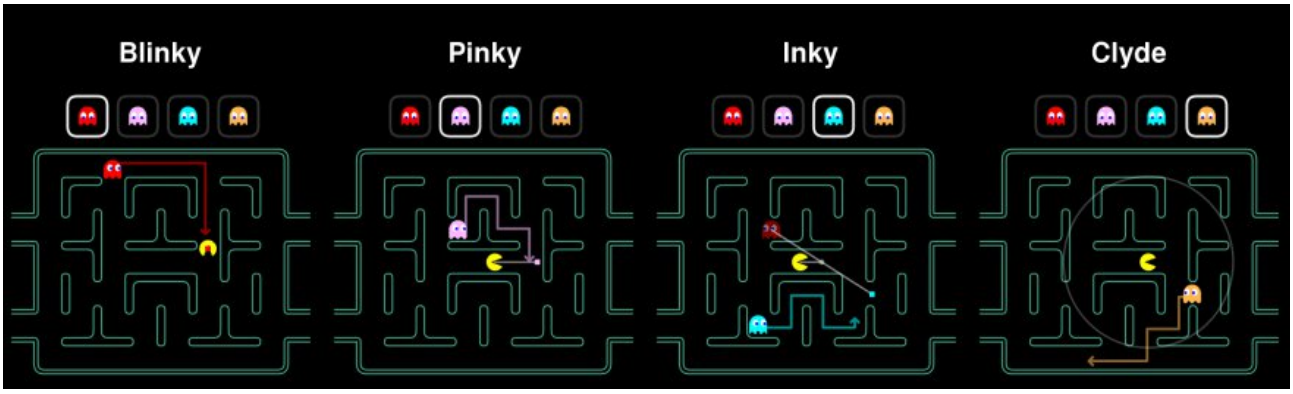
Die Ecken Teilen sich wie folgt auf:
Roter Geist - obere rechte Ecke
Pinker Geist - obere linke Ecke
Orangener Geist - untere rechte Ecke
Cyan Geist - untere rechte Ecke

```
}else if(!getStart()){
    if(getPath().isEmpty())
        pathFinding(getTopRight().get((int)(Math.random()*getTopRight().size()-1)));
}
```

Klasse: GhostRed | Zeilen: 34 -36

“Scatter-Phase” vom rotem Geist

“Chase-Phase” - Klasse : Ghost (Package Pacman)



https://dev.to/code2bits/pac-man-patterns--ghost-movement-strategy-pattern-1k1a
abgerufen: 24.06.2019

In der bereits erwähnten “Chase-Phase” jagt der Geist den Pacman. Jeder Geist hat dabei eine unterschiedliche Herangehensweise (siehe Abbildung).
Der rote Geist ist der aggressivste und versucht den Pacman direkt zu jagen in dem er den kürzesten weg zu den Koordinaten vom Pacman sucht.

```
public void calculatePath(Pac pac) {
    t+=0.01;
    if(!getDeath()&&!getStart()) { //if its not dead or in start mode calculate a path
        if(getChase()&&!getFrightened()) {//if chase mode is active
            if(t>1||getPath().isEmpty()) {
                pathFinding(new Point(pac.getX(),pac.getY())); //straight up chase pacman
                t=0;
            }
```

Klasse: GhostRed | Zeilen: 20 -29

Wenn der Geist nicht Tot ist oder sich am Spiel-Anfang befindet und sich in der “Chase-Phase” befindet wird einfach der kürzeste weg zum den Pacman X und Y Koordinaten berechnet. Eine erneute Berechnung findet ca. alle 1.3 Sekunden statt oder wenn der alte Weg “aufgebraucht” wurde sprich die Liste des alten Weges leer ist.

Der Pinke Geist ist etwas hinterhältig, denn dieser versucht sich vor den pacman zu platzieren. (siehe Grafik)

```
@Override
public void calculatePath(Pac pac) {
    t+=0.01;
    if(!getDeath()&&!getStart()) { //if its not dead or in start mode calculate a path
        if(getChase()&&!getFrightened()) { //if chase mode is active
            if(t>0.6||getPath().isEmpty()) { //after 1 seconds calculate a new path or when the current path is empty
                b=false; //reset found path boolean
                for(int i=80;i>60;i--) { //searches 80 to 70 pixels infront of pacman (
                    if(getEveryPath().contains(new Point(pac.getX()+pac.getDirectionX()*i,pac.getY()+pac.getDirectionY()*i))) {
                        pathFinding(new Point(pac.getX()+pac.getDirectionX()*i,pac.getY()+pac.getDirectionY()*i));
                        b=true;
                        break;
                    }
                }
                if(!b) //if there is no possible path 80-70 pixels infront of pacman
                    pathFinding(new Point(pac.getX(),pac.getY())); //straight up chase pacman like the red ghost
                t=0;
            }
```

Klasse: GhostPink | Zeilen: 21 -38

Ist der Pinke Geist in der “Chase-Phase” schaut er von 100 bis 20 Pixel vor Pacman, in die Richtung in der sich der Pacman bewegt. Es wird überprüft ob der Punkt sich in der Liste mit allen begehbaren Punkten befindet. Gibt es einen Punkt der zwischen 100 und 20 Pixeln vor Pacman ist wird dieser als Endpunkt des Pathfinding Algorithmus benutzt.
Findet er keinen Punkt (z.B. wenn der Pacman gegen eine der äußeren Wände schaut) bewegt sich der Geist zu einem zufälligen Punkt im Level. Der Weg vom pinken Geist wird ca. jede Sekunde erneut berechnet oder wenn das alte Ziel erreicht wurde.

Aus zeitlichen Gründen haben wir es nicht geschafft die Charaktere der anderen beiden Geister zu Simulieren. Der Charakter des Cyan Ghost ist sehr komplex und scheint sehr zufällig. Der orangene Geist versucht es den Pacman zu vermeiden.
In dieser implementieren von Pacman bewegen sich die beiden Geister nach dem gleichen Charakter, nämlich zufällig.

```
@Override
public void calculatePath(Pac pac) {
    if(!getDeath()&&!getStart()) {
        if(getChase()&&!getFrightened()) {
            if(getPath().isEmpty()) //if the ghost is at its destination calculate a new path
                pathFinding(getEveryPath().get((int)(Math.random()*getEveryPath().size()-1)));
```

Klasse: GhostOrange | Zeilen: 16 -21

Es wird sich einfach ein zufälliger Punkt von allen begehbaren Punkten als Endpunkt des pathfindings genutzt. Der Weg wird erst dann erneut berechnet wenn er den Zielpunkt vom letztem Weg erreicht hat.

“Namen in den HighScores suchen” - Klassen: Sorting (Package Menue)

In unserer Implementierung der Highscore Ausgabe werden nur die besten 10 Scores der jeweiligen Spiele ausgegeben. Es gibt eine Motivation den Namen in dieser Liste zu Suchen um so den Platz zu ermitteln. Suchen ist schneller als einfach eine Liste durch zu scrollen und den Namen zu suchen.

Wir suchen den Namen xenia. Dieser befindet sich auf Platz 15 und ist deshalb nicht sichtbar.

Nachdem wir alle Namen mit beim Anfangsbuchstaben x gesucht haben finden wir xenia an platz 15 mit einem score von 1000.

Das suchen funktioniert auch mit mehr als einem Buchstaben. Und ignoriert auch Groß- und Kleinschreibung.

Implementiert wurde das ganze mit einem Mergesort der zuerst alle Buchstaben aufsteigend sortiert und danach wird nach jedem einzelnen Buchstaben gesucht mittels Binary-Search.

```
public static void mergeSortChar(ArrayList<Player> a, int p, int r, int t) {
    if (p < r) {
        int q = (p + r) / 2;
        mergeSortChar(a, p, q,t);
        mergeSortChar(a, q + 1, r,t);
        mergeChar(a, p, q, r,t);
    }
}
```

Klasse: Sorting | Zeilen: 33 -40

Der Parameter t steht für den Buchstaben an der Stelle t vom Namen des Spielers.

```
public static void mergeChar(ArrayList<Player> a, int p, int q, int r,int t) {
    int lsize = q - p + 1;
    int rsize = r - q;
    ArrayList<Player> la = new ArrayList<Player>();
    ArrayList<Player> ra = new ArrayList<Player>();
    for (int i = 0; i < lsize; ++i)
        la.add(a.get(p + i));
    for (int j = 0; j < rsize; ++j)
        ra.add(a.get(q + 1 + j));
    int i = 0;
    int j = 0;
    int k = p;
    while (i < lsize && j < rsize) {
        if (la.get(i).getName().toLowerCase().charAt(t) < ra.get(j).getName().toLowerCase().charAt(t)) {
            a.set(k, la.get(i));
            i++;
        } else {
            a.set(k, ra.get(j));
            j++;
        }
        k++;
    }
    while (i < lsize) {
        a.set(k, la.get(i));
        i++;
        k++;
    }
    while (j < rsize) {
        a.set(k, ra.get(j));
        j++;
        k++;
    }
}
```

Klasse: Sorting | Zeilen: 42 -74

Hier werden die Teil-Listen zusammengeführt und dabei in aufsteigender Reihenfolge Sortiert. Man beachte das die Buchstaben erst in Kleinbuchstaben umgewandelt werden müssen damit der Vergleich ohne Rücksicht auf Groß- und Kleinschreibung gelingt.

```
for(int t=0;t<b.length();t++) { //do this for every character in the string
    //first sort the characters
    sortChar(coppyOfScores,t);
    sortChar(searchList,t);

    int k=binarySearch(coppyOfScores,b.toLowerCase().charAt(t),0,coppyOfScores.size()-1,t); //k= index of found character

    if(k==-1) { //nothing found
        coppyOfScores.clear();
        return coppyOfScores;
    }else {
        int o=k; //obere grenze
        int u=k; //untere grenze
        //get douplicate characters
        for(int i=k+1;i<coppyOfScores.size();i++)
            if(coppyOfScores.get(i).getName().toLowerCase().charAt(t)==coppyOfScores.get(k).getName().toLowerCase().charAt(t))
                o++;
            else
                break;

        for(int i=k-1;i>0;i--)
            if(coppyOfScores.get(i).getName().toLowerCase().charAt(t)==coppyOfScores.get(k).getName().toLowerCase().charAt(t))
                u--;
            else
                break;
```

Klasse: Sorting | Zeilen: 168 -215

Wichtig ist hier das auch doppelt vorkommende Buchstaben beachtet werden. Finden wir einen Buchstaben müssen wir auch noch darüber und darunter in der Liste schauen, sonst finden wir nicht alle Einträge die wir suchen. Diese Suche muss für jeden einzelnen Buchstaben passieren damit wir nach und nach die Namen aus sieben die nicht mit den Buchstaben übereinstimmen.

Heap Sort:

Am Anfang muss der sogenannte Heap erstellt werden. Dabei wird die Liste in ein binären Baum umgewandelt. Dies geschieht mit Hilfe der for-Schleife und dem rekursiven Aufruf von dem Befehl “heap”.

```
// Heap erstellen
for (int i = n / 2 - 1; i >= 0; i--)
    heap(al, n, i);
```

Klasse: Sorting | Zeilen: 125-127

Im Befehl “heap” selber wird zuerst die übergebene Variable “i” als größtes Element gespeichert. Die Kinder können jeweils mit 2*i+1 und 2*+2 ermittelt werden, da wir ja ein binären Baum erstellt haben. Danach wird geprüft ob das linke Kind größer ist als die Wurzel und ob das rechte Kind größer ist. Wenn es größer ist, wird es entsprechend getauscht. Dann wird die Methode rekursiv aufgerufen, um die Arrayliste komplett durchzugehen und immer wieder neue Teilbäume zu erstellen.

```
public static void heap(ArrayList<Player> al, int n, int i) {
    int largest = i; //Größte als Wurzel
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    //linkes Kind größer als die Wurzel
    if (l < n && al.get(l).getScore() < al.get(largest).getScore())
        largest = l;

    // rechtes Kind größer als das Größte
    if (r < n && al.get(r).getScore() < al.get(largest).getScore())
        largest = r;

    // größte ist größer als die Wurzel
    if (largest != i) {
        Player swap = al.get(i);
        al.set(i, al.get(largest));
        al.set(largest, swap);

        // rekursiver Durchgang durch den nächsten Teilbaum
        heap(al, n, largest);
    }
```

Klasse: Sorting | Zeilen: 141-163

Somit ist der Baum sortiert und das größte Element ist nun die Wurzel. Diese Element wechselt die Position an das Ende der Liste. Nun wird erneut das größte Element gesucht, jedoch nur von 0 bis zu der Schranke nachdem die Elemente schon sortiert sind. Wenn dies geschehen ist, ist die Liste absteigend sortiert.

```
//Bewege aktuelle Wurzel zum Ende
Player temp = al.get(0);
al.set(0, al.get(i));
al.set(i, temp);

//größter im Heap wird rausgesucht
heap(al, i, 0);
```

Klasse: Sorting | Zeilen: 155-158

Genutzte Algorithmen aus der Informatik 2 Vorlesung:
-Kürzester Weg Algorithmus (Verbesserung vom Dijkstra-Algorithmus)
Klasse: Node && Ghost | Zeilen: 530 -597
-Mergesort
Klasse: Sorting | Zeilen: 33 -120
-Heapsort
Klasse: Sorting | Zeilen: 122 -163
-Binary Search
Klasse: Sorting | Zeilen: 224 -239

6 Entwurf und Umsetzung

In diesem Projekt wurde das Java Runtime Environment 1.8 benutzt.
Außerdem wird die openFX-Version 13 für das Projekt benutzt.

Klassendiagramm

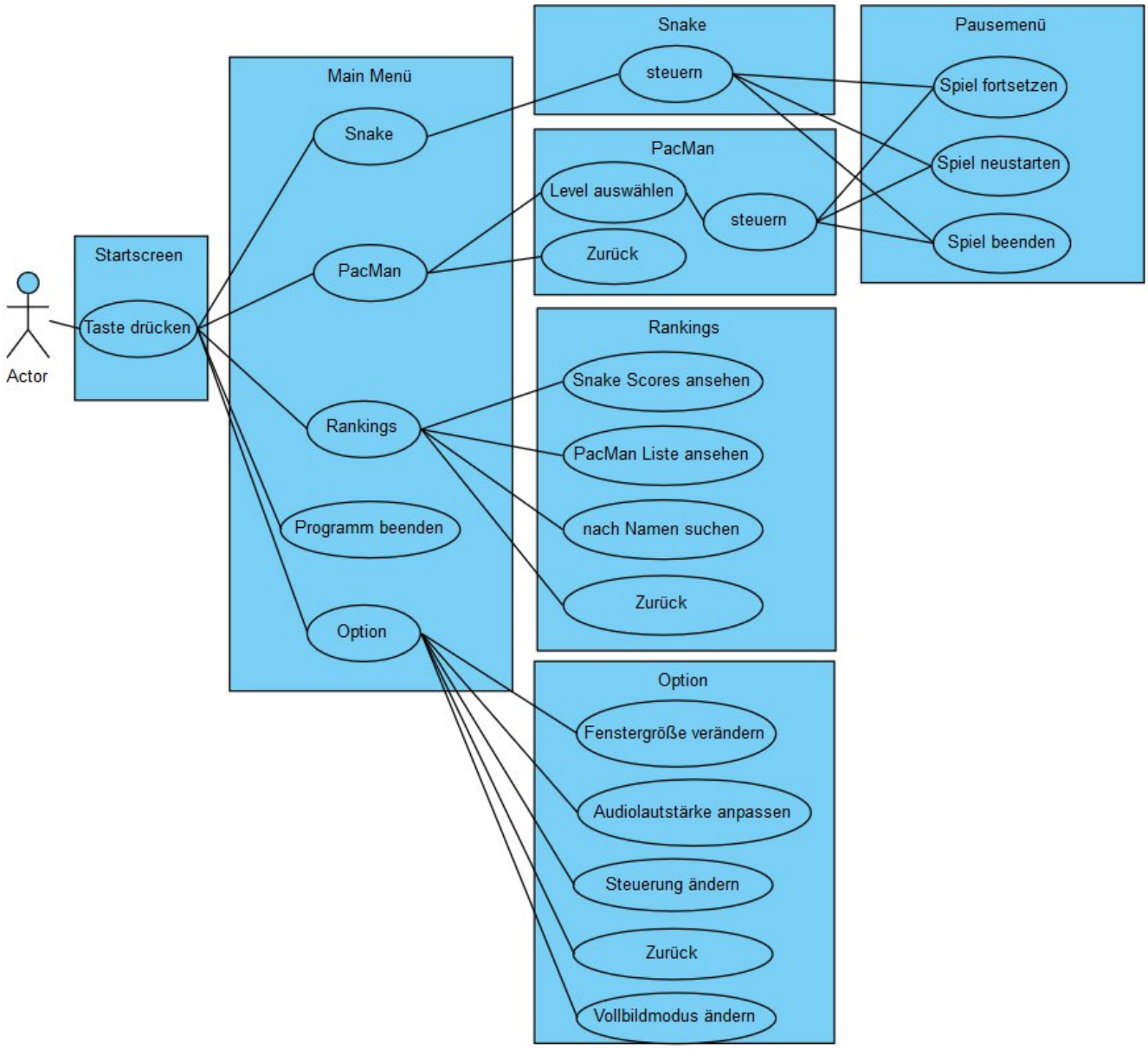
Klassendiagramm siehe: Arcade-Automat Klassendiagramm.pdf im Abgabeordner.

Sequenzdiagramm

Sequenzdiagramm siehe: Pacman loop Sequenzdiagramm.pdf im Abgabeordner.

Das Sequenzdiagramm beschreibt einen Durchlauf der Hauptschleife des vom Pacman Spiel.
Als erstes wird beim Pacman Objekt die Methode update() aufgerufen, welche die Position des Pacman verändert. Diese Methode liefert keine Rückgabe da sie lediglich die Objekte innerhalb der Pacman Klasse verschiebt.
Als nächstes wird überprüft ob ob der Pacman ein Objekt im Level aufgesammelt hat.
Auch hier keine Rückmeldung da alles innerhalb der Level Klasse geschieht.
Die Geister werden als nächstes aktualisiert sprich sie suchen einen neuen Pfad und werden weiter gesetzt.
Das Interface wird auch aktualisiert, da es z.B. den Score-Text aktualisieren muss.
Als nächstes wird in der Variable s der Rückgabelwert der checkColissionGhost() Methode gespeichert, welche auf dem pac Objekt ausgeführt wird.
0= kein Geist kollidiert mit dem Pacman
1=Geist kollidiert mit Pacman wobei der Pacman keinen Bonus vom Power-Ball besitzt.
2=Geist kollidiert mit Pacman wobei der Pacman den Bonus vom Power-Ball besitzt.

Im Fall 1 ruft sich das GameLoop Objekt selbst auf mit der pacDeath() Methode und bei Fall 2 mit der ghostDeath() Methode.
Beim Fall 0 wird die schleife wiederholt.


Actor
Startscreen
Taste drücken
Main Menü
Snake
PacMan
Rankings
Programm beenden
Option
Snake
steuern
PacMan
Level auswählen
steuern
Zurück
Rankings
Snake Scores ansehen
PacMan Liste ansehen
nach Namen suchen
Zurück
Option
Fenstergröße verändern
Audiolautstärke anpassen
Steuerung ändern
Zurück
Vollbildmodus ändern
Pausemenü
Spiel fortsetzen
Spiel neustarten
Spiel beenden

7 Evaluation

Eine wichtige Methode die bei Pacman getestet werden muss, ist die Bewegungsfunktion die alle Charaktere nutzen. Wir müssen sicherstellen das alle nötigen Objekte für den Spielverlauf mit bewegt werden und das wir pro Aufruf der Methode uns nur um 1 Pixel in die jeweilige Richtung bewegen.

```
public void move() {
    switch(getDirectionX()){
        case 1:
            actualMove(1,0);
            faceDirection(1,0);
            break;
        case -1:
            actualMove(-1,0);
            faceDirection(-1,0);
    }

    switch(getDirectionY()){
        case 1:
            actualMove(0,1);
            faceDirection(0,1);
            break;
        case -1:
            actualMove(0,-1);
            faceDirection(0,-1);
    }
}
```

Klasse: Character | Zeilen: 234 - 254

Der Charakter bewegt sich anhand der X und Y Richtung die er momentan besitzt. Die faceDirection() Methode ist abstrakt und verändert das aussehen des Charakters damit er auch visuell in die Richtung schaut in die er sich bewegt.

```
public void actualMove(int dx, int dy) {
    moveHitBox(dx,dy); //move hitbox
    if(level.checkColissionWalls(hitBox.getBoundsInParent())) {
        moveHitBox(-dx, -dy);
        animation.pause();
    }else {
        animation.play();
        x+=dx;
        y+=dy;
        moveForm();
        moveHitBox2();
    }
}
```

Klasse: Character | Zeilen: 268 - 280

Die actualMove() Methode ist das Herzstück der Bewegung aller Charaktere. Erst wird eine HitBox in die Richtung vorausgeschickt um zu schauen ob sie mit einer Wand kollidiert. Ist dies nicht der Fall, kann sich der Charakter dorthin bewegen. Ist eine Wand im Weg darf sich der Charakter nicht weiter bewegen und die Bewegungsanimation des Charakters muss gestoppt werden.
Für den ersten Fall sieht ein Test für eine Bewegung in X Richtung wie folgt aus:

```
public void move1() {
    pac.setDirectionX(1);
    pac.move();
    assertSame(1, pac.getX() );
    assertSame(0, pac.getY() );
    assertSame((int)pac.getForm().get(0).getTranslateX(),1);
    assertSame((int)pac.getForm().get(0).getTranslateY(),0);
    assertSame((int)(pac.getHitBox().getTranslateX()+pac.getHitBox().getWidth()/2),pac.getX());
    assertSame((int)(pac.getHitBox().getTranslateY()+pac.getHitBox().getWidth()/2),pac.getY());
    assertSame((int)(pac.getHitBox2().getTranslateX()+pac.getHitBox2().getWidth()/2),pac.getX());
    assertSame((int)(pac.getHitBox2().getTranslateY()+pac.getHitBox2().getWidth()/2),pac.getY());
}
```

Klasse: CharacterTest | Zeilen: 29 - 40

In weiteren Tests wird auch für negative Richtungen und Bewegung in Y oder in Y und X Richtung getestet.

```
public void move2() {
    pac.setDirectionY(1);
    pac.move();
    assertSame(0, pac.getY() );
    assertSame(0, pac.getX() );
    assertSame((int)pac.getForm().get(0).getTranslateX(), 0 );
    assertSame((int)pac.getForm().get(0).getTranslateY(), 0 );
    assertSame((int)(pac.getHitBox().getTranslateX()+pac.getHitBox().getWidth()/2),pac.getX());
    assertSame((int)(pac.getHitBox().getTranslateY()+pac.getHitBox().getWidth()/2),pac.getY());
}
```

Klasse: CharacterTest | Zeilen: 42 - 52

Wäre eine Wand Rechts vom Pacman würde ein Test für eine Bewegung in X Richtung so aussehen. Wichtig ist hier das die HitBox auch wieder am Ursprungsort ist.

Ein sehr nützlicher Test wurde genutzt um zu testen ob der Weg finde Algorithmus vom Geist auch das richtige Ziel gefunden hat.

```
public void everyPath() {
    red.pathFinding(new Point(pac.getX(),pac.getY()));
    assertSame(red.getPath().get(0),new Point(pac.getX(),pac.getY()));
}
```

Klasse: GhostTest | Zeilen: 33 - 36

Der Endpunkt des errechneten Weges muss dabei der erste Punkt in der Liste mit den Wegpunkten sein. Hier wurde der weg zum direkt Pacman berechnet.

Um Sicherzustellen das die Charaktere auch wieder an ihren Uhrsprungort gesetzt werden sobald das Spiel resetet wird oder das Spiel gestartet wird wurden diese Tests benutzt.

```java
@Test
public void reset() {
    pac.reset();
    assertSame(0, pac.getDirectionX() );
    assertSame(0, pac.getDirectionY() );
    assertSame(0, pac.getDirectionNX() );
    assertSame(0, pac.getDirectionNY() );
    assertTrue(pac.getStartX()==pac.getX() );
    assertTrue(pac.getStartY()==pac.getY() );
}

@Test
public void setup() {
    pac.setup(root);
    assertSame(0, pac.getDirectionX() );
    assertSame(0, pac.getDirectionY() );
    assertSame(0, pac.getDirectionNX() );
    assertSame(0, pac.getDirectionNY() );
    assertTrue(pac.getStartX()==pac.getX() );
    assertTrue(pac.getStartY()==pac.getY() );
}
```

Klasse: CharacterTest | Zeilen: 64 - 84

Auch die Richtungen müssen zurückgesetzt werden damit sich die Charaktere nicht sofort bewegen wenn das Spiel weitergeht.

```java
@Test
public void move() { //Testen von Verschieben der Koordinaten um eine Einheit nach links
    loop.getSnakeControler().refreshSnake();
    int pos = Snake.snakeList.get(0).getX()-UI.pointsizeX;
    assertTrue(Snake.snakeList.get(0).getX()==pos);
}
```

Dies wäre ein Test ob die Koordinaten der Schlange sich korrekt ändern wenn sich die Schlange bewegt.

Das vollständige Testen des Snake-Spiels und einiger PacMan- und Menüklassen war auf Grund von Problemen mit Bild- und Sounddateien nicht möglich.

8 Zusammenfassung und Ausblick

Wir hoffen wir konnten die 80er Jahre noch einmal zurück in die Gegenwart bringen um Nostalgie zu erwecken oder vielleicht auch einfach Menschen Freude am Spielen zu bereiten.
Wir haben es erreicht, dass man die Arcade-Spiele wie PacMan und Snake auf einem Computer mit einer lauffähigen Java-Version spielen kann. Der Automat ist so gestaltet das man leicht weitere Spiele hinzufügen kann. So könnte man in Zukunft eine Automaten machen, der alle Spiele dieses Zeitraums beinhaltet.
Zudem ist die Pacman-Version so konzipiert das man leicht neue Level ohne großen Aufwand erstellen kann. Man könnte eine Art Level-Builder programmieren um schnell seine eigenen pacman Level zu realisieren und zu spielen.

9 Literatur

Quellen:

8bit Font
https://fonts.google.com/specimen/Press+Start+2P?selection.family=Press+Start+2P
abgerufen: 10.05.2019

BackgroundVideo
https://www.youtube.com/watch?v=hbD6Cl4HCIk
abgerufen: 20.05.2019

MainMusic
https://www.youtube.com/watch?v=i94rqSASTLU&t=1128s
abgerufen: 20.05.2019

Pacman Sound effects
https://www.classicgaming.cc/classics/pac-man/sounds
abgerufen: 10.05.2019

Pacman Sound effects
http://samplekrate.com/drumkit.php?id=34
abgerufen: 10.06.2019

Pacman Death Sound
http://www.orangefreesounds.com/pacman-death-sound/
abgerufen: 10.06.2019

informationen über die Geister
https://dev.to/code2bits/pac-man-patterns--ghost-movement-strategy-pattern-1k1a
abgerufen: 19.06.2019

Früchte für die PowerUp's:
https://st2.depositphotos.com/2935785/5189/v/450/depositphotos_51899519-stock-illustration-set-of-pixel-icons-fruits.jpg
abgerufen: 18.05.2019

Backround Snake:
http://www.thelogicgame.com/games/mahjongunlimited/inc/background6.jpg
abgerufen: 20.05.2019

Death Sound:
https://www.youtube.com/watch?v=bPIyOaSwFos
abgerufen(25.06.2019

Original Spiel von Namco.