Dokumentation zum Spiel Breakout

Vorwort

Diese Dokumentation beschreibt die Umsetzung des Spiels „Breakout“ von M.H.. Die Umsetzung fand im im Rahmen des MOOC von OpenHPI statt.

Features und ausgewählte Codebeispiele

Nachfolgend wird erläutert welche besonderen Features umgesetzt wurden und ausgewählte Features werden mit Codebeispielen erklärt.

Aufbau der Wall

Die Methode buildWall in der Klasse Wall ist für die Definition der einzelnen Blöcke in der Wand zuständig. Beim Aufruf werden der Methode u.a. die Spalten und Reihenanzahl übergeben. Die Wand wird anhand der Konstanten für den sichtbaren Bereich oben in der Mitte platziert.

Es existiert ein besonderes Mauerstück (Brick), welches eine besondere Aktion ausführt. Dieses hier als magicBrick benannte Mauerstück wird per Zufallszahl (siehe erste Zeile) an eine zufällige Position in der Mauer platziert. Jedes Brick besitzt hierzu die boolean-Eigenschaft isMagicBrick, welche nur bei dem einen auf wahr gesetzt wird. Später wird das MagicBrick dazu verwendet, dem Spieler bei einem Treffer des Bricks zusätzliche Zeit zu geben. Der generelle Aufbau der Mauer erfolgt über zwei verschachtelte for-Schleifen, jeweils für die Spalten und Reihen. Darin wird insbesondere die Position der Bricks bestimmt.

```
private void buildWall(Game game, int rows, int columns) {

    int randomBrick = (int) ((Math.random()*(rows*columns))+1);
    boolean isMagicBrick=false;
    int brickNumber=0;

    int wallLeftBegin = (SCREEN_X / 2) - ((columns/2) * BRICK_WIDTH);
    int wallTopBegin = (SCREEN_Y / 4) - ((rows/2) * BRICK_HEIGHT);

    for(int row=1; row<=rows; row++){
        for(int col=1; col<=columns; col++){
            brickPosX = wallLeftBegin + (BRICK_WIDTH) * (col-1) + (BRICK_WIDTH/2);
            brickPosY = wallTopBegin + (BRICK_HEIGHT) * (row-1) + (BRICK_HEIGHT/2);
            brickNumber++;
            if(brickNumber==randomBrick){
                isMagicBrick=true;
            }else{
                isMagicBrick=false;
            }
            wall.add(new Brick(game, new Point(brickPosX, brickPosY),
                    new Dimension(BRICK_WIDTH, BRICK_HEIGHT), isMagicBrick));
        }
    }
}
```

In der Display-Methode wird ständig überprüft ob die Wall bereits „tot“ ist. Eine Schleife prüft hierzu jeden Brick. Ist jeder Brick tot, so wird die Variable „allBricksDead“ gesetzt, welche anhand der Methode isAllBricksDead für andere Klassen zur Verfügung steht.

```
@Override
public void display() {
    allBricksDead=true;
    for(Brick brick : wall){
        if(!brick.isDead()){
            brick.display();
            allBricksDead=false;
        }
    }
}

public boolean isAllBricksDead(){
    return allBricksDead;
}
```

Collision Logic

Eine der größten Herausforderungen ist die zuverlässige Kollisionserkennung des Balls mit den anderen Gegenständen. Hierzu wird beispielhaft die Kollision mit den Bricks erläutert und was in welchem Fall geschehen soll.

```
for(Brick brick : wall){

    if(!brick.isDead()){
        Line2D.Double brickLineBottom = new Line2D.Double(brick.getX()-brick.getWidth()/2, b
        Line2D.Double brickLineTop = new Line2D.Double(brick.getX()-brick.getWidth()/2, bric
        Line2D.Double brickLineLeft = new Line2D.Double(brick.getX()-brick.getWidth()/2, bri
        Line2D.Double brickLineRight = new Line2D.Double(brick.getX()+brick.getWidth()/2, br

        // Ball kommt von unten
        if(ball.isMovingUpwards()){

            if(ballCollidesWithLine(ball,brickLineBottom)!=null){
                ball.moveTo(ball.getX(), brick.getY()+(brick.getHeight()/2)+BALL_DIAM/2+1);
                ball.bounceY();
                brick.nextStatus();
                game.increaseScore(100);
                increaseTime(brick, timer);

            }
            else if(ballCollidesWithLine(ball,brickLineLeft)!=null){
                ball.moveTo(brick.getX()-(brick.getWidth()/2)-BALL_DIAM/2-1,ball.getY());
                ball.bounceX();
                brick.nextStatus();
                game.increaseScore(100);
                increaseTime(brick, timer);
            }
            else if(ballCollidesWithLine(ball,brickLineRight)!=null){
                ball.moveTo(brick.getX()+(brick.getWidth()/2)+BALL_DIAM/2+1,ball.getY());
                ball.bounceX();
                brick.nextStatus();
                game.increaseScore(100);
                increaseTime(brick, timer);
            }
        }
```

Die Methode checkCollision wird ständig aufgerufen um auf Kollisionen zu überprüfen. In dieser wird u.a. die Kollision mit den einzelnen Bricks überprüft. Zunächst wird für ein Brick für alle Seiten des Rechteck eine Linie erzeugt. Die Methode ballCollidesWithLine überprüft ob der Ball eine Linie überquert hat. Dies geschieht durch den Aufruf der Methode getIntersectionPointOfLines auf, welche als Eingaben zwei Linien erwartet. Die erste Linie ist die zuletzt zurückgelegte Strecke des Balls. Hierzu wurden die Variablen oldBallLocation.X und oldBallLocation.Y erstellt. Es wird eine Linie von der alten Ballposition zur aktuellen Ballposition erzeugt. Die zweite Linie ist die Außenlinie des Bricks. Als Ergebnis liefert die methode getIntersectionPointOfLines null oder den Schnittpunkt zurück, was die Methode ballCollidesWithLine weiter hochreicht. Somit kann, je nach Richtung des Balls, bei Kollision mit einer Linie des Bricks genau definiert werden was dann geschehen soll. In diesem Fall ein Wechsel der Richtungen X oder Y und die Erhöhung der Punktzahl.

Diese Kollisionserkennung wird ebenfalls für das Paddle verwendet, wie nachfolgender Code zeigt. Zusätzlich wird das Verhalten beim Abprallen des Balls vom Paddle definiert. Wenn paddle und Ball aus der selben Richtung kommen, so soll der Winkel flacher werden, was physikalisch zu erwarten wäre. Das Gegenteil geschieht bei entgegengesetzter Richtung. Wird das Paddle nicht bewegt, so erhält der Ball wieder die ursprünglichen 45-Grad Winkel. Hierzu wurden in Ball die Methoden setDirectionLower, setDirectionHigher und setDirectionNormal definiert, welche den Vektor des Balls ändern.

```
} else if(ballCollidesWithLine(ball,paddleLineTop)!=null){
    ball.moveTo(ball.getX(), ball.getY()-(paddle.getHeight()/2)-BALL_DIAM/2-1);

    if(paddle.isMovingRight() && ball.isMovingRight()){
        ball.setDirectionLower();
    }else if(paddle.isMovingRight() && ball.isMovingLeft()){
        ball.setDirectionHigher();
    }else if(paddle.isMovingLeft() && ball.isMovingLeft()){
        ball.setDirectionLower();
    }else if(paddle.isMovingLeft() && ball.isMovingRight()){
        ball.setDirectionHigher();
    }else if(!paddle.isMovingLeft() && !paddle.isMovingRight()){
        ball.setDirectionNormal();
    }

    ball.bounceY();
```

Zeitanzeige für den GameScreen

Der Zeitcounter wird mithilfe eines Threads umgesetzt. Hierzu wird die Klasse TimerCounter definiert, welche die Sekunden in einem Thread herunterzählt. Die run-Methode des Threads enthält eine Schleift, welche so lange läuft bis die Variable killed=true ist. Diese wird auf true gesetzt, wenn die aktuelle Spielrunde beendet ist, damit der Thread nicht weiterläuft. Anhand der Variable „running“ innerhalb der while-Schleife wird geprüft ob das Spiel pausiert ist. Eine Pausierung erfolgt in der Klasse GameScreen, sobald der Ball ins Aus fliegt oder die Leertaste gedrückt wurde.

Die Klasse Timer ist für die Anzeige der Zeit verantwortlich. Sie erhält über Umwege die aktuelle

Sekundenzahl des Threads. Diese verwendet sie zur Anzeige eines Zeitbalkens. Für diesen wurde sich entschieden, damit die verbliebene Zeit auch bei schnellem Spiel leicht vom Spieler zu erfassen ist. Innerhalb der display-Methode ist folgender Code für die Anzeige des Zeitbalkens verantwortlich:

```
for(int i=0; i<=secInt; i++){
    if(secInt<=10){
        game.fill(255,0,0);
    }else if(secInt<=30){
        game.fill(210,105,30);
    }else{
        game.fill(34,139,34);
    }
    game.strokeWeight(0);
    game.rect(i*3+100, game.height-23, 1, 8);
}
```

Je nach Zeit färbt sich der Balken anders.

Beschleunigung des Balls in Abhängigkeit der Zeit

In der update-methode von GameScreen ist folgender Code zu finden:

```
if(t.getSeconds()%10==0 && !speedUpdated){
    speedUpdated=true;
    ballSpeed++;
}else if(t.getSeconds()==145 || t.getSeconds()==85 || t.getSeconds()==45 || t.getSeconds()==25){
    speedUpdated=false;
}
currentBall.setSpeed(ballSpeed);
```

Dieser beschleunigt den Ball, wenn die aktuelle Sekundenzahl durch ohne Rest durch 10 teilbar ist und die Variable speedUpdated false ist. Die Variable ist notwendig, da ansonsten der Ball während dieser Sekunde um das vielfache beschleunigen würde (die update-methode wird schließlich kontinuierlich aufgerufen). speedUpdated wird regelmäßig zwischen beliebigen Zehner-Schritten zurückgesetzt. In diesem Falle bedeutet ein zurücksetzen bei Sekunde 145, dass der Ball in der Sekunde 140 um Eins beschleunigt wird. Die nächste Beschleunigung ist bei Sekunde 80, da speedUpdated bei Sekunde 85 zurückgesetzt wurde...

Known Bugs und nicht umgesetzte Features

Aus Zeitgründen konnten folgende Bugs nicht behoben / folgende Features nicht implementiert werden:

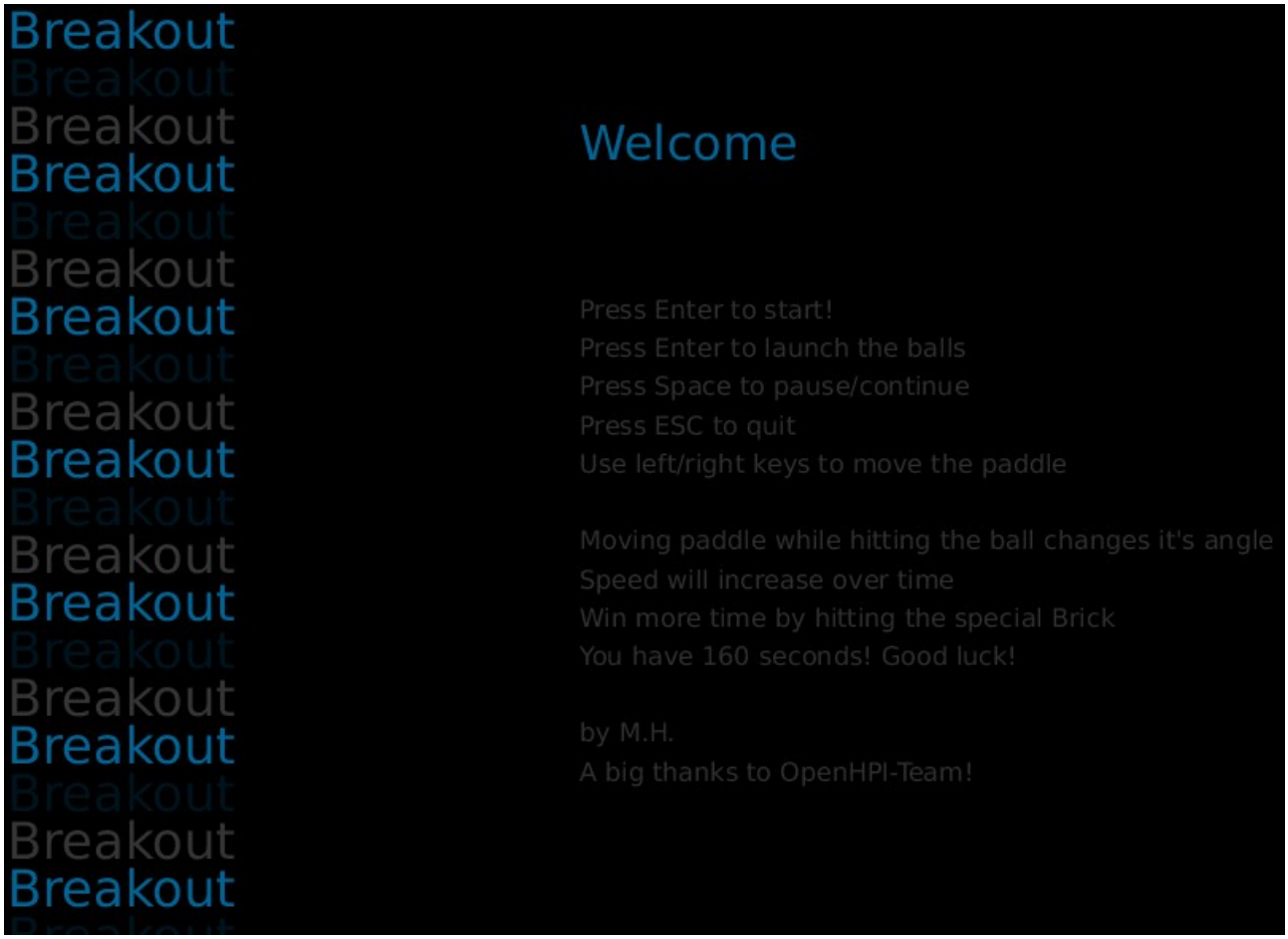
- Der Ball kann in seltenen Fällen durch die Bricks fliegen. Hierzu müsste nochmal die Kollisionserkennung analysiert werden. Bei Ideen bitte melden!

- Highscores wurden nicht umgesetzt

Screenshots & Anleitung

Startbildschirm

Die Anleitung ist dem Startbildschirm des Spiels zu entnehmen.



GameScreen

Spielmodus mit voller Zeitanzeige. Wenn der bricks mit der Beschriftung „+10“ getroffen wird, so bekommt der Spieler mehr Zeit, was auch im Zeitbalken berücksichtigt wird. Der +10 bricks kann entsprechend seiner Lebensdauer 3-mal um 10 Sekunden die Zeit erhöhen.

Der Schwierigkeitsgrad es Spiels ist recht hoch. Durch überlegtes Bewegen des Paddles kann der Ball auf die Bricks gezielt werden (siehe CollisionLogic), damit alle Bricks in der gegebenen Zeit getroffen werden.

Game-Over Screen

Neben dem Game-Over Screen gibt es noch den Win-Screen. Da der Screenshot hierzu ein erfolgreiches Spiel voraussetzt und Sonntags-Abends hierzu keine Zeit ist, fehlt der Screenshot ;) ...Viel Spass beim spielen!