|  |  |
| --- | --- |
| Gruppe Schmid, Platzer, Künzi, 28 September 2016 | Klasse I3q |

|  |
| --- |
|  |

Projekt Star Blaster

Technische Dokumentation

|  |
| --- |
|  |

|  |  |
| --- | --- |
| **Projektname** | Star Blaster |
| **Version** | 0.5 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Status** | in\_Arbeit | in\_Pruefung | genehmigt |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | X |  |  |  |  |  |

|  |  |
| --- | --- |
| Personenkreis | |
| Autoren | Michael Schmid, Casimir Platzer, Kevin Künzi |

|  |  |  |  |
| --- | --- | --- | --- |
| Kontrolle | | | |
| **Wann** | **Version** | **Wer** | **Beschreibung** |
| 28.09.2016 | 0.1 | Kevin Künzi | Dokument erstellt |
| 29.12.2016 | 0.2 | Casimir Platzer | Kapitel angepasst |
| 06.01.2017 | 0.3 | Kevin Künzi | Bilder eingefügt |
| 16.01.2017 | 0.5 | Michael Schmid | Kapitel angepasst. |
|  |  |  |  |

Inhaltsverzeichnis

1 Einleitung 3

2 Package Main 4

2.1 Klasse Game 4

2.2 Game Panel 4

2.3 Config 4

2.4 ArgumentChecker 4

3 Package GameState 5

3.1 Klasse GameState 5

3.2 Klasse GameStateManager 5

3.3 Menü-Klassen 6

3.4 Klasse StateSinglePlayer 6

3.5 Klasse StateMultiPlayer 6

4 Package TileMap 7

4.1 Klasse Tile 7

4.2 Klasse TileMap 7

5 Package Input 9

5.1 Klasse InputHandler 9

6 Package Entity 10

6.1 Enum ObjectType 10

6.2 Enum Team 10

6.3 Klasse SpaceObject 10

6.4 Klasse SpaceObjectMoving 11

6.5 Klasse Animation 12

7 Package Painter 13

7.1 Klasse PaintComponent 13

8 Package Pickups 14

8.1 Klasse EnergyPickup 14

8.2 Klasse HealthPickup 14

8.3 Klasse MissilePickup 15

8.4 Klasse MissileUpgrade 16

8.5 Klasse PhaserUpgrade 16

# Einleitung

Super Star Battles ist ein in Java geschriebenes Action Spiel. Graphisch setzt es auf Swing und TileMaps. Genauere Infos zum Inhalt des Spiels befinden sich in der Spielbeschreibung.

Diese Dokumentation soll eine rasche Übersicht über die verwendeten Klassen und den Gedanken dahinter geben. Dabei soll auf die wichtigsten Besonderheiten des Codes eingegangen werden. Grundkenntnisse in der Java-Programmierung werden für dieses Dokument vorausgesetzt, selbsterklärende Methoden, wie zum Beispiel getter- und setter-Methoden werden hier nicht näher aufgeführt. Genauere Infos zu den Funktionen finden Sich in den Java Code Kommentaren.

# Package Main

Im Main Package befinden sich 4 Klassen. Diese müssen für Anpassungen/Erweiterungen des GamePlays normalerweise nicht angepasst werden.

## Klasse Game

Beinhaltet die main-Methode. Die Klasse startet ein JfFrame und setzt ein GamePanel als ContentPane.

## Game Panel

Das GamePanel beinhaltet das Swing Image sowie den GameStateManager. Im Game Panel findet der Game Loop statt, welcher dafür sorgt, dass das Spiel in konstanten Intervallen von 16 Millisekunden die beiden Methoden update() und draw() des aktuellen GameStates aufgerufen werden. Das gewählte Intervall von 16 Millisekunden führt dazu, dass das Spiel mit 60 Bildern pro Sekunde (FPS) ausgeführt wird.

Ebenfalls implementiert die Klasse GamePanel die Klasse KeyListener. Mit den beiden Methoden keyPressed und keyReleased werden die vom Spieler eingegebenen Tastatur-Befehle an den GameStateManager weitergegeben.

Mehr Infos zu dem Game Loop Pattern befinden sich auf der sehr empfehlenswerten Seite: <http://gameprogrammingpatterns.com/game-loop.html>

## Config

In dieser Datei werden einige Balance Technische Variablen gesetzt, wie zum Beispiel der Startgesundheit und Bewegungsgeschwindigkeit der Spieler oder der Schaden der Standartrakete. Diese Zentralisierung der Variablen hat zum Ziel, das Spiel auf einfache Weise zu balancen.

## ArgumentChecker

Casimir

# Package GameState

Die Verwaltung der verschiedenen Spielzustände wird über verschiedene GameStates realisiert. Ein GameState kann ein spielbares Level beinhalten oder auch ein Menü, von welchem aus der Spieler zu wiederum anderen GameStates navigieren kann, zum Beispiel vom Hauptmenü (GameState StateMenu) zum Einspieler-Modus (StateSinglePlayer).

Ein GameState ist ein „Level“ des Spiels. Jeder Level und Modus des Spiels hat eine eigene gameState Klasse. In der Spielversion 1.00 sind, dass die zwei gameStates „OnePlayerVsBotState“ und „TwoPlayerLevel1State“. Der gameState beinhaltet unterschiedliche Models ( Klassen des Package „models“ wie Players, Raketen, Powerups) beeinhalten. Der Vorteil dieser Klassen liegen in der einfachen Erweiterbarkeit des Spiels. Für ein neues Level müssen nur ein neuer GameState sowie ein neues TileMap erstellt werden.

## Klasse GameState

Die abstrakte Klasse GameState bildet die Grundlage für sämtliche Instanzen von GameStates. Für sämtliche Objekte, welche später in den spielbaren Levels vorkommen, werden separate Listen geführt. Momentan gibt es folgende Listen:

* listPlayer
* listEnemy
* listTurret
* listPhaser
* listPhaserBig
* listMissile
* listLaser
* listHealthPickup
* listEnergyPickup
* listMissilePickup
* listPhaserUpgrade
* listMissileUpgrade

Wird eine neue Klasse eingeführt, zum Beispiel eine neue Art Waffe oder Raumschiff, dann muss für diese eine neue Liste eingeführt werden, sowie Methoden zum Hinzufügen oder Entfernen von Objekten für dieser Liste.

Ausserdem hat jeder GameState die Methode getGSM, welche als Rückgabewert den aktuellen GameStateManager hat.

## Klasse GameStateManager

Der GameStateManager beinhaltet alle GameStates und verwaltet diese. Die Klasse reicht den GameLoop vom GamePanel an den CurrentState (momentan aktiver GameState weiter).

Beim Wechsel zu einem bestimmten GameState wird jeweils die init-Methode des neu zu setzenden GameStates ausgeführt.

## Menü-Klassen

Mit den Klassen **StateMenu**, **StateControlScreen**, **StateScoreScreen** und **StateScoreScreenMulti** wird die gesamte Menüführung des Spiels geregelt. Beim Starten des Spiels ist im GameStateManager der aktuelle GameState StateMenu. Die Navigation in sämtlichen Menüs erfolgt über die Cursor-Tasten. Mit der Enter-Taste wird die aktuelle Wahl bestätigt und entsprechend der nächste GameState im GameStateManager als aktueller GameState gesetzt. Mit der Escape-Taste wird in jedem Menü wieder zurück zum Hauptmenü gewechselt, also der aktuelle GameState im GameStateManager wieder auf StateMenu gesetzt.

## Klasse StateSinglePlayer

In der Klasse StateSinglePlayer, welche ebenfalls die Klasse GameState erweitert, wird der Einspieler-Modus des Spiels verwaltet. Nebst den Listen mit sämtlichen Objekten, welche in der Superklasse GameState definiert sind, hat diese Klasse noch eine Instanz der Klasse TileMap zur Darstellung der Spielwelt (Map). Zum Übergeben der zu verwendenden Tiles in der Map wird die Methode loadTiles der Instanz von TileMap aufgerufen. Der Aufbau der Map selber wird mit der Methode loadMap der Instanz von TileMap aufgerufen.

In der Methode init wird beim Wechsel zum GameState StateSinglePlayer die aktuelle System-Zeit in der Variable timeStart gespeichert. Diese wird dazu verwendet, um die verstrichene Zeit während des Spiels festzuhalten. Dazu wird bei jedem Aufruf der update-Methode die Startzeit von der aktuellen Systemzeit subtrahiert.

In der Methode update wird geprüft, ob das Spiel beendet ist, entweder weil die Zeit abgelaufen ist, weil alle gegnerischen Schiffe oder weil das Schiff des Spielers selber zerstört wurde(n). Um zu prüfen, ob alle gegnerischen Schiffe und Turrets zerstört wurden, werden in dem GameState die beiden boolean-Werte allShipsDestroyed und allTurretsDestroyed geführt. Jedes Mal, wenn ein Schiff zerstört und damit aus der entsprechenden Liste entfernt wird, wird geprüft ob dies das letzte Schiff in der Liste war, und je nach dem der Wert allShipsDestroyed auf true gesetzt. Das Vorgehen ist entsprechend für die Variable allTurretsDestroyed dasselbe, mit der entsprechenden List für die Turrets.

Wenn das Spiel zu Ende ist, wird im GameStateManager der aktuelle GameState auf StateScoreScreen gewechselt.

## Klasse StateMultiPlayer

Mit der Klasse StateMultiPlayer wird der Mehrspieler-Modus des Spiels verwaltet. Die Spielwelt wird hier gleich wie bei der Klasse StateSinglePlayer über eine Instanz der Klasse TileMap erstellt. Dazu werden ebenfalls die beiden Methoden loadTiles und loadMap der Klasse TileMap aufgerufen.

In der Methode update wird ebenfalls wieder geprüft, ob das Spiel beendet ist. Dies ist dann der Fall, wenn entweder das Raumschiffs des Spieler 1 oder des Spieler 2 zerstört wurde. Dazu werden die Methoden getDestroyed der Klasse Player für die beiden Spieler aufgerufen. Wenn eines der beiden Schiffe zerstört wurde, wird im GameStateManager der aktuelle GameState auf StateScoreScreenMulti gewechselt.

# Package TileMap

Die Spielwelt, in welcher der Spieler sein Raumschiff bewegen kann, ist nach dem Prinzip einer TileMap aufgebaut. Das bedeutet, dass das Spielfeld aus mehreren "Kacheln", den Tiles, besteht, welche jeweils eine eigene Textur und einen eigenen Typ haben. Dies wird mit den nachfolgenden beiden Klassen Tile und TileMap realisiert.

## Klasse Tile

**Variablen**

**private int type:** beinhaltet die Information, ob andere Objekte mit dieser Tile kollidieren oder nicht.

**Konstruktor**

Dem Konstruktor wird ein Bild mit der Textur für die Tile, sowie ein Integer-Wert, welcher den Typ der Kachel definiert, wobei 0 = "NORMAL" und 1 = "BLOCKED".

**Methoden**

Keine besonderen Methoden.

## Klasse TileMap

**Variablen:**

**private int tween:** der Wert wird im Konstruktor definiert und wird bei der Kameraführung (Map-Positionierung) verwendet. -> Methode setPosition.

**private int[][] map:** Informationen zur Map werden in einem zwei-dimensionalen Integer-Array gespeichert

**private Tile[][] tiles:** Sämtliche Tiles werden in diesem zweidimensionalen Array gespeichert.

**Konstruktor:**

Dem Konstruktor wird ein Integer-Wert übergeben, welcher die Grösse der einzelnen Tiles definiert. Die beiden Werte numRowsToDraw und numColsToDraw, welche die Information enthalten, wie viele Zeilen und Spalten später gezeichnet werden sollen, werden ebenfalls aus diesem Wert abgeleitet.

**Methoden**

**public void loadTiles(String s)**

Der übergebene String muss ein Pfad zu einer Bilddatei sein, welches die gewünschten Texturen für die Tiles enthält.Dieses Bild wird anhand der TileSize mit einem geschachteltetn Loop unterteilt in mehrere Einzelbilder.

**Achtung:** Momentan ist hier fix hinterlegt, dass das übergebene Bild in 14 Reihen von Teilbildern unterteilt wird, da das von uns verwendete Textur-File entsprechend aufgebaut ist.

**public void loadMap(String s)**

Mit dieser Methodewird ein Text-File geladen, mit welchem der Aufbau der Map bestimmt wird, also die Anzahl Tiles und welches Tile an welcher Stelle steht.

**public int getType(int row, int col)**

Diese Methode gibt als Rückgabewert den entsprechenden integer-Wert des Tiles zurück, welches mittels übergebener Zeile und Spalte ermittelt wird.

**public void setPosition(double x, double y)**

Wenn der Spieler sein Raumschiff durch die Map bewegt muss sich die Karte mitbewegen, was mit dieser Methode sichergestellt wird. Die übergebene Position wird noch mit dem tween-Wert angepasst, damit die Kamera-Bewegung dabei etwas flüssiger wird.

Die private Methode private void fixBounds() wird aufgerufen, damit die Position der Karte nicht unter den Mindestwert oder über den Höchstwert fällt.

**public void draw(Grahics2D g)**

Mit den Variablen rowOffset und colOffset ist der Wert gegeben, ab wo die Karte gezeichnet werden muss. Mit den beiden Werten numRowsToDraw und numColsToDraw ist ausserdem bekannt, wieviele Tiles gezeichnet werden müssen.

In einem verschachtelten Loop werden für die Zeilen und Spalten die Tiles aus der map (int[][]) ausgelesen und mit der drawImage-Methode vom übergebenen Graphics2D-Objekt gezeichnet.

# Package Input

## Klasse InputHandler

Die Klasse InputHandler ist dafür zuständig, dass der Input des Spielers korrekt verarbeitet wird. Der Spieler gibt dabei seine Inputs über die Computer-Tastatur ein. Die Klasse GamePanel, welche die Klasse KeyListener implementiert, gibt die Inputs weiter an den GameStateManager, welche diese wiederum zusammen mit dem aktuellen GameState an den InputHandler weitergibt.

**Variablen**

Keine Besonderheiten.

**Konstruktor**

Keine Besonderheiten.

**Methoden**

**public void handlePressed(int k, GameState state)**

Abhängig vom übergebenen GameState werden mit dieser Methode die vom Benutzer gepressten Tasten verwaltet. Der übergebene Integer-Wert wird verglichen mit bestimmten KeyEvents, so wird für den GameState StateSinglePlayer mit dem KeyEvent VK\_LEFT zum Beispiel die Methode player.setLeft(true) aufgerufen.

Eine Ausnahme bildet dabei die Escape-Taste. Unabhängig vom aktiven GameState wird dabei die Methode goToMenu() vom GameStateManager aufgerufen, worauf dieser den aktuellen GameState zum GameState StateMenu wechselt.

**public void handleReleased(int k, GameState state)**

Für die beiden GameStates StateSinglePlayer und StateMultiPlayer ist es ebenfalls notwendig, dass auch der Input in Form von losgelassen Tasten an den InputHandler übergeben werden. Das Vorgehen ist dabei dasselbe wie bei der Methode handlePressed.

# Package Entity

In diesem Package enthalten ist die Klasse SpaceObject, welche zusammen mit deren Unterklasse SpaceObjectMoving die Grundlage für sämtliche Objekte bildet, welche auf der Map dargestellt werden: Raumschiffe, Geschosse inklusive Explosionen, Turrets, Pickups und Upgrades.

## Enum ObjectType

In sämtlichen Unterklassen von SpaceObject wird ein ObjectType definiert aufgrund der in der Enum-Klasse vorgebenen Ausprägungen.

## Enum Team

In sämtlichen Unterklassen von SpaceObject wird ein Team definiert aufgrund der in der Enum-Klasse vorgebenen Ausprägungen.

## Klasse SpaceObject

Jedes Objekt, welches zu einer TileMap hinzugefügt werden soll, erweitert die Klasse SpaceObject.

**Variablen**

**protected GameState stateActual:** mit stateActual wird der GameState definiert, zu welchem das SpaceObject hinzugefügt wird. Der GameState wird dem Konstruktor übergeben.

**protected Animation animation:** die Darstellung der SpaceObject-Instanzen erfolgt über Instanzen der Klasse Animation, welche für jeden Zustand des SpaceObjects die entsprechenden Bilder aus der Klasse PaintComponent laden.

**protected int angle:** für jedes SpaceObject wird mit dieser Variable der Winkel festgehalten. Je nach Winkel wird bei der Darstellung das Bild des SpaceObjects entsprechend rotiert.

**protected TileMap tileMap:** mit der Variable tileMap wird die Instanz von TileMap definiert, auf welcher das SpaceObject hinzugefügt wird. Die TileMap wird dem Konstruktor übergeben.

**protected double xmap** und **protected double ymap:** mit diesen Variablen wird die aktuelle Position der Map festgehalten. Diese Werte werden zum Beispiel in der Methode notOnScreen() benötigt

**protected int cwidth** und **protected int cheight:** mit den beiden Variablen werden die Dimensionen für die Kollisions-Box für das SpaceObject definiert. Diese kann theoretisch unterschiedlich von der effektiven Grösse des Objekts sein, also von der Grösse des Bildes, welches auf der TileMap gezeichnet wird.

**Konstruktor**

Dem Konstruktor werden eine TileMap und ein GameState übergeben. Damit ist für jedes SpaceObject später eindeutig definiert, auf welcher TileMap dieses zu positionieren ist und in welchem GameState das Objekt zu der entsprechenden Liste hinzugefügt werden muss.

**Methoden**

**Public boolean intersects(SpaceObject o)**

Die Methode prüft, ob sich das SpaceObject mit dem in der Methode übergebenen SpaceObject überschneidet. Dazu wird die intersects-Methode von der Klasse Rectangle verwendet, die Dimensionen der Rectangles der SpaceObjects wiederum basieren auf den Werten cwidth und cheight.

**Public boolean notOnScreen()**

Die Methode prüft, ob das SpaceObject auf der angezeigten Spielfläche erscheint oder nicht. Der Rückgabewert „true“ bedeutet in dem Fall, dass das Objekt nicht in dem Bereich der auf dem Bildschirm angezeigten Spielfläche enthalten ist.

## Klasse SpaceObjectMoving

Die Klasse erbt von der Klasse SpaceObject. Die Idee dahinter ist, dass sich nicht alle Objekte auch bewegen können und dass somit zur besseren Übersicht der Code in zwei Klassen aufgeteilt werden kann.

**Variablen**

protected double dx und protected double dy: mit diesen beiden Variablen ist geregelt, in welche Richtung sich das SpaceObject bewegt.

**protected double xdest** und **protected double ydest:** mit diesen Variablen wird die Position definiert, welche das Objekt mit dem nächsten Update haben soll.

**Konstruktor**

Dem Konstruktor werden eine TileMap und ein GameState übergeben. Mit diesen Werten wird wiederum der Konstruktor der Superklasse SpaceObject aufgerufen.

**Methoden**

**public void calculateCorners(double x, double y)**

Für die aktuelle Position (x und y) des Objekts werden die Typen der umliegenden Tiles geprüft und die entsprechenden boolean-Werte für das SpaceObjectMoving gesetzt.

**public void checkTileMapCollision()**

Mit dieser Methode wird geprüft, ob das Objekt mit dem nächsten Update (Update-Methode vom GameStateManager) mit der Map kollidiert oder nicht. Bei einer Kollision werden dabei die Richtungsvektoren (dx und / oder dy) entsprechend auf 0 gesetzt.

**public void getNextPosition()**

Abhängig von den boolean-Werte für die Richtung des Objekts left, right, up und down werden die beiden Richtungsvektoren dx und dy, sowie der Winkel des Objekts bestimmt.

## Klasse Animation

Objekte, welche auf der Map angezeigt werden, bestehen aus verschiedenen BufferedImage-Arrays. Mit der Klasse Animation wird definiert, wann welche Bilder für ein Objekt gezeichnet werden soll und wie lange dabei die einzelnen Bilder angezeigt werden sollen.

**Variablen**

**private BufferedImage[] frames:** die verschiedenen Einzelbilder für eine bestimmte Animation werden mit dieser Variable definiert

**private long delay:** mit der Variable delay wird angegeben, wie lange jedes Bild einer bestimmten Animation angezeigt wird

**private boolean playedOnce:** wenn die Animation einmal abgespielt wurde, wird dieser Wert auf true gesetzt. Dies ist zum Beispiel für die Animation von Explosionen wichtig, da diese nicht mehrmals abgespielt werden sollen.

**Konstruktor**

Keine Besonderheiten.

**Methoden**

**public void setFrames(BufferedImage[] frames)**

Normale setter-Methode, wobei beim Aufruf dieser Methode jeweils die Startzeit der Animation festgehalten wird.

**public void update()**

Die Startzeit der Animation wird benötigt, um die verstrichene Zeit (Variable elapsed) zu berechnen. Sobald diese Zeit grösser als der gesetzte delay-Wert ist, wird das nächste Bild aus dem BufferedImage-Array als aktuell zu zeichnendes Bild definiert. Sobald jedes Bild einmal während der gesamten delay-Dauer angezeigt wurde, wird der Wert playedOnce auf „true“ gesetzt.

# Package Painter

In der Painter Package existiert nur eine einzige Klasse: die PaintComponent. Mit dieser Klasse wollen wir das zeichnen der Objekte vom Objekt selbst abkapseln. So weiss zum Beispiel die Turret Klasse nicht, wie es sich selbst zeichnet. Es liefert der PaintComponent nur seine Daten (Grösse sowie Position), wie der Turret zu zeichnen ist liegt ganz in der PaintComponent.

## Klasse PaintComponent

Die Klasse ist für das gesamte Zeichnen aller Objekte / States verantwortlich. Mit der PaintComponent wollen wir das Zeichnen der Objekte vom Objekt selbst abkapseln. So weiss zum Beispiel die Turret Klasse nicht, wie es sich selbst zeichnet. Es liefert der PaintComponent nur seine Daten (Grösse sowie Position), wie der Turret zu zeichnen ist liegt ganz in der PaintComponent.

**Variablen**

**private State state:** Die Klasse enthält jeden State, damit Sie auf die Objekte darin zugreifen kann.

**private bufferedImage images:** Die Klasse enthält mehrere BuffferedImages.

**private boolean playedOnce:** wenn die Animation einmal abgespielt wurde, wird dieser Wert auf true gesetzt. Dies ist zum Beispiel für die Animation von Explosionen wichtig, da diese nicht mehrmals abgespielt werden sollen.

**Konstruktor**

Im Konstruktor werden alle Bilder aus den Dateien gelesen und in die diversen bufferedImage Variablen gespeichert. Dies hat den Vorteil, dass die Bilddateien nur einmalig beim instanzieren der PaintComponent geladen werden müssen.

**Methoden**

**public void drawMultiplayer(Graphics g, StateMultiPlayer multiPlayer)**

Jeder State hat seine eigene Draw Methode. Stellvertretend für alle wird hier kurz drawMultiplayer erklärt.

Diese Methode zeichnet alle SpaceObjects (i.e. Pickups, Spieler, Gegner, Schüsse) sowie den Hintergrund und die Player Status (Spielergesundheit etc). Dabei wird durch jede SpaceObject Liste des States durchiteriert und das Object dann auf g gezeichnet.

# Package Pickups

In diesem Package sind die einzelnen Pickups implementiert. Jeder der Klassen extends SpaceObject.

## Klasse EnergyPickup

Die Klasse implementiert das EnergyPickup, welches dem Schiff, das darüberfliegt, eine gewisse Anzahl Energy zurückgibt.

**Variablen**

**private int timer:** Ein Timer, welche für das reaktivieren der Upgrades nötig ist.

**private boolean active:** Speichert den «Status» des Pickups.

**private int amount:** Die Anzahl der wiederhergestellten Energie, wenn ein Schiff über das Pickup fliegt.

**Konstruktor**

Keine Besonderheiten.

**Methoden**

**public void update()**

Je nach der boolean Variable active macht diese Funktion folgendes:

Falls active true ist, überprüft die Methode, ob eine Kollision mit einem Spieler vorhanden ist. Falls ja, wird bei dem Spieler die Methode addEnergy(int amount) aufgerufen und das EnergyPickup setzt active auf false.

Falls active false ist, wird der Timer reduziert. Falls der Timer 0 wird, wird das Pickup wieder auf active = true gesetzt.

In beiden Fällen wird danach die animation geupdatet.

## Klasse HealthPickup

Die Klasse implementiert das HealthPickup, welches dem Schiff, das darüber fliegt, eine gewisse Anzahl Gesundheit zurückgibt.

**Variablen**

**private int timer:** Ein Timer, welche für das reaktivieren der Upgrades nötig ist.

**private boolean active:** Speichert den «Status» des Pickups.

**private int amount:** Die Anzahl der wiederhergestellten Gesundheit, wenn ein Schiff über das Pickup fliegt.

**Konstruktor**

Keine Besonderheiten.

**Methoden**

**public void update()**

Je nach der boolean Variable active macht diese Funktion folgendes:

Falls active true ist, überprüft die Methode, ob eine Kollision mit einem Spieler vorhanden ist. Falls ja, wird bei dem Spieler die Methode addHealth(int amount) aufgerufen und setzt active auf false.

Falls active false ist, wird der Timer reduziert. Falls der Timer 0 wird, wird das Pickup wieder auf active = true gesetzt.

In beiden Fällen wird danach die animation geupdatet.

## Klasse MissilePickup

Die Klasse implementiert das MissilePickup, welches dem Schiff, das darüber fliegt, eine gewisse Anzahl Raketenmunition zurückgibt.

**Variablen**

**private int timer:** Ein Timer, welche für das reaktivieren der Upgrades nötig ist.

**private boolean active:** Speichert den «Status» des Pickups.

**private int amount:** Die Anzahl der wiederhergestellten Munition, wenn ein Schiff über das Pickup fliegt.

**Konstruktor**

Keine Besonderheiten.

**Methoden**

**public void update()**

Je nach der boolean Variable active macht diese Funktion folgendes:

Falls active true ist, überprüft die Methode, ob eine Kollision mit einem Spieler vorhanden ist. Falls ja, wird bei dem Spieler die Methode addMissileAmmo(int amount) aufgerufen und setzt active auf false.

Falls active false ist, wird der Timer reduziert. Falls der Timer 0 wird, wird das Pickup wieder auf active = true gesetzt.

In beiden Fällen wird danach die animation geupdatet.

## Klasse MissileUpgrade

Die Klasse implementiert das Upgrade auf den doppelten Missilewerfer des Schiffs. Nach dem Pickup dieser Klasse verdoppelt sich die Anzahl abgeschossener Raketen von 1 auf 2.

Im Gegensatz zu den Pickups brauchen diese Missiles keine timer oder active Variablen, da Sie zu jederzeit aktiv sind.

**Variablen**

Keine Besonderheiten.

**Konstruktor**

Keine Besonderheiten.

**Methoden**

**public void update()**

Die Methode aktualiseriert die Animation und überprüft die Kollision mit einem Spieler. Falls eine Kollision stattfindet, wird beim Spieler die boolean DoubleMissile auf true gesetzt.

## Klasse PhaserUpgrade

Die Klasse implementiert das Upgrade auf den doppelten Phaser des Schiffs. Nach dem Pickup dieser Klasse verdoppelt sich die Anzahl abgeschossener Phaser von 1 auf 2.

**Variablen**

Keine Besonderheiten.

**Konstruktor**

Keine Besonderheiten.

**Methoden**

**public void update()**

Die Methode aktualiseriert die Animation und überprüft die Kollision mit einem Spieler. Falls eine Kollision stattfindet, wird beim Spieler die boolean DoublePhaser auf true gesetzt.