ASCIIPinball

Prüfungsstudienarbeit zur Vorlesung „Java Programmierung“

Prüfer: Prof. Dr. Andreas Berl

Teilnehmer 1: Muhr Philipp, 00692629, philipp.muhr@stud.th-deg.de, AI
Name, Matrikelnummer, Email, Studiengang

Teilnehmer 2: Mican Michael, 00692390, michael.mican@stud.th-deg.de, AI
Name, Matrikelnummer, Email, Studiengang

Semester: SS 2019
Datum der Abgabe: 23.06.2019

Erreichte Punktezahl:	_____
Note:	_____
Erstkorrektur Unterschrift:	_____
Zweitkorrektur Unterschrift:	_____

Inhalt

Kurzbeschreibung des Spiels

Bei Pinball (auch Flippertisch) versucht man einen Ball möglichst lange auf der Spielfläche zu halten, um so einen hohen Score zu erzielen. Desto länger der Ball sich auf der Spielfläche befindet, desto höher ist die Wahrscheinlichkeit Punkte, durch Kollisionen mit Gegenständen, zu erzielen. Der Ball wird beim Start üblicherweiser aus einem bestimmten Startbereich auf die Spielfläche geschossen. Die sogenannten Flipper Finger dienen dazu den Ball im Spiel zu halten, denn der Ball kann zwischen Ihnen nach unten in ein Loch fallen und dadurch geht dieser verloren. Außerdem kann der Ball durch einen „Auslaufkanal“, welcher sich in der Regel neben den Flipper Fingern befindet, ins Aus gehen. Insgesamt besitzt jeder Spieler jeweils drei Bälle in einem Spiel. Wenn alle Bälle verloren sind, ist das Spiel vorbei. Der Spieler, der die höchste Punktzahl erreicht hat hat gewonnen.

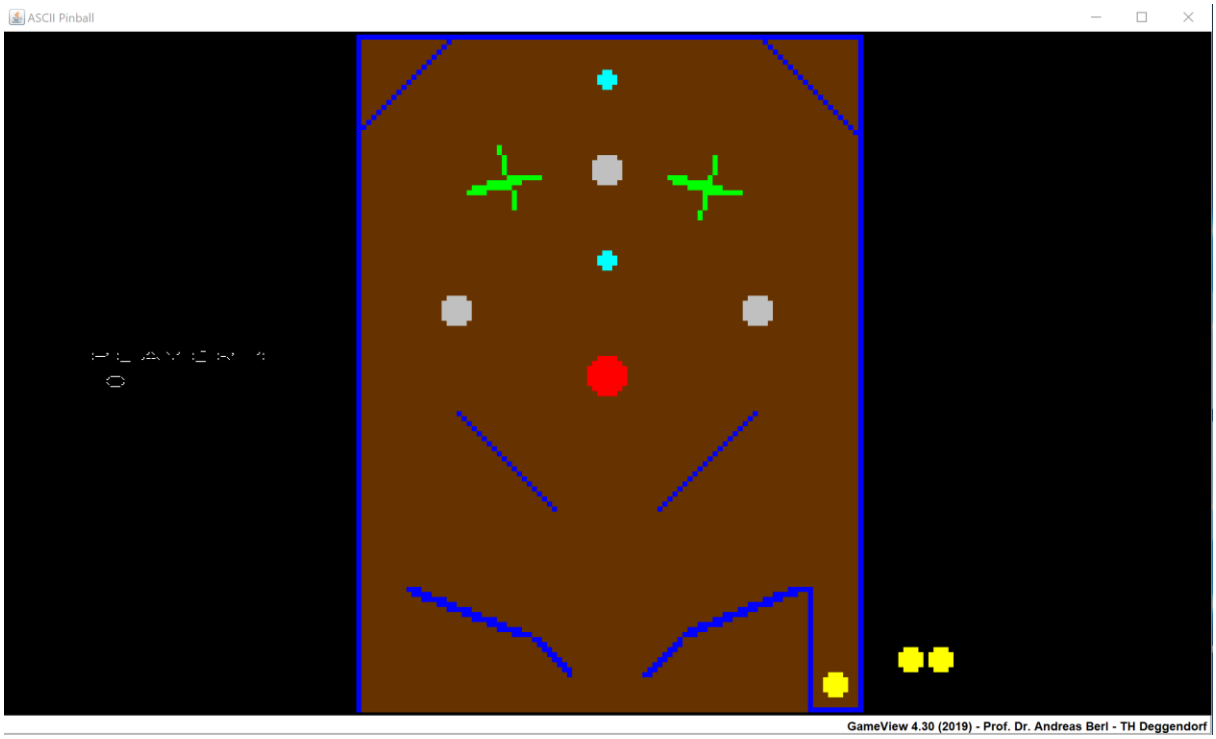

Abb. 1: ASCIIPinball Spielfenster

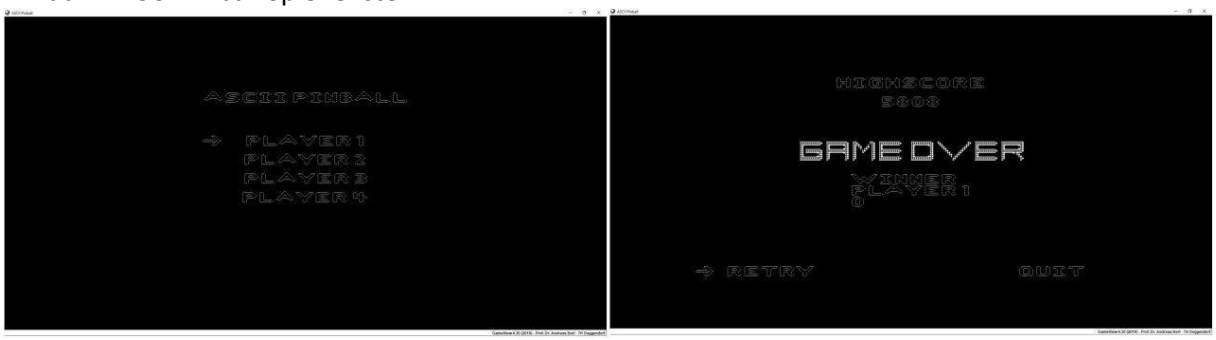

Abb. 2: Start- und Game Over-Screen

Im finalen Spiel (Abb.1) sind verschiedene Objekte zu sehen. Der gelbe Punkt stellt den Ball dar, welcher sich über die Spielfläche bewegt. In der Abbildung ist weiterhin zu sehen, wie der Ball am Anfang einer Runde in der Abschussvorrichtung liegt. Wenn diese Vorrichtung betätigt wird, schießt sie den Ball ins Spiel und versperrt die Abschussrampe, damit der Ball nicht mehr in diese zurückfallen kann. Die drei grauen Punkte werden Point Doors genannt. Sie geben dem Spieler 1000 Punkte, sobald er es schafft alle drei mindesetens einmal mit dem Ball zu berühren und sie so zum Leuchten zu bringen. Die zwei blauen Punkte stellen Teleporter dar. Wenn der Ball einen Teleporter passiert, wird

Abb. 3: Der Flippertisch „White Water“ der Firma Williams

der Ball zum anderen blauen Punkt teleportiert, dabei behält er seine Geschwindigkeit und seine Richtung bei. Der rote Bumper in der Mitte stößt den Ball aktiv ab und der Spieler bekommt bei Kontakt 150 Punkte. Die grünen Kreuze oben rechts und links drehen sich um Ihren Mittelpunkt. Der Ball prallt von diesen zwar ab, schreibt dem Spieler aber keine Punkte gut. Die Lücke links unten am Rand des Flippers stellt einen Auslaufkanal dar. Wenn der Ball dort ins Aus fällt geht er verloren. Links neben dem Flippertisch ist zu sehen, welcher Spieler an der Reihe ist und wie viele Punkte dieser bis jetzt erzielt hat. Auf der unteren rechten Seite sind die übrigen Bälle dargestellt, die der jeweilige Spieler noch hat. Weiterhin hat das Spiel einen Startbildschirm und einen Game Over Bildschirm, in welchen man verschiedene Auswahlmöglichkeiten hat. (Abb. 2)

Ein echter Flippertisch in der realen Welt (Abb. 3) ist mechanisch. Hier sorgen physikalische Abläufe dafür, dass der Ball nach unten fällt, von Hindernissen abprallt und nicht eine unendliche Geschwindigkeit erreicht. Bei unserem virtuellen Tisch wurde eine Physik-Simulation implementiert, welche diese Eigenschaften abbilden soll.

Software-Entwurf

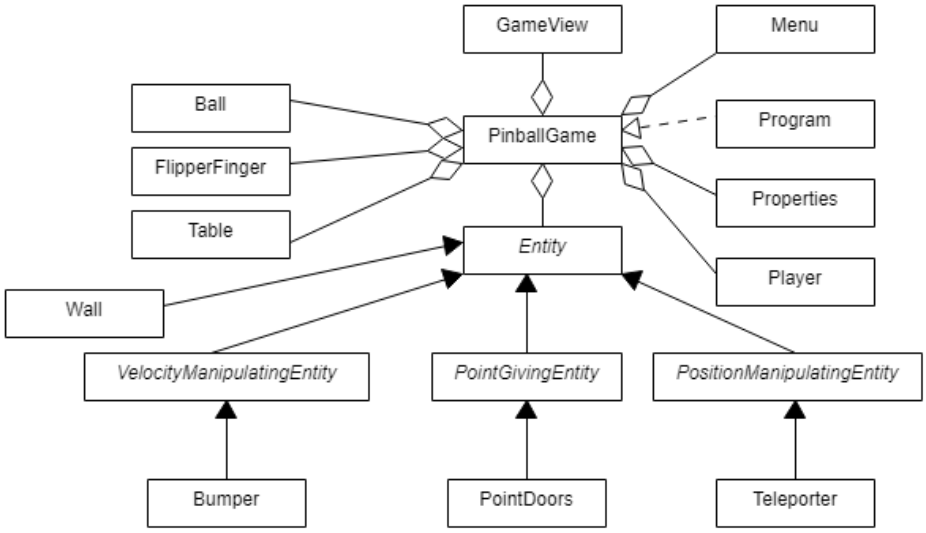

Abb. 4: Ursprünglicher Software-Entwurf

Die Abbildung 4 zeigt den ursprünglichen Software-Entwurf. Dieses Klassendiagram ist noch deutlich gröber als das letztendlich Finale (siehe Abb.6). Das grundsätzliche Modell ist gleichgeblieben, es wurde nur deutlich verfeinert und durch mehrere Klassen ausgebaut. Diese Veränderung erklärt sich vor allem durch die ansonsten extrem langen und unübersichtlichen Klassen und dem Hinzukommen von neuen Funktionalitäten (z.B. Abspielen von Geräuschen).

Das Modell sieht vor die eigentliche Spiellogik in PinballGame zu implementieren, welche dann kontrolliert von der Klasse Programm ausgeführt bzw. gesteuert wird. Die Kollisionsabfrage wird von den Entities übernommen. Hierfür gibt die PinballGame Klasse den aktuellen Ball an die Entities weiter. Diese überprüfen, ob der Ball mit ihnen kollidiert und geben dementsprechend einen „abgeprallten Ball“ zurück und teilen einen eventuellen Punktezuwachs mit. Eine genauere Beschreibung des finalen Software- Entwurfs ist dem Kapitel Implementierung zu entnehmen.

Abb. 5: Erstes, sehr simples Mock Up

Das erste Mock-Up (Abb. 5) ist sehr minimalistisch, und sollte zunächst die Spielfläche darstellen. Zwischen diesem Mock Up und dem finalen Spiel wurde vieles verändert. Unter anderem ist die Spielfläche nun farbig gestaltet und der Ball ist größer als nur ein char Element. Es wurden Ascii-Art-Fonts verwendet, um den Score, und den Text sichtbarer darzustellen. Außerdem gibt es einen Startbildschirm und einen Endbildschirm, um den Spieler nicht direkt in das Spiel zu werfen. Um die leeren Bereiche links und rechts des Spielfelds zu füllen, wird eine ASCII-Art Datei eingelesen und über den gesamten Hintergrund kopiert. Der eigentliche Spieltisch besteht im Endprodukt aus farbigen Pixeln und nicht, wie im Mock Up vorgesehen, aus ASCII-Zeichen.

Implementierung

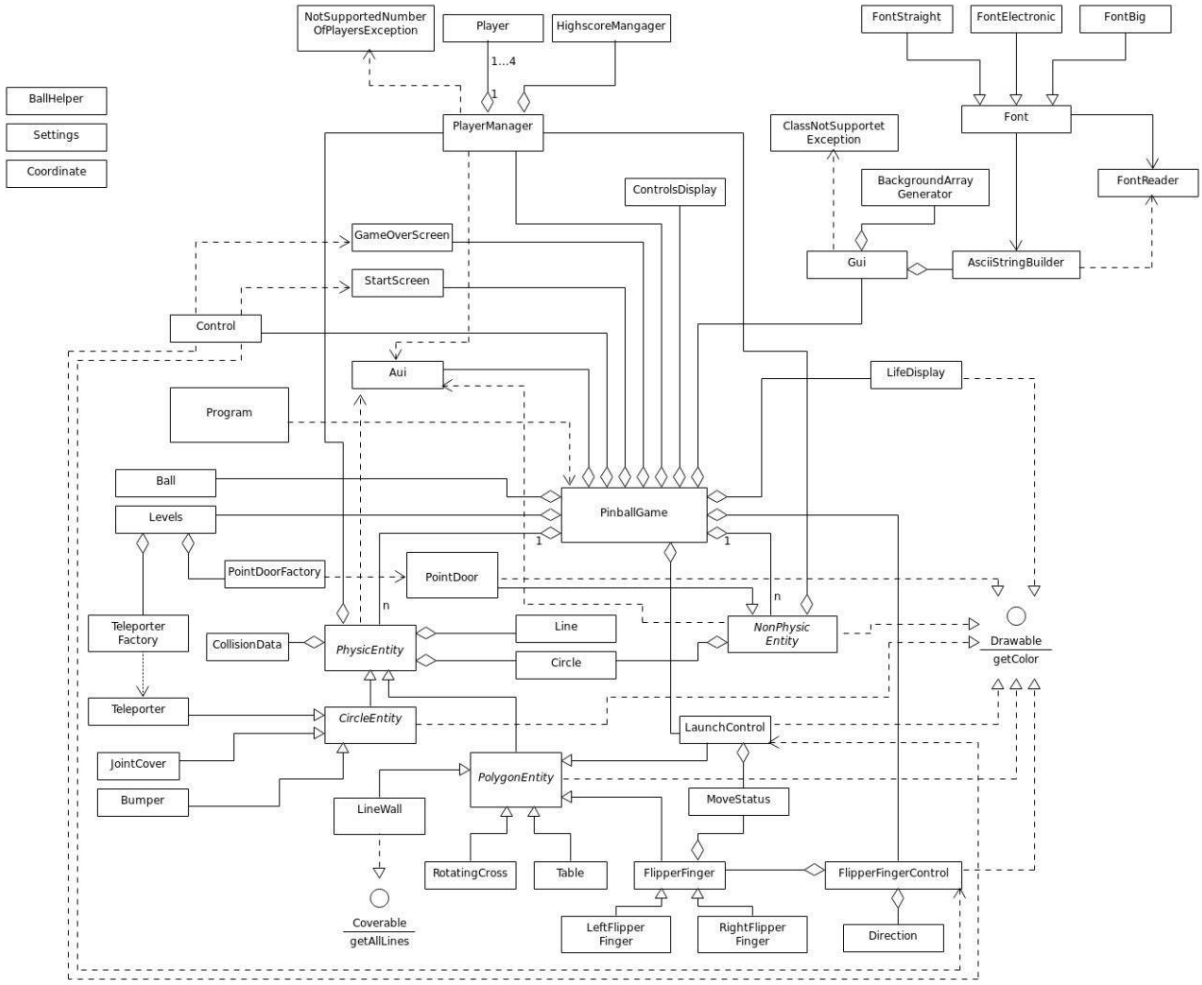

Abb. 6: Das finale Klassendiagramm mit allen Klassen und den wichtigsten Beziehungen

Abbildung 6 zeigt das Klassendiagram reduziert auf die wichtigsten Beziehungen, da das Klassendiagramm ansonsten sehr unleserlich geworden wäre. Die Klasse PinballGame stellt die zentrale Logik des Programms dar. Sie instanziiert alle wichtigen Klassen und macht sie miteinander bekannt. Sie bietet außerdem die drei Steuerungsmethoden „init()“ (Initialisiert/Resetet das Spielfeld), „simulateTick()“ (Bewegt alles um einen Schritt und berechnet eventuelles abprallen) und „printAll()“ (Gibt alle zu zeichnenden Objekte an die Gui Klasse).

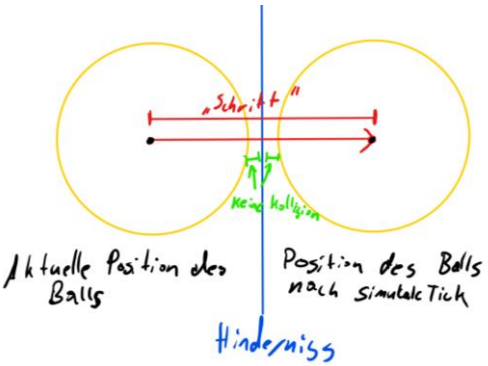

Abb. 7: „Tunneln“ eines Balls durch ein Hindernis bei zu großer Schrittweite

Diese drei Steuerungsmethoden werden von der Klasse „Program“, in der sich auch die main-Methode befindet, aufgerufen. Bei Aufruf des Spiels wird hier eine Instanz der Klasse PinballGame erstellt und dann in einem Loop die Steuerungselemente aufgerufen. Für einen Frame (Ein Aufruf der Methode printAll()) werden 75 Schritte berechnet (75-mal simulateTick() aufgerufen), wofür das Program insgesamt 17ms Zeit hat. Wird es früher fertig, schläft das Programm die restliche Zeit. Es wurde sich für diese nicht frame-synchrone Simulation entschieden, um die einzelnen Schritte so klein zu machen,

damit ein „Überspringen“ bzw. „Durchtunneln“ (siehe Abb. 7) des Balles durch Wände, vorallem bei hohen Geschwindigkeiten, vermieden wird.
Die Ball Klasse gehört zusammen mit allen Physic- und NonPhysicEntities zu der Gruppe der Objekte. Alle Objekte benutzen ein eigenes feines Koordinatensystem anstatt des Gittersystems der GameView. Jedes Objekt wird dann beim Zeichnen auf dieses Gittersystem gemapped/umgerechnet. Dies wurde vorallem aus ästhetischen Gründen gemacht. So sind nämlich viel flüssigerere und vorallem dynamischere „Animationen“ möglich, da ein hochaufgelöster, flüssiger Bewegungsablauf auf eine niedriger aufgelöste Fläche projeziert wird. Außerdem ist es so möglich einzelne Schritte pro Tick unter eine Pixelgröße zu reduzieren, um so ein Durchtunneln, wie in Abbildung 7 gezeigt, zu vermeiden.
Der Ball selbst berechnet unter Berücksichtigung von Gravitation und Reibungskraft seine zukünftige Position, mit der dann die Physic- und NonPhysicEntities rechnen können. Nach der Kollisionsprüfung adaptiert der Ball die Eigenschaften des abgeprallten Balles (bei aufgetretener Kollision) bzw. übernimmt seine zuvor berechneten Parameter (bei keiner Kollision).
Für alle Berechnungen, allen voran den in den PhysikEntities, stellt die Klasse BallHelper zwei große Hilfs-Methoden zur Verfügung (joinBalls() und convertDirection()). Diese Methoden sind erforderlich, um Ergebnisse zu vereinheitlichen und zu vereinfachen.
PhysikEntities ist die Elternklasse aller Objekte, von welchen der Ball abprallen kann. Sie stellt die grundsätzliche Schnittstelle (updateBall()) zur PinballGame Klasse zur Verfügung. In dieser wird nach Kollisionen gesucht und bei Eintritt einer solchen die formspezifische interactWithBall() Methode aufgerufen, um so die neue Ballrichtung zurückzugeben. Außerdem wird bei Kollision mithilfe des PlayerManagers dem aktuellen Spieler ein, vom Objekt abhängiger, Score gutgeschrieben. Zusätzlich enthält die Klasse Kollisionsdaten (Objekte der Klasse collisionData) in der der Kollisionspunkt, sowie das Kollisionsobjekt gespeichert wird. Von PhysicEntity erben Polygonal- und CircleEntity. Diese implementieren die Kollisionsprüfung und den „Abprallalgorithmus“. Polygonial Entities bestehen hierbei aus Linien (Line Klasse) und CircleEntities aus Kreisen (Circle Klasse). Line & Circle definieren ihre Form und bieten getter Methoden an, um diese abzurufen. An unterster Stelle der Physik-Vererbungs-Hierarchie stehen nun die letztendlichen Objekte. Aktuell implementiert sind LineWall, Bumper, RotatingCross, Teleporter und JointCover. Diese definieren das Aussehen, die Position und die Eigenschaften. Sie implementieren auch das vererbte Interface „Drawable“ und gibt so die Farbe in der das Objekt dargestellt werden soll zurück. Es gibt auch einige besondere Klassen, die von PolygonEntity erben. Namentlich sind das FlipperFinger und LaunchControl. Diese beinhalten zusätzlich zur Definition Ihrer Position und Aussehens auch noch Funktionen, die Ihr Auftreten verändern. Diese Veränderungen werden von der Klasse Control gesteuert. Diese Klasse analysiert alle Tasteneingaben des Spielers und ruft jeweils die Methoden der zu steuernden Klassen auf.
NonPhysicEntity verhält sich ähnlich wie PhysicEntity, jedoch wird hier keine Kollision festgestellt und es kommt auch nie zu einem Abprall. Davon erbt die Klasse PointDoor. PointDoor selbst stellt aufgrund des Abstandes von sich zum Ball fest ob der Ball über die Fläche gerollt ist und schaltet seinen Status dabei auf an. PointDoors sind untereinander verlinkt; wenn alle verlinkten PointDoors den Status an haben werden sie ausgeschaltet und der Spieler bekommt 1000 Punkte gutgeschrieben. Dies wird über den PlayerManager realisiert.
Der PlayerManager verwaltet die Spieler. Sie bildet die Ansprechstation für alle Statusanfragen/Status änderungen für Spieler. Sie erstellt bei Instanziierung der Spieleranzahl eine bestimmte Anzahl an Playerinstanzen und wirft eine unchecked UnsupportedNumberOfPlayersException, wenn der Programmierer es ermöglicht mehr als 4 Spieler anzulegen. Außerdem verwendet die Klasse PlayerManager den HighscoreManager, um zu überprüfen ob ein neuer Highscore erzielt wurde und gibt diesen auf dem Game Over Screen aus. Die Klasse Player speichert spieler-spezifische Variabeln, wie die verbleibenden Bälle/Leben oder den Score des Spielers.

Die Klasse Gui verwaltet alle Ausgaben des Programms. Sie bekommt bei Instanziierung eine Referenz auf die GameView. Sie enthält alle Algorithmen, um aus den Informationen aus den Klassen das jeweilige Objekt zu zeichen. Dafür muss von dem Physikkoordinatensystem in das GameView Gitter umgerechnet werden. Außerdem kann die Gui mithilfe des AsciiStringBuilders String in AsciiString konvertieren. Dafür benötigt der AsciiStringBuilder eine Referenz auf ein Font Objekt, welches die Schriftart des AsciiStrings definiert. Damit der Code nicht zu viele Zeilen beinhalten würde, wurde das Einlesen dieser Fonts in den FontReader ausgelegt. Darüber hinaus kann die Klasse Gui mithilfe des BackroundArrayGenerator ein AsciiArt aus dem resource Ordner einlesen und über den Hintergrund kopieren, um so die schwarzen Ränder etwas zu beleben.
Aui verwaltet alle Audio Ausgaben. Sie wird von anderen Klassen aufgerufen, um einen Sound abzuspielen.
GameOverScreen und StartScreen verwalten den Anfangs- und Endbildschirm. Der StartScreen definiert die Anzahl der Spieler, der GameOverScreen ermöglicht den Neustart und das Beenden des Spiels.
In der Klasse Levels ist der Level definiert. Außerdem werden dort JointCover an die Enden jeder Coverable Entity generiert, um ein „Aufspießen“ zu verhindern. Dies ist erforderlich aufgrund eines bis zum finalen Release nicht gefundenen Fehlers in der Physik, welcher dazu führt, dass der Ball in einer Wand stecken bleibt.

In diesem Projekt wurden verschiedene Design Pattern verwendet. So ist die Klasse Settings eine statische Klasse und nach dem Immutable Design Pattern aufgebaut, welche es verhindert zur Laufzeit Veränderungen an dieser vorzunehmen. Außerdem wurde das Factory Design Pattern für die Erstellung von Teleportern und PointDoors verwendet, da diese Referenzen aufeinander aufbauen. Die Factory erstellt Instanzen, verlinkt diese und gibt dann alle Instanzen in einem Array zurück. Mithilfe des package-private Zugriffmodifizierers wird der Konstruktor der eigentlichen PointDoor und Teleport Klasse gekapselt, sodass nur die Factory Instanzen erstellen kann. Es wurde in Erwägung gezogen, die Klasse Gui, Aui, Control und PinballGame mit einem Singelton Designpattern zu gestalten. Aufgrund von mangelnder Zeit wurde diese Idee jedoch verworfen.
Der gesamte Workflow zwischen den zwei Programmierern wurde durch ein Git Repository sichergestellt. Vor dem Hinzufügen einer neuen Funktionalität wurden Unittests durchgeführt. Auch das debugging Feature von IntelliJ wurde vorallem bei der Implementierung der Physik verwendet, um Logikfehler zu identifizieren und anschließend auszubessern.

Beschreibung einer besonderen Herausforderung bei der Umsetzung des Spiels

Eine besondere Herausforderung bei der Umsetzung des Spiels lag unter anderem in der Physik. Schon im Software Entwurf wurde ersichtlich, dass die Physik einen nicht zu vernachlässigenden Teil der Software ausmachen wird. Grundsätzlich fußt die Physiklogik auf zwei Klassen. Gravitations und Reibungskraft werden in der Klasse Ball verwaltet, wohingegen die Kollisions- und Abprall-Logik in der PhysicEntity Klasse bearbeitet wird. Hierbei ist zwischen Circular- und Polygonial-Entity zu unterscheiden, da abhängig davon, ob die Abpralloberfläche eine Linie oder ein Kreis ist, die Algorithmik sich unterscheidet.

Um die Gravitation zu simulieren, wird die Geschwindigkeit des Balls, die in eine bestimmte Richtung wirkt, mithilfe von trigonometrischen Funktionen in eine in x-Richtung („nach Rechts“) und eine in y-Richtung („nach Oben“) wirkende Geschwindigkeit aufgeteilt. Nur von der y-Geschwindigkeit wird nun ein fixer Gravitationswert abgezogen. Somit beschleunigt der Ball jeden Tick nur nach unten. Nach dieser Gravitationsanpassung wird die Flugrichtung neu berechnet und die zwei Geschwindigkeiten werden wieder zu einer Geschwindigkeit („Velocity“) zusammengeführt. Von dieser wird nun noch abschließend eine Reibungskraft abgezogen, um „Energie“ bzw. „Tempo“ aus dem System zu nehmen.

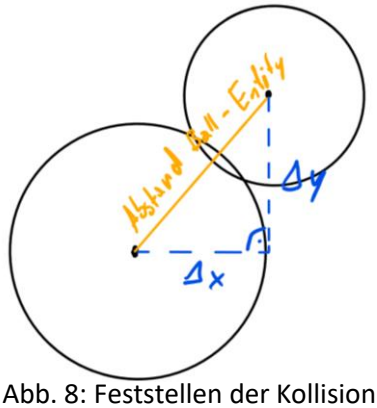

Abb. 8: Feststellen der Kollision bei Kreis – Ball Kollision mit dem Satz des Pythagoras

Damit der Ball abprallen kann, muss zunächst festgestellt werden ob zwischen der aktuellen Position des Balls und der zukünftigen Position des Balls es zu einer Kollision gekommen ist. Hierfür muss man zwischen PolygonEntities und CircleEntities unterscheiden.
Bei der Kollision mit einem Kreis wird mithilfe des „Satzes von Pythagoras“ der Abstand zwischen der Position des Balles im nächsten Tick und der zu untersuchenden kreisförmigen Entity berechnet (siehe Abb. 8). Wenn dieser Abstand geringer oder gleich der Summe aus Radius des Balles und Radius der Entity ist, ist es zu einer Kollision gekommen.
Bei Kollisionen mit PolygonEntities (Entities die aus Linien bestehen) ist die Beantwortung der Kollisionsfrage etwas komplexer. Zunächst werden alle Linien (Strecken) zu Geraden umgerechnet. Anschließend wird ein Lot durch die Zukunftsposition des Balles gezogen. Danach wird der Schnittpunkt des Lots mit der Gerade berechnet (Lotfußpunkt) (siehe Abb. 9). Dieser Schnittpunkt wird anschließend evaluiert ob er auf dem Geraden Element bzw. der (begrenzten) Strecke liegt, also tatsächlich die Linie (Strecke) schneidet. Hierfür wird der Abstand zu den Endpunkten der Linie berechnet (Satz des Pythagoras), wenn der Abstand von einer der beiden Punkte länger ist als die Länge der Strecke, ist der Schnittpunkt auf der Gerade. Ist der Schnittpunkt maximal eine Radiuslänge des Balls entfernt, streift der Ball die Gerade und es ist ebenfalls eine Kollision aufgetreten. Nach der Evaluierung wird mit Hilfe des „Satzes von Pythagoras“ der Abstand zwischen dem nächsten Schnittpunkt/Kollisionspunkt und der zukünftigen Position des Balls errechnet. Ist dieser Abstand geringer oder gleich dem Radius des Balls, so ist eine Kollision aufgetreten (siehe Abb. 9).

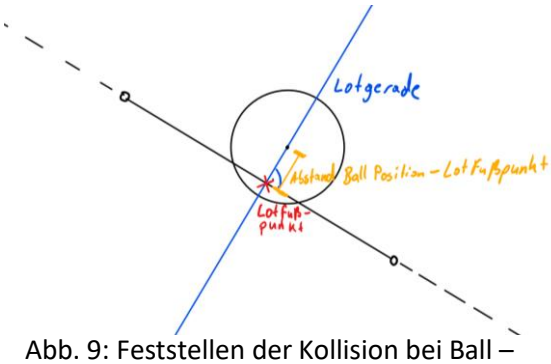

Abb. 9: Feststellen der Kollision bei Ball – Linie Kollision

Sobald eine Kollision festgestellt wurde, muss eine neue Richtung des Balls und möglicherweise auch eine neue Geschwindigkeit (z.B. bei Kollision mit einem Bumper) errechnet werden.

Bei einer Kollision mit einer Linie wird nun zunächst die Steigung der kollidierenden Linie errechnet. Bei einer vollwertigen Kollision (kein Streifen der Linie) ist dies dieselbe Steigung wie die Linie hat. Streift der Ball nur die Linie ist die Abprallsteigung die Senkrechte auf die Steigung der Geraden, die den Kollisionspunkt (Eckpunkt der Linie) und den Mittelpunkt des Balls schneidet (siehe Abb. 10).

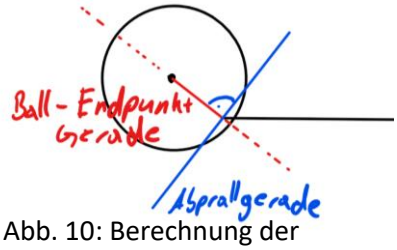

Abb. 10: Berechnung der Abprallgeraden in Streiffällen

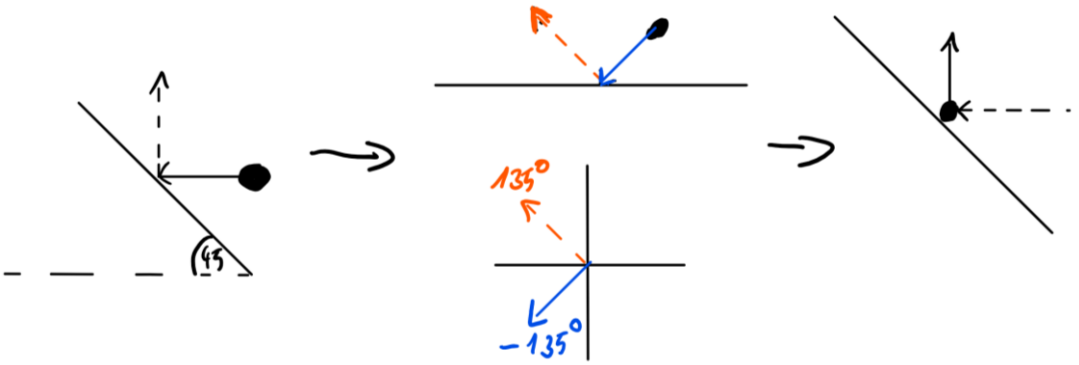

Abb. 11: Berechnung der neuen Richtung des Balles nach Abprall

Diese Steigung wird nun verwendet, um den Winkel der Linie zur x-Achse zu berechnen. Dieser Winkel wird von der Flugrichtung des Balls abgezogen (um einen Abprall mit einer waagerechten Gerade zu simulieren) und anschließend nach dem Motto Einfallswinkel gleich Ausfallswinkel umgerechnet. Hierfür wird das Vorzeichen der (auf -180° – 180° konvertierten) Flugrichtung umgedreht. Abschließend wird der zuvor abgezogenen Winkel wieder aufaddiert (siehe Abb. 11).

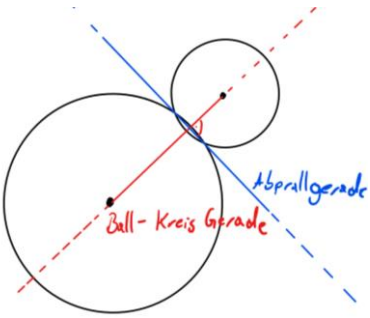

Abb. 12: Berechnung der Abprallgeraden bei Ball – Kreis

Bei einer Kollision mit einem Kreis wird eine Senkrechte auf die Gerade, die die Mittelpunkte des Balls und des kollidierenden Kreises schneidet, konstruiert. Diese Senkrechte ist die Abprallsteigung (siehe Abb. 12). Mithilfe dieser Steigung wird analog zur Rechnung mit Linien der Abprall des Balls berechnet.

Arbeitsteilung

Die Arbeitsteilung ist der Tabelle (Abb. 13) zu entnehmen. Das Programmierteam hat anhand dieser dargestellt, wer welche Klassen bzw. Aufgabenteile, wie Tonaufnahmen oder das Mock Up übernommen hat. Manche Bereiche wurden gemeinsam bearbeitet.

Die Klasse Control ist durch Pair-Programming entstanden, da sich die Entwickler hier einig sein mussten, wie die Tasten definiert werden sollen.

Außerdem wurde dieses Verfahren in verschiedensten Vorlesungen bereits empfohlen. Weiterhin wurde die Klasse vom jeweiligen Programmierer nach gewünschter Aktion erweitert.

Die statische Klasse Settings ist deshalb als gemeinsam definiert, da in dieser Klasse hauptsächlich Einstellungen zum Spielverhalten bestimmt wurden. Dabei sind unterschiedliche Variablen entstanden, die für einige Klassen notwendig sind. Michael Mican

und Philipp Muhr haben sich somit je nach Bedarf die richtigen Variablen angelegt, wenn diese als notwendig empfunden wurden.
BallHelper wurde generiert, um die Klasse Ball aufzuteilen, da diese sich bereits über zu viele Zeilen an Code erstreckte. In BallHelper wurde die Methode joinBalls() von Herrn Muhr und convertDirection() von Herrn Mican bearbeitet.
Tonaufnahmen wurden sowohl von Herrn Muhr, als auch von Herrn Mican aufgenommen, wobei die Bearbeitung der Töne Herr Mican übernahm. Dieser recherchierte außerdem weitere Töne für die Implementierung im Internet.
Mit Ausarbeitung ist die finale Abgabe gemeint. Die Ausarbeitung wurde von beiden Team-Mitgliedern zusammen ausgearbeitet.
Die jeweiligen Ascii-Arts, die in dem Spiel verwendet wurden, wurden separat von beiden Projekterstellern im Internet ermittelt und bearbeitet. Dabei wurden allerdings nicht alle Resultate benutzt, da man eine gewisse Auswahl generieren wollte, aus welcher man sich für bestimmte Symbole zusammen entschied.

Klasse	Autor
AsciiStringBuilder	M
FontReader	M
Control	M & P
PinballGame	M
Programm	M
ClassNotSupportedException	M
NotSupportedNumberOfPlayersException	M
Font	P
FontBig	P
FontElectronic	P
FontStraight	P
BackgroundArrayGenerator	M
ControlsDisplay	M
GameOverScreen	P
Gui	M
LifeDisplay	M
StartScreen	P
Coverable	M
Drawable	M
Levels	P
Directions	P
FlipperFinger	P
FlipperFingerControl	P
LeftFlipperFinger	P
RightFlipperFinger	P
MoveStatus	P
LaunchControl	P
PointDoor	M
PointDoorFactory	M
NonPhysicEntity	M

Teleporter	M
TeleporterFactory	M
Bumper	M
CircleEntity	M
JointCover	M
LineWall	M
PolygonEntity	M
RotatingCross	M
Table	M
PhysicEntity	M
Ball	M
HighScoreManager	M
Player	M
PlayerManager	M
Circle	M
Line	M
Aui	M
CollisionData	M
Coordinate	M
Settings	M & P
BallHelper	M & P
MockUp	P
Klassen/Software Entwurf	M
Graphische Darstellung des Klassendiagrams	P
Tonaufnahmen	M
Ausarbeitung	M & P
ASCII Art	M & P

Abb. 13: Arbeitsteilungstabelle
M = Michael Mican, P= Philipp Muhr