**Dokumentation**

Jan-Philipp Roslan

Jahrgangsstufe 13 (G9)

Lippe-Berufskolleg

**Inhaltsverzeichnis**

**Vorwort2**

**Motivation3**

**Ideenfindung4**

**Umsetzung5**

**Spielanleitung10**

**Vorwort**

Hier im Vorwort sollen einige allgemeine Informationen erklärt werden und meine Einsendung als solches beschrieben werden.

Mein Projekt nennt sich „Verse“ und ist ein 2D-Puzzle-Jump’n Run. Angelehnt an klassische Genre-Vertreter wie Super Mario, „springt“ man auch in Verse durch die Level löst anhand kleiner Rätsel das Level. Jedoch besitzt Verse eine Funktion, durch die sich das Spiel von den anderen unterscheidet:

In Verse bewegt sich die Zeit nicht unbedingt nach vorn. Im Spiel hat man die Möglichkeit die Zeit, für einen bestimmten Zeitraum zurückzuspulen. Dabei bleibt die Umgebung unangetastet, jedoch wird der Spieler zurückgesetzt. Durch diese Funktion ergeben sich einige interessante Rätselmöglichkeiten.

Diese Demo enthält lediglich ein Level welches repräsentativ für die Spielidee besteht. Alle Inhalte, wie Texturen, Musik und Sounds sind für private Zwecke frei nutzbar im Sinne von Lizenzen wie ‚Creative-Commons‘. Die Verweise zu den Quellen dieser Daten befinden sich in der Textdatei „Licenses,txt“.

Anfangs waren eigentlich mehr Level und Items, sowie verschiedene Gegnertypen geplant, jedoch konnte ich diese Funktionen nicht implementieren, da die Zeit bis zur Abgabe nicht mehr für diese reichte. Unter anderem ist dies dadurch geschuldet, dass ich das Projekt alleine anging und ich nun kurz vor meinem Abitur stehe und damit die Vorabiklausuren einhergehen.

**Motivation**

Von dem Wettbewerb erfahren habe ich durch meinen Klassenlehrer. Dieser informiert uns innerhalb unserer Klasse darüber. Zuerst wollten einige Klassenkameraden und ich an dem Wettbewerb als Gruppe teilnehmen, jedoch stiegen diese relativ schnell aus Mangel an Interesse aus.

Meine Motivation an dem Wettbewerb trotzdem weiterhin alleine teilzunehmen sind die Erfahrungen, die man durch solch ein Ereignis erhalten kann und vor allem die Thematik der Spiele. Videospiele habe ich selber bisher nur in sehr kleinem Maßstab programmiert, da ich mich eher auf Desktop- und Webanwendungen fokussiert habe und den Wettbewerb daher als Lernerfahrung und als Möglichkeit sehe, für mich komplett neue Thematiken, die vor allem in der Spieleentwicklung wichtig sind (Vektorraum, Kollisionssysteme, etc..) kennenzulernen und in der Programmierung einsetzen zu können.

**Ideenfindung**

Die Idee ein eigenes Spiel für den Wettbewerb zu entwickeln habe ich sehr schnell gefasst, da mir wichtig war möglichst viel selber zu machen, ohne vorher viele Dinge und Ideen als Referenz für mein Projekt zu nutzen. Des Weiteren ist es sehr interessant ein komplettes Spiel selbst zu entwickeln und somit einmal den kompletten Entwicklungsablauf durchzuführen. Dazu gehören vor allem das anfängliche Designen und Festlegung des Aufbaus, die Programmierung der „Spiel-Engine“ und das konkrete Erstellen des Spieles mit Medien wie Musik und Texturen.

Um das Projekt innerhalb der gegebenen Zeit fertig zu stellen zu können stand zudem schnell die Entscheidung für ein 2D-Plattformer fest, weil diese vom Aufbau sehr simpel gestaltet sind, jedoch noch genug Möglichkeiten für eigene Ideen bieten.

Um ein Spiel zu entwickeln, welches sich von anderen Genre-Vertretern abhebt, sollte das Projekt eine Funktion besitzen, die es von diesen unterscheidet. Hierbei kam mir die Idee der Zeitmanipulation. Zeit ist ein Element von Videospielen welches kaum berührt und somit noch seltener als ein tragendes Spielelement genutzt wird. Dabei bietet es sehr viele Möglichkeiten den Spielfluss zu verändern und ist eine perfekte Grundlage für neue und kreative Spielideen.

Interessant ist zudem, dass viele Spieler gar nicht gewohnt sind den zeitlichen Ablauf eines Spiels zu ändern und somit kaum Erfahrung haben, wie solche Rätsel zu lösen sind. In diesem Aspekt ist also noch viel Potenzial vorhanden, besondere Spielideen mit der Funktion zu erstellen.

**Umsetzung**

Das Spiel ist in Java implementiert, da Java meine primäre Programmiersprache ist und ich am meisten Erfahrung mit ihr habe.

Des Weiteren bietet Java mit dem aktuellsten GUI-Framework ,JavaFX' eine ausgezeichnete Grundlage für die Programmierung eines solchen Projekts. JavaFX ist ähnlich wie Swing (der Vorgänger von JavaFX und langjähriges Standard-GUI-Framework) ein, in die Sprache integriertes, Framework um grafische Nutzeroberflächen zu erstellen. Im Gegensatz zu Swing jedoch, baut JavaFX auf einen dreidimensionalen Hardware-Renderer auf. Auf Windows wird dabei i.d.R. DirectX verwendet und auf anderen Systemen entsprechend OpenGL. Dadurch ist jedes JavaFX-GUI eine vollständige, hardware-beschleunigte, 3D-Umgebung. Zusammen mit den standardmäßigen und vordefinierten Controls, wie Knöpfen, Bildcontainern und Textfeldern bietet es damit eine geeignete Basis für ein 2D-Videospiel.

Da eine JavaFX eine tatsächliche 3D-Umgebung darstellt wird eine virtulle Kamera verwendet um die Inhalte im Raum passend darzustellen. In normalen Nutzeroberflächen wo kein dreidimensionaler Raum benötigt wird ist dies eine sogenannte Parallelkamera, mit der alle Objekte parallel zu den Koordinantenachsen gezeichnet sind und Aspekte wie Sichtfeldgröße und Verzerrung außer Acht gelassen werden. Dies ist zudem auch automatisch bereits die richtige Kameraeinstellung für das Spiel, da genau die Eigenschaft der Parallelität benötigt wird.

Anders als andere 3D-Umgebungen benutzt JavaFX jedoch einen Szenengraph um die Objekte hierarchisch anzuordnen. Mit diesem können Objekte zueinander untergeordnet werden. Dadurch lassen sich Objekte in 'Parent' und 'Child'-Objekte unterteilen. Wenn also Objekte einem 'Parent-Objekt' untergeordnet werden, modifizieren sich ihre Werte bereits, wenn man die Eigenschaften des Vaterobjektes verändert. (Beispiel: Koordinatenposition)

**Spiel-Engine**

Während JavaFX sich um die Ausgabe der Objekte auf den Bildschirm kümmert und auch als Grundlegende Basis für Medien, wie Musik und Sounds dient, benötigt das Spiel auch eine Ablauflogik. Zu dieser gehören Dinge, wie Kollisionsabfragen, Maus- und Tastatureingaben und Update-Intervalle. Diese Dinge und noch mehr werden oft in einer sog. Spiel-Engine zusammengefasst. Sie bildet das Grundgerüst des Spiels. Mit ihr wird auch eine Modularität ermöglicht, die später die Spiellogik von abstrakteren Dingen wie z.B. die Darstellung der Objekte auf dem Bildschirm trennt.

Die Programmierung der Spiel-Engine war der komplexeste Teil bei dem Projekt, da mein Ziel war, sie möglichst abstrakt und modular zu halten um sie auch in späteren Projekte nutzen zu können. Des Weiteren mussten in ihr sehr viele Grundlagenprozesse implementiert werden, die umfangreicher als die spätere Spiellogik sind, da ich keine weiteren Bibliotheken genutzt habe, außer eine kleine Werkzeugbibliothek, die ich selber geschrieben habe. Dem Projekt entsprechend, habe ich meine Engine 'VerseFX' genannt.

VerseFX basiert auf einem Level-orientierten Prinzip. Ein Level besteht aus Objekten die während der Laufzeit dargestellt werden sollen und miteinander interagieren können. So sind also das Hauptmenü und die einzelnen Spiellevel jeweils ein Level. Wird ein anderes Level geladen (zum Beispiel beim Wechsel vom Hauptmenü und das erste Spiellevel), werden alle Objekte aus dem alten Level gelöscht und die neuen Objekte werden in die Szenerie geladen. Dadurch erfolgt eine saubere Trennung zwischen einzelnen Szenen, oft auch 'Views' genannt. Damit erfolgt eine Durchsetzung des MVC-Prinzip wie es heute in der GUI-Entwicklung üblich ist. Unterstützt wird dieses Prinzip mit der Möglichkeit in JavaFX die GUIs als XML-Dateien zu entwickeln und somit die Darstellung einer Szene (View) strikt von dem dahinter liegenden Code (Controller) zu trennen.

Dies war auch meine erste Absicht in meinem Projekt. Die Level sollten in XML beschrieben werden und dahinter liegende Controller sollten in Java-Code mit dem restlichen Ablauf kommunizieren. Dafür bietet JavaFX auch bereits alle nötigen Mittel. Jedoch entschied ich mich letztendlich dagegen, da eine konsequente Durchsetzung einer XML-Beschreibung der Level die Erweiterung des XML-Parsers nötig gemacht und somit viel Zeit in Anspruch genommen hätte. So sind nun die Level in Java-Code beschrieben, jedoch ist es jederzeit prinzipiell möglich XML einzusetzen, was auch generell zu empfehlen ist.

Die Engine basiert, wie alle gängigen Spiel-Engines auf ein Frame-basiertes Aktualisierungskonzept. Dies bedeutet dass das Bild mehrmals pro Sekunde neu gezeichnet wird und die Objekte auch entsprechend ihren Status kontrollieren und ihre eigene Logik (Bewegung, Kollisionsabfragen) durchführen. VerseFX benutzt dynamische Frame-Zeiten, somit wird nach Berechnung eines Frames, direkt der nächste berechnet. In besonders rechenintensiven Szenen bedeutet dies ein Einbrechen der Frame-Rate. Um dies entgegenzuwirken und eine Verlangsamung des Spiels zu verhindert, wird das Konzept der Delta-Zeit genutzt. Dabei wird die Zeit seit dem letzten Frame dazu genutzt auf die Geschwindigkeit des Spiels zu schließen. Multipliziert man diese Zeit nun mit den Bewegungen von Objekten erhält man zeitunabhängige Bewegungen, die trotz schwankender Frame-Raten gleich bleiben.

Den Einstiegspunkt für die JVM bildet die Klasse Main.java. Sie initialisiert ein Objekt vom Typ FXGame und fängt zudem Eingaben von Maus und Tastatur ab, welches sie der Singleton-Instanz von InputManager übergibt. Des Weiteren definiert sie die Aktualisierunslogik für das für das Spiel. Dafür bietet JavaFX bereits die Klasse AnimationTimer, die für frame-basierte Animationen und Logik konzipiert ist und Code solange in einer Schleife ausführt, bis der Timer gestoppt wird.

FXGame.java ist das Interface, welches eine Klasse implementieren muss, damit sie als Oberklasse für das gesamte Spiel dienen kann. Dementsprechend implementiert die Klasse Game.java dies und stellt somit die Einstiegsklasse für das eigentliche Spiel dar.

Um den Prozess des Laden eines Levels zu vereinfachen und automatisieren existiert die Singleton-Klasse LevelLoader.java. Sie ist in der Lage das alte Level zu entladen und das neue zu laden. Dabei sammelt es die Objekte eines Levels und fügt sie dem JavaFX-Szenengraph hinzu und aktualisiert ihren Zustand pro Frame. Außerdem ist sie in der Lage Objekte für die Kollisionsüberprüfung zu registrieren.

Level-Implementationen erben von der abstrakten Klasse Level.java, welches ein gemeinsames Interface für Level zur Verfügung stellt und Grundfunktionalitäten wie das Löschen von Objekten bereitstellt. Objekte innerhalb eines Levels speichern das Level, in welchem sie sich gerade befinden. Dafür dient das Interface ObjectManager.java, welches den Zugriff auf Level.java abstrahiert und den Objekten nur Zugriff auf bestimmte Methoden gibt.

Der pro-Frame-Aktualisierungszyklus des folgendermaßen aus:

1. Main.java definiert AnimationTimer und ruft zyklisch die update-Methode von Game.java auf

2. Game.java ruft die tick-Methode von LevelLoader.java auf.

3. LevelLoader.java ruft die update-Methode des derzeit geladenen Levels auf.

4. Das passende Level ruft die update-Methoden seiner Objekte auf.

Der Objektsammler ist in der Klasse ObjectBatch.java implementiert. Der LevelLoader initialisiert jeweils einen für das Level, welches geladen werden soll und übergibt den Sammler dem Level während der Level init-Methode. So kann das Level dem Sammler Objekte übergeben, die später dem LevelLoader zum Zeichnen übergeben werden. Zudem nimmt der Sammler registrieren von Objekten für die Kollisionsabfrage entgegen.

Objekte, die in VerseFX auf dem Bildschirm gezeichnet werden, müssen von der abstrakten Klasse BasicNode.java erben. Sie definiert vor allem Methoden zum Bewegen und Skalieren und implementiert die Interfaces Timeable.java (Objekt zeitlich rückwärts bewegen), Collidable.java (Implementiert die Kollidierbarkeit mit anderen Objekten) und Updateable.java (markiert das Objekte für frame-gesteuerte Aktualisierung).

Die Klasse DrawImage.java erbt von BasicNode. Sie stellt eine nicht animierte 2D-Textur da. Die Klasse AnimImage.java erbt wiederum von DrawImage und erweitert sie um Animationen. Die Einzelbilder der Animation sind dabei mit in der Textur als Einzelbilder gespeichert. Auf dem Bildschirm wird also immer nur ein Ausschnitt der ganzen Texture gezeigt, die das jeweilige Einzelbild zeigen.

Die einzelnen konkreten Spielobjekte (Spielfigur, Wände, Knöpfe) erben von diesen beiden Klassen. Sie müssen zudem einen einzigartigen Namen besitzen, damit sie im Kollisionssystem unterschieden werden können.

Ausserdem erbt die Klasse DrawText.java von BasicNode. Sie stellt Text auf dem Bildschirm dar. Genutzt wird sie im Hauptmenü für die Texte und Menüknöpfe.

VerseFX beinhaltet auch noch weitere Klassen, die von BasicNode erben, sie werden jedoch in dem Spiel nicht genutzt und sollen somit hier nicht weiter von Relevanz sein.

**Kolllisionssystem**

Die Klasse BasicNode besitzt ein Objekt vom Typ BoundingBox. Eine BoundingBox stellt eine Box um ein Objekt herum dar. Durch diese Speicherung der Ausdehnungen und Position eines Objekts lassen sich simple Kollisionsabfragen durchführen, indem man prüft ob sich zwei Boxen schneiden. Beim Objektsammler werden die Objekte, mit der Angabe der Namen ihrer Kollisionsziele, für das Kollisionssystem im LevelLoader registriert. Dieser prüft pro Frame, ob ein Objekt mit einem seiner Kollisionsziele kollidiert. Ist dies der Fall können innerhalb eines Objekte mittels getCollisions() diese Objekte zurückgegeben werden. Zudem wird bei einer Kollision die Methode intersects() innerhalb eines Objektes aufgerufen. Diese ist abstrakt in BasicNode definiert und kann mit einer Ereignislogik versehen werden, wenn ein Kindobjekt von BasicNode diese überschreibt.

Der Nachteil an diesem Kollisionskonzept ist, dass eine Kollision erst erkannt werden kann, sobald sie tatsächlich eintritt. Oft ist es aber nötig, dass eine Kollision vor Eintreten erkannt wird. So zum Beispiel wenn der Spieler in der Spielwelt den Boden nicht kollidiert sondern auf ihm steht.

Für diese Anforderung existiert noch ein zweites Kollisionssystem. Für dieses besitzt BasicNode neben den normalen Bewegungsmethoden (moveX(), moveY(),...) noch weitere, die dieses Kollisionssystem nutzen (moveColX(), moveColY(),...). Bei diesen Methoden wird das Objekt nicht direkt bewegt, sondern wird einer Queue hinzugefügt, die überprüft ob bei diesen Bewegungen, falls sie ausgeführt werden, eine Kollision entstehen würde. Ist das der Fall, wird das Objekt nicht bewegt. Analog wird das Objekt bewegt, wenn der Fall nicht eintritt.

**Spiellogik**

Die Wände (GrassWall.java / StoneWall.java) in der Spielwelt erben von DrawImage und besitzen keine weitere Logik.

Die Knöpfe, die der Spieler betätigen kann, sind in der Klasse Switch.java implementiert. Diese erbt von AnimImage, damit, bei Betätigung der Schalter, diese sichtlich umgelegt werden. In seiner intersect-Methode überprüft ein Schalter, ob er mit der Spielfigur kollidiert. Sollte dies der Fall sein und der Spieler drückt zugleich auch die ‘F’-Taste auf der Tastatur, wird der Schalter umgelegt. Dabei wird die Ereignislogik ausgeführt, die mit der Methode setOnTriggered(..) definiert wurde. Im Spiellevel steuert die Ereignislogik eines jedes Knopfes die unterschiedlichen Türen.

Die Spiellogik der Spielfigur prüft pro Frame ob eine Richtungstaste gedrückt wird. Ist das der Fall bewegt sich der Spieler unter Betrachtungen von Kollisionen (*siehe Abs. Kollisonssystem*). Um eine Gravitation zu imitieren wird eine exponential-Funktion genutzt, bei der die Parameter so gewählt sind, dass eine typische Sprungkraft zustande kommt, wie man es von ähnlichen Spielen gewohnt ist.

Die Zielflagge des Levels ist in der Klasse LevelFinish.java implementiert. Sie überprüft ob der Spieler sie berührt. Ist das der Fall spielt sie den Sound für das Abschließen des Levels ab. Das Level überprüft zudem auch, ob der Spieler das Ziel erreicht hat. Wenn er es erreicht, lädt es wieder in das Hauptmenü.

**Zeitmodifikation**

BasicNode.java implementiert das Timeable-Interface, welches die Klasse um die Funktion des Zurückspulens der Zeit erweitert. Im Prinzip baut diese Funktionen auf einen simplen *Stack auf*, der um die Funktion erweitert wurdeüberlaufen zu dürfen und am Anfang wieder anfängt und die ersten Objekte wieder überschreibt. Im Spiel hat der Spieler einen Stack mit der Größe von 3000 Einträgen. So können 3000 Positionen des Spielers festgehalten werden. Pro Frame wird die Position des Spielers in diesem Stack gespeichert. Ist der Stack voll, werden die ersten Einträge überschrieben, so sind im Stack die 3000 neuesten Positionen des Spielers gespeichert. Aktiviert der Spieler nun die Zeitzurückspulung, wird der Stack geleert, bis kein Element mehr vorhanden ist. Während dieses Prozesses werden alle anderen Funktionen der Spielfigur deaktiviert und erst wieder freigegeben, sobald der Prozess abgeschlossen ist.

Die Implementation der Zeitmodifikation befindet sich in der tick-Methode von BasicNode.java.

**Spielanleitung**

**Spielziel**

Ziel des Spiels ist es mit der Figur an der Zielflagge an zu kommen. Um das Level zu abzuschließen muss ein Rätsel aus Türen durchquert werden. Die meisten dieser Türen versperren dem Spieler den Weg und so müssen, mithilfe von Schaltern, die Türen so geöffnet und geschlossen werden, dass sich ein Weg für den Spieler zur Flagge ergibt.

Nun ist es aber nicht möglich normal dieses Rätsel zu lösen, sondern es muss geschickt die Zeit zurückgespult werden um einige Wege überhaupt erst zu ermöglichen. Dabei muss beachtet werden, dass die Spielfigur bis zu 3 Sekunden zurückgesetzt wird. Beim Zurückspulen ist die Spielfigur in der Lage Türen zu durchqueren, wenn diese zwischendurch geschlossen wurden.

**Steuerung**

In Verse steuert man die Spielfigur mit den Tasten „A“ und „D“ nach links bzw. rechts. Springen kann man mit der Leertaste.

Sollte einmal eine Situation entstehen, in der es aufgrund fehlbenutzter Schalter nicht weitergeht, kann das Level mit der „R“-Taste zurückgesetzt werden.

Mit der „E“-Taste wird das Zurückspulen der Zeit gestartet.

Mit Drücken der „F“-Taste können die Schalter in der Spielwelt umgelegt werden.