Prozedurale Levelgenerierung für rasterbasierte Mehrspieler-Rätselspiele

Moritz Kertesz

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Januar 2017

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 25. Januar 2017

Moritz Kertesz

Inhaltsverzeichnis

Kurzfassung

Diese Arbeit stellt mehrere Methoden zur prozeduralen Generierung von Spielinhalten vor und evaluiert diese im Bezug auf ihre Anwendbarkeit in rasterbasierten Mehrspieler-Rätselspielen. Aus einer Kombination der vielversprechendsten Konzepte wird ein Level-Generator erstellt, welcher in das Spiel *Vancouver Maneuver* integriert wird. Der Zielzustand des Spiels wird vom Algorithmus als Ausgangsbasis herangezogen, um den Lösungsweg in umgekehrt chronologischer Reihenfolge aufzubauen. Zufällige Spiel-Aktionen werden aneinander gereiht und dafür benötigte Spielelemente werden erzeugt. Diese Vorgehensweise stellt die Lösbarkeit der generierten Level-Instanzen sicher, sodass keine nachträgliche Überprüfung notwendig ist.

Abstract

This thesis introduces multiple methods for procedural content generation in games and evaluates how suitable they are for usage in grid-based multiplayer puzzle games. The most promising concepts get combined to create a level generator which is then implemented in the game *Vancouver Maneuver*. The algorithm uses the goal state of the game as a base to build a solution in reverse chronological order. Random player actions get chained together and required game elements for these actions get created. This procedure ensures the solvability of the generated level instances so that no additional verification is necessary.

Kapitel 1

Einleitung

1.1 Motivation

Während sich die Videospielindustrie in einem stetigen Wachstum befindet, steigen auch die Produktionskosten und der erforderliche Aufwand, um abwechslungsreiche und bedeutungsvolle Spielinhalte zu erstellen, die den hohen Erwartungen der Spieler gerecht werden. Dies gilt sowohl für renommierte *AAA*-Studios, deren Interesse es ist Arbeitskräfte und Entwicklungszeit einzusparen, als auch für aufsteigende unabhängige Entwickler, welche die dazu notwendigen Ressourcen erst gar nicht aufbringen können. Nicht zuletzt aus diesem Grund stellt die prozedurale Generierung von Inhalten für Computerspiele ein breites Feld dar, in welchem über die letzten Jahrzehnte hinweg aktiv Forschung betrieben wurde.

Es gibt kaum eine Komponente in Videospielen, bei der nicht versucht wird sie algorithmisch zu erstellen. Von audiovisuellen Elementen wie 3D-Modellen, Texturen, Animationen, Musik und Soundeffekten, über Verhaltensroutinen für computergesteuerte Entitäten, bis hin zur Automatisierung von Storytelling und Leveldesign bleibt kein Bereich unangetastet. Diese Arbeit geht dabei auf den letzteren Aspekt ein und konzentriert sich ganz spezifisch auf den Teilbereich der automatisierten Levelgenerierung für rasterbasierte Mehrspieler-Rätselspiele. Die Klassifikationen Rätselspiel und Mehrspieler-Spiel wurden aus einem bestimmten Grund gewählt: Auch wenn es mehrere Arbeiten gibt, die jeweils einen der beiden Aspekte bearbeiten, so ist in der Forschung vor allem in dieser Kombination nur sehr wenig Material vorhanden.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es einen Algorithmus zu finden, der in der Lage ist eine unbegrenzte bzw. möglichst große Anzahl an nicht-trivialen Rätsel-Level-Instanzen für das Videospiel *Vancouver Maneuver* zu generieren. Die-

ses wurde im Laufe des Studiums als mehrsemestriges Projekt umgesetzt. Da es sich dabei um ein kooperatives Spielerlebnis handelt, liegt bei der Generierung besonderer Fokus darauf, dass alle Spieler bei der Lösung des Rätsels involviert sein und sich gegenseitig helfen müssen, um ans Ziel zu gelangen. Ein komplettes Level soll also nicht allein durch Aktionen eines einzelnen Spielers abschließbar sein, sondern idealerweise häufig abwechselnde Handlungen aller Spieler erfordern. Natürlich muss ebenso die generelle Lösbarkeit der einzelnen Levels garantiert werden.

Eine weiteres Ziel ist es den Algorithmus so zu gestalten, dass er eine hohe Kontrollierbarkeit aufweist. Der Generator soll einige benutzerdefinierbare Parameter bereitstellen und Ergebnisse liefern, die diesen Einstellungen möglichst exakt entsprechen. Abgesehen von davon soll der Generierungsprozess ohne weitere Eingriffe ablaufen, optimalerweise sogar in so geringer Zeit, dass die Levels zur Laufzeit des Spiels generiert werden können.

Auf die visuelle Erscheinung der Levels wird primär kein Wert gelegt, sondern nur darauf, dass logisch funktionierende Rätsel einer gewünschten Komplexität generiert werden. Da also keine optisch ansprechende Geometrie erzeugt werden soll, wurde eine Beschränkung auf rasterbasierte Levels festgelegt. Obwohl prinzipiell nur Levels für *Vancouver Maneuver* generiert werden und keine Universallösung für alle Mehrspieler-Rätselspiele geliefert werden kann, sollen die verwendeten Konzepte in ähnlichen Spielen anwendbar sein und als Ansatzpunkt für weitere Arbeiten dienen.

1.3 Aufbau dieser Arbeit

Die Grundstruktur dieser Arbeit orientiert sich in groben Zügen an Bernhard Handlers Masterarbeit mit dem Titel *Prozedurale Levelgenerierung für 2D Plattformspiele* [4], da sie als Inspiration für die Themenwahl diente und sich die grundlegende Aufgabe der automatisierten Level-Erstellung für ein einzelnes Genre sehr ähnelt. Die in Rätselspielen anwendbaren Methoden unterscheiden sich jedoch grundlegend von jenen der Plattformspiele.

Kapitel 2 soll zunächst einen generellen Überblick über die Möglichkeiten der prozeduralen Inhaltsgenerierung in Computerspielen bieten. Neben einer Begriffsdefinition werden einzelne Anwendungsgebiete beschrieben und nennenswerte Beispiele aufgelistet.

In Kapitel 3 wird definiert welche Eigenschaften ein Rätselspiel aufweisen muss, um als solches kategorisiert zu werden. Da sich Spiele innerhalb dieses Genres sehr voneinander unterscheiden können, werden auch hier einige Beispiele angeführt und anhand einer Liste von typischen Elementen wird das exakte Subgenre eingegrenzt, mit dem sich diese Arbeit beschäftigt.

Anschließend wird in Kapitel 4 das Spiel *Vancouver Maneuver* vorgestellt, für welches in weiterer Folge Levels generiert werden. Es wird auf alle wichtigen Details eingegangen und anhand eines manuell erstellten Levels

gezeigt, wie ein typisches Spiel abläuft.

Nachfolgend wird in Kapitel 5 eine Reihe von Ansätzen und Methoden beschrieben, die sich zur Generierung von Rätseln besonders anbieten und daher für das Projekt in Erwägung gezogen wurden. Es wird erläutert, ob der jeweilige Algorithmus sich für den Einsatz in *Vancouver Maneuver* eignet.

Der eigens entwickelte Ansatz, wird in Kapitel 6 veranschaulicht. Darin wird unter anderem geschildert, welche Teilaspekte der zuvor vorgestellten Methoden in die eigene Vorgehensweise eingearbeitet werden konnten.

Kapitel 7 schildert die Implementierungsdetails des verwendeten Verfahrens und liefert weitere Informationen zu den einzelnen Generierungsschritten. Es werden unter anderem die Programmarchitektur und einzele Teilalgorithmen beschrieben.

In Kapitel 8 wird der umgesetzte Ansatz analysiert und aufgrund seiner Effizienz und der Kontrollierbarkeit einzelner Eigenschaften wie z. B. Schwierigkeitsgrad bewertet und mit zuvor manuell erstellten Levels verglichen.

Kapitel 9 besteht aus einer Zusammenfassung und einem Fazit zum durchgeführten Projekt. Abschließend wird noch ein Ausblick auf weitere mögliche Entwicklungen im Bereich der prozeduralen Generierung gewagt.

Kapitel 2

Prozedurale Generierung von Spielinhalten

2.1 Definition

Für den Terminus der prozeduralen Generierung in Spielen existieren viele Definitionen, die in ihrem Detailgrad stark variieren. Unter anderem beschreibt Smelik [15] diesen als Überbegriff für Algorithmen, die automatisch spezifische Inhalte, basierend auf einer begrenzten Anzahl an Eingabeparametern, erstellen. Da sie diese geringe Anzahl an Eingangsdaten in eine weit größere Datenmenge umwandeln werden sie von Roden u. a. [14] auch als Daten-Amplifikations-Algorithmen bezeichnet. Hendrikx u. a. [5] hingegen inkludieren die Bewertung und Selektion interessanter erzeugter Instanzen in der Begriffsbeschreibung. Togelius u. a. [18] wiederum beschreiben prozedurale Generierung schlicht als automatische Erstellung von Inhalten durch Algorithmen. Für sie stellt sie also einfach das Gegenteil zur manuellen Generierung dar.

2.2 Anwendungsgebiete

Es gibt keinen universalen Ansatz für die Generierung von Spielinhalten und sehr spezifische Lösungen für die jeweiligen Problemstellungen müssen dabei gefunden werden. Daher sollen nachfolgend grobe Kategorien vorgestellt werden, in denen sich prozedurale Generierung abspielen kann. Hendrikx u. a. [5] unterteilen die generierbaren Komponenten eines Videospiel in sechs, aufeinander aufbauende, Kategorien. Im Folgenden werden die ersten fünf davon beschrieben (*Derived Content* wird wegen Irrelevanz ausgelassen).

2.2.1 *Game-Bits*

Game-Bits sind die elementarsten Bestandteile eines Spiels. Dinge wie Texturen und Audiospuren werden als abstrakte Teile beschrieben, die in weiterer Folge zu konkreten interagierbaren Objekten, wie Gebäuden oder Vegetation zusammengesetzt werden. Auch wenn vor allem komplexere Texturen noch durch die manuelle Arbeit eines Designers erstellt werden müssen, existieren viele, meistens auf Rauschfunktionen basierende, Methoden, um simplere Strukturen von Materialien wie Holz oder Marmor zu erstellen. Im Audiobereich wird sowohl versucht die Komposition von Musik zu automatisieren, als auch die Generierung von Soundeffekten betrieben. Ebenso stellen 3D-Modelle und deren Animationen häufig das Motiv für prozedurale Generierung dar.

2.2.2 *Game-Space*

Der *Game-Space* ist die mit *Game-Bits* gefüllte Welt in der das Spiel abläuft. Hier wird aufgrund der unterschiedlichen Anforderungen zwischen *Indoor*- und *Outdoor*-Leveln unterschieden. Um *Indoor*-Level zu erzeugen eignen sich Techniken wie zellulare Automaten [7] zur Erzeugung höhlenartiger Strukturen, Grammatik-Ansätze (siehe Abschnitt 5.3) oder eine breite Auswahl an Labyrinth-Generatoren für *Dungeons* und auch genetische Algorithmen (siehe Abschnitt 5.4). Für *Outdoor*-Level existieren vollautomatische Geländegeneratoren, aber auch prozedurale Werkzeuge, die Designer nicht vollends ersetzen, sondern deren Arbeitsabläufe beschleunigen sollen.

2.2.3 *Game-Systems*

Zu den *Game-Systems* zählen Simulationen wie Ökosysteme, Straßennetzwerke, Städte oder das Verhalten von Entitäten. Der Einsatz von solchen Systemen kann den *Game-Space* für den Spieler weit glaubhafter und interessanter machen. Ökosysteme können durch *Producer/Consumer*-Systeme modelliert werden, die z. B. komplexe Nahrungsketten und Fortpflanzungsmechanismen simulieren. Bei der Generierung von Straßennetzwerken und Städten kommen L-Systeme (siehe Abschnitt 5.3.3), Agenten-Simulationen und Tensor-Felder zum Einsatz [15]. Um Verhaltensweisen zu simulieren, wird auf das Forschungsgebiet der künstlichen Intelligenz zurückgegriffen.

2.2.4 *Game-Scenarios*

Im Bereich der *Game-Scenarios* ist die Generierung von Rätseln und Storytelling angesiedelt. Die Story eines Spiels zu generieren erscheint besonders reizvoll, da die Interaktivität in der Natur des Spiels liegt und sich diese dynamisch an die Handlungen des Spielers anpassen könnte. Dazu können komplexe Systeme und künstliche Intelligenz eingesetzt werden. Es gilt da-

bei eine gute Balance zwischen strukturierten Storylines, die den Richtlinien des Storytelling folgen, und dem Einfluss des Spielers auf die Handlung zu erreichen. Rätsel bilden zwar ein eigenes Videospiel-Genre, sie sind aber auch oft Bestandteil vieler anderer Spiele. Auf jene Algorithmen, die häufig verwendet werden um Rätsel zu erzeugen, wird in Kapitel 5 eingegangen.

2.2.5 *Game-Design*

Das *Game-Design* umfasst die Regeln und Ziele eines Spiels und stellt damit den Hauptaspekt eines Spiels dar. Doch auch in diesem Gebiet wird an prozeduraler Generierung geforscht. Auch wenn die Ergebnisse nur wenig Komplexität aufweisen, können z. B. mit einem Generator von Nelson u. a. [11] Spielsysteme basierend auf der Eingabe von Wörtern oder Texturen erstellt werden. Über eine Datenbank werden eine zu den Eingabedaten passende Thematik, das Genre und mehrere Spielelemente kombiniert, um ein logisch intaktes Spiel zu generieren.

2.3 Online- und Offline-Algorithmen

Algorithmen zur prozeduralen Generierung von Spielen werden häufig anhand ihrer Laufzeit als Online- oder Offline-Algorithmen klassifiziert.

2.3.1 Online-Algorithmen

Eine Prozedur wird als online bezeichnet, wenn sie schnell genug terminiert, um in Echtzeit ausgeführt zu werden. Das ermöglicht es, dem Spieler bei jedem Neustart ein einzigartiges Spielerlebnis zu verschaffen, ohne dabei störende Ladezeiten zu verursachen. Ebenso kann dadurch Speicherplatz gespart werden, da die Menge an Eingangsdaten für die Generierung geringer als die Menge der Ausgangsdaten ist. Weiters ermöglichen Online-Algorithmen Inhalte, die sich während der Laufzeit dynamisch an die Verhaltensweise des Spielers anpassen, was von Yannakakis u. a. [19] als *Experience-Driven Content* bezeichnet wird und immer häufiger Verwendung findet.

2.3.2 Offline-Algorithmen

Im Gegensatz dazu spricht man von einem Offline-Algorithmus, wenn dieser eine zu lange Laufzeit hat, um während des Spielens ausgeführt werden zu können. Die Erzeugnisse des jeweiligen Algorithmus müssen daher vorberechnet und auf der Festplatte gespeichert werden. Offline-Algorithmen werden hauptsächlich dazu genutzt Arbeitsabläufe in der Entwicklungsphase zu beschleunigen.

Abbildung 2.1: Nahezu jede Kleinigkeit in *.kkrieger* (2004) wurde prozedural generiert um die Dateigröße des Spiels möglichst gering zu halten [31].

2.4 Beispiele

2.4.1 *.kkrieger*

Bei *.kkrieger* [22] handelt es sich um einen Ego-Shooter der durch die Motivation entstand, ein Spiel mit einer Dateigröße von unter 96 Kilobyte zu entwickeln (siehe Abbildung 2.1). Das Spiel hat dennoch lange Ladezeiten, da alle Ressourcen beim Start prozedural erzeugt werden müssen. So werden etwa die 3D-Modelle aus primitiven Körpern wie Kugeln, Quadern und Zylindern zusammengestellt und deformiert, während die Texturen durch die Kombination verschiedener Formeln erzeugt werden. Es wurde hier also kein Wert darauf gelegt neue Inhalte zu generieren, sondern nur bestehende Inhalte durch möglichst kurze Beschreibungen abzuspeichern. Da es auch nur ein einziges Level beinhaltet, ist *.kkrieger* eher als Tech-Demo zu verstehen und nicht als vollwertiges Spiel. Alle hier generierten Elemente sind der Kategorie *Game-Bits* zuzuordnen.

2.4.2 *Borderlands 2*

Borderlands 2 [25] ist ebenfalls ein Ego-Shooter, mit der Besonderheit, dass nahezu alle Waffen durch prozedurale Generierung erzeugt werden (siehe Abbildung 2.2). Diese werden modular aus kleinen vorgefertigten 3D-Modellen randomisiert zusammengebaut. Auch die Attribute jeder einzelnen Waffe werden zufällig generiert. Hierbei werden Formeln verwendet, die sicherstellen, dass keine Erzeugnisse entstehen, die die Spielbalance durcheinander bringen. Auch hier beläuft sich die Generierung nur auf *Game-Bits*.

Abbildung 2.2: In *Borderlands 2* (2012) können ungefähr 17,75 Millionen einzigartige prozedural generierte Waffen eingesammelt zu werden [30].

Abbildung 2.3: Einer der ungefähr 18 Trillionen einzigartigen Planeten aus *No Man's Sky* (2016) [32]. Fast alle sichtbaren Komponenten des Spiels machen Gebrauch von prozeduraler Generierung.

2.4.3 *No Man's Sky*

No Man's Sky [23] ist ein Weltraum-*Survival*-Spiel, welches sowohl die Generierung von *Game-Bits*, *Game-Spaces* als auch *Game-Systems* einsetzt (siehe Abbildung 2.3). Jeder einzelne Planet, den der Spieler erforschen kann, ist prozedural erstellt. Von der Form des Terrains, über die Vegetation bis zum das Farbschema hin wird alles generiert. Spezies von Tieren, die auf manchen Planeten zu entdecken sind, werden aus vordefinierten Modellen zusammengebaut, verzerrt, gebogen etc. und mit prozeduralen Animationen zum Leben erweckt. Auch deren Verhalten und das von feindlich und freundlich gesinnten Nicht-Spieler-Charakteren unterliegt dem Generierungsprozess.

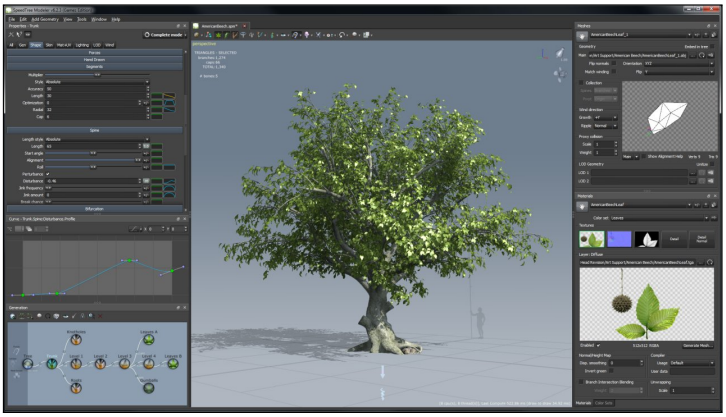


Abbildung 2.4: Mit *SpeedTree* können durch einen knotenbasierten Workflow zahlreiche Arten von Vegetation erzeugt werden [35].

2.4.4 *SpeedTree*

SpeedTree [28] ist kein Videospiel, sondern ein Werkzeug zum Generieren von Bäumen und anderen organischen Strukturen (siehe Abbildung 2.4). Designer ersparen sich die Modellierung von mehreren leicht unterschiedlichen Baummodellen, indem sie gewisse Parameter, wie z. B. die Länge der Blätter oder die Winkel in denen sie wachsen, einstellen und den Rest der Arbeit dem Generator überlassen. *SpeedTree* erlaubt auch das manuelle Eingreifen während und nach der Generierung. Es ist in der Videospielindustrie mittlerweile ein Branchen-Standard, was die Erstellung von Vegetation angeht, und wird auch bei Filmproduktionen eingesetzt.

Kapitel 3

Rätselspiele

Dieses Kapitel dient dazu, die Kriterien zu vermitteln, welche ein Videospiel erfüllen muss, um in das Genre der Rätselspiele zu fallen. Da es dafür keine eindeutige und universale Definition gibt, soll hierdurch erläutert werden, wovon die Rede ist, wenn der Begriff im Laufe der Arbeit erwähnt wird. Anschließend werden die Spielelemente vorgestellt, die unter anderem in *Vancouver Maneuver* zum Einsatz kommen und auch generell in diesem Genre oft vertreten sind. Zuletzt wird eine Reihe von kommerziell erfolgreichen Spielen angeführt, welche die relevanten Spielmechaniken aufweisen.

3.1 Definition

Der Artenreichtum von Rätselspielen ist so groß, dass es schwer fällt eine genaue Definition dafür zu formulieren. Grundsätzlich basieren sie auf der Anwendung von Logik, um eine gewisse Problemstellung zu lösen. Hier wird ausschließlich auf Rätselspiele eingegangen, die kein Vorwissen vom Spieler, bis auf die Kenntnis der Spielregeln, und logische Kombinationsgabe erfordern. Die Zusammenhänge der einzelnen Rätselelemente müssen innerhalb des Spiels erfahrbar sein. Dem Spieler werden, bildlich gesprochen, gewisse Bausteine in die Hand gelegt, die er mithilfe von Hinweisen in der richtigen Reihenfolge anordnen muss. Die Bausteine können z. B. die Zahlen von *Sudoku* sein, die anhand der Limitierungen bereits eingetragener Zahlen in die leeren Zellen eines Rasters eingefüllt werden müssen. Es können aber auch einfach die verschiedenen Handlungsmöglichkeiten, die dem Spieler zur Verfügung stehen, in eine zeitlich korrekte Abfolge gebracht werden müssen. Der Spieler betreibt dabei gedanklich eine informierte Suche im Möglichkeitsraum des Rätselspiels. Im Rahmen dieser Arbeit wird das Spektrum der Rätselspiele auf solche limitiert, in denen die Spieler selbst als Charaktere in der Spielwelt abgebildet werden und, durch Manipulation und Bewegung von Objekten in dieser Welt, versuchen einen Zielzustand herzustellen.

3.2 Typische Elemente

Es gibt eine Reihe von wiederkehrenden Spielelementen im Genre der Rätselspiele. Nachfolgend sind die häufigsten und auch für diese Arbeit wichtigsten Objekte aufgelistet.

3.2.1 Spieleravatar

Der Avatar dient dazu, dem Spieler graphisch anzuzeigen, an welcher Position in der Spielwelt er sich befindet. Der Freiheitsgrad, mit dem der Avatar bewegt werden kann, ist je nach Spiel sehr unterschiedlich. Objekte können nur benutzt werden, wenn der Avatar sich z. B. im selben Raum oder direkt neben diesem befindet. Im Falle von Spielen in der Ego-Perspektive wird zusätzlich die Wahrnehmung des Spielers limitiert, was eine explorative Komponente zum Spiel hinzufügt. Das Vorhandenseins eines Spieleravatars ist in Rätselspielen nicht zwingend notwendig. Bei Abwesenheit eines solchen hat der Spieler meist freie Sicht auf das gesamte aktuelle Level und wird in der Wahl seiner Aktionen nicht durch die Position seines Avatars eingeschränkt. Oft möchte man in Spielen aber genau diese örtliche Abhängigkeit erreichen. Für die meisten nachfolgenden Spielelemente ist das Vorhandensein eines Spieleravatars eine Voraussetzung.

3.2.2 Schlüssel und Schloss

Ist es das Ziel des Spielers einen gewissen Ort zu erreichen, besteht die Herausforderung in der Regel darin schrittweise Zugang zu verschlossenen Bereichen zu erlangen. Die Barrieren, die diese Bereiche trennen, werden meist durch Türen, Laserschranken oder auch magische Hindernisse visualisiert. Es muss also vor dem Durchschreiten der jeweiligen Blockade ein entsprechender Schlüssel eingesammelt werden.

Der Einsatz des Schlüssel-Schloss-Prinzips ist kein eindeutiges Anzeichen dafür, dass es sich um ein Rätselspiel handelt. Es wird auch außerhalb des Rätselspiel-Genres im Leveldesign häufig eingesetzt, um die Linearität des Levels aufzubrechen. Wenn etwa mehrere Schlüssel erforderlich sind um eine Barriere zu öffnen, so kann der Spieler selbst entscheiden welchen er zuerst einsammeln möchte. Durch den Einsatz von Schlüssel und Schloss wird die Menge an *Backtracking*, die der Spieler durchführen muss erhöht. Das heißt, dass bereits besuchte Areale nochmal besucht werden müssen. Dadurch wird der Spieler dazu gebracht die Umgebung zu erforschen und die Spielzeit wird erhöht, ohne neue Level-Abschnitte erstellen zu müssen. Häufig nehmen in Spielen auch andere Gegenstände die Funktion eines Schlüssels ein. Ein Paradebeispiel dafür sind die Spiele der *The Legend of Zelda*-Serie (siehe Abschnitt 3.3.4).

3.2.3 Schalter

Ein ganz wichtiger Bestandteil vieler Rätselspiele sind Schalter. Diese verhalten sich in der Regel nach einem von mehreren verschiedenen Mustern:

Bleibender Effekt: Der Schalter verharrt nach dem Benutzen im betätigten Status und löst ein Ereignis aus. Die resultierende Spielmechanik entspricht dem eben besprochenen Schlüssel-Schloss-Prinzip.

Effekt während dem Halten: Der Effekt des Schalters ist nur aktiv solange er betätigt wird. Im Normalfall bedeutet dies, dass er mit einer Kiste oder von einem Mitspieler zu einem gewissen Zeitpunkt betätigt werden muss. Um zu verhindern, dass auch hier eine Schlüssel-Schloss-Situation entsteht, müssen für ein Objekt mehrere Schalter präsent sein, die an verschiedenen Zeitpunkten zu betätigen sind.

Zeit-Schalter: Der Effekt des Schalters tritt bei Betätigung sofort ein, wird aber nach einer vorgegebenen Zeit wieder rückgängig gemacht. In rundenbasierten Spielen kann die Anzahl der Felder, die mit dem Avatar passiert werden, als Maßeinheit der Zeit verwendet werden.

Zustände umschalten: Mehrmaliges Betätigen des Schalters wechselt zwischen zwei oder mehreren Zuständen hin und her. Interessant werden diese Schalter dadurch, dass sie dem Spieler zwar gewisse Wege öffnen, jedoch andere gleichzeitig verschließen. Alternativ können im Level auch mehrere zusammengehörige Schalter verteilt sein, die bei Betätigung auf nur genau einen der möglichen Zustände umschaltet. Der Spieler muss dadurch nicht bloß an einen, sondern je nach Situation verschiedene Orte erreichen, um Zustände umzuschalten.

3.2.4 Kisten

Kisten werden in Spielen, wie eben erwähnt, häufig dazu verwendet, um Schalter zu betätigen. Sie können in dreidimensionalen Spielen auch genutzt werden, um an zu hoch gelegene Orte zu gelangen. Kisten können für den Spieler auch eine Blockade darstellen, die aus dem Weg geräumt werden muss, oder sie müssen vom Spieler selbst dazu genutzt werden andere Mechanismen zu blockieren. Abhängig davon ob die Kisten vom Spieler getragen, geschoben oder gezogen werden können, ergeben sich unterschiedliche Mechaniken. So baut etwa das Spiel *Sokoban* einzig und allein auf der Mechanik des Schiebens von Kisten auf (siehe Abschnitt 3.3.1).

3.2.5 Laser und Spiegel

Ein wiederkehrendes Element in Rätselspielen ist die Verwendung von Laser- bzw. Lichtstrahlen. Dabei sind normalerweise zwei Objekte involviert: Ein Emitter bildet den Ursprungspunkt des Lasers, der einen Strahl in eine bestimmte Richtung aussendet, und ein Sensor stellt das Ziel dar, auf wel-

ches dieser treffen soll. Beide dieser Vorrichtungen sind meist, jedoch nicht zwingend, statisch im Level positioniert und weit voneinander entfernt. Der Spieler muss mithilfe anderer Gegenstände dafür sorgen, dass der Strahl den richtigen Pfad entlang verläuft. So kann der Laser z. B. mit Spiegeln reflektiert werden oder wie in *Portal 2* mit einem Würfel, der mit einem Linsensystem versehen ist, in eine gewünschte Richtung weitergeleitet werden (siehe Abschnitt 3.3.2). *The Talos Priciple* verwendet zusätzlich zwei verschiedene Farben von Lasern, um die Komplexität der Rätsel zu erhöhen (siehe Abschnitt 3.3.3). Oft sind Laser in Spielen so realisiert, dass sich zwei Strahlen nicht kreuzen dürfen, da sie sich ansonsten blockieren.

3.2.6 Unterschiedliche Fähigkeiten

Um in Mehrspieler-Spielen zu erreichen, dass alle Spieler zur Lösung eines Problems beitragen, können die dazu erforderlichen Fähigkeiten oder Gegenstände unter den Spielern aufgeteilt werden, anstatt allen dasselbe Set an Fähigkeiten zu geben. Auch wenn ein Spieler alleine immer noch in der Lage ist, das Level komplett durchzudenken, erfordert dann zumindest die Ausführung Handlungen aller Spieler. *The Lost Vikings* [27] ist unter anderem ein erfolgreicher Rätsel-Plattformer, der von dieser Mechanik Gebrauch macht.

3.3 Beispiele

Nachfolgend werden Beispiele erfolgreicher Spiele aufgelistet, die der oben angeführten Definition entsprechen und dabei helfen sollen ein besseres Bild vom Einsatz und Zusammenspiel dieser Elemente zu zeichnen. Der Großteil davon beinhaltet die bisher beschriebenen typischen Spielelemente. Alle erwähnten Spiele wurden auch getestet, um ein Gefühl dafür zu entwickeln, was die einzelnen Rätsel ansprechend und fordernd macht.

3.3.1 *Sokoban*

Sokoban [26] ist ein Spiel, in dem es einzig und allein darum geht Kisten an dafür vordefinierte Felder zu bewegen. Die Schwierigkeit entsteht dadurch, dass nur das Schieben und nicht das Ziehen der Kisten möglich ist. Durch diesen Umstand können sogenannte *Deadlocks* entstehen. Das sind irreversible unlösbare Situationen, in denen, im Falle dieses Spiels, Kisten nicht mehr aus einer Ecke oder von einer Wand wegbewegt werden können. Die Schwierigkeit des Spiels besteht darin ebendiese zu vermeiden. In den meisten Implementierungen können Züge jedoch auch rückgängig gemacht werden, um frustrierende Repetition zu vermeiden. In Abbildung 3.1 ist das erste Level des Originalspiels zu sehen.

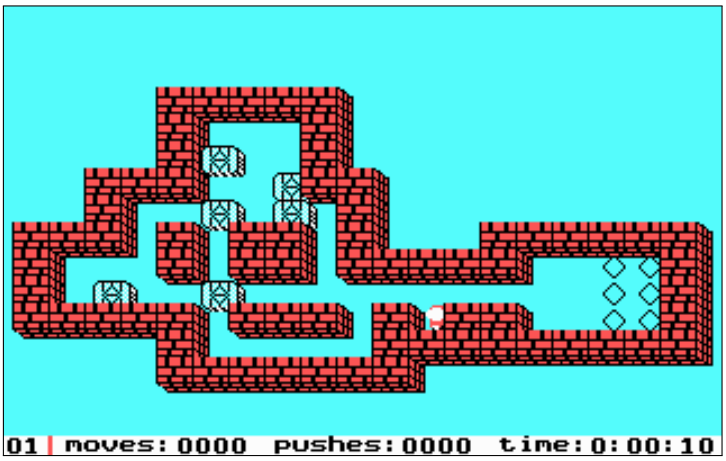


Abbildung 3.1: Das erste Level der Original-Version, rekonstruiert in der *MS-DOS*-Variante von *Sokoban* (1988) [34].

3.3.2 *Portal 2*

Portal 2 [20] ist ein physikbasiertes Rätselspiel in der Ego-Perspektive. Der Spieler kann mit seinem Avatar dabei durch das Level laufen, springen und an Wänden mittels einer Portal-Kanone je ein orangenes und blaues Portal erscheinen lassen. Dies funktioniert nur an dafür vorgesehenen Stellen, um das Vorankommen nicht zu einfach zu gestalten und gleichzeitig die Möglichkeiten, die der Spieler durchdenken muss, etwas zu reduzieren.

Die Rätsel basieren häufig darauf, dass der Spieler beim Hindurchspringen durch die Portale Schwung sammelt, um dann damit über einen andernfalls unüberwindbaren Abgrund zu springen. Abseits von den namensgebenden Portalen finden sich mit Schaltern, Kisten, Lasern und mehr auch alle der hier vorgestellten klassischen Rätselelemente.

Das Spiel bietet einen Mehrspieler-Modus für zwei Personen an (siehe Abbildung 3.2). Die Spieler besitzen dieselben Fähigkeiten wie im Einzelspieler-Modus und können daher gemeinsam insgesamt vier Portale erstellen. Die Kooperation der beiden Spieler wird meist dadurch erreicht, dass sie sich in zwei abgetrennten Bereichen bewegen, die aber durch Fenster oder ähnliches verbunden sind, wodurch sie Portale schießen können um ihrem Kollegen weiterzuhelfen. Dadurch sind die beiden Spieler trotz ihrer identischen Fähigkeiten aufeinander angewiesen.

3.3.3 *The Talos Principle*

In *The Talos Principle* [24] erwarten den Spieler, neben existentiellen und philosophischen Fragen, vor allem knifflige Rätsel in verwinkelten rasterbasierten 3D-Arealen. Es gilt durch diese Bereiche in der Ego-Perspektive zu

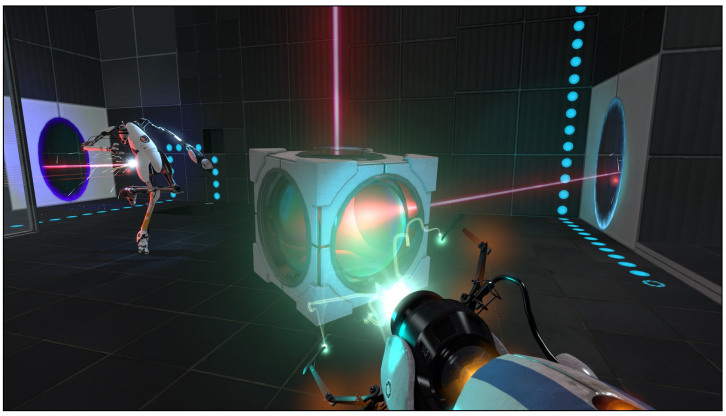


Abbildung 3.2: Eine Szene aus dem Mehrspielermodus von *Portal 2* (2011). Ein Laserstrahl wird durch ein Portal hindurch umgeleitet [33].

navigieren, um zu einem Schlüssel zu gelangen, der für das Voranschreiten in höhere Levels benötigt wird. Dabei versperren Hindernisse wie Kraftfelder, Minen und Geschütztürme den Weg zum Ziel. Zu Beginn stehen dem Spieler bloß sogenannte *Jammer* zur Verfügung, mit denen diese blockiert werden können, indem auf diese gezielt wird. Nach kurzer Zeit kommen Prismen hinzu, mit deren Hilfe verschiedenfarbige Laser zwischen Endpunkten zu verbinden sind, um die eben genannten Barrieren zu deaktivieren. Obwohl der Großteil der Levels meist sehr flach aufgebaut ist, kann hierbei die in einem 3D-Spiel zur Verfügung stehende Vertikalität genutzt werden, um z. B. zu verhindern, dass sich zwei Laserstrahlen kreuzen, indem einer der Strahlen angehoben wird wenn eines der Prismen auf eine erhöhte Position gestellt wird. Weitere Elemente sind Kisten, Schalter und Ventilatoren, die bestimmte Gegenstände und auch den Spieleravatar über Wände hinweg transportieren können. Der Reiz vieler Rätsel dieses Spiel entsteht dadurch, dass die vorhandenen Rätselelemente mindestens zwei Verwendungszwecke haben. Ein Prisma kann z. B. in gleicher Weise wie eine Kiste genutzt werden, um einen Schalter zu betätigen, während die Kiste zusätzlich dem Spieler helfen kann höher gelegene Ebenen zu erreichen. Durch diese Mehrfachfunktionen entstehen mehr Möglichkeiten, die der Spieler in seinem Kopf durchlaufen muss, was die Rätsel komplexerer erscheinen lässt.

Besonders interessant sind auch eine Handvoll sogenannter *Recording-Puzzles* (siehe Abbildung 3.3), die im Laufe des Spiels zu bewältigen sind, bei denen der Spieler seine Bewegungen aufzeichnen und abschließend wieder abspielen muss. Dabei materialisiert sich die Aufzeichnung und gibt die Bewegungen des Spielers wieder. Steht der Spieler bei der Aufzeichnung auf einem Schalter, der eine Laserschranke öffnet, kann er nun beim Abspielen hindurch laufen, da seine Projektion den Schalter gedrückt hält. Oft sind wiederholte Aufnahmen notwendig, die jedes Mal um ein paar Spieler-

Abbildung 3.3: Ein *Recording-Puzzle* aus *The Talos Principle* (2014) [37]. Die blau leuchtenden Objekte geben die Bewegungen der Aufnahme wieder.

Aktionen länger werden, welche durch die vorherige Aufnahme möglich gemacht wurden. Ignoriert man den Timing-Aspekt, fühlen sich diese Aufgaben ähnlich an wie ein Mehrspieler-Modus, bei dem der Spieler jedoch beide Avatare selbst steuert.

3.3.4 *The Legend of Zelda: Twilight Princess*

Bei den Spielen aus der *The Legend of Zelda*-Reihe handelt es sich um eine Mischung aus Abenteuer und Rätselspielen. Stellvertretend für alle Spiele der Serie wird hier *The Legend of Zelda: Twilight Princess* [21] vorgestellt, da in Abschnitt 5.3.2 darauf Bezug genommen wird. Der Spieler muss im Laufe des Spiels mehrere *Dungeons* durchqueren, welche sich aus einer Mischung von Rätseln und Kampfsequenzen zusammensetzen. Der grundsätzliche Ablauf dieser Levels basiert auf einer Vielzahl an Schlüssel/Schloss-Rätseln, die gleich auf zweierlei Arten funktionieren. In den Levels sind mehrere gleichartige Schlüssel zu finden, die an allen verschlossenen Türen anwendbar sind, dann jedoch aus dem Inventar des Spielers verschwinden. Beim Leveldesign musste daher darauf geachtet werden, dass sich der Spieler nicht aussperren kann. Das heißt, hinter allen Türen, welche der Spieler in einer anderen Reihenfolge öffnet, als im Lösungsweg des Designers vorgesehen ist, muss stets noch ein weiterer Schlüssel zu finden sein. Das Schlüssel-Schloss-Prinzip ist aber auch auf eine andere Weise im Spiel enthalten, die nicht ganz so offensichtlich ist. Der Spieler sammelt im Laufe des Spiels eine Menge an Gegenständen, die sowohl zum Kampf als auch zum Lösen von kleineren Rätseln in den einzelnen Räumen einsetzbar sind. Diese Gegenstände fungieren aber zusätzlich auch als Schlüssel. In einem Level müssen z. B. Schalter mit einem Boomerang betätigt werden, um Zugang zu einem neuen Bereich zu erlangen. Jenen Boomerang erhält der Spieler jedoch erst zu einem späte-

Abbildung 3.4: Screenshot des Seeschreins aus *The Legend of Zelda - Twilight Princess* (2006) [36]. Hier muss eine große Treppe rotiert werden, um, für den Spiel-Fortschritt notwendige, Räume zu erreichen.

ren Zeitpunkt des Levels. Es handelt sich hierbei also um einen Schlüssel dessen Anwendung lediglich mit einer Geschicklichkeitsaufgabe verbunden wird. Häufig werden in diesem Spiel auch Schalter eingesetzt, um die Zugänglichkeit einzelner Räume zu manipulieren. In Abbildung 3.4 ist ein Level zu sehen, in dem mittels Schaltern die Rotation einer großen Treppe im Zentrum gesteuert werden, welche die Erreichbarkeit entlegener Orte möglich macht.

Kapitel 4

Vancouver Maneuver

Vancouver Maneuver ist ein kooperatives *Augmented-Reality*-Brettspiel (siehe Abbildung 4.1) für zwei Spieler, die mit ihren Smartphones über eine WLAN-Verbindung kommunizieren. Ein vordefiniertes ausgedrucktes Bild liegt in der Mitte eines Tisches, das für die Smartphones als *Image-Tracker* dient, den die Smartphones identifizieren können und auf welchen die Spielwelt mittels 3D-Modellen projiziert wird. Dieser *Augmented-Reality*-Aspekt ist aber für diese Arbeit unerheblich und wird daher gänzlich vernachlässigt.

4.1 Spielablauf

Der Ablauf des Spiels teilt sich in zwei aufeinander folgende Akte auf. Zuerst muss ein Schlüssel für einen Tresor gefunden werden und anschließend muss der besagte Tresor damit geöffnet werden. Dann ist die Siegesbedingung erfüllt. Der Schlüssel ist im Normalfall in einem von mehreren Schränken zu

Abbildung 4.1: Die Sicht eines Spielers auf ein manuell designtes Level aus *Vancouver Maneuver* durch sein Smartphone.

finden, von denen die restlichen leer sind und nur zur Ablenkung für den Spieler dienen. Die Spieler teilen sich dabei kein Inventar, d. h. es muss genau jener Spieler, der den Schlüssel gefunden hat, auch zum Tresor gelangen. Rundenbasiert begeben sich diese auf andere Felder und führen diverse Aktionen aus, wie etwa Kisten zu verschieben, Schalter zu betätigen oder Spiegel zu drehen. Das sind bereits alle Interaktionen, die den Spielern zur Verfügung stehen. Sie starten in einem von mehreren quadratischen Räumen, jedoch nicht zwangsweise im gleichen. Eine Besonderheit des Spiels ist, dass die Räume jederzeit nach Belieben gedreht und an andere Rasterpositionen verschoben werden dürfen. Die Spieler können anliegende Räume allerdings nur betreten, wenn deren Türen alle exakt aneinander passen. Einige der Türen sind versperrt und können nur durch Schalter und Kisten, Laser-Emitter und Sensoren oder Kombinationen dieser Mechanismen geöffnet werden (siehe Abschnitt 3.2). Zusätzlich gibt es Überwachungskameras, welche ähnlich wie verschlossenen Türe als Blockaden funktionieren und das Betreten der Felder, die sich in deren Sichtkegel befinden, unmöglich machen. Um durch das Level zu navigieren, müssen die beiden Spieler zusammenzuarbeiten und darauf achten, dass stets der richtige Weg für den jeweils anderen Spieler geöffnet ist. Die Spieldauer bisheriger manuell erstellter Levels beträgt durchschnittlich fünf bis zehn Minuten.

4.2 Beispiel eines manuell erstellten Levels

Das folgende Beispiel illustriert wie der konkrete Spielablauf vonstattengeht. Es soll ein Bild davon verschaffen, welchen Schwierigkeitsgrad von Menschenhand geschaffene Levels aufweisen. Durch die Generierung sollen Levels ähnlichen Ausmaßes entstehen und deren Komplexität im Idealfall übertroffen werden.

4.2.1 Ausgangssituation

Die typische Ausgangssituation der meisten manuell designten Levels ist ein Spielbrett, bestehend aus vier Räumen. Das trifft auch auf das Beispiel-Level zu, welches in Abbildung 4.2 gezeigt wird. Beide Spieler P1 und P2 starten hierbei in Raum R3. Zuerst müssen sie den Schlüssel in Raum R4 erreichen und dann damit zum Tresor in Raum R2 gelangen.

Mittels punktierten Linien wird gezeigt welche Objekte auf andere Einfluss nehmen. Hierbei ist zu beachten, dass alle Objekte auf der Abbildung in deaktiviertem Zustand zu sehen sind. Das Schlosssymbol bei den Türen D6 und D7 zeigt an, dass diese standardmäßig geschlossen sind. Das heißt sie sind beim Start des Spiels bereits geöffnet, da die Kiste C1 auf dem Bodenschalter FS1 liegt, der den beiden Türen das Signal liefert ihren Status zu ändern. Würde eine punktierte Linie auf eine offene Tür zeigen, so würde sich diese bei Betätigung des Schalters also schließen. Selbiges gilt für die

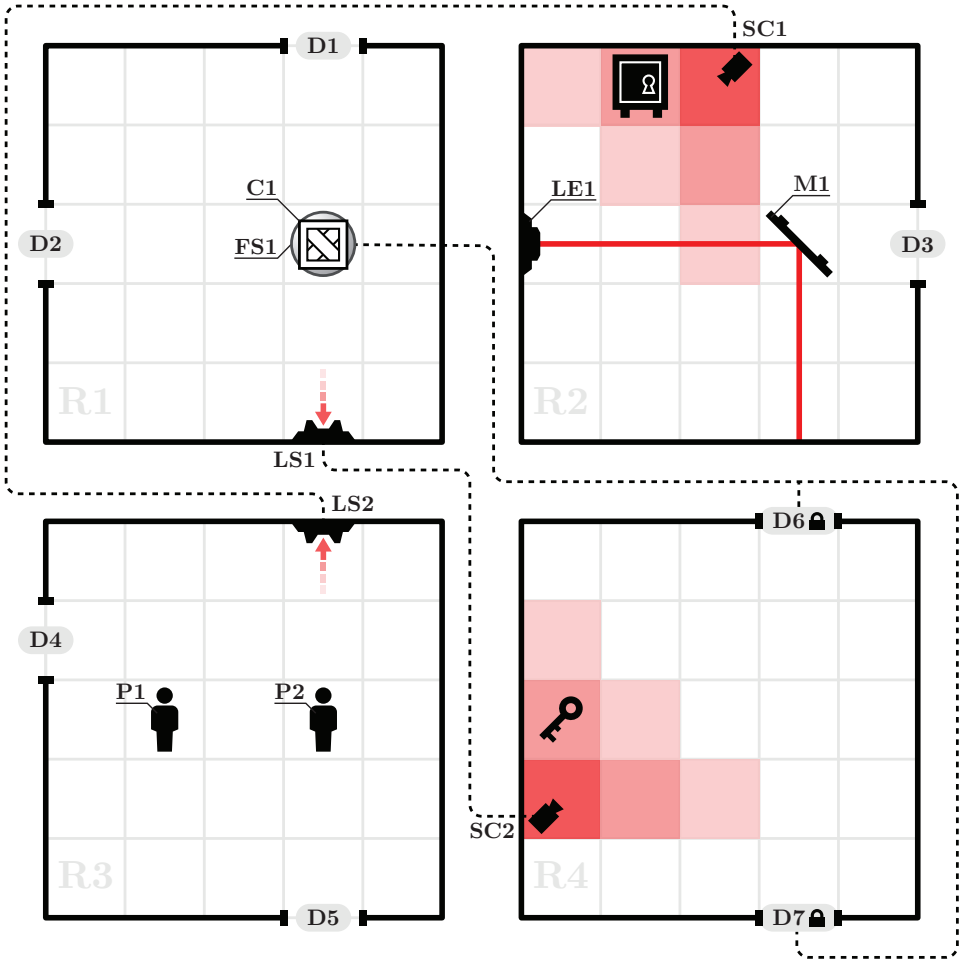


Abbildung 4.2: Diagramm eines manuell erstellten Levels aus *Vancouver Maneuver*. Bis auf den Schlüssel und den Tresor haben alle Objekte einen eindeutigen Bezeichner. Die Auslassungen in den Wänden stellen Türen dar.

Überwachungskameras SC1 und SC2, die sich wenn sie kein Signal erhalten im eingeschalteten Zustand befinden. Die rot schattierten Felder zeigen den Sichtbereich dieser an, der nicht betreten werden darf. Trifft also ein Laserstrahl auf den Sensor LS1, so erhält die Kamera SC2 ein Signal und wird dadurch ausgeschaltet. Ab diesem Zeitpunkt dürfen die rot markierten Felder im Sichtkegel der Kamera dann betreten werden.

4.2.2 Lösungsweg

Zwischen den einzelnen Schritten ist es oft notwendig die Räume zu drehen und anders aneinander zu legen. Dies wird nicht explizit als Lösungsschritt angeführt, sondern es wird nur aufgelistet durch welche Türen die Spieler

hindurch schreiten müssen. Welche Drehungen dafür notwendig sind, ergibt sich von selbst. Der Lösungsweg lautet wie folgt:

1. Spieler P1 geht entweder durch die Türen D4 und D7 oder durch D5 und D6 um in Raum R4 zu gelangen.
2. P2 geht durch D5 und D1 in R1.
3. P2 schiebt die Kiste C1 von Boden-Schalter FS1 hinunter. Die Türen D6 und D7 schließen sich.
4. P2 zieht den Spiegel M1 durch T3 und T2 hindurch und auf FS1 hinauf. D6 und D7 öffnen sich wieder.
5. P2 dreht den Spiegel um 90° gegen den Uhrzeigersinn und deaktiviert mittels dem Laser-Sensor LS1 die Kamera SC2.
6. P1 kann nun den Schlüssel nehmen und durch D7 und D4 wieder zurück in den Raum R3 gehen. Dabei darf P1 am Ende des Zuges nicht den Weg zu Laser-Sensor LS2 blockieren.
7. P2 dreht M1 so, dass LS2 aktiviert wird. Raum R3 muss so an R1 angelegt sein, dass der Laserstrahl durch T1 und T5 hindurch kann. Die Ursprungsposition des Schlüssels ist nun wieder versperrt, aber der Tresor ist nun erreichbar.
8. Da P1 im Besitz des Schlüssels ist, muss dieser nun zum Tresor navigieren und das Level dort erfolgreich abschließen.

Bei dieser Abfolge wurde viermal der Spieler gewechselt. P1 kam dreimal zum Zug, während P2 nur zweimal handeln durfte. P2 hatte jedoch anfangs ein längeres Segment zu bewältigen, das aus mehreren Schritten bestand. Die Aufteilung der Spielzeit kann daher als ungefähr gleich bezeichnet werden. Die Länge dieses Rätsels wird für *Vancouver Maneuver* schon als zufriedenstellend angesehen, da die angegebenen Schritte nur der ideale und kürzeste Lösungsweg sind. Wird das Spiel von Menschen gespielt, ist zu beachten, dass die Spieler sich zu Beginn erst einmal im Level Orientierung verschaffen und die Zusammenhänge der Elemente durch Ausprobieren erfahren müssen. Weiters können bei der Ausführung auch Fehler gemacht werden, welche rückgängig gemacht werden müssen und so die Spielzeit verlängern.

Kapitel 5

Erwogene Ansätze

Im folgenden Kapitel werden jene Methoden zur Levelgenerierung vorgestellt die für das Projekt in Erwägung gezogen wurden. Diese Ansätze werden für Spiele verschiedenster Genres eingesetzt, unter anderem in Rätselspielen. Es wird darauf eingegangen, ob die einzelnen Techniken auch für das Subgenre geeignet sind, welchem *Vancouver Maneuver* angehört.

5.1 *Brute-Force*-Suche mit *Solver*

Der konzeptuell einfachste Ansatz, um ein Level zu erzeugen, ist die *Brute-Force*-Suche. Dabei werden prinzipiell alle möglichen Konfigurationen eines Systems generiert und auf die jeweiligen Anforderungen geprüft, weswegen sie ebenfalls erschöpfende Suche genannt wird. Mit jeder neuen Variable, die bei diesem System eingeführt wird, steigt der erforderliche Aufwand exponentiell an. Wenn für eine Problemstellung jedoch keine bessere Lösung bekannt ist, kann die erschöpfende Suche oft der einzige Weg sein eine Lösung zu finden. Für kleine Suchräume stellt sie allerdings oft eine angebrachte Herangehensweise dar.

Eine häufige Anwendung der *Brute-Force*-Suche ist auch das Knacken von Passwörtern. Dabei werden im einfachsten Fall alle möglichen Zeichenketten iterativ ausprobiert, bis die Lösung gefunden wird oder eine gewisse Passwort-Länge erreicht ist. Da hierbei eine einzige richtige Kombination existiert, muss der gesamte Lösungsraum abgesucht werden. In anderen Systemen kann jedoch eine Vielzahl an Lösungen vorhanden sein.

Unter Umständen kann es Sinn ergeben eine geringere Anzahl an zufälligen Kombinationen zu testen. Dies ist der Fall, wenn die Vermutung besteht, dass mehrere Lösungen existieren und der Suchraum so groß ist, dass er nicht in akzeptabler Laufzeit komplett durchsucht werden kann. Dadurch verringert sich jedoch die Wahrscheinlichkeit, dass tatsächlich eine Lösung gefunden wird.

5.1.1 Aufbau des Suchbaums

Um den Lösungsraum von *Vancouver Maneuver* zu durchsuchen, eignet sich eine Baumstruktur. Im Falle des oben genannten Passwort-Beispiels wäre es zwar am effizientesten, mit verschachtelten Schleifen über alle möglichen Zeichen zu iterieren, zur Veranschaulichung wird hier jedoch trotzdem erklärt wie es mithilfe eines Baumes ablaufen würde: Der Wurzelknoten wäre eine leere Zeichenkette. Von dieser Zeichenkette werden alle Möglichkeiten berechnet, wie sie erweitert werden könnte. Angenommen, das Passwort erlaubt englische alphanumerische Zeichen, so werden 36 Kindsknoten expandiert, von denen jeder eines der möglichen Passwörter mit einer Länge von 1 enthalten. Jeder einzelne Knoten muss darauf überprüft werden ob er die Lösung enthält. Das ganze wird dann für jedes Blatt im Baum wiederholt, bis eine gewünschte Suchtiefe erreicht ist.

Auf den konkreten Fall vom *Vancouver Maneuver* umgelegt würde dies bedeuten, dass mit einem leeren Spielbrett begonnen wird. Es werden nun alle Möglichkeiten errechnet, wie und wo ein neues Spielelement platziert werden kann und diese als Kindsknoten angefügt. In der ersten Iteration muss ein Raum platziert werden, da für das Positionieren der anderen Spielelemente das Vorhandensein eines Raumes eine Voraussetzung ist. Die Spielsituation wird dann einem *Solver* übergeben der die Lösbarkeit und Schwierigkeit des Levels verifizieren kann. Im nächsten Schritt bestehen nun schon mehr Möglichkeiten, wie das Spielbrett erweitert werden kann. Es könnte ein weiterer Raum, ein Spieler, ein Schalter, eine Kiste, eine Tür etc. erzeugt und an eine zufällige Position platziert werden.

Da der Lösungsraum von *Vancouver Maneuver* für eine iterative Vorgehensweise zu groß ist, muss auf eine Baumstruktur und das zufällige Hinzufügen von Spielelementen zurückgegriffen werden. Wie in Abbildung 5.1 simplifiziert zu sehen ist, können Duplikate entstehen, wenn dieselben Elemente in unterschiedlicher Reihenfolge hinzugefügt werden. Diese sollten in einer effizienten Implementierung identifiziert und gelöscht werden, da deren Lösbarkeit und die all ihrer Kinder sonst mehrfach geprüft wird. Solche Redundanzen sind bei einem ohnehin schon langsamen Algorithmus wie diesem auf jeden Fall zu vermeiden. Da Duplikate nur an derselben Suchtiefe auftreten können, wäre es möglich jedes Mal vor Hinzufügen eines Kindsknotens nur jene Elemente zu überprüfen, welche bereits auf derselben Tiefe vorhandenen sind.

5.1.2 *Solver*

Um die Spielbarkeit der Levels zu testen, wird ein sogenannter *Solver* benötigt. Dieser beginnt im Ausgangszustand des Levels, ermittelt alle möglichen Züge für diesen und berechnet alle Stadien in denen sich das Spiel nach Durchführung dieser Züge befinden kann. Dabei wird analog zur in

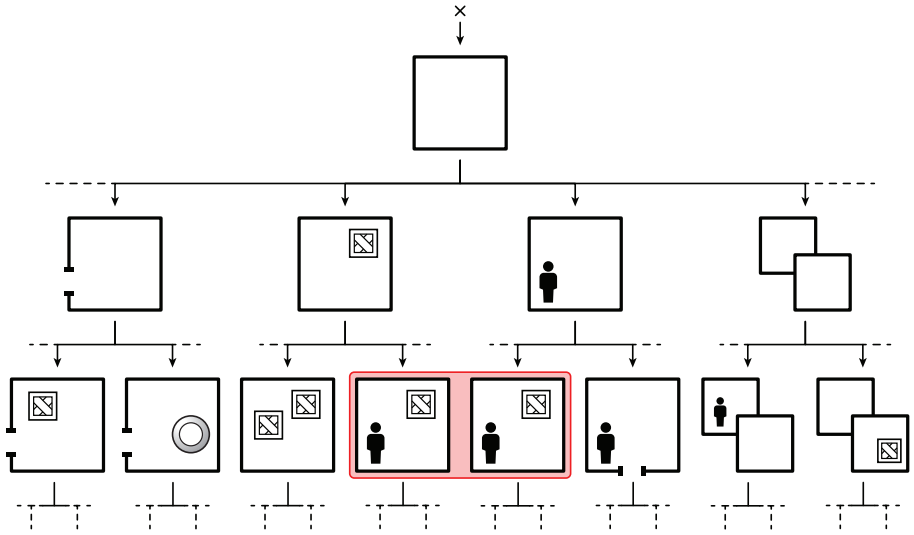


Abbildung 5.1: Hier ist die Erstellung eines Suchbaumes zu sehen. Es wird in jeder Generation ein Spielelement an einer zufälligen Position hinzugefügt. Wenn dieselben Schritte in einer unterschiedlichen Reihenfolge auftreten, kann es zu Duplikaten kommen. Bei der Expansion der Kindsknoten sollte dies also verhindert werden.

Abschnitt 5.1.1 beschriebenen Generierung der potentiellen Levels, ein Baum aufgebaut. Der Ablauf ist in Abbildung 5.2 durch ein vereinfachtes Beispiel dargestellt. Der Ausgangszustand des Levels wird durch Knoten 0 dargestellt. Von diesem ausgehend werden alle möglichen Zustände berechnet, die anhand der Spielregeln eintreten können. Dem Spieler stehen dabei Bewegungen in eine von vier Richtungen zur Verfügung. In diesem Beispiel ist es zusätzlich nicht möglich Kisten zu ziehen, sondern nur zu schieben. Es wird überpüft, ob der jeweilige Zug ausgeführt werden kann und wenn dies der Fall ist, wird ein neuer Knoten mit dem resultierenden Spielzustand erstellt. Die Züge werden dabei stehts in der selben Reihenfolge expandiert. Jeder Knoten unterläuft einer Prüfung, ob die Siegesbedingung erfüllt ist, was hier der Fall ist, wenn der Spieler den Raum verlassen hat. In Knoten 14 ist dieser Zustand erreicht und alle nachfolgenden Knoten müssen nicht mehr getestet werden, weswegen Nummer 15 ausgegraut dargestellt ist.

Da bewegliche Entitäten involviert sind, treten häufig wiederkehrende Spielsituationen auf. Diese müssen erkannt und verworfen werden, um redundante Berechnungen zu verhindern. Wenn die Suchtiefe des *Solvers* nicht limitert wird, kommt es ansonst zu einer Endlosschleife. Daher müssen alle errechneten Zustände für Vergleiche gespeichert werden. Dies führt zu massivem Speicherverbrauch bei großen Suchtiefen, weshalb die Zustände in möglichst kompakter Form abgespeichert werden und idealerweise einen

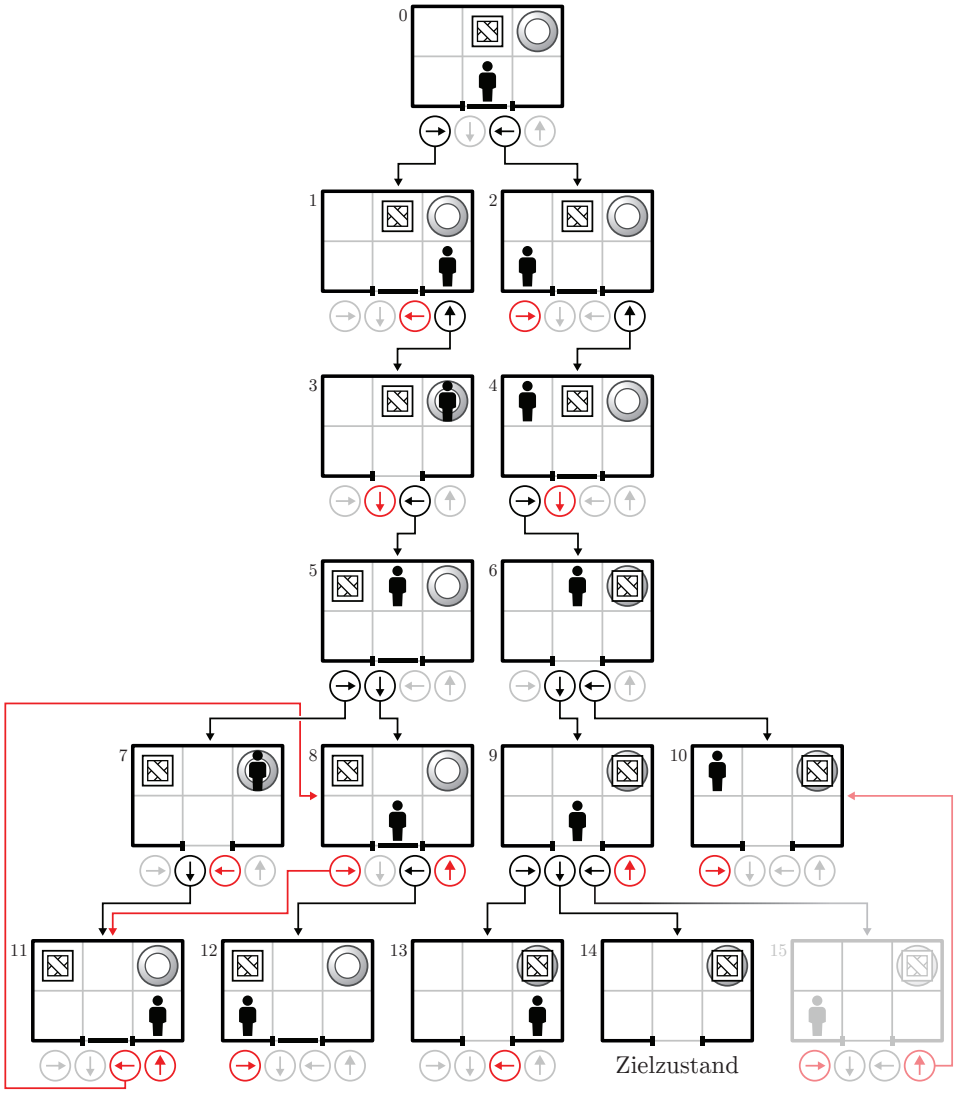


Abbildung 5.2: Visualisierung eines *Solvers*. Der Spieler soll mithilfe der Kiste den Schalter betätigen, um die Tür zu öffnen und das Level zu verlassen. Die eingekreisten Pfeile stellen die vier möglichen Züge des Spielers dar. Ausgegraute Züge sind im aktuellen Zustand nicht möglich, während rote Pfeile einen Zug kennzeichnen, durch den ein wiederkehrender Spielzustand auftritt (welcher der vorangegangene Knoten ist, falls nicht durch einen roten Pfeil anders angegeben).

Hashcode für schnelle Vergleiche generieren sollten. Am häufigsten treten Wiederholungen auf, wenn der Spieler das Gegenteil des vorherigen Zuges ausführt. Es liegt also nahe, diese resultierenden Zustände dieser Aktionen erst gar nicht zu expandieren. Jedoch ist in Abbildung 5.2 ebenso zu se-

hen, dass in Knoten 5 und 6 dieses Phänomen nicht auftritt: Der Spieler verschiebt dort durch seinen Zug auch eine Kiste, welche jedoch bei einer Bewegung in die Gegenrichtung auf ihrer Position verharrt. Gerade bei komplexeren Spielregeln als diesen ist es schwer vorauszusagen, ob eine Aktion die Vorangegangene rückgängig macht.

Durch den *Solver* können wahlweise nur der schnellste oder alle Lösungswege gefunden werden, je nachdem ob mittels Breiten- oder Tiefensuche vorgegangen wird. Da allerdings, selbst wenn mehrere Lösungswege vorhanden sind, der kürzeste Weg am meisten über die Schwierigkeit des getesteten Levels aussagt, ist der schnellste Weg grundsätzlich ausreichend, um die generierten Instanzen zu bewerten. Der *Solver* kann daher nicht nur die Lösbarkeit, sondern näherungsweise auch die Komplexität einzelner Level bewerten. Das ist vor allem nützlich, um zu einfache Rätsel sofort von Lösungen auszuschließen. Besonders in geringen Suchtiefen ist anzunehmen, das sehr viele triviale Lösungen existieren. Knoten, die sich also sehr nah an der Wurzel befinden, könnten also vom *Solver* übersprungen werden, der seine Tests erst ab einer gewissen Minimaltiefe ausführt. Diese Optimierung hat allerdings nur begrenzten Wert, da gerade die einfachen Level-Instanzen weniger Laufzeit beanspruchen. Da der *Solver* als Teil der *Brute-Force*-Suche eingesetzt werden soll und mit der Tiefe des primären Suchbaumes auch die Tiefe des *Solver*-Baumes wächst, steigt die Laufzeit zusätzlich exponentiell an. Das Ausführen des Ersteren alleine kann aber schon für relativ kleine Level sehr zeitintensiv sein.

Bei jenen Levels, von denen bekannt ist, dass sie eine Lösung besitzen, ist es bei der Suche von Vorteil eine Heuristik zu verwenden, die hilft, vielversprechende Knoten zuerst zu durchsuchen. Da aber bei der *Brute-Force*-Generierung meist keine Lösung zu erwarten ist, ergibt die Verwendung einer Heuristik hier keinen Sinn. Um bessere Laufzeiten zu erreichen, kann der *Solver* eine vereinfachte Levelstruktur verwenden, die nur die einzelnen Interaktionen der Spieler repräsentiert, anstatt jedes tatsächliche Feld zu überprüfen. Ein *Solver* kann jedoch auch für andere Ansätze verwendet werden, um sicherzustellen, dass deren Generationsprozess korrekte Ergebnisse liefert.

5.2 *Template-Stitching*

Template-Stitching bezeichnet das zufällige Aneinanderfügen von vordefinierten Teilstücken. Bei der Erstellung dieser muss darauf geachtet werden, dass die verwendeten Teile zusammenpassende Endstücke aufweisen um nahtlose Übergänge zu ermöglichen. Wenn die Ausmaße dieser Segmente sehr groß gewählt werden, ist es für den Entwickler einfacher bedeutungsvolle Spielelemente innerhalb dieser zu platzieren, die einen Bezug zueinander haben sollen. Umso kleiner die einzelnen Teilstücke jedoch sind, umso schwe-

rer wird es dem Spieler fallen diese nach wiederholten Spieldurchläufen als solche zu identifizieren. An ein als Start gekennzeichnetes Teilstück wird schrittweise ein zufälliger Teil nach dem anderen angeordnet. Sollte es hierbei zu Überlappungen kommen, wird das jeweilige Teil für diese Stelle als unpassend markiert und der Algorithmus springt einen oder wenn notwendig mehrere Schritte zurück. Über die Anzahl der Anknüpfungspunkte lässt sich die Linearität eines Levels gut kontrollieren. Mit nur einem Endstück entsteht eine Sackgasse, mit zwei Enden werden lineare Strukturen gebildet und größere Anzahlen verursachen Verzweigungen. Die *Templates* können verschiedenen Gruppen zugewiesen werden, um größere Kontrollierbarkeit zu ermöglichen. Dadurch kann die minimale und maximale Menge an Teilen einzelner Kategorien gesteuert werden.

Diese Methode ist nur bedingt zur Rätselgenerierung geeignet. Tayler u. a. [17] verwenden diese Methode jedoch z. B. als Teilaspekt in der Generierung von *Sokoban*-Levels dar. Durch die Aneinanderreihung vordefinierter Blöcke aus 3 mal 3 Spielfeldern wird garantiert, dass keine unerreichbaren oder zu engen Gänge entstehen, durch die keine Kisten geschoben werden können. Für sämtliche Blöcke sind bis zu 4 zusätzliche umgebende Felder definiert, welche deren Anknüpfpunkte repräsentieren. Diese *Templates* können auch gedreht und gespiegelt werden, um ein Spielfeld zu bilden. Die tatsächliche Positionierung der Kisten und des Spieleravatars passiert in einem separaten Schritt, wo sie zufällig platziert werden und sich dann im Zielzustand des Levels befinden, von dem aus rückwärts Spieleraktionen generiert werden, bis eine gewisse Länge erreicht ist.

Template-Stitching wird hauptsächlich dafür verwendet räumliche Anordnungen zu generieren. Die Anwendung dieser Technik in *Vancouver Maneuver* wird aber durch die Spielmechanik verhindert, die es erlaubt einzelne Räume zu drehen und anders zu verbinden. Es ist daher nicht garantiert, dass der Spieler die einzelnen *Templates* in der Reihenfolge betritt, die vom Algorithmus dafür vorgesehen wurde. Aus diesem Grund wird dieser Ansatz als ungeeignet eingestuft.

5.3 Generative Grammatiken

Generative Grammatiken werden ebenfalls als Ersetzungssysteme bezeichnet und stammen ursprünglich aus der Linguistik [2]. Damit werden Systeme beschrieben, welche mittels eines Sets von Regeln syntaktisch korrekte Sätze einer Sprache generieren können. Da diese Regeln meist Rekursionen enthalten, kann im Normalfall eine unbegrenzt hohe Anzahl an Sätzen gebildet werden.

5.3.1 Terminologie

Eine generative Grammatik besteht aus einem Alphabet, einer oder mehreren Transformationsregeln und einem Startsymbol, welches auch als Axiom bezeichnet wird. Das Alphabet setzt sich aus beliebig vielen Symbolen zusammen, wobei zwischen terminalen Konstanten und non-terminalen Variablen unterschieden wird. Die Transformationsregeln beschreiben wodurch einzelne Symbole ersetzt werden können. Sie bestehen aus einer linken und einer rechten Seite, welche den Ausgangs- bzw. den Zielzustand darstellen. Ein terminales Symbol zeichnet sich dadurch aus, dass es bei keiner Regel im Set auf der linken Seite steht. Umgekehrt gilt für non-terminale Symbole, dass eine entsprechende Regel existieren muss. Die Regeln können solang angewandt werden, bis nur noch terminale Symbole vorhanden sind oder nach einer vordefinierten Anzahl an Iterationen abgebrochen wird. Generative Grammatiken haben einige wichtige Eigenschaften anhand derer sie kategorisiert werden:

Kontextfrei/kontextsensitiv

Bei kontextfreien Grammatiken steht auf der linken Seite aller Regeln jeweils nur ein Symbol. Kontextsensitive Regeln hingegen beinhalten Informationen über die Umgebung des Symbols, so muss im folgenden Beispiel etwa A von B und C umschlossen sein, damit eine Ersetzung mit AB stattfinden darf. Diese beiden Schreibweisen repräsentieren dieselbe Regel:

$$B < A > C \rightarrow AB,$$
$$BAC \rightarrow BABC.$$

Propagativ/non-propagativ

Eine Grammatik gilt als propagativ, wenn auf der rechten Seite der Regeln mehr Symbole stehen als auf der linken. Dadurch wächst die Anzahl der Symbole bei jeder neuen Regelanwendung an. Bei non-propagativen Grammatiken haben Regeln also auf beiden Seiten die gleiche Anzahl an Symbolen. Dadurch bleiben die Ergebnisse stehts gleich lang wie das Axiom.

Deterministisch/stochastisch

Bei deterministischen Grammatiken existiert im Regelsatz jeweils nur eine Regel mit der gleichen linken Seite. Das bedeutet, dass jede Regel zum gleichen Ergebnis führt, egal wie oft sie angewandt wird. Stochastische Regelsätze weisen hingegen mehrere Regeln mit der gleichen linken Seite auf, von denen eine zufällig angewandt wird. Diese können mit Wahrscheinlichkeiten versehen werden, um die Häufigkeit der Anwendung zu kontrollieren:

$$A(0.75) \rightarrow AB,$$
$$A(0.25) \rightarrow A.$$

Parallel/sequenziell

Die Regelanwendung kann entweder parallel oder sequenziell ablaufen. Bei der parallelen Ausführung wird über alle Symbole iteriert und es werden so viele Regeln wie möglich angewandt. Bei der sequenziellen Ausführung hingegen kommt in jedem Generierungsschritt immer nur eine Regel auf einmal zum Einsatz.

Parametrisch

Einzelne Symbole können mit Parametern versehen werden. Im diesem Beispiel werden A die Werte 0 und 1 zugewiesen:

$$A(0, 1).$$

Transformationsregeln können darauf basierende Bedingungen enthalten und diese Parameter gegebenenfalls modifizieren. Durch Bezeichner kann auf beiden Seiten der Regel Bezug auf einzelne Parameter genommen werden:

$$A(x, y) : x == 0 \rightarrow A(1, y + 5)B(2, 3).$$

5.3.2 *String*-Grammatiken

Die einfachste Form einer generativen Grammatik ist eine sogenannte *String*-Grammatik. Damit wird eine einzelne Zeichenkette manipuliert. In einer Zeichenkette können nicht nur lineare Strukturen, sondern etwa mittels Klammerung Baumstrukturen geschaffen werden, welche vor allem bei L-Systemen Verwendung finden (siehe Abschnitt 5.3.3).

Spielelemente können durch Grammatiksymbole abgebildet werden, während Transformationsregeln die Zusammenhänge dieser Elemente beschreiben können. Dormans u. a. [3] analysieren z. B. den Aufbau des Wald-Tempels aus *The Legend of Zelda: Twilight Princess* (siehe Abschnitt 3.3.4) und leiten generelle Regeln her mit denen derartige Missionen generiert werden können. Unter anderem beschreiben sie wie dies mittels *String*-Grammatiken geschehen könnte. Die folgenden Regeln sind diesen Arbeiten entnommen und sollen illustrieren worauf beim Design einer generativen Grammatik zu achten ist. Diese enthalten bereits grundlegende Logik, wie z. B. dass ein Schloss hinter seinem zugehörigen Schlüssel platziert werden muss:

$$\begin{aligned} \mathit{Dungeon} &\rightarrow \mathit{Hindernis} + \mathit{schatz}, \\ \mathit{Hindernis} &\rightarrow \mathit{schlüssel} + \mathit{Hindernis} + \mathit{schloss} + \mathit{Hindernis}, \\ \mathit{Hindernis} &\rightarrow \mathit{monster} + \mathit{Hindernis}, \\ \mathit{Hindernis} &\rightarrow \mathit{raum}. \end{aligned}$$

Obwohl mit diesem simplen Regelsatz syntaktisch korrekte Levels generiert werden können, sind nicht alle Ergebnisse für den Einsatz in einem Spiel

geeignet. Das Ergebnis $raum + schatz$ wäre mit dieser Grammatik möglich und als tatsächliches Level viel zu kurz. Bessere Regeln implizieren z. B. eine Mindestlänge bzw. eine im Leveldesign typische Drei-Akt-Struktur:

$$\begin{aligned} Dungeon \rightarrow\ & Hindernis + Hindernis + Hindernis + \\ & Hindernis + schatz, \\ Dungeon \rightarrow\ & Wächter + Hindernis + Miniboss + belohnung + \\ & Hindernis + Levelboss + schatz. \end{aligned}$$

Dies zeigt wie wichtig ein guter Regelsatz ist, um zufriedenstellende Ergebnisse zu erhalten. Wenn neue Regeln hinzugefügt werden, muss auch darauf geachtet werden, dass logische Zusammenhänge alter Regeln nicht zerstört werden.

5.3.3 L-Systeme

L-Systeme sind parallele Ersetzungssysteme. Sie sind nach Aristid Lindenmayer benannt, von dem sie entworfen wurden, um den Wachstumsprozess von Algen zu modellieren. L-Systeme werden häufig im Bereich der Pflanzengenerierung eingesetzt, wo ihre Eigenschaft der Selbstähnlichkeit ausgenutzt wird. In der einfachsten Form wird dies mittels *Turtle Graphics* realisiert. Dabei werden die einzelnen Symbole einer *String*-Grammatik als Zeichenoperationen interpretiert, die relativ zu einem Cursor, der sogenannten *Turtle*, ausgeführt werden [12]. Weiters werden L-Systeme bei der Generierung von Stadtmodellen eingesetzt [13] und auch zur generativen Musik-Komposition werden sie herangezogen [10].

5.3.4 Graph-Grammatiken

Graph-Grammatiken funktionieren analog zu den *String*-Grammatiken. Die durch Kanten verbundenen Knoten eines Graphen stellen dabei die Symbole des Alphabets dar. Folglich bestehen die Produktionsregeln aus kleinen Teilgraphen anstatt aus Zeichenketten. Die Anwendung einer Produktionsregel ist in Abbildung 5.3 zu sehen. Die Knoten werden dabei mit numerischen Bezeichnern versehen, um sie auf beiden Seiten eindeutig zu identifizieren. Der Graph wird zu Beginn nach einer Knotenkonfiguration durchsucht, welche der linken Seite der Regel entspricht. Ist eine solche vorhanden, kann die Regel angewandt werden und alle Kanten auf der linken Seite werden aus dem Graph entfernt. Knoten, deren Bezeichner nur auf der rechten Seite aufscheint, werden neu erstellt. Ebenso werden jene gelöscht, deren Bezeichner nur auf der linken Seite vermerkt ist. Im letzten Schritt werden alle Kanten die auf der rechten Seite zu sehen sind erstellt und mit den entsprechenden Knoten verknüpft. Kanten können zusätzliche Eigenschaften aufweisen. Sie können eine Gewichtung aufweisen, gerichtet oder ungerichtet sein und einen beliebigen Typ zugewiesen bekommen. Produktionsregeln können auch

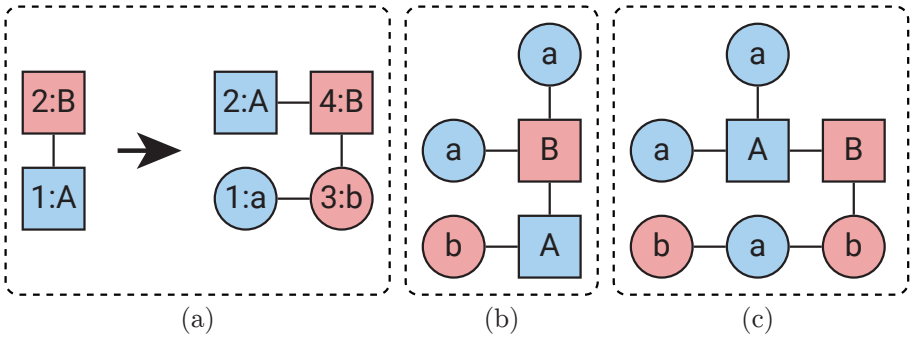


Abbildung 5.3: Die Transformation eines Graphen [3]: (a) Beispiel einer Produktionsregel, (b) Graph vor der Anwendung der Regel, (c) Graph nach der Anwendung der Regel.

negative Anwendungsbedingungen enthalten. Das heißt, sie können auf der linken Seite speziell gekennzeichnete Kanten enthalten, die nicht in der Konfiguration vorhanden sein dürfen.

Dormans und Bakkes [3] nutzen Graph-Grammatiken in einem zweistufigen Prozess, um Levels für Adventure-Games zu erzeugen. Sie generieren zuerst eine Mission und wandeln diese anschließend in räumliche Strukturen um. Abbildung 5.4 zeigt die Generierung einer Missionsstruktur, welcher das Schlüssel-Schloss-Prinzip zugrunde liegt. Die generierten Missionen beschränken sich darauf, dass der Spieler vom Start ins Ziel gelangen muss und auf dem Weg dorthin gewisse Aufgaben erledigen muss. Es wird mit einem linearen Aufbau von acht *Task*-Knoten T begonnen, die von einem Start-Knoten S und einem Ziel-Knoten G umgeben sind. Hierbei ist zu beachten, dass die Knoten mit gerichteten Kanten verbunden sind, die von Start in Richtung Ziel zeigen. Das bedeutet aber nicht, dass sich der Spieler nur in diese Richtung bewegen kann, sondern beschreibt bloß zu welchen Bereichen er zuerst Zugang erlangt. Um den linearen Aufbau aufzubrechen wird Regel (b) angewandt, welche einen *Task*-Knoten aus der linearen Abfolge entfernt und seitlich wieder anknüpft. Mittels Regel (c) werden Schlüssel und Schlösser eingebaut. Die gerichteten Kanten garantieren, dass der Schlüssel nicht hinter einem Schloss platziert wird und somit für den Spieler erreichbar ist. Die grünen Pfeile beschreiben hierbei keine örtlichen Zusammenhänge, sondern welche Schlüssel-Schloss-Paare zusammengehören. Da Schlösser jedoch nutzlos sind, wenn deren Schlüssel direkt vor ihnen positioniert ist, müssen sie noch auseinander bewegt werden. Hierfür sind die beiden Regeln (f) und (g) zuständig, die einerseits ein Schloss nach vorne und anderseits einen Schlüssel nach hinten bewegen. Alleine durch diese vier relativ simplen Regeln lassen sich also bereits einfache aber korrekte Missionsstrukturen erzeugen. Dormans u. a. [3] fahren mit der Anwendung einer *Shape*-Grammatik fort, um die Räumlichkeiten des Levels zu generieren. Diese enthält Regeln,

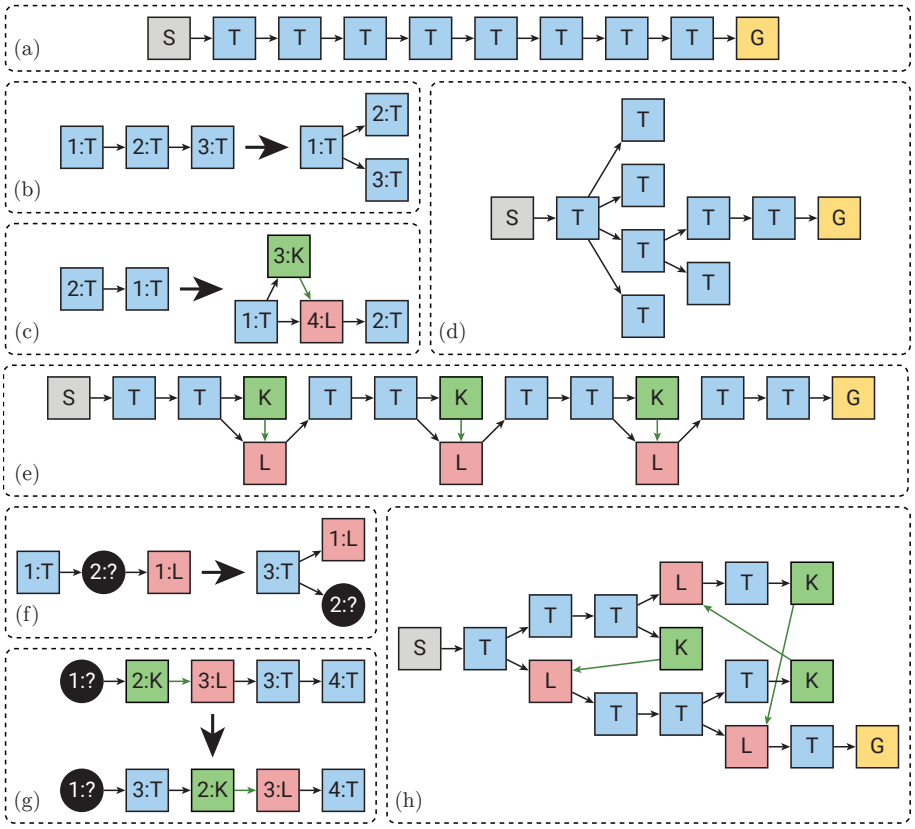


Abbildung 5.4: Missionsgenerierung für einen *Dungeon* eines *Adventure*-Spiels nach Dormans und Bakkes [3]: (a) stellt die Ausgangssituation dar, (b) ist eine Regel, um eine Verzweigung herzustellen, (c) fügt jeweils einen Schlüssel und ein Schloss hinzu, (d) zeigt den Graph nach vierfacher Anwendung von (b), (e) zeigt den Graph nach dreifacher Anwendung von Regel (c), (f) ist eine Regel, um ein Schloss nach vorne zu bewegen, (g) verschiebt einen Schlüssel nach hinten, (h) zeigt die Endsituation nach mehrfacher Anwendung der Regeln (b), (c), (f) und (g).

welche die Missionsstruktur in räumliche Formen umwandelt.

Van der Linden u. a. [9] verfolgen mit dem eigens entworfenen Editor *Entika* einen ähnlichen Ansatz, welcher jedoch als *Gameplay-Grammar* betitelt wurde. Designer können mit diesem Editor Spieleraktionen, Objekte deren Interaktionen definieren. Daraus wird eine Sequenz von Aktionen generiert (siehe Abbildung 5.5). Diese kann auch mehrere Pfade enthalten und Aktionen können in Sub-Aktionen unterteilt werden, um mehr Komplexität zu schaffen. Anschließend werden zuvor definierte Entitäten der Sequenz entsprechend in der Spielwelt platziert.

Graph-Grammatiken bieten sich an, um die Beziehungen der Spielelemente von *Vancouver Maneuver* nachzumodellieren, ähnlich wie es hier mit

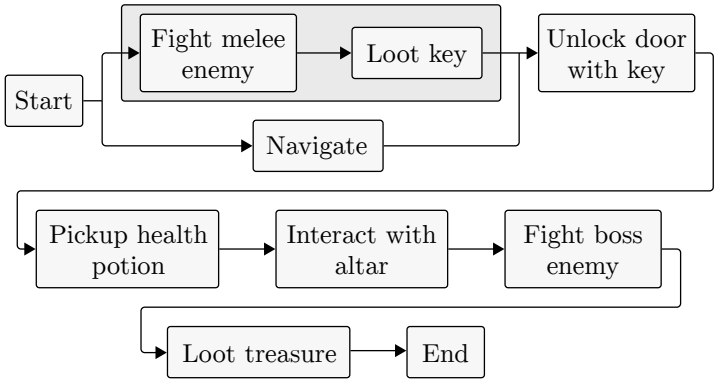


Abbildung 5.5: Eine mittels *Entika* unter Verwendung eines *Gameplay-Grammars* generierte Sequenz von Spieleraktionen [9].

dem Schlüssel-Schloss-Prinzip vorgezeigt wurde. Die einzelnen Elemente die sich gegenseitig aktivieren, können auch zuerst gemeinsam in einem Raum platziert werden, um anschließend mittels, der Spiellogik angepassten, Regeln in andere Räume verschoben zu werden. Im eben beschriebenen Generierungsprozess waren jedoch, bis auf den Spieler, keine beweglichen Objekte involviert. Der Spieler wurde sogar nicht einmal als Element im Graph abgebildet. Es ist daher nicht sichergestellt, wie gut der Ansatz dazu geeignet ist, bewegliche Objekte abzubilden. Bei der Erstellung der Transformationsregeln muss vor allem darauf geachtet werden.

5.4 Genetische Algorithmen

Genetische Algorithmen gehören zur Gruppe der suchbasierten Generierungsmethoden. Nach dem Vorbild der natürlichen Evolution wird durch Reproduktion, Mutation, Rekombination und Selektion versucht die optimale Lösung eines Problems zu finden. Zu Beginn wird eine Population an Lösungskandidaten erstellt. Die Population besteht aus zufälligen Instanzen, die gleichmäßig über den kompletten Lösungsraum verteilt sein sollten. Besteht eine Vermutung, an welchem Punkt sich die optimale Lösung in diesem Raum ungefähr befindet, so kann die Startpopulation kontrolliert in dessen Nähe generiert werden, um den Suchprozess zu beschleunigen.

Eine Fitness-Funktion, die spezifisch an das jeweilige Problem angepasst werden muss, bewertet die einzelnen Kandidaten. Jene mit einer schlechten Bewertung werden aus der Population entfernt, während jene mit guter Beurteilung zur Reproduktion herangezogen werden. Bei sehr großen Populationen können Kandidaten auch nur stichproben-artig evaluiert und gekreuzt werden. Durch Rekombination einzelner Teile von guten Kandidaten, erben die neuen Generationen Eigenschaften der ursprünglichen Instanzen.

Mutationen sind dafür verantwortlich neue zufällige Eigenschaften einzuführen und verhindern, dass auf lokale Maxima im Lösungsraum zugesteuert wird. Nachdem ausreichend Kandidaten mit guter Bewertung gekreuzt und mutiert wurden, besteht eine hohe Wahrscheinlichkeit, dass sich in der neuen Population solche mit noch besseren Fitness-Werten befinden. Dieser Ablauf wird nun so oft wiederholt bis eine Abbruchbedingung erfüllt ist. Das kann z. B. eine gewisse Zahl an Iterationen sein oder das Finden einer Lösung die eine zufriedenstellende Bewertung aufweist.

Genetische Algorithmen finden unter anderem in der Generierung von Rätseln Anwendung. Unter anderem wurden evolutionäre Algorithmen von Ashlock [1] und LeBaron u. a. [8] dazu genutzt, um Labyrinthe zu generieren, welche gewissen Rätsel-Logiken gerecht werden müssen. Die Funktionsweise dieser Rätsel ist jedoch leider zu unterschiedlich, um deren Methodik für *Vancouver Maneuver* ohne Weiteres zu adaptieren. Es ist außerdem sehr schwer dafür eine gute Fitness-Funktion zu erstellen. Aufgrund dieser Tatsachen sind Erfolgsaussichten mit der dieser Methodik eher gering.

5.5 *Constraint-Propagation*

Eine weitere Technik, die für Rätselspiele interessant sein kann, ist *Contraint-Programming*. Ein *Constraint-Satisfation-Problem* wird als Triplett (X, D, C) beschrieben. Es besteht dabei aus einer begrenzten Menge an Variablen X. Für jede Variable $x \in X$ existiert eine begrenzte Menge an zuweisbaren Werten $D(x)$. C ist die Menge der Bedingungen, denen die Werte den Variablen entsprechen müssen. Die Lösung des Problems ist dann gefunden, wenn eine Belegung der Variablen in X erfolgt ist, die alle Bedingungen in C erfüllen. Hierzu wird die *Constraint-Propagation* genutzt, bei der aus bestehenden Bedingungen neue generiert werden, welche verwendet werden, um die Wertebereiche der einzelnen Variablen einschränken.

Jefferson u. a. [6] entwickelten das Spiel *Combination* für die Erforschung der Levelgenerierung mittels *Constraint-Propagation*. Es ist eine Modifikation des bekannten Damenproblems, bei dem es gilt acht Damen auf einem Schachbrett so zu positionieren, dass sie einander nicht schlagen können. In derselben Weise müssen in *Combination* Spielsteine, die dem Läufer, dem Turm und der Dame eines Schachspiels entsprechen, auf Feldern platziert werden. Die Steine haben verschiedene Farben und nur gleichfarbige dürfen sich gegenseitig attackieren. Systeme dieser Art bieten sich an, als *Constraint-Satisfation-Problem* modelliert zu werden. Durch zusätzliche Wände, die Strahlen blockieren können, und Löcher, in denen keine Steine platziert werden dürfen, konnte jedoch nicht dieselbe Strategie wie für das Damenproblem verwendet werden. *Constraint-Propagation* wird in diesem Fall auf zweierlei Arten genutzt: Einerseits werden die einzelnen Levels damit generiert und auf Korrektheit überprüft, andererseits wird die Laufzeit

des Algorithmus als Maß für die Schwierigkeit des Levels herangezogen. Dazu wird dieser mehrere Male mit zufälligen Start-Einstellungen ausgeführt und die durchschnittliche Laufzeit berechnet, um zu verhindern, dass die Lösung durch Zufall sofort gefunden wird. Der Ansatz ist allerdings besser geeignet für Rätsel in denen keine beweglichen Objekte bzw. keine zeitlichen Zustandsveränderungen vorkommen. Die Lösungskonfiguration des Rätsels ist statisch und jeder Teil hat nur einen einzigen richtigen Ort, an dem er platziert werden muss. Im Gegensatz dazu kann eine Kiste in *Vancouver Maneuver* zu verschiedenen Zeitpunkten verschiedene Positionen einnehmen. Daher kann dieser Ansatz nicht übernommen werden.

5.6 *Answer-Set-Programming*

Smith u. a. [16] schlagen *Answer-Set-Programming* als guten Ansatz für prozedurale Inhaltsgenerierung vor. Dabei handelt es sich um eine Form der deklarativen Programmierung, die auf besonders schwierige Suchprobleme ausgelegt ist. Dabei beschreibt der Entwickler nur was zu berechnen ist, jedoch nicht wie es zu berechnen ist. Ein großer Nachteil dieser Methode ist, dass die Laufzeit eines solchen Algorithmus länger als jene eines problemspezifischen Ansatzes ist. *Answer-Set-Programming* opfert Performanz im Gegenzug für erhöhte Expressivität und verminderte Komplexität. Mit steigender verfügbarer Rechenkraft wird diese Methode vermutlich an Relevanz gewinnen. Da Geschwindigkeit bei der Wahl des Ansatzes jedoch eine große Rolle spielt, wird diese Methode verworfen.

Kapitel 6

Eigener Ansatz

Keiner der in Kapitel 5 beschriebenen Ansätze ist ohne weitere Modifikationen auf *Vancouver Maneuver* anwendbar. Daher wurde ein Hybridansatz entwickelt, der Teilaspekte dieser Methoden mit eigenen Ideen kombiniert. Wie bei den Graph-Grammatiken (siehe Abschnitt 5.3.4) wird immer von der selben minimalen Startstruktur ausgegangen. Durch die Anwendung von korrekt gestalteten Transformationsregeln nimmt die Komplexität der ursprünglichen Struktur zu, während deren Lösbarkeit in jedem Stadium garantiert ist. Die einzelnen Regeln entsprechen dabei rückwärts ausgeführten Spieleraktionen. Die Spielregeln von *Vancouver Maneuver* verhindern das Erstellen von Transformationsregeln, die sicherstellen, dass durch das Hinzufügen eines neuen Spielelementes der Lösungsweg verlängert wird. Der Grund dafür ist, dass beinahe jedes Objekt, sogar jeder Raum, mehrmals verwendet werden soll. Dadurch kann es vorkommen, dass später hinzugefügte Elemente zuvor generierte Rätsel sehr stark vereinfachen. Um diesem Umstand entgegenzuwirken, wird nach jeder Regelanwendung überprüft, ob sich der Lösungsweg im Vergleich zum vorhergehenden Zustand verlängert hat. Ist dies nicht der Fall, wird die zuletzt angewandte Regel rückgängig gemacht. Der Ansatz wurde daher als *Backtracking*-Algorithmus entworfen. Der Benutzer liefert Eingabedaten in der Form von Minimal- und Maximalmengen der jeweiligen Spielelemente und der gewünschten Länge des Lösungsweges. Nach jeder Transformation wird überprüft, ob eine Spielsituation erreicht wurde, die diesen Daten entspricht. Diese wird in einem zweiten Generierungsschritt in die tatsächliche Levelgeometrie umgewandelt und den Spielelementen werden konkrete Spielfelder zugeteilt.

6.1 Generierung einer Levelstruktur

Die Ausgangssituation wird durch das minimale Level definiert, welches die Siegesbedingung des Spiels erfüllt. Im Fall von *Vancouver Maneuver* ist dies ein geöffneter Tresor. Die erste Generierungsaktion ist daher dafür verant-

Abbildung 6.1: Das einfachste lösbare Level. Der Spieler P ist bereits im Besitz des Schlüssels K und steht direkt im selben Bereich wie der Tresor S.

wortlich einen Tresor zu erstellen. Diese zieht einige weitere Aktionen nach sich: Der Tresor benötigt einen Raum, in dem er platziert werden kann. Da noch kein Raum existiert, zieht dies wiederum nach sich, einen zu erstellen und weiters den Tresor darin zu platzieren.

Die letzte Spieleraktion eines jeden Spieldurchlaufs ist stets das Öffnen des Tresors. Daum wird ebendiese Aktion rückwärts ausgeführt und es wird versucht den Tresor zu schließen. Dazu werden zwei zusätzliche Elemente benötigt: Ein Spieler, der vom Tresor aus erreichbar ist und ein Schlüssel, den der Spieler in sein Inventar aufnehmen kann. Weil noch kein Spieler existiert, muss auch dieser erstellt werden und direkt neben dem Tresor platziert werden. Die einzelnen Generierungsschritte ziehen also notwendigerweise das Platzieren und Modifizieren mehrerer Objekte auf einmal nach sich, um die Lösbarkeit nach jedem Schritt zu garantieren. Nun befindet sich das Level in dem Zustand, der in Abbildung 6.1 dargestellt ist.

Als nächster Schritt wird versucht, die Abstände zusammenhängender Objekte zu erhöhen. Zu Beginn besteht nur eine Beziehung zwischen dem Tresor S und dem Schlüssel K, die eine Distanz von 0 aufweisen. Jedoch können weder Tresor noch Schlüssel direkt verschoben werden. Der Schlüssel befindet sich zurzeit im Inventar des Spielers, daher muss stellvertretend der Spieler bewegt werden. Es wird errechnet, welche Bereiche eine höhere Distanz zum Tresor aufweisen. Da momentan nur ein Raum existiert, muss ein weiterer erstellt werden, in den der Spieler verschoben werden kann. Das Erstellen eines Raumes zieht eine weitere Aktion nach sich, nämlich zugehörige Türen zu erstellen. In Abbildung 6.2 wird gezeigt, dass sich die Distanz $d(S, K)$ zwischen Tresor S und Schlüssel K in der Tat auf 1 erhöht. Den Spieler trennt nun bereits die Aktion, durch die Tür zu gehen, vom Ziel. Wird die exakt gleiche Regel erneut angewandt, ergeben sich jedoch verschiedene Türkombinationen, von denen eine zufällige ausprobiert wird. Danach muss erneut die Distanz zwischen den beiden Objekten gemessen werden, um zu verifizieren, dass die Generierungsaktion auch wirklich die Komplexität des Levels erhöht hat.

Da dies noch weit entfernt von einem herausfordernden Rätsel ist, könnte der nächste Generierungschritt sein, die Tür zu verschließen und ein Element zu platzieren, welches diese öffnet. Das könnten dann ein oder mehrere Schalter sein, die gedrückt gehalten werden müssen. Da der Spieler aus die-

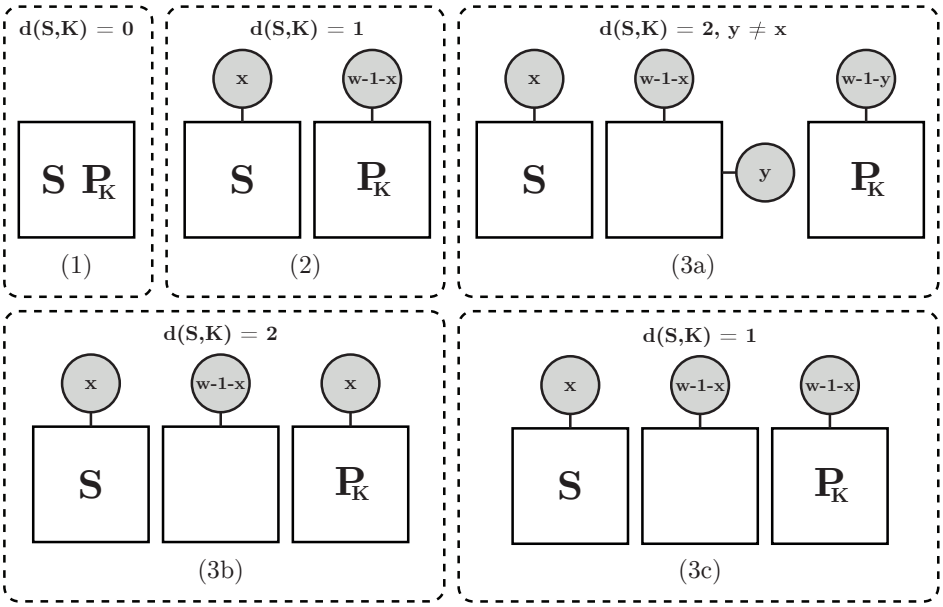


Abbildung 6.2: Das Hinzufügen eines neuen Raumes zieht stets die Erstellung von mindestens einer Tür nach sich. Diese kann an einer von 5 Stellen positioniert werden (dargestellt durch die hellgrauen Knoten). Für jede Tür an einer Stelle x muss eine gegenüberliegende Tür im Spielbrett vorhanden sein. (2) stellt den Zustand nach (1) dar, während (3a), (3b) und (3c) verschiedene mögliche Zustände nach (2) darstellen. Zustand (3c) wird zwar ausprobiert aber anschließend verworfen, weil sich durch diesen der Lösungsweg nicht erhöht hat.

sem Grund also nicht gleichzeitig durch die Tür gehen und auf dem Schalter stehen kann, muss weiters eine Kiste dort platziert werden. Alternativ könnte auch der zweite Spieler an dieser Stelle ins Spiel gebracht werden. Das Objekt, welches am Schalter steht, muss auch noch davon wegbewegt werden. Es wird entweder einer der bestehenden Räume dafür genutzt oder wieder ein neuer erstellt. Erst dann gilt die Aktion als beendet, da sonst keine Verlängerung des Lösungsweges stattgefunden hat und der Generator die Aktion rückgängig macht. Da die Kiste nicht durch jene Tür verschoben werden kann, die sie mittels Schalter öffnen soll, muss in diesem Fall ein neuer erstellt werden. Es kann aber durchaus passieren, dass die Kiste nach mehreren Generierungsschritten auf Umwegen im Raum hinter der verschlossenen Tür angelangt.

In ähnlicher Weise kann ein Objekt wie der Tresor, der Schlüssel oder eine Kiste in eine Kamerazone verschoben werden, um sie für den Spieler unerreichbar zu machen. Die nachfolgenden Aktionen betreffend der Erstellung des Schalters laufen identisch zu denen der Türen ab und sind in Abbildung 6.3 dargestellt.

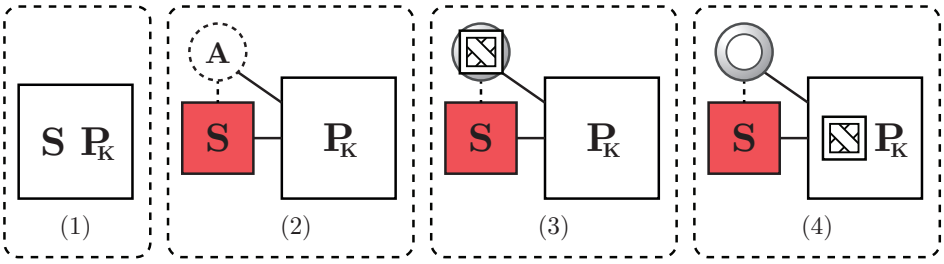


Abbildung 6.3: Das Hinzufügen einer Kamerazone erfordert immer die Erstellung eines Aktivator-Platzhalters, wie in (2) zu sehen ist. Dieser muss in (3) z. B. durch einen Schalter ausgetauscht werden, auf welchem entweder ein Spieler oder eine Kiste platziert wird. In (4) wird die Distanz des neu erstellten Objektes zum zugehörigen Schalter erhöht.

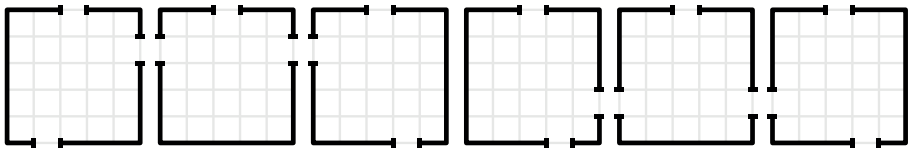


Abbildung 6.4: Für einen Raum mit Türen an den Positionen 2, 3 und 1 gibt es 6 verschiedene mögliche Platzierungen. Die erste Tür wird stets oben platziert, da die Drehung des Raumes später ohnehin zufällig gewählt wird und nur die Lage der einzelen Türen zueinander eine Rolle spielt.

Da die Wiederverwendung von Objekten die Komplexität des Levels erhöht und daher anzustreben ist, wird beim Platzieren des nächsten Schalters zunächst überprüft, ob die bereits existierende Kiste in Reichweite ist. Wenn dies der Fall ist, wird keine neue erstellt, sondern diese mit dem Schalter verknüpft. Falls mehrere Kandidaten zur Verknüpfung existieren, kann für jeden errechnet werden wie viele zusätzliche Lösungsschritte dessen Benutzung bedeuten würde und je nach gewünschter Level-Länge kann dann für einen entschieden werden.

6.2 Umwandlung in ein konkretes Spielfeld

Nachdem die abstrahierte Levelstruktur erstellt wurde, muss dieses in konkrete Spielfelder umgewandelt werden, um das Level auch tatsächlich spielbar zu machen. Diese Umwandlung erfolgt für jeden einzelnen Raum unabhängig und beginnt mit der Platzierung der Türen, die mit ebendiesem Raum verbunden sind (siehe Abbildung 6.4). Anschließend werden die Bereiche, welche sich innerhalb des Raumes befinden, platziert. Die zuvor angewandten Simplifizierungen müssen hierbei berücksichtigt werden. In den generierten Spielbrettern ist es z. B. unmöglich, dass eine Kamerazone di-

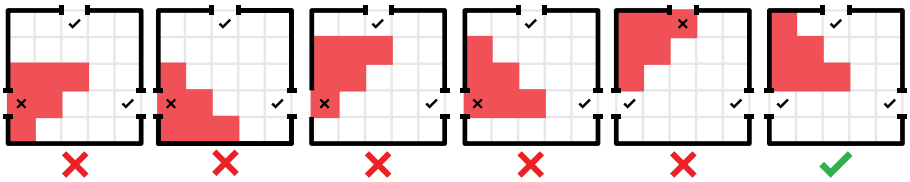


Abbildung 6.5: Es gibt 24 mögliche Positionen und Ausrichtungen einer Kamera in einem Raum mit einer Breite von 5. Es werden alle ausprobiert, bis eine erreicht ist, bei der kein Feld vor einer Tür blockiert ist. Wird keine gefunden, muss Konfiguration der Türen neu vorgenommen werden und danach dieser Schritt wiederholt werden.

rekt an eine Tür angrenzt. Daher darf jene auf dem konkreten Spielfeld auch nie so platziert werden (siehe Abbildung 6.5). Die Tür wäre ansonsten indirekt verschlossen, was in der abstrahierten Struktur jedoch nicht vermerkt is. Das Level könnte dadurch möglicherweise unlösbar werden. Weiters werden die Schalter positioniert, für die selbiges gilt wie für die Kamerazonen, weswegen sie nicht vor einer Tür platziert werden dürfen. Zuletzt werden die beweglichen Objekte auf ein zufälliges Spielfeld ihres zugeordneten Bereiches gestellt. Es besteht die Möglichkeit, dass nicht alle Objekte ohne Kollisionen platziert werden können, in welchem Fall der Generator erneut bei der Erstellung der Missionsstruktur beginnen muss. Andernfalls ist die Generierung eines Levels damit abgeschlossen.

Kapitel 7

Implementierung

7.1 Verwendete Tools und Bibliotheken

Vancouver Maneuver wurde aufgrund großer Erfahrungswerte mittels C# in der *Unity-Engine* [38] umgesetzt. Für das *Augmented-Reality-Tracking* wurde die ursprünglich von *Qualcomm* entwickelte *Vuforia*-Bibliothek [29] verwendet. Die Netzwerkkommunikation wurde mithilfe der *Unity*-eigenen *Unet*-Bibliothek verwirklicht. Da jedoch für die Generierung der Levels weder der *Augmented-Reality-* noch der Netzwerk-Aspekt der Applikation von Bedeutung war, wurde die Grundlogik in einem neuen *Unity*-Projekt ohne die Verwendung jener Bibliotheken umgesetzt. Dadurch konnten gewisse Programmteile einfach übernommen werden, während vor allem die Abwesenheit der Netzwerk-Logik den Arbeitsablauf erleichterte.

7.2 Programmarchitektur

In diesem Projekt koexistieren mehrere Versionen der gleichen Klassen in verschiedenen *Namespaces*. Der Grund dafür ist, dass der Generator vereinfachte Formen dieser Klassen benötigt, um in annehmbarer Zeit terminieren zu können. Diese werden dann in einem zweiten Schritt in Komponenten für *Unity* umgewandelt, um diese zu visualisieren und spielbar zu machen. Im Folgenden werden die wichtigsten Klassen des Generators näher beschrieben.

`IActionGenerator`

Objekte, die das Interface `IActionGenerator` implementieren, stellen eine Methode zur Verfügung, welche alle Aktionen ermittelt, die derzeit mit dem betreffenden Objekt möglich sind. Der Generator verwaltet eine Liste aller existierenden `IActionGenerator`-Objekte. Zu Beginn jedes Generierungsschrittes iteriert er über jene Liste und sammelt alle verfügbaren Aktionen, von denen er eine zufällige auswählt, ausführt und aus der Liste löscht.

`GenerationAction`

Da der Generator nach dem *Backtracking*-Verfahren vorgeht, muss jeder Generierungsschritt reversibel sein. Alle Aktionen leiten daher von der abstrakten Basisklasse `GenerationAction` ab, von der die abstrakten Methoden `Perform()` und `Undo()` implementiert werden müssen. Mehrere solcher Aktionen können aneinandergekettet werden, um einen einzelnen Generierungsschritt abzubilden. Nach Ausführung der gesamten Kette muss die Lösbarkeit des Levels gewährleistet sein.

`IActivator`

Das Interface `IActivator` wird von allen Spielelementen implementiert, die etwas aktivieren können. Diese stellen damit eine Methode zur Verfügung, mit der sich feststellen lässt, in welchem Zustand sich dieses befindet.

`Activatable`

Der Typ `Activatable` verwaltet eine Liste von verbundenen Aktivatoren, von denen wahlweise alle oder nur ein einzeler aktiv sein müssen, damit das `Activatable` als aktiviert gilt. Aktivierbare Objekte halten eine Referenz auf eine `Activatable`-Instanz. Mithilfe dieser Klasse lassen sich die Beziehungen zwischen Schaltern, Türen und Kamerazonen modellieren.

`Area`

`Area` stellt die abstrakte Basisklasse für die begehbaren Bereiche des Spiels dar. Sie verwaltet sowohl eine Liste von `Placeable`-Objekten, die sich derzeit darin aufhalten, als auch eine weitere Liste ihrer direkten benachbarten Bereiche. Eine virtuelle Methode `IsVisitable()` ist vorhanden, die zurückgibt, ob die `Area` überhaupt betretbar ist. Ebenso können mit weiteren Methoden die Distanzen zu anderen Bereichen bzw. deren generelle Erreichbarkeit errechnet werden (siehe Abschnitt 7.3).

`CamArea`

Die Klasse `CamArea` ist eine Ableitung von `Area` und hält eine Referenz auf ein `Activatable`-Objekt, anhand dessen bestimmt wird, ob die Kamerazone im aktuellen Zustand betreten werden darf oder ob sie versperrt ist.

`FloorSwitch`

`FloorSwitch` leitet von `Area` ab und überschreibt `IsVisitable()` so, dass ein Schalter nur betretbar ist, falls sich kein anderes Objekt darauf befindet. Weiters wird das `IActivator`-Interface implementiert, welches einen positiven Wert zurückliefert, wenn ein Objekt auf dem Schalter steht.

`Room`

`Room` leitet von der Klasse `Area` ab und erweitert diese um ein Array von `Door`-Objekten, das die Türen an den einzelnen Seiten eines Raumes repräsentiert. Die Methode zum Errechnen der erreichbaren Bereiche wird so überschrieben, dass nicht nur direkte Nachbarn, sondern auch Räume, die mittels den verbundenen Türen erreichbar sind zurückgeliefert werden.

`Door`

Da Räume in *Vancouver Maneuver* nicht direkt miteinander verbunden werden dürfen, muss dies mittels `Door`-Objekten geschehen. Diese sind mit einem `DoorConnector` verknüpft, der eine Liste aller Türen, die sich auf der entgegengesetzten Seite befinden, zurückliefert. Von diesen aus kann wiederum der alle verbunden Räume geschlossen werden.

`Placeable`

`Placeable` stellt die abstrakte Basisklasse aller Spielelemente dar, die in einer `Area` platziert und verschoben werden können. Die Anzahl der durchgeführten Bewegungen wird aufgezeichnet, was für den Generator wertvolle Informationen liefert.

7.3 Erreichbarkeitsberechnung

Die Berechnung der Erreichbarkeit einzelner Objekte ist notwendig für den *Solver* und auch im Generierungsprozess wird die Distanzmessung zwischen einzelnen Objekten benötigt, welche auf dem selben Algrothmus beruht.

Um die Erreichbarkeit einzelner Bereiche zu testen, wird allen `Area`-Objekten ein Distanzwert von -1 zugeordnet. Ein Startpunkt wird gewählt, der auf einem *Stack* gespeichert und dessen Distanz auf 0 gesetzt wird. Solange sich Objekte auf dem *Stack* befinden, wird die oberste `Area` entfernt und alle benachbarten Bereiche auf dem *Stack* platziert, deren Distanzwert nicht -1 entspricht. Sollte einer dieser Bereiche nicht erreichbar sein (weil z. B. eine Tür verschlossen ist), wird dessen Distanz auf -2 gesetzt, anderfalls wird der aktuelle Wert kopiert und um 1 erhöht. Nachdem diese Prozedur abgeschlossen ist, kann aus jeder einzelnen `Area` die entsprechende Distanz zum Startpunkt ausgelesen werden.

Es ist jedoch nicht ganz so einfach zu ermitteln, welche Räume erreicht werden können und welche nicht. Dies wird durch die Türen des jeweiligen Raumes und deren Position in der Wand bestimmt (nummeriert von 0 bis 4 im Uhrzeigersinn). Ein Raum mit einer Breite w und einer Tür an der Stelle i kann nur von einem anderen Raum betreten werden, der eine Tür an der Stelle $w - 1 - i$ in einer seiner Wände enthält. Dies gibt jedoch nicht an,

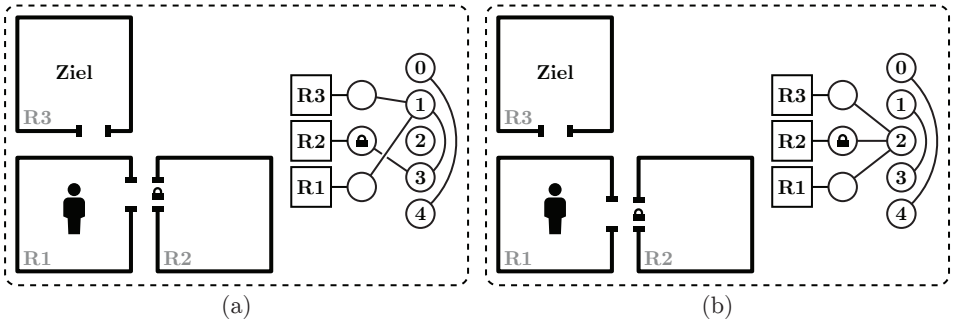


Abbildung 7.1: Die abgebildeten Spielbretter sind identisch, bis auf die Wandpositionen der Türen. Es ergibt sich ein Spezialfall für ungerade Raumbreiten: (a) R3 ist von R1 aus nicht erreichbar, da die Tür in R2 versperrt ist (wäre aber erreichbar, wenn dies nicht der Fall wäre), (b) R3 ist von R1 erreichbar, obwohl die Tür in R2 versperrt ist.

ob die Räume tatsächlich aneinander gelegt wurden, sondern nur, dass dies potentiell möglich ist. Um diese Strukturen schneller traversieren zu können, werden sogenannte Tür-Konnektoren eingeführt. Diese verwalten eine Liste aller Türen mit ihrer Wandposition und speichern eine Referenz auf jenen Konnektor, der die gegenüberliegenden Türen repräsentiert. Für jedes `Board` werden fünf dieser Objekte erstellt, eines für jede mögliche Wandposition. Wenn der Pfadfindungsalgorithmus Knoten expandiert und auf eine Tür trifft, verweist diese auf ihren entsprechenden Tür-Konnektor. Es wird dann nicht direkt die Tür-Liste des Konnektors expandiert, sondern die Liste des verbundenen Konnektors. Dadurch muss nicht ständig über alle Türen iteriert werden, um deren Konnektivität zu überprüfen. Es ist jedoch nicht dafür geeignet, um die Pfade von Lasern verifizieren. Dazu müssten alle potentiell erreichbaren Räume umpositioniert werden, um deren tatsächliche Erreichbarkeit zu ermitteln.

Abbildung 7.1 zeigt wie sich der mittlere `DoorConnector` verhält, wenn die Raumbreite des Spielbretts eine ungerade Zahl ist. Dieser verweist auf sich selbst, wodurch mittige Türen mehr Verbindungen aufweisen als andere. Dieser Spezialfall ist einer der Hauptgründe weswegen die Pfadfindung auf diese Weise implementiert wurde.

Besondere Vorsicht ist weiters auch bei Schaltern geboten. Wenn sich ein Objekt auf einem Schalter befindet, gilt dieser als aktiviert. Das bedeutet, alle Türen die an diesem Schalter hängen, gelten als geöffnet. Würde der Spieler jetzt jedoch vom Schalter hinunterschreiten, würden sich die entsprechenden Türen schließen. Eine Berechnung während der Spieler am Schalter steht hätte jedoch ergeben, dass Bereiche hinter jenen Tür als erreichbar gelten. Ist ein Bodenschalter der Ausgangspunkt der Erreichbarkeitsberechnung, muss dieser also temporär deaktiviert werden.

Kapitel 8

Analyse

Das grundlegende Ziel, einen funktionierenden Level-Generator für *Vancouver Maneuver* zu erstellen, konnte erfüllt werden, wenn auch nicht alle ursprünglichen Spielelemente umgesetzt werden konnten. Es gilt nun jedoch herauszufinden, welche der zuvor festgelegten Anforderungen der entwickelte Ansatz erfüllen kann. Einerseits wird der Algorithmus selbst anhand verschiedener Kritierien, wie z. B. seiner Performanz, beurteilt. Andererseits werden dessen Resultate mit Levels verglichen, welche von einem Mitglied des Projektteams von *Vancouver Maneuver* zuvor manuell entworfenen wurden. Dazu werden einige messbare Kennzahlen von Levels definiert und gegenübergestellt.

8.1 Manuell erstellte Levels

Im Laufe des Entwicklungsprozesses von *Vancouver Maneuver* wurden insgesamt vier Levels händisch erstellt, die ansteigenden Schwierigkeitsgrad aufweisen. In Tabelle 8.1 werden deren definierende Eigenschaften aufgelistet. Der Generator wurde demententsprechend mit Parametern betrieben, die diesen Wertebereichen entsprechen. In allen Levels wurde jeweils nur ein Tresor und ein zugehöriger Schlüssel verteilt, der in einem der Kästen zu finden ist. Auffällig ist, dass die Kistenanzahl in allen Levels sehr gering ist, um häufige Spielerwechsel zu erzwingen.

8.2 Effizienz

8.2.1 Laufzeit

Eine der Anforderungen an den Generator war, dass er schnell genug ist, um vor dem Start eines Levels ausgeführt werden zu können. Die Laufzeit des Algorithmus hängt sehr stark von eingestellten Parametern ab. Werden Minimal- und Maximalwerte eingestellt, die jenen der manuell entworfe-

	Lvl. 1	Lvl. 2	Lvl. 3	Lvl. 4	Ds.
Räume	3	4	4	4	3,75
Türen	6	5	7	7	6,25
Schalter	2	3	1	4	2,5
Kisten	1	1	1	0	0,75
Laser-Emitter	1	0	1	0	0,5
Laser-Sensoren	1	0	2	0	0,75
Spiegel	1	0	1	0	0,5
Kameras	1	2	2	3	2
Kästen	1	1	1	3	1,5
Tresore	1	1	1	1	1
Lösungsschritte	7	6	9	13	8,75
Spielerwechsel	1	1	2	5	2,25

Tabelle 8.1: Die wichtigsten Eigenschaften der vier manuell erstellten Levels. In der rechten Spalte sind jeweiligen Durchschnittswerte aufgelistet.

nen Levels entsprechen, terminiert der Generator im Normalfall in einigen wenigen Sekunden. Er kann unter diesen Bedingungen daher zur Laufzeit eingesetzt werden, wenn eine kurze Ladezeit vom Spieler als akzeptabel angesehen wird. Zusätzlich ist dieses Ergebnis erheblich beeinflusst von der Hardware auf der die Tests ausgeführt wurden und vor allem auf mobilen Geräten sind schlechtere Laufzeiten anzunehmen.

8.2.2 Speicherverbrauch

Der entwickelte Ansatz wendet *Backtracking* an, was eine Tiefensuche impliziert. Der Speicherverbrauch hält sich dadurch sehr gering, weil beim Traversieren des Suchbaumes immer nur der aktuelle Ast im Speicher gehalten werden muss. Der *Solver*, der zur Verifizierung der generierten Lösungen angewandt wird, benötigt weit größere Mengen an Speicherplatz, da Information über alle bereits besuchten Spielzustände in einer *Hashmap* gespeichert werden. Er geht mittels Breitensuche vor, um die kürzeste Lösung zu finden, weshalb zusätzlich eine große Liste von expandierten Spielzuständen verwaltet werden muss.

8.3 Kontrollierbarkeit

8.3.1 Objektanzahl

Bei Häufigkeit einzelner Elemente ist direkt kontrollierbar und das primäre Kriterium, wann der Algorithmus stoppen soll. Dadurch ist garantiert, dass eine Lösung gefunden wird, die den Parametern exakt entspricht, falls sie existiert. Eine Suche nach einem Level mit mindestens drei Räumen und maximal zwei Türen wird hingegen kein Ergebnis liefern, da jeder Raum mindestens eine Tür benötigt. Der Generator wird zwar einen dritten Raum erzeugen, diesen Schritt aber sofort wieder rückgängig machen, wenn festgestellt wird, dass die Maximalanzahl an Türen bereits erreicht wurde. Der Generator ist nicht in der Lage solche Fehlkonfigurationen zu erkennen und wird unnötigerweise den kompletten Lösungsraum durchsuchen.

8.3.2 Schwierigkeitsgrad

Der gewünschte Schwierigkeitsgrad kann nur ungefähr über die Länge des Lösungsweges bestimmt werden. Da alle Bewegungen der Spieler zwischen Interaktionen mit Objekten zu einem einzelnen Lösungsschritt zusammengefasst werden, ist der Lösungsweg ein guter Indikator für die Schwierigkeit eines Levels. Die Länge einzelner Laufwege fällt daher nicht ins Gewicht. Zusätzlich müssen jedoch die Maximal-Objektanzahlen hoch genug eingestellt sein, damit genügend Bereiche und Objekte für entsprechend lange Lösungswege vorhanden ist.

8.3.3 Multiple Lösungswege

Der Generator trimmt gezielt Aktionen, welche zuvor generierte Rätselteile beeinflussen. Er ist daher nicht in der Lage multiple Lösungswege zu generieren. Dies wird jedoch als positive Eigenschaft des Generators angesehen.

8.3.4 Nutzung des Mehrspieler-Aspektes

Einer der Eingabeparameter für den Generator ist die Wahrscheinlichkeit, mit welcher ein Spielerwechsel auftreten soll. Jedes Mal bevor der Generator die möglichen Aktionen des aktuellen Spielzustandes errechnet, wird anhand dieser Wahrscheinlichkeit bestimmt, welcher Spieler im nächsten Zug aktiv sein soll. Darauf basierend wird nicht komplett zufällig eine der Aktionen ausgewählt, sondern zuerst jene ausprobiert, die den besagten Spieler involvieren. Dies ist jedoch keine Garantie dafür, dass, wenn ein Wert von 1 angegeben wird, tatsächlich jede Runde ein Spielerwechsel stattfindet. Es können Situationen auftreten, in denen dies einfach nicht möglich ist und trotzdem der gleiche Spieler erneut zum Zug kommt. Dieser Umstand tritt jedoch selten auf.

Kapitel 9

Schlussbemerkungen

9.1 Zusammenfassung

Innerhalb dieser Arbeit wurden mehrere verschiedene Herangehensweisen zur Erstellung von Spielinhalten präsentiert und deren Eignung zur Generierung von Levels für das Spiel *Vancouver Maneuver* evaluiert, welches als rasterbasiertes Mehrspieler-Rätselspiel klassifiziert werden kann. Da sich keines der vorgestellten Verfahren perfekt für die Anwendung im Spiel eignete, wurde ein Hybridansatz entwickelt, der einzelne Konzepte aus den vielversprechendsten Methoden kombiniert. In Anlehnung an den Graph-Grammatik-Ansatz wird dabei zuerst einem minimales Spielfeld erstellt, welches anhand von gewissen Regeln schrittweise mit neuen Spielelementen erweitert wird. Dies wurde als *Backtracking*-Algorithmus realisiert: Wenn eine unlösbare Situation auftritt, werden alle Aktionen, die seit dem letzten lösbaren Spielzustand durchgeführt wurden, rückgänging gemacht und die nächste verfügbare Aktion wird ausprobiert. Der Generierung geht eine Benutzereingabe voraus, bei der gewisse Parameter festgelegt werden können, die das gewünschte Endresultat beschreiben (wie z. B. die Länge des Lösungsweges). Ist eine diesen Daten entsprechende Konfiguration gefunden, wird in einem zweiten Schritt die generierte Struktur in tatsächliche Levelgeometrie umgewandelt. Die einzelnen Objekte werden dabei auf konkrete Spielfelder platziert. Hier wird erneut mittels *Backtracking* vorgegangen, um etwaige Kollisionen einzelner Objekte zu vermeiden. Mit einem *Solver* kann anschließend sichergestellt werden, dass die generierte Level-Instanz tatsächlich lösbar sind. Leider konnten nicht die gesamten Spielregeln von *Vancouver Maneuver* im Generator umgesetzt werden, da diese einen einen riesigen Lösungsraum verursachen, der nicht in angemessener Laufzeit abgesucht werden kann. Um die Problemgröße zu reduzieren und einen funktionierenden Generator zu erstellen, mussten diverse Simplifizierungen vorgenommen werden. Unter anderem wurden die laser-basierten Spielelemente entfernt, da diese besonders viele zusätzliche Überprüfungen notwendig machten. Al-

leine aufgrund dieser Tatsache haben die Ergebnisse des Generators bereits einen weit geringeren Variantenreichtum als Levels, die von Menschenhand erstellt wurden. Die anderen Spielelemente betreffend weist der Generator jedoch eine hohe Kontrollierbarkeit auf. Es können Spielbretter generiert werden, die den Benutzereingaben exakt entsprechen, vorausgesetzt, dass eine entsprechende Lösung existiert.

9.2 Fazit

Die Erzeugung von Levels für *Vancouver Maneuver* erwies sich als schwierig. Besonders die drehbaren Räume legen der Generierungsmethode erhebliche Limitierungen auf. Das Erstellen von Türen wird z. B. ab einer gewissen Anzahl problematisch, da keine neue hinzugefügt werden kann, welche von keinem der bereits existierenden Räume erreichbar ist. Einzelne Generierungsschritte müssen sehr vorsichtig designt werden, um nicht mit zuvor erstellten Rätseln zu interferieren. Dadurch wird nur ein Teilbereich des gesamten Lösungsraumes durchsucht und möglicherweise besonders interessante und innovative Rätselkombinationen, die mehrere unterschiedliche Spielelemente kombinieren, werden von vorherein ausgeschlossen. Es wird vermutet, dass für Spiele mit statischen Levelstrukturen bessere Erfolgsaussichten bestehen. Obwohl mit dem implementierten Ansatz nicht die kompletten Spielregeln abgebildet werden konnten, war er grundsätzlich erfolgreich dabei funktionierende Rätselstrukturen zu erzeugen. Die Länge des Lösungsweges und die Anzahlen einzelner Spielelemente konnten sehr direkt steuerbar gemacht werden. Ebenso kann der Generator Pfade mit häufigen Spielerwechseln favorisieren und sicherstellen, dass beide Spieler gleichmäßig am Lösungsweg beteiligt sind, was eines der Hauptziele darstellte.

9.3 Ausblick

Der Forschungsstand im Bereich der prozeduralen Generierung variiert zwischen den einzelnen Spiele-Genres und auch innerhalb der Genres sehr stark. Gerade für den Typ von Rätselspielen, der hier in dieser Arbeit behandelt wirds, besteht noch viel Potenzial für die Zukunft. Um die tatsächlich fühlbare Level-Qualität zu überprüfen, war ursprünglich geplant Testspiele durchzuführen, deren Teilnehmer einen Fragebogen ausfüllen sollten. Dies war wegen zeitlichen Komplikationen nicht möglich, würde aber den nächsten Schritt darstellen, um den Erfolg dieser Arbeit besser einzustufen.

Anhang A

Inhalt der CD-ROM

A.1 Masterarbeit

Pfad: /

Kertesz_Moritz_2017.pdf Masterarbeit

A.2 Literatur

Pfad: /Literatur/

*.pdf . . . . . Kopien der Literatur- und Online-Quellen

A.3 Projektdateien

Pfad: /ThesisProject/

Executables/ . . . . . Ausführbare Dateien für Windows
Source/ . . . . . *Unity*-Projekt

A.4 Abbildungen

Pfad: /Abbildungen/

*.pdf . . . . . Vektorgrafiken
*.png . . . . . Screenshots

Quellenverzeichnis

Literatur

[1] Daniel Ashlock. „Automatic Generation of Game Elements via Evolution“. In: *Proceedings of the 2010 IEEE Symposium on Computational Intelligence and Games*. Copenhagen, Denmark: IEEE, Aug. 2010, S. 289–296 (siehe S. 34).

[2] Noam Chomsky. „Three Models for the Description of Language“. *IRE Transactions on Information Theory* 2.3 (Sep. 1956), S. 113–124 (siehe S. 27).

[3] Joris Dormans und Sander Bakkes. „Generating Missions and Spaces for Adaptable Play Experiences“. *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (Sep. 2011), S. 216–228 (siehe S. 29, 31, 32).

[4] Bernhard Handler. „Prozedurale Levelgenerierung für 2D Plattformspiele“. Masterarbeit. Hagenberg im Mühlkreis, Austria: University of Applied Sciences Upper Austria, Interactive Media, Okt. 2012 (siehe S. 2).

[5] Mark Hendrikx u. a. „Procedural Content Generation for Games: A Survey“. *ACM Transactions on Multimedia Computing, Communications, and Applications* 9.1 (Feb. 2012), S. 1–22 (siehe S. 4).

[6] Christopher Jefferson, Wendy Moncur und Karen E. Petrie. „Combination: Automated Generation of Puzzles with Constraints“. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC ’11. Taichung, Taiwan: ACM, März 2011, S. 907–912 (siehe S. 34).

[7] Lawrence Johnson, Georgios N. Yannakakis und Togelius Julian. „Cellular Automata for Real-time Generation of Infinite Cave Levels“. In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. PCGames ’10. Monterey, CA, USA: ACM, Juni 2010, 10:1–10:4 (siehe S. 5).

[8] Dean LeBaron, Logan Mitchell und Dan Ventura. „Intelligent Content Generation via Abstraction, Evolution and Reinforcement“. In: *Proceedings of the 2015 AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Santa Cruz, CA, USA: AAAI, Sep. 2015, S. 36–41 (siehe S. 34).

[9] Roland van der Linden, Ricardo Lopes und Rafael Bidarra. „Designing Procedurally Generated Levels“. In: *Proceedings of the IDPv2 2013 - Workshop on Artificial Intelligence in the Game Design Process, co-located with the Ninth AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment*. Palo Alto, CA, USA: AAAI, Okt. 2013, S. 41–47 (siehe S. 32, 33).

[10] Stelios Manousakis. „Musical L-Systems“. Masterarbeit. Den Haag, Netherlands: The Royal Conservatoire, Sonology, Juni 2006 (siehe S. 30).

[11] Michael Nelson Mark J. and Mateas. „Towards Automated Game Design“. In: *Proceedings of the AI*IA 2007: Artificial Intelligence and Human-Oriented Computing: 10th Congress of the Italian Association for Artificial Intelligence*. Hrsg. von Roberto Basili und Maria Teresa Pazienza. Rome, Italy: Springer, Sep. 2007, S. 626–637 (siehe S. 6).

[12] Gabriela Ochoa. „On Genetic Algorithms and Lindenmayer Systems“. In: *Proceedings of the 5th International Conference PPSN V: Parallel Problem Solving from Nature*. Hrsg. von Agoston E. Eiben u. a. Amsterdam, Netherlands: Springer, Sep. 1998, S. 335–344 (siehe S. 30).

[13] Yoav I. H. Parish und Pascal Müller. „Procedural Modeling of Cities“. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. Los Angeles, CA, USA: ACM, Aug. 2001, S. 301–308 (siehe S. 30).

[14] Timothy Roden und Ian Parberry. „From Artistry to Automation: A Structured Methodology for Procedural Content Creation“. In: *Proceedings of the 3rd International Conference on Entertainment Computing*. Hrsg. von Matthias Rauterberg. ICEC 2004. Eindhoven, Netherlands: Springer, Sep. 2004, S. 151–156 (siehe S. 4).

[15] Ruben M. Smelik. „A Declarative Approach to Procedural Generation of Virtual Worlds“. Diss. Delft University of Technology, Computer Graphics und CAD/CAM Group, Nov. 2011 (siehe S. 4, 5).

[16] Adam M. Smith und Michael Mateas. „Answer Set Programming for Procedural Content Generation: A Design Space Approach“. *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (Sep. 2011), S. 187–200 (siehe S. 35).

[17] Joshua Taylor und Ian Parberry. „Procedural Generation of Sokoban Levels“. In: *Proceedings of the 6th International North American Conference on Intelligent Games and Simulation GAMEON-NA*. Galway, Ireland: EUROSIS, Aug. 2011, S. 5–12 (siehe S. 27).

[18] Julian Togelius u. a. „What is Procedural Content Generation?: Mario on the Borderline“. In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. PCGames '11. Bordeaux, France: ACM, Juni 2011, 3:1–3:6 (siehe S. 4).

[19] Georgios N. Yannakakis und Julian Togelius. „Experience-Driven Procedural Content Generation“. *IEEE Transactions on Affective Computing* 2.3 (Juli 2011), S. 147–161 (siehe S. 6).

Games

[20] Valve Corporation. *Portal 2*. Microsoft Windows, OS X, Linux, PlayStation 3, Xbox 360. 2011. URL: http://www.thinkwithportals.com/ (siehe S. 14).

[21] Nintendo EAD. *The Legend of Zelda: Twilight Princess*. Gamecube, Wii, Wii U. 2011. URL: http://www.zelda.com/tp/ (siehe S. 16).

[22] Farbrausch. *.kkrieger*. Microsoft Windows. 2004 (siehe S. 7).

[23] Hello Games. *No Man's Sky*. Microsoft Windows, PlayStation 4. 2016. URL: http://www.nomanssky.com/ (siehe S. 8).

[24] Croteam Ltd. *The Talos Principle*. Microsoft Windows, OS X, Linux, Android, PlayStation 4. 2014. URL: http://www.croteam.com/talosprinciple/ (siehe S. 14).

[25] Gearbox Software. *Borderlands 2*. Microsoft Windows, OS X, Linux, PlayStation 3, PlayStation Vita, Xbox360, PlayStation 4, Xbox One, Nvidia Shield. 2012. URL: http://www.gearboxsoftware.com/game/borderlands-2/ (siehe S. 7).

[26] Inc. Spectrum Holobyte. *Soko-Ban*. MS-DOS. 1988 (siehe S. 13).

[27] Silicon & Synapse. *The Lost Vikings*. Amiga, Amiga CD32, Game Boy Advance, MS-DOS, Mega Drive/Genesis, Super NES, PlayStation. 1992 (siehe S. 13).

Online-Quellen

[28] Interactive Data Visualization Inc. *SpeedTree*. 2000. URL: http://www.speedtree.com/ (siehe S. 9).

[29] Qualcomm Inc. *Vuforia Augmented Reality SDK*. 2005. URL: https://www.vuforia.com/ (siehe S. 41).

[30] *Screenshot von Borderlands 2*. URL: https://cdn-enterprise.discourse.org/gearbox/uploads/default/original/3X/2/1/2147502143a75236c48ddd3546caef7acda430a5.jpg (besucht am 02. 08. 2016) (siehe S. 8).

[31] *Screenshot von .kkrieger*. URL: https://upload.wikimedia.org/wikipedia/en/c/ce/Kkrieger_screenshot.jpg (besucht am 02. 08. 2016) (siehe S. 7).

[32] *Screenshot von No Man's Sky*. URL: http://cdn.idigitaltimes.com/sites/idigitaltimes.com/files/2016/06/08/no-mans-sky-screenshot.jpg (besucht am 02. 08. 2016) (siehe S. 8).

[33] *Screenshot von Portal 2*. URL: http://vignette2.wikia.nocookie.net/half-life/images/7/7f/Portal_2_coop_jan_22_3.jpg/revision/latest?cb=20110205173221&path-prefix=en (besucht am 22. 02. 2016) (siehe S. 15).

[34] *Screenshot von Sokoban*. URL: http://www.abandonia.com/files/games/231/Sokoban_2.png (besucht am 02. 08. 2016) (siehe S. 14).

[35] *Screenshot von SpeedTree*. URL: http://www.speedtree.com/images/modeler_lrg.jpg (besucht am 02. 08. 2016) (siehe S. 9).

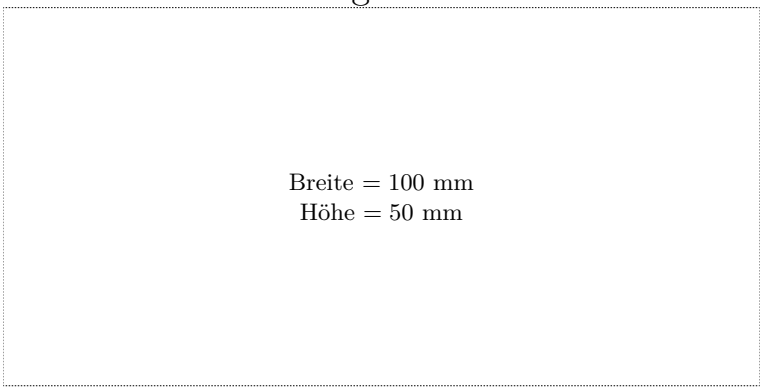
[36] *Screenshot von The Legend of Zelda: Twilight Princess*. URL: https://thefortressofhelixity.files.wordpress.com/2014/05/lakebed-temple-main-room-tp.jpg (besucht am 22. 02. 2016) (siehe S. 17).

[37] *Screenshot von The Talos Principle*. URL: http://scientificgamer.com/blog/wp-content/uploads/2015/07/talos_doubler.jpg (besucht am 02. 08. 2016) (siehe S. 16).

[38] Unity Technologies. *Unity*. 2005. URL: https://unity3d.com/ (siehe S. 41).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —