# Prozedurale Generation von Rätseln und Aufgaben in Videospielen

Michael Egger

28. September 2017

# 1 Einleitung

Videospiele gewinnen immer weiter an Popularität. Laut der Entertainment Software Association (ESA) spielten im Jahr 2003 50% aller US-Amerikaner Videospiele, welches einen Umsatz von rund 7 Milliarden US$ bewirkte. [1] Heute sind es 63% der amerikanischen Haushalte, in denen mindestens eine Person wohnt, die wöchentlich mehr als 3 Stunden mit Videospielen verbringt. (http://essentialfacts.theesa.com/#key-facts, 18.08.2017) In Deutschland wurde laut Bundesverband Interaktive Unterhaltungssoftware (BIU) in den Jahren 2015 und 2016 jeweils ein Umsatz von 2.9 Milliarden verzeichnet. (https://www.biu-online.de/marktdaten/gesamtmarkt-digitale-spiele-2016/, 18.08.2017) Die händische Entwicklung von Videospielen konnte bisher sicherstellen, dass Quantität und Qualität von Inhalten in Videospielen dem Markt gerecht werden. Über das letzte Jahrzehnt steigte jedoch die Anzahl der Konsumenten und die Produktionskosten [1] für Spiele exponentiell an. Spielinhalte sind ein wichtiger Faktor, um die Spieler auf Dauer unterhalten zu können.

## 1.1 Prozedurale Generation für Videospiele (PCG-G)

Die sogenannte "Procedural Content Generation for Games" (PCG-G) versucht, durch Automatisierung und Unterstützung der Inhaltserstellung diesen schnell wachsenden Anforderungen gerecht zu werden. Mark Hendrikx et al. [3] beschreiben sechs Hauptklassen von Inhalten in Spielen, die prozedural generiert werden können. Als "Game Bits" bezeichnen sie elementare Einheiten des Spieles, welche für den Spielablauf meist nicht essentiell sind. Weiters unterscheiden sie zwischen konkreten- sowie abstrakten Game Bits. Als konkrete Game Bits werde beispielsweise Bäume beschrieben, mit denen interagiert werden kann. Abstrakte Game Bits können beispielsweise Texturen sowie Audiospuren sein.
Als sogenannten "Game Space" wird die Umgebung bezeichnet, in dem das Spiel stattfindet. Diese ist mit "Game Bits" befüllt. Hendrikx et al. [3] bezeichnen Innen- und Außenarchitektur oder Landschaft, sowie Wasserbereiche wie Flüsse oder Seen, als "Game Space". Diese Kategorie ist für den Spieler sehr wichtig, da oft die Interpretation eines Spieles und dessen Spielwelt mit dem "Game Space" beginnt.
"Game Systems" sind meist Algorithmen, welche die Spielwelt oder deren Inhalte simulieren und sie somit glaubwürdiger für den Spieler machen soll. Zum Beispiel Systeme, welche das Zusammenspiel zwischen Vegetation und der Außenarchitektur bzw. der Landschaft simulieren, oder auch Systeme welche ein Ökosystem, Straßennetz, oder das Verhalten eines virtuellen Charakters simulieren.
Als "Game Scenarios" beschreiben dem Spieler eine Reihenfolge, in welcher sich

das Spiel entwickeln soll. Unter Anderem werden in [3] Puzzles als "Game Scenario" bezeichnet. Puzzles werden als Probleme definiert, welche der Spieler durch bereits vorhandenem Wissen und früheren Erfahrungen, oder durch systematisches Erkunden der potentiellen Lösungen lösen kann. Auch die Geschichte, also die "Story" eines Spieles, sowie das Konzept mehrerer Levels , wird als "Game Scenario" bezeichnet. Beide können einen gewünschten Ablauf im Spiel beschreiben, wie das Fortschreiten in einer Geschichte, oder das Freischalten eines neuen Levels.
Das "Game Design" beinhaltet Regeln (Welche Möglichkeiten hat der Spieler?) und Ziele (Was ist das Ziel des Spielers?), sowie ästhetische Konzepte wie der Grafikstil des Spieles. Dieser Bereich wird oft in "System Design" (Design der mathematischen Konzepte und Regeln, welchen das Spiel folgen soll) und "World Design" (Design des Kontextes, der Umgebung und der Geschichte des Spieles) kategorisiert.
"Derived Content", also der "abgeleitete Inhalt", wird von Hendrikx et al. [3] als Inhalt bezeichnet, der als Nebenprodukt der Spielwelt erstellt werden kann. So könnten zum Beispiel Nachrichten basierend auf Aktionen des Spielers generiert werden. Auch die Bestenlisten in Videospielen werden von [3] in diesen Bereich kategorisiert.

## 1.2 Populäre Beispiele

In den letzten Jahren wurden viele kommerzielle Spiele veröffentlicht, welche PCG-G verwenden. [3] nennt an dieser Stelle zum Beispiel die "Diablo" Serie (Erstveröffentlichung im Jahr 2000) und "Minecraft" (veröffentlicht im Jahr 2009), welche den "Game Space" prozedural generieren, als auch "EVE Online" (2003), welches "Game Bits", "Game Space" und "Game Scenarios" prozedural generieren kann. In diesem Abschnitt möchte ich zwei populäre Beispiele präsentieren.

### 1.2.1 Rogue, Universität von Berkeley, Kalifornien (1980)

Rogue wurde 1980 veröffentlicht und von Michael Toy, Glenn Wichman und später auch von Ken Arnold entwickelt. Spieler steuern in diesem Spiel einen Charakter um mehrere Levels eines Kerkers zu erforschen. Dabei können sie diverse Gegenstände und Schätze sammeln, die sie auf ihrem Weg unterstützen soll. Rogue ist ein Rundenbasiertes-Spiel, welches auf einem auf ASCII-Zeichen basierendem Spielfeld ausgetragen wird.
Bekannt wurde das Spiel in den 1980er Jahren vorallem bei College-Studenten und Computeraffinen Benutzern bekannt.
Rogue verwendete prozedurale Generation für die Erstellung der Levels, der Monster und der Schätze, die zu finden sind, sodass kein Spieldurchlauf einem anderen ähnelt.
Obwohl Rogue nicht das erste Spiel seiner Art mit prozeduraler Generation war, werden weiterhin viele Spiele veröffentlicht, die sich von Rogue inspirieren lassen. [5]

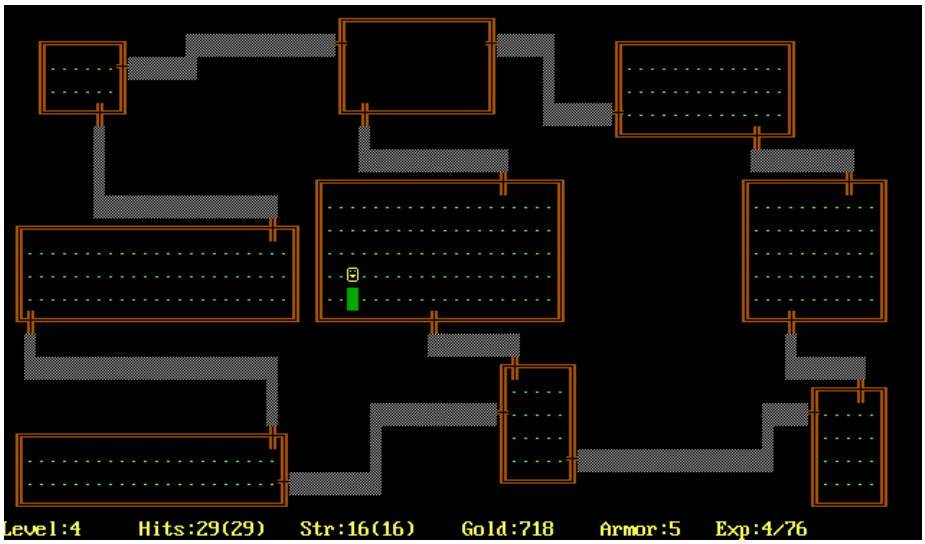


Abbildung 1: Quelle: https://en.wikipedia.org/wiki/Rogue_(video_game)

### 1.2.2 No Man's Sky, Hello Games, 2016

No Man's Sky ist ein Action-Adventure Überlebens Spiel [4], welches von Hello Games entwickelt und im Jahr 2016 veröffentlicht wurde. Spieler können sich frei in einem prozedural generierten und deterministischen Universum bewegen, welches über 18 Quintillionen ($1.8x10^{19}$)) Planeten beinhaltet, jeder mit eigener Flora und Fauna.
Fast alle Elemente des Spiels sind prozedural generiert, zum Beispiel Sternensysteme, Planeten und deren Ökosysteme sowie Tiere und deren Verhaltensstrukturen und Gebäude.

# 2 Problemstellung

Viele Videospiele verwenden Algorithmen, um Spielinhalte prozedural zu generieren. Bei den populären Beispielen sehen wir, dass nicht nur aktuelle Spiele solche Verfahren verwenden. Die Idee dahinter entstand schon in den 1980er Jahren.
Meist beschränken sich diese Verfahren jedoch auf das Generieren von virtuellen Welten oder Gegenständen. Das Spiel No Man's Sky führte das Prinzip sogar so weit, dass vollständige Planeten in der Spielwelt prozedural generiert werden konnten. Leider sind Spiele, die prozedural generierten Inhalt verwenden, einiger Kritik ausgesetzt: generierten virtuellen Welten fehlt oft eine spielerische Tiefe, wie sie bei manuell erstellten Welten und Inhalten zu finden ist. Dies führt daher, dass die verwendeten Algorithmen zwar nach gewissen Regeln arbeiten, diese jedoch meist keine spezifischeren Spielmechaniken berücksichtigen können.

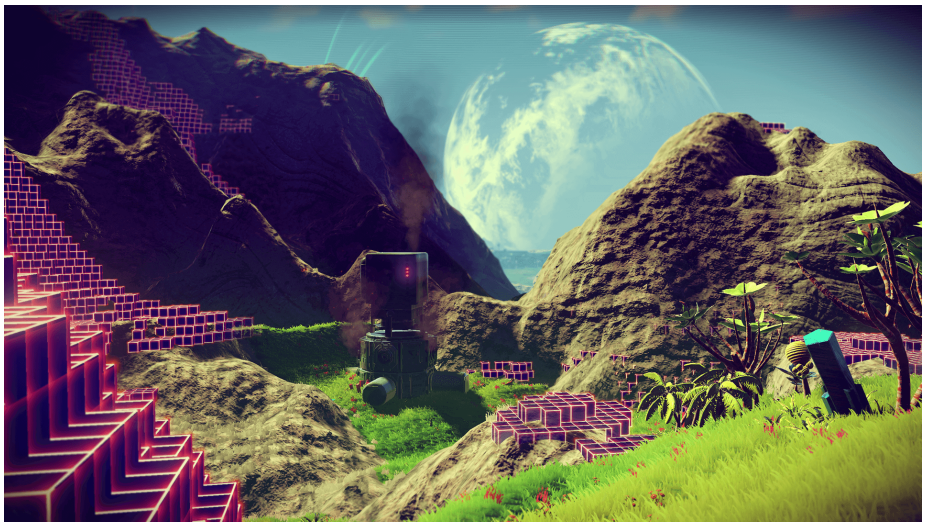

Abbildung 2: Quelle: http://nmswp.azureedge.net/wp-content/uploads/2017/02/ResourceLab.png

Die Idee zu dieser Arbeit versucht an diesem Punkt anzusetzen. Ich möchte Spieleentwicklern die Möglichkeit geben, in prozeduralen virtuellen Welten zusätzliche prozedurale Aufgaben und Rätsel einbinden zu können. Dieses System sollte Hand in Hand mit der Generation von Welten und Umgebungen agieren können.

# 3 Die Idee

Der Ansatz zum Lösen dieses Problems, welcher in dieser Arbeit präsentiert wird, baut auf folgendem Gedanken auf: Puzzles und Aufgaben in Videospielen lassen sich auf eine Folge von Aktionen des Spielers mit der Spielwelt abstrahieren. Sobald der Spieler diese Aktionen (optional in einer bestimmten Reihenfolge) abgeschlossen hat, ist die Aufgabe, beziehungsweise das Puzzle, erfüllt. Die Art der Aktionen und die Spielwelt sind nur vom Kontext des Spieles selbst abhängig. Das entwickelte System sollte also darauf aufbauen, dass mit einem gegebenen Kontext des Spieles, eine solche Folge von Aktionen des Spielers mit der Spielwelt erstellen lässt. Entwickler sollen die Möglichkeit haben, einen Kontext an das System zu übermitteln, und ein generiertes Ergebnis zu erhalten. Dieses Ergebnis sollte der Spieleentwickler wieder algorithmisch (prozedural) in das Videospiel einbinden können.
Dieser Ansatz sollte dazu führen, dass Entwickler zu Beginn des Spieles einen Kontext erstellen, aus diesem Kontext ein Puzzle generieren, um diese wiederum prozedural in ihre Spielwelt zu integrieren. Das Ziel sollte sein, dass der Spieler mit jedem Starten des Spiels eine neue Welt, mit neuen Aufgaben und Puzzles erhält.

# 4 Related Work

Clara Fernández-Vara und Alec Thomson [2] beschreiben eine Möglichkeit Puzzles in Adventure-Spielen zu generieren. Solche Puzzles zeichnen sich oft durch eine tiefere Verbindung mit der Erzählung des Spieles aus. Als Beispiel für solche Puzzles nennen sie das Kombinieren von zwei Gegenständen, das Finden eines passenden Schlüssels, oder in einer Konversation die richtigen Antworten zu geben, um im Spiel weiter zu kommen. Das System welches in [2] vorgestellt wird, verwendet eine Datenbank an Gegenständen und Charakteren, sowie eine sogenannte Puzzle-Map als Eingabe. Weiters wird der Name eines Gegenstandes angegeben, den der Spieler durch das Erfüllen des Puzzles erhalten soll. Die Puzzle-Map beschreibt die Struktur der Puzzles. Sie ergibt sich aus einer Kombination von verschiedenen Bausteinen. Diese Puzzle-Map ist dann Ausgangspunkt für die Generation von zufälligen Puzzles.

# 5 Diskussion

Während der Vorbereitung zu dieser Arbeit wurden viele Ansätze durchdacht, um dieses Problem zu lösen. Ich möchte an dieser Stelle einige dieser Lösungsansätze präsentieren, und darauf eingehen, warum diese ausgeschlossen wurden. Weiters werde ich näher auf den von uns verwendeten Lösungsansatz eingehen.

## 5.1 Die Anforderungen an das System

Von Beginn dieses Projektes an wurden sehr spezifische Anforderungen an eine potentielle Lösung zum Generieren solcher Inhalte gestellt. Das entwickelte System sollte

- in gängiger Software zur Entwicklung von Spielen (sogenannte S̈pieleengines)̈ verwendbar sein
- eine möglichst einfache Einrichtung des Systems gewähren
- mit wenig Aufwand möglichst viele verschiedene Ergebnisse liefern
- möglichst Effizient in Laufzeit und Speicherverbrauch sein
- für möglichst viele Kategorien von Spielen verwendbar und daher Anpassungsfähig und Allgemein sein

Um ein Bild davon zu bekommen, wie das System aufgebaut sein soll, wurde ein konkretes Anwendungsbeispiel erstellt.

Ein Usecase

1. EntwicklerIn bindet die Software in die Engine ein

2. EntwicklerIn gibt Kontext/Universum des Spiels an (Objekte, Interaktionen, Regeln) an.
3. Software generiert zur Laufzeit ein zufälliges Rätsel
4. EntwicklerIn kann das erstellte Rätsel interpretieren und in die Umgebungsgeneration einbinden. Das Rätsel wird somit direkt mit in die Spielwelt aufgenommen.
5. Bei einer Aktion des Spielers gibt EntwicklerIn die Interaktion an das Rätsel-System weiter.
6. Rätsel behandelt das Event und gibt EntwicklerIn die aktualisierten Stati der Objekte. Diese können wieder interpretiert und die Spielwelt aktualisiert werden.

Es wird also ein System gesucht, welches dem Entwickler die Möglichkeit gibt, eine Beschreibung der Spielwelt anzugeben und die Ergebnisse des Systems prozedural in das Spiel einzubauen, und den Fortschritt des Spielers im Puzzle wiederum prozedural durch das System bestimmen lassen. Somit sollte der Spieleentwickler nur eine passende Anbindung an das System benötigen, um damit viele verschiedene Spielinhalte generieren zu können.

Das System das in dieser Arbeit präsentiert wird, versucht diese Anforderungen umzusetzen. Zunächst wird die grundlegende Funktionsweise des entwickelten Systems erklärt. Im weiteren möchte ich eine formale Beschreibung bereitstellen, wie dieses System funktioniert.

## 5.2 Eine grundlegende Informelle Beschreibung

### 5.2.1 Puzzle

Ein Puzzle besitzt in diesem System eine (gerichtete) Graphen-ähnliche Struktur. Die Knoten in diesem Graphen sind Paare aus Spielobjekten - und Zielstatus. Die Verbindungen beschreiben Abhängigkeiten zwischen den Knoten.

### 5.2.2 Die Spielobjekte

Spielobjekte werden vom Entwickler an das System übergeben. Damit gibt der Entwickler einen Teil des Kontextes des Spieles an. Spielobjekte können gewöhnliche Objekte wie Schlüssel, Türen oder Schalter sein, aber auch Objekte wie Gegner oder Gegenstände können Spielobjekte sein. Eine wichtige Eigenschaft, die Spielobjekte im Allgemeinen erfüllen müssen, ist der Besitz und die potentielle Änderung des Status durch Aktionen des Spielers. Ein Schlüssel als Spielobjekte kann diese Anforderungen beispielsweise mit den Stati N̈icht gefundenünd G̈efundenërfüllen, welche etwa beschreiben, ob der Spieler diesen Schlüssel bereits aufgenommen hat oder noch nicht. Eine Tür kann beispielsweise Öffenöder G̈eschlossenśein. Ein Gegner kann K̈.Oöder B̈ei Bewusstseinśein. Wofür diese Stati und potentielle Änderungen des Status benötigt werden, werde ich im nächsten Abschnitt näher erläutern.

### 5.2.3 Die Knoten des Puzzle-Graphen

Wie oben schon erwähnt werden Puzzles in diesem System als Graphen dargestellt. Die Knoten dieses Graphen bestehen aus Spielobjekt und Status - Paaren. Wie im letzten Abschnitt schon kurz erläutert wurde, müssen Spielobjekte die Eigenschaft aufweisen, ihren Status ändern zu können. Diese Eigenschaft wird für die Art und Weise benötigt, wie der Graph traversiert werden kann. Ein Knoten gilt dann als erfüllt, wenn der Status des referenzierten Spielobjektes dem des Knoten entspricht. Ein Knoten mit dem Paar (Tür, Offen) ist beispielsweise genau dann erfüllt, wenn die Tür den Status Öffenbesitzt.

### 5.2.4 Die Kanten des Puzzle-Graphen

Wie etwas weiter oben erwähnt, beschreiben die Kanten des Graphen die Abhängigkeiten zwischen den Knoten. Diese geben an, wie der Graph, beziehungsweise später auch das Puzzle, traversiert werden kann. Ein Knoten mit dem Paar Schalter äls Spielobjekt und Benutztäls Status zeigt auf ein Knoten mit dem Paar Tür äls Spielobjekt und Öffen äls Status. In diesem System bedeutet das, dass der Knoten mit einer referenzierten Tür abhängig vom Knoten mit dem referenzierten Schalter ist. Welche Auswirkungen dies hat, wird im Abschnitt zu den Aktionen des Spielers näher erläutert.

### 5.2.5 Die Aktionen des Spielers

Um diesen Puzzle-Graphen zu traversieren, werden sogenannte Events, also die Aktionen des Spielers, verwendet. Ein Event besteht immer aus einem referenzierten Spielobjekt und einer Funktion, die Statusänderungen dieses Objektes beschreibt. Ein Beispiel für ein solches Event wäre das Öffnen einer Türe des Spielers. Das Event referenziert das Spielobjekt Türünd beschreibt eine Statusänderung von Geschlossenäch Öffen: Der Entwickler muss diese Events beim Erstellen des Kontextes des Spieles, neben den Spielobjekten, definieren und angeben. Jedes Mal wenn der Spieler versucht, eine Tür zu öffnen, wird dieses Event an das System übergeben.

### 5.2.6 Events und der Puzzle-Graph

Bei bestimmten Aktionen des Spielers mit der Spielwelt kann der Entwickler Events an das System übergeben. In der Beschreibung der Knoten des Puzzle-Graphen wurde beschrieben, dass Knoten erfüllt sind, sobald der Status des vom Knoten referenzierten Spielobjektes dem referenzierten Status des Knotens entspricht. Diese Statusänderungen werden von Events hervorgerufen.
Sobald ein Event an das System übergeben wurde, wird dieses an der passenden Stelle ausgeführt, und Statusänderungen von Objekten werden gegebenenfalls an den Entwickler zurückgeliefert. Mit Hilfe dieser Statusänderungen kann der Entwickler wiederum die dazugehörigen Stati der Objekte im Spiel selbst verändern, und dem Spieler somit den Fortschritt grafisch darstellen, wie z.B. das Öffnen der Tür in der Spielwelt. Durch die Abhängigkeiten von Knoten

ist es auch möglich, dass ein Event keine Statusänderungen hervorruft, da der zugehörige Knoten Abhängigkeiten besitzt, die zuvor noch nicht erfüllt wurden. Das führt dazu, dass der Entwickler keine Statusänderung erhält. Dadurch bleibt eventuell eine Tür verschlossen, und der Spieler weiß, dass er diese noch nicht öffnen kann.
Sollte diese Abhängigkeit ein Knoten sein, der einen verwendeten Schalter beschreibt, so kann der Spieler die Tür erst öffnen, sobald der Schalter betätigt wurde. Dieses Verhalten wird durch das System automatisch erzeugt, da Statusänderungen nur ausgeführt werden, wenn es dafür einen passenden Knoten gibt, dessen Abhängigkeiten erfüllt sind.

Um diese Begriffe wie Knoten, Kanten und Events genauer zu erläutern, beschreibe ich diese im nächsten Abschnitt formell:

## 5.3 Formale Beschreibung

### 5.3.1 Allgemein

Es gibt viele verschiedene Arten von Puzzles, wie z.B. logische Puzzles (Sudoku, Sokoban, Rubiks Würfel), als auch Wortpuzzles (Kreuzworträtsel) oder Zahlenpuzzles (wie z.B. T̈inde die Nächste Zahl in der Reihe: 2 4 8 16 ?̈). Diese Puzzles können alle verschieden definiert werden, jedoch werde ich mich auf ein Art von Puzzles in Videospielen konzentrieren und definiere diese auf eine allgemeine Weise.

Zu Beginn definiere ich die hier verwendete Terminologie:

**Definition 5.1.** Ein *Puzzle* in Videospielen ist eine Serie an Aufgaben, die ein Spieler zu erledigen hat, um das Puzzle zu lösen.

Weiters definieren wir welche Arten von Aufgaben dies sind und führen *Aktionen* ein.

**Definition 5.2.** Eine *Aktion* beschreibt all das, was in einem Spiel passiert. Diese kann entweder aktiv vom Spieler selbst, oder passiv vom Spiel ausgelöst werden.
Wenn eine Aktion auf einen Teil des Spieles ausgeführt wird, nennen wir dies *Interaktion*.

Eine Aktion kann somit alles sein, was in einem Spiel passiert; ausgehend entweder vom Spieler, oder vom Spiel selbst. Eine Interaktion ist somit eine Aktion mit einem bestimmten Teil des Spieles.
Ein Beispiel für eine Aktion wäre das Springen oder Laufen des Spielers.
Ein spezifisches Beispiel für eine Interaktion könnte das Öffnen einer Tür, oder das Aufheben eines Schlüssels sein. Die Tür und der Schlüssel wären somit ein Teil des Spiels.

**Definition 5.3.** Ein *Spielobjekt* beschreibt alle Teile des Spiels, mit denen interagiert werden kann.

Ein Spielobjekt kann somit jedes interagierbare Objekt in eine Spielwelt sein, wie z.B. eine Tür zum Öffnen oder ein Schlüssel zum Aufheben.

Da wir nun die allgemeine Terminologie definiert haben, können wir nun die Grundidee der prozeduralen Generation von Puzzles folgen:

**Grundidee.** Die meisten Puzzles in Videospielen können als eine Menge an Aktionen oder Interaktionen mit Spielobjekten beschrieben werden, welche in einer bestimmten Reihenfolge erfüllt werden müssen, um das Puzzle zu lösen. Das Puzzle gilt als gelöst, wenn die letzte Aktion erfüllt wurde. Diese Aktionen, Interaktionen und Spielobjekte sind abhängig vom Universum, in welchem das Spiel stattfindet.

**Generation von Puzzles.** Da ein Puzzle als eine Menge an Aktionen, oder Interaktionen mit Spielobjekten definiert werden kann, sollte es möglich sein, diese Menge prozedural zu generieren. Mit Hilfe eines gegebenen Universums kann ein lösbares Puzzle generiert werden.

Weiters werde ich kurz auf den Unterschied zwei verschiedener Darstellungsweisen von Puzzles eingehen.

### 5.3.2 Aktionsbasierte Puzzles

Puzzles wurden oben als eine Serie von Aufgaben definiert, die erfüllt werden müssen, um das Puzzle abzuschließen. Im aktionsbasierten Ansatz sind diese Aufgaben Aktionen. Ein Aktionsbasiertes Puzzle ist abgeschlossen, wenn alle Aktionen abgeschlossen sind.

**Definition 5.1.** Eine Aktion wird *relevant* genannt, wenn sie für das erfolgreiche Abschließen des Puzzles abgeschlossen sein muss.

Das bedeutet, dass nur noch relevante Aktionen abgeschlossen sein müssen, um das Puzzle abzuschließen. Daher können auch Aktionen existieren, die für das Puzzle nicht relevant sind, jedoch etwas mehr Variaton in diese Menge bringen.

**Definition 5.1.1.** Ein *Aktionsbasiertes Puzzle* ist ein Tupel bestehend aus einer Menge an Aktionen $A$, eine Menge an relevanten Aktionen $R \subseteq A$ und einer irreflexiven, transitiven Relation $O(A)$.

**Definition 5.2.** Aktion $a \in A$ kann abgeschlossen werden, $\iff$ $\forall b \in A$, $b <_O a$: b ist abgeschlossen.

**Definition 5.3.** Aktionsbasiertes Puzzle $P$ heißt *erfüllt* $\iff$ $\forall a \in R$ : a ist abgeschlossen.

Um den Ansatz des Statusbasierten Puzzles zu beschreiben muss die Bedeutung eines *Status* definiert werden:

**Definition 5.4.** Der *Status* eines Spielobjektes beschreibt den Status des dazugehörigen Objektes in der Spielwelt.

Der Statusbasierte Ansatz definiert diese Aufgaben als *Ausführung einer Aktion, oder einer Interaktion mit einem bestimmten Spielobjekt, um den benötigten Status zu erreichen.*

### 5.3.3 Statusbasierte Puzzles

Als ersten möchte ich die Defnition eines Spielobjektes auf den Kontext von Statusbasierten Puzzles erweitern.

**Definition 5.1.** Ein *Statusübergang* ist eine Funktion welche, abhängig der gegebenen Aktion oder Interaktion, den Status eines Spielobjektes verändert.

**Definition 5.1.1.** Ein *Statusbasiertes Spielobjekt* ist ein Tupel bestehend aus einem Anfangsstatus $S_0$, dem derzeitigen Status $S_C$ und einer Menge and Statusübergängen F.
Im Weiteren nennen wir Statusbasierte Spielobjekte einfachheitshalber Spielobjekte.

In dieser Definition wurde das Spielobjekt um einen Anfangsstatus, welcher beschreibt, in welchem Status das Spielobjekt im Statusbasierten Puzzle startet, um den derzeitigen Status und der Möglichkeit den derzeitigen Status zu verändern, erweitert.

Die Definition eines Statusbasierten Puzzles ist ähnlich zur Definition des Aktionsbasierten Ansatzes, wird jedoch um sogenannte Events erweitert.

**Definition 5.2.** Ein Event e ist ein Tupel bestehend aus einem zugehörigen Spielobjekt G und einem Namen $\nu$. Ein Event beschreibt Aktionen und Interaktionen im Statusbasierten Ansatz.

Events beschreiben hiermit Aktionen und Interaktionen. Events lösen Statusübergänge in Statusbasierten Puzzles aus. Sie werden, anders als Aktionen oder Interaktionen, vom Puzzle selbst verarbeitet. Um genauer zu beschreiben was dies bedeutet, werde ich die Baublöcke eines Statusbasierten Puzzles vorstellen, die sogenannten Puzzlenodes.

**Definition 5.3.** Eine *Puzzlenode*, also ein Knoten im Puzzle, ist ein Paar bestehend aus einem Spielobjekt $G$ und einem Zielstatus $S_G$.

Eine Puzzlenode ist eine simple Struktur, die als Informationsträger dient. Ein naheliegender Vergleich sind Knoten in einem Graphen. Jedes Statusbasierte Puzzle ist aus mehreren solcher Puzzlenodes aufgebaut. Weiters werden wir definieren, welche Attribute von Puzzlenodes gehalten werden:

**Definition 5.4.** Eine Puzzlenode wird *relevant* genannt, wenn diese für das Erfüllen des Puzzles abgeschlossen sein muss.

Ähnlich zur Relevanz von Aktionen sind Puzzlenodes, die nicht relevant sind, eine Möglichkeit mehr Variation in einem Puzzle zu gewähren.

**Definition 5.5.** Eine Puzzlenode $n \in N$ wird *abgeschlossen* genannt $\iff$ G hat aktuellen Status $S_G$.
Puzzlenode n wird *aktiv* genannt $\iff \forall q \in R$, $q <_O n$: q ist abgeschlossen.
Zwei Puzzlenodes $a, b \in N$ sind *gleich* $\iff G_a = G_b$ und $S_{Ga} = S_{Gb}$. Dann schreiben wir $a = b$.
Sei $G_e$ das Spielobjekt, auf das sich ein Event e bezieht, und $G_n$ das Spielobjekt, auf das sich Puzzlenode n bezieht. Puzzlenode n heißt *kompatibel* mit Event e $\iff G_e = G_n$ und n ist aktiv.

Mit diesen Definitionen können wir schließlich das Statusbasierte Puzzle definieren:

**Definition 5.6.** Ein *Statusbasiertes Puzzle* $P$ ist ein Tupel bestehend aus einer Menge an Puzzlenodes N, einer Menge an Relevanten Puzzlenodes $R \subseteq N$, einer irreflexiven, transitiven Relation $O(N)$ und einer Menge an Events E. Im Folgenden nennen wir Statusbasierte Puzzles einfach *Puzzle*.

**Definition 5.7.** Eine Puzzlenode n kann nur anschließend an ein Event e äbgeschlossen; beziehungsweise ïicht mehr abgeschlossen; werden.

Events werden von Puzzlenodes verarbeitet. Nur Puzzlenodes, die ein Event verarbeitet haben, können abgeschlossen werden. Dazu muss definiert werden welche Puzzlenode zum Verarbeiten eines Events ausgewählt wird. Eine Puzzlenode n ist nur dann berechtigt, Event e zu verarbeiten, wenn n zu e kompatibel ist.

**Definition 5.8.** Ein Event e wird immer vom **kleinsten** (im Bezug auf Relation O), aktiven kompatiblen Puzzlenode verarbeitet.
Sollte keine solche zur Verfügung stehen, wird e vom **größten** (im Bezug auf Relation O) abgeschlossenen kompatiblen Puzzlenode n verarbeitet $\iff \forall q \in R$, $n <_O q$: q ist nicht abgeschlossen.

Mit diesen letzten beiden Definitionen wird es möglich in der Puzzlestruktur Schritte *zurück* zu tätigen. Wenn eine kleinste, aktive, kompatible Puzzlenode n gefunden wird, wird dieses ausgewählt um das Event zu verarbeiten. Dadurch kann n abgeschlossen werden, und das nächste Event wird von der nachkommenden Puzzlenode verarbeitet. Dadurch wird das Puzzle um *einen Schritt weiter* dargestellt. Sollte keine solche Puzzlenode gefunden werden, wird die größte, abgeschlossene kompatible Puzzlenode m gewählt. Puzzlenode m kann dadurch ïicht mehr abgeschlossen sein; womit das Puzzle um *einen Schritt zurück* dargestellt wird.

In Aktionsbasierten Puzzles ist dies nicht direkt möglich. Aktionen können nicht rückgängig gemacht werden. Obwohl das durch Schleifen von Aktionen möglich sein könnte, ist es nicht trivial, solche Strukturen korrekt zu generieren, ohne die Lösbarkeit des Puzzles zu beeinträchtigen.

### 5.3.4 Prozedurale Generation von Puzzles

Damit Puzzles garantiert lösbar sind, müssen einige Eigenschaften erfüllt werden.

**Lösbarkeit.** Ein Puzzle p ist lösbar $\iff$ p enthält keine Deadlocks.

Weiters, derzeit beobachtete Deadlocks werden generiert durch

- Zirkuläre Abhängigkeiten
- Exklusive Abhängigkeiten
- Direkte Gleichheit

Für die folgenden Definitionen verwenden wir ein Statusbasiertes Puzzle P mit der gegebenen transitiven, irreflexiven Relation O.

**Definition 5.1.** Ein Puzzle P enthält eine **zirkuläre Abhängigkeit** $\iff$ $\exists a, b, c \in N$ mit $a <_O b$, $b <_O c$ and $c <_O a$.

Der Fall in Definition 5.1 wird durch die Transitivität sowie der Irreflexivität der gegeben Relation O verhindert.

**Definition 5.2.** Ein Puzzle P enthält eine **Exklusive Abhängigkeit** $\iff$ $\exists a, b, c \in N$ mit $b <_O a$ und $c <_O a$, wo gilt: $G_b = G_c$ aber $S_{Gb} \neq S_{Gc}$.

**Definition 5.3.** Ein Puzzle P enthält eine **Direkte Gleichheit** $\iff \exists a, b \in N$ mit $a <_O b$ und $a = b$.

Direkte Gleichheit wird durch die Irreflexivität der gegebenen Relation O verhindert.

Um ein Puzzle zu generieren, wird ein Universum, in welchem das Puzzle stattfindet, benötigt.

**Definition 6.** Ein Universum (im Kontext von generierten Puzzles) ist ein Tupel bestehend aus einer Menge an Spielobjekten, einer Menge an Events, und einer optionalen Menge an logischen Regeln.

Das Universum definiert den Kontext des Puzzles. Das generierte Puzzle ist damit eine zufällige Serie an Statusbasierten Puzzlenodes, in welcher die Anordnung (die Relationen) optional von weiteren logischen Regeln beschränkt werden kann, und die Inhalte der Puzzlenodes abhängig von den gegebenen Spielobjekten sind. Die Events beschreiben alle Aktionen oder Interaktionen die in der jeweiligen Spielwelt möglich sind und eventuell in das Puzzle einfließen sollen. Das Puzzle wird diese Events verarbeiten und die Stati der Spielobjekte dementsprechend anpassen.

Während Spielobjekte und Events definieren, was in einer Spielwelt passieren kann, beschreiben die logischen Regeln was in dieser Spielwelt nicht passieren sollte, beziehungsweise in dem Kontext der Spielwelt keinen Sinn ergibt. Diese logischen Regeln beschreiben dadurch die logischen Eigenschaften des gegebenen Universums. Ein einfaches Beispiel wäre Folgendes:
*In diesem Universum muss der Spieler immer zuerst einen Schlüssel aufgehoben*

*haben, um eine Tür zu öffnen.*
Es sollte nie notwendig sein, solche logischen Regeln, um ein vollständiges, lösbares Puzzle zu generieren. Diese sollten lediglich dazu dienen, die Lösungsmenge der zufällig generierten Puzzles einzuschränken.

**Ein Gedanke zu den logischen Regeln.** Gegeben sei ein Universum ohne logische Regeln. Die Lösungsmenge der generierten Puzzle besteht aus allen Permutationen von (lösbaren) Anordnungen der Puzzlenodes. Da die meisten Puzzles in Videospielen als eine Serie von Interaktionen mit Spielobjekten beschrieben werden können, sollte es möglich sein, jedes spezifische, lösbare Puzzle durch angeben von genügend logischen Regeln zu erhalten.

Sollte ein Benutzer ein spezifisches Puzzle generieren wollen, sollte dies durch eine genügend große Anzahl and logischen Regeln im Universum möglich sein, jedes mal genau dieses spezifische Puzzle zu erhalten.

## 6.1 Implementation

Das in dieser Arbeit beschriebene System wurde in C++ mit Hilfe von Visual Studio erstellt. Natürlich sollte es auch möglich sein, das selbe Ergebnis in jeder beliebigen äquivalenten Programmiersprache zu erhalten. Grund für die Implementation in C++ war die Möglichkeiten zur Einbindung in der Unreal Engine 4 von Epic Games [**?**]. Da die Unreal Engine 4 auf C++ basiert, und auch benutzerdefinierte Module direkt als C++ Objekte erstellt werden, war diese Entscheidung naheliegend.

In diesem Kapitel möchte ich etwas genauer auf meine Implementation in C++ eingehen, sowie im Weiteren eine Beispielanwendung in C++ präsentieren, und anschließend die Einbindung des Systems in die Unreal Engine 4 beschreiben.

### 6.1.1 Implementation in C++

Meine Implementation des Systems liefert eine Bibliothek von Funktionen, welche die Möglichkeit bietet, ein Puzzle nach den in Abschnitt 5.3 und Abschnitt 5.2 beschriebenen Eigenschaften zu generieren.

Hier eine kurze Übersicht über die verfügbaren Objekte des Systems. Die Frontend-Objekte werden vom Benutzer für das generieren von Puzzles benötigt, während die Backend-Objekte allein vom System selbst benutzt werden.

**Frontend Objekte**

- Puzzle.h
- PuzzleGenerator.h
- PuzzleObject.h
- PuzzleState.h

- PuzzleEvent.h
- PuzzleRule.h
- StateTransition.h
- PuzzleRule.h
- IPuzzleObjectMetaData.h
- PuzzleUpdateListener.h

**Backend Objekte**

- PuzzleRelation.h
- PuzzleRandomizer.h
- PuzzleNode.h

**Puzzle.h** Das *Puzzle* Objekt ist die Grundlegende Datenstruktur des generierten Puzzles. Dieses Objekt hält alle nötigen Informationen über den Fortschritt des Puzzles, bearbeitet Events (genauere Beschreibung in Paragraph 6.1.1), und informiert den Benutzer über Änderungen im Puzzle, beziehungsweise über Auswirkungen von Events im Puzzle. Ein *Puzzle* Objekt wird von dem *PuzzleGenerator* Objekt (6.1.1) automatisch erstellt und befüllt.

**PuzzleGenerator.h** Der *PuzzleGenerator* ist für die automatische Generierung von Puzzles zuständig. Diese Klasse ist der Einstiegspunkt für den Benutzer. Der PuzzleGenerator benötigt zum vollständigen Erstellen eines Puzzles ein oder mehrere *PuzzleObject* Objekte, eine Menge an *PuzzleEvent* Objekte (6.1.1), sowie optional eine Menge an *PuzzleRule* (6.1.1) Objekte.

```
/* Init Puzzle Generator*/
PuzzleGenerator* puzzGen = new PuzzleGenerator();

/* Generate Puzzle */
Puzzle* P = puzzGen->generatePuzzle(objects, events, rules);
```

Ausschnitt 6.1: Ein Beispielaufruf zum Generieren eines Puzzles mit dem PuzzleGenerator Objekt. objects ist hierbei ein Array von PuzzleObject Objekten, events ein Array von PuzzleEvent Objekten sowie rules ein Array von PuzzleRule Objekten

**PuzzleObject.h** Das *PuzzleObject* Objekt wird vom Benutzer verwendet, um Spielobjekte zu beschreiben, welche im generierten Puzzle verwendet werden sollen. Diese Objekte müssen mit einem Namen, mehreren möglichen Stati (per *PuzzleState* Objekte (6.1.1)), sowie einer *StateTransition* (6.1.1) erstellt werden. Ein Beispiel für solch ein PuzzleObject wäre zum Beispiel ein Schalter, der zwei Stati *Benutzt* und *Unbenutzt* aufweist, sowie eine StateTransition von *Unbenutzt nach Benutzt* durch das *PuzzleEvent* (6.1.1 ) mit dem Namen "Benutze".

```
PuzzleObject* leverObject = new PuzzleObject("Lever");

PuzzleState* s_Unused = new PuzzleState("Unused");
PuzzleState* s_Used = new PuzzleState("Used");

StateTransition* T = new StateTransition();
T->addTransition("Use", *s_Unused, *s_Used);

leverObject->setStateTransition(*T);
```

Ausschnitt 6.2: Ein Beispiel zum Erstellen eines Schalter-Objektes als PuzzleObject, mit zwei Stati, und einer State-Transition

Zusätzlich bietet das System dem Entwickler eine Möglichkeit weitere Metadaten an ein *PuzzleObject* Objekt zu binden. Genaueres dazu wird in Abschnitt zu *IPuzzleObjectMetaData.h* beschrieben.

**PuzzleState.h** Das *PuzzleState* Objekt beschreibt einen Status des *PuzzleObject* Objektes. Der Status kann sich durch eine definierte *StateTransition* und ein hervorrufendes *PuzzleEvent* ändern. Der Status des Objektes beschreibt einerseits den Status im Spiel, sowie den Status im Puzzle selbst. Der Fortschritt eines Puzzles ergibt sich aus mehreren *PuzzleObject* und deren aktuellen Status. Hat ein *PuzzleObject* den im *PuzzleNode* (genauere Beschreibung unter 6.1.1) definierten Zielstatus, so ist dieser Abschnitt des Puzzles erfüllt.

**PuzzleEvent.h** Das *PuzzleEvent* Objekt beschreibt eine Aktion des Spielers. Der Benutzer kann solche Events für sein Spiel definieren. Diese Events können wiederum bei *StateTransition* Objekte verwendet werden, um einen Statusübergang bei einem referenzierten *PuzzleObject* hervorzurufen. Events können an das *Puzzle* Objekt übergeben werden, welches dann von diesem verarbeitet wird. Der Benutzer muss diese Events direkt mit den Aktionen des Spielers koppeln. Sollte zum Beispiel ein Spieler einen Schalter betätigen, so sollte das Event "Use" (wie in Ausschnitt 6.2 beschrieben) an das *Puzzle* Objekt übergeben werden.

**PuzzleRule.h** Ein *PuzzleRule* Objekt beschreibt gewisse Regeln, die von der Generation des Puzzles eingehalten werden müssen. Bei der Erstellung dieser Arbeit bietet dieses Objekt nur die **BEFORE** Regel. Diese Regelart beschreibt, dass ein definiertes Objekt in der Puzzlegeneration *immer* vor dem anderen

Objekt stehen muss. Als Beispiel könnte eine Regel erforderlich sein, die beschreibt, dass ein "Schalter" Objekt immer vor dem "Tür" Objekt stehen muss. Es existiert auch die Möglichkeit zur Einschränkung mit Hilfe des Status von Objekten. So kann zum Beispiel eine Regel erstellt werden, sodass ein Schalterobjekt mit dem Status "Unbenutzt" immer vor dem Schalterobjekt mit dem Status "Benutzt" stehen muss. Diese Regeln bieten eine Möglichkeit, um gewisse logische Regeln für den Spieler und das Spiel zu gewährleisten.

```
1 /*
2 Lever muss in der Relation vor Door sein
3 PuzzleRule(PuzzleObject* o1, PuzzleState* s1, PuzzleObject*
      o2, PuzzleState* s2, PuzzleRule::EPuzzleRuleType)
4 */
5 PuzzleRule * rule = new PuzzleRule(lever, nullptr, door,
      nullptr,
6 PuzzleRule::E_PuzzleRuleType::BEFORE);
```

**StateTransition.h** Ein *StateTransition* Objekt beschreibt den Statusübergang eines Objektes. Statusübergänge werden von *PuzzleEvent* Objekten, also Events, hervorgerufen. Eine *StateTransition* muss an ein zuständiges *PuzzleObject* angehängt werden. Ein Beispiel zur Erstellung einer StateTransition wird in Ausschnitt 6.2 gezeigt.

**IPuzzleObjectMetaData.h** Das Interface *IPuzzleObjectMetaData* erlaubt es dem Entwickler benutzerdefinierte Daten an ein *PuzzleObject* Objekt anzuhängen. Diese Möglichkeit ist nützlich um gegebenenfalls direkte Referenzen zu Spielobjekten zu hinterlegen, oder um sonstige benötigte Daten mit diesem *PuzzleObject* Objekt zu verbinden.

**PuzzleNode.h** Die *PuzzleNode* ist das Baustück eines Puzzles. Diese werden ausschließlich vom *PuzzleGenerator* generiert und an das *Puzzle* Objekt übergeben. Eine *PuzzleNode* besteht immer aus einem *PuzzleObject* sowie einem Zielstatus des Typs *PuzzleState*. Außerdem speichert eine PuzzleNode dessen derzeitigen Status. Eine PuzzleNode kann *inaktiv*, *aktiv* oder *erledigt* sein. Wie weiter oben beschrieben, werden Events vom *Puzzle* Objekt verarbeitet. Das *Puzzle* Objekt bestimmt eine *PuzzleNode*, die in der Lage ist, das übergebene Event zu bearbeiten. Wie in Definition in Abschnitt 5.3 beschrieben, werden also nur *PuzzleNode* Objekte ausgesucht, die entweder aktiv oder erledigt sind. Wenn eine PuzzleNode ein solches Event bearbeitet, wird gegebenenfalls der Status des referenzierten *PuzzleObject* aktualisiert. Sind alle notwendigen PuzzleNode Objekte erledigt, so ist das Puzzle selbst auch erledigt.

**PuzzleUpdateListener.h** Das Interface *PuzzleUpdateListener* bietet dem Benutzer die Möglichkeit, über Änderungen der Stati von PuzzleNode Objekten

informiert zu werden. So werden alle Statusänderungen durch das Bearbeiten eines Events vom Puzzle an das verwendete *PuzzleUpdateListener* Objekt übergeben. Dieses Interface bietet zum Zeitpunkt dieser Arbeit folgende Methoden:

**onPuzzleComplete()** Wird aufgerufen, sobald das Puzzle abgeschlossen ist, also alle notwendigen PuzzleNode Objekte erledigt beziehungsweise erfüllt sind.

**onObjectStateChange(PuzzleObject* O)** Wird aufgerufen, sobald eine Statusänderung des PuzzleObject O hervorgerufen wurde (durch ein Event).

**onNodeActive(PuzzleNode* N)** Wird aufgerufen, sobald eine PuzzleNode N den eigenen Status auf aktiv setzt, also in Zukunft von auch Events bearbeiten kann.

**onNodeComplete(PuzzleNode* N)** Wird aufgerufen, sobald eine PuzzleNode N den eigenen Status auf erledigt setzt.

**onNodeIncomplete(PuzzleNode* N)** Wird aufgerufen, sobald eine PuzzleNode N den eigenen Status auf inaktiv setzt.

Jede dieser Funktionen kann dazu verwendet werden mit jeder Änderung im Puzzle, automatisiert Änderungen in der Spielwelt vorzunehmen. Der *PuzzleUpdateListener* ist also die Schnittstelle für Entwickler, um die Darstellung der Objekte in der Spielwelt mit dem Status im Puzzle selbst abzugleichen. Zum Beispiel könnte bei einem Schalter Objekt, welches den Status auf "Benutzt" setzt, die Darstellung des Schalters automatisiert auf eine andere Position (so zum Beispiel als benutzten Schalter) in der Spielwelt gesetzt werden. Eine Tür die geöffnet wurde, könnte mit dieser Schnittstelle in der Spielwelt als offene Tür gezeigt werden. Dies ermöglicht dem Benutzer eine prozedurale Aktualisierung der Darstellung in der Spielwelt.

**PuzzleRelation.h** Das *PuzzleRelation* Objekt beschreibt die Hierarchie zwischen den *PuzzleNode* Objekten. Diese wird einerseits vom PuzzleGenerator Objekt verwendet, um ein korrektes Puzzle zu generieren, als auch vom Puzzle Objekt selbst um die passende PuzzleNode zu finden, welche ein Event bearbeiten darf. Zum Beispiel dürfen nur solche PuzzleNode Objekte ein Event bearbeiten, wenn sie alle vorhergehenden PuzzleNode Objekte erledigt sind, und sie somit aktiv sind.

**PuzzleRandomizer.h** Das *PuzzleRandomizer* Objekt stellt verschiedene Funktionen zur Verfügung, um benötigte Zufallsfaktoren für die Generierung eines Puzzles zu berechnen.

**PUZZGEN_TYPES.h** Diese Header-Datei beschreibt lediglich einige benutzerdefinierte Typen für das Puzzle System. Die verwendeten Typen sind:

**T_PuzzleNodePair** Ein Paar zweier PuzzleNode* Referenzen. Diese werden hauptsächlich von der PuzzleRelation - Klasse verwendet.

**T_PuzzlePairList** Eine Liste von T_PuzzleNodePair

**T_PuzzleNodeList** Eine Liste von PuzzleNode* Referenzen

**T_PuzzleObjectList** Eine Liste von PuzzleObject* Referenzen

**T_PuzzleEventList** Eine Liste von PuzzleEvent* Referenzen

**T_PuzzleRuleList** Eine Liste von PuzzleRule Objekten

**T_PuzzleObjectStatePair** Ein Paar von PuzzleObject* Referenz und PuzzleState* Referenz.

## 6.2 Beispielanwendung der Implementation

Um die Verwendung des Systems deutlicher darstellen zu können, werde ich in diesem Abschnitt ein kleines Anwendungsbeispiel präsentieren. Zuerst wird eine Anwendung als C++ - Konsolenprogramm vorgestellt. Danach wird die Einbindung und Anwendung in der Unreal Engine 4 näher erläutert.

### 6.2.1 Beispielanwendung in C++

Zuerst muss die Bibliothek des *PuzzGen* Systems (das in dieser Arbeit entwickelte System) als *Include*-Verzeichnis eingebunden werden. Je nach Entwicklungsumgebung kann sich dieser Schritt unterscheiden. Für dieses Beispiel wird *Visual Studio Express 2015 für Windows* verwendet. Dort kann das Include-Verzeichnis in den Eigenschaften des Projektes hinzugefügt werden. (Rechtsklick auf Projekt, und anschließend Eigenschaften wählen). Unter dem Reiter *C/C++ -¿ Allgemein* findet sich die Option für Zusätzliche Includeverzeichnisse.

Die benötigten Includes für das System selbst beschränken sich auf die Header-Datei *PuzzGen.h*. Diese inkludiert alle restlichen benötigten Header-Dateien. Um eine textuelle Darstellung des derzeitigen Puzzle-Fortschrittes zu erhalten, werden wir auch noch den *iostream* für Ein- und Ausgabe inkludieren.

```
1 #include <iostream>
2 #include "PuzzGen.h"
```

Zuerst werden wir die benötigten Variablen definieren.

```
1 Puzzle* P;
2 T_PuzzleObjectList objects;
3 T_PuzzleEventList events;
4 T_PuzzleRuleList rules;
5 PuzzleGenerator* puzzGen;
```

Anschließend werden die Objekte, Events, und Statusübergänge definiert und ein Puzzle generiert.

```
int main() {

  /* Erstelle Objekt Schalter*/
  PuzzleObject* Schalter = new PuzzleObject("Schalter");

  /* Die öMglichen Stati unseren neuen Schalter Objektes*/
  PuzzleState* S_Unbenutzt = new PuzzleState("Unbenutzt");
  PuzzleState* S_Benutzt = new PuzzleState("Benutzt");

  /* Setze den Anfangsstatus des Objektes */
  Schalter->setCurrentState(*S_Unbenutzt);

  /* State Transition definieren üfr Schalter
  Event E_BenutzeSchalter ändert den Status des Objektes von
      S_Unbenutzt auf S_Benutzt
  */
  StateTransition* T_Schalter = new StateTransition();
  T_Schalter->addTransition("E_BenutzeSchalter",
      *S_Unbenutzt, *S_Benutzt);
  Schalter->setStateTransition(*T_Schalter);

  /* Das Event E_BenutzeSchalter exisitert noch nicht, also
      ümssen wir es erstellen
  Der Name des Events ist üfr die State Transition
      notwendig. Events werden aktuell über deren Namen
      identifiziert.
  */
  PuzzleEvent* E_BenutzeSchalter = new
      PuzzleEvent("E_BenutzeSchalter", Schalter);

  /*
  Unser PuzzleGenerator öbentigt alle definierten Events,
      Objekte und Regeln.
  Also ümssen wir unser Schalterobjekt und das Event zu den
      Listen ühinzufgen.
  */
  objects.push_back(Schalter);
  events.push_back(E_BenutzeSchalter);

  /* Jetzt öknnen wir ein Puzzle generieren lassen. Wir
      haben jedoch nur ein Objekt,
  jedoch öknnen wir so üüberprfen, dass es funktioniert hat.
  */
  puzzGen = new PuzzleGenerator();
  P = puzzGen->generatePuzzle(objects, events, rules);

  /* Das Puzzle Objekt und das PuzzleObject beinhaltet eine
```

```
      toString methode, um das aktuelle
39    Puzzle auszugeben.
40    */
41    std::cout << Schalter->getTextualRepresentation() <<
          std::endl;
42    std::cout << P->getSimpleTextualRepresentation() <<
          std::endl;
43
44    /* Warten auf Input. */
45    int inputVar;
46    std::cin >> inputVar;
47
48    return 0;
49  }
```

Ausschnitt 6.7:

```
Gameobject: Schalter
 > StateTransition:
> E_BenutzeSchalter: <Unbenutzt -> Benutzt>
 > CurrentState:
  - Unbenutzt
 > Reachable States:
  - Benutzt

<<< Puzzle >>>
>> Puzzlenode (Schalter | Benutzt) :: ACTIVE <>

$$$ Relation $$$
<<<<>>>>
```

In der Ausgabe in der Konsole sehen wir dann Informationen über unser Schalter Objekt, sowie über das Puzzle. Es wurde eine PuzzleNode generiert. (Schalter—Benutzt) bedeutet, dass der Zielstatus dieser PuzzleNode der Status "Benutzt" ist. Die Node ist aktiv, also ist sie in der Lage, Events zu bearbeiten. Die Relation ist derzeit leer, da wir nur ein Objekt angegeben haben. Es kann ganz einfach ein zweites Objekt hinzugefügt werden.

```
1   PuzzleObject* Tuer = new PuzzleObject("Tuer");
2   PuzzleState* S_Offen = new PuzzleState("Offen");
3   PuzzleState* S_Geschlossen = new
        PuzzleState("Geschlossen");
4   Tuer->setCurrentState(*S_Geschlossen);
5   StateTransition* T_Tuer = new StateTransition();
6   T_Tuer->addTransition("E_OeffneTuer", *S_Geschlossen,
        *S_Offen);
7   Tuer->setStateTransition(*T_Tuer);
8   PuzzleEvent* E_OeffneTuer = new
        PuzzleEvent("E_OeffneTuer", Tuer);
9   objects.push_back(Tuer);
10  events.push_back(E_OeffneTuer);
```

Sobald das Tür-Objekt hinzugefügt wurde, sehen wir diese Ausgabe.

Ausschnitt 6.9:

```
Gameobject: Schalter
 > StateTransition:
> E_BenutzeSchalter: <Unbenutzt -> Benutzt>
 > CurrentState:
  - Unbenutzt
 > Reachable States:
  - Benutzt
```

```
<<< Puzzle >>>
>> Puzzlenode (Schalter | Benutzt) :: ACTIVE <>

>> Puzzlenode (Tuer | Offen) :: INCOMPLETE <>

$$$ Relation $$$
[Schalter::Benutzt] <<< [Tuer::Offen]
<<<<>>>>
```

Wie zu sehen ist, wurden nun 2 PuzzleNode Objekte generiert. Aus diesen 2 PuzzleNode Objekten wurde eine Relation generiert, welche definiert, dass das Objekt Schalter den Status "benutzt" haben muss, bevor die PuzzleNode mit dem Objekt Tuer ein Event bearbeiten kann (da sie INCOMPLETE, also inaktiv ist), und damit den Status "offen" erreichen kann. In einem Spiel würde das bedeuten, dass der Spieler zuerst den Schalter benutzen muss, ehe er die Tür öffnen kann.

Zum Zeitpunkt dieser Arbeit werden vom PuzzleGenerator Objekt genau so viele Nodes erstellt wie die Anzahl der übergebenen Objekte. Durch die Anpassung dieser Zahl können auch PuzzleNodes generiert werden, die doppelt verwendet werden, um zum Beispiel Wiederholungen von Spielaktionen zu erlauben. Ein Beispiel für solch eine Wiederholung wäre das Aktivieren eines Schalters, das Öffnen einer Tür, gefolgt vom wiederbetätigen des ersten Schalters und dem Öffnen einer zweiten Türe.

Natürlich können über den selben Weg mehrere Objekte hinzugefügt werden. Wenn wir ein drittes Objekt hinzufügen, erhöht sich natürlich auch die Anzahl der Möglichen Ergebnisse. Mit dem folgenden Codebeispiel können wir ein weiteres Objekt erstellen - in diesem Fall einen Schlüssel.

```
1   PuzzleObject* Schluessel = new PuzzleObject("Schluessel");
2   PuzzleState* S_NichtAufgehoben = new
        PuzzleState("NichtAufgehoben");
3   PuzzleState* S_Aufgehoben = new PuzzleState("Aufgehoben");
4   Schluessel->setCurrentState(*S_NichtAufgehoben);
5   StateTransition* T_Schluessel = new StateTransition();
6   T_Schluessel->addTransition("E_NimmSchluessel",
        *S_NichtAufgehoben, *S_Aufgehoben);
7   Schluessel->setStateTransition(*T_Schluessel);
8   PuzzleEvent* E_NimmSchluessel = new
        PuzzleEvent("E_NimmSchluessel", Schluessel);
9   objects.push_back(Schluessel);
10  events.push_back(E_NimmSchluessel);
```

Ab diesem Punkt können sich die Ergebnisse natürlich unterscheiden, da die Generierung der Relation zufallsbasiert ist. Jedoch sollte das Ergebnis in etwa so aussehen:

Ausschnitt 6.11:

```
<<< Puzzle >>>
>> Puzzlenode (Tuer | Offen) :: ACTIVE <>

>> Puzzlenode (Schalter | Benutzt) :: INCOMPLETE <>

>> Puzzlenode (Schluessel | Aufgehoben) :: INCOMPLETE <>

$$$ Relation $$$
[Tuer::Offen] <<< [Schalter::Benutzt]
[Schalter::Benutzt] <<< [Schluessel::Aufgehoben]
<<<<>>>>
```

Damit nun Events auf dieses Puzzle ausgeführt werden können, werden wir eine simple Eingabemechanik erstellen. Es sollen die möglichen Events in einer Liste dargestellt werden, und als Auswahlmöglichkeit zur Verfügung stehen. Der neue Status des Puzzles wird nach einem Event erneut ausgegeben. Mit der Eingabe 0 lässt sich das Programm schließen, mit Eingaben ¿0 kann ein Event ausgewählt werden, welches ausgeführt werden soll.

```
1  /* Simple Eingabefunktion zum Testen der Events */
2    int exitProgram = 0;
3    /* Wiederhole solange keine 0 eingegeben wurde */
4    while (!exitProgram) {
5      system("cls"); // öLschen des Konsolen screens
6      std::cout << P->getSimpleTextualRepresentation() <<
           std::endl; // Simple Ausgabe des derzeitigen Puzzle
           Statuses
7      /* Ausgabe der ömglichen Events in einer nummerierten
           Liste*/
8      std::cout << "<<<␣Eingabe␣>>>" << std::endl;
9      int i = 1;
10     for (T_PuzzleEventList::iterator it = events.begin();
           it != events.end(); ++it) {
11       PuzzleEvent *e = *it;
12       printf_s(">%d:␣%s::%s␣\n", i++,
             e->getEventName().c_str(),
             e->getRelatedObject()->getObjectName().c_str(),
             stdout);
13     }
14
15     /* Lesen des Input */
16     int inputInt = NULL;
17     char inputChar;
```

```
18
19      std::cin >> inputChar;
20      inputInt = inputChar - '0';
21
22      /* Auswerten der Eingabe. Eingabe unter 1 ßschliet das
            Programm, ansonsten wird das Event üausgefhrt*/
23      if (inputInt < 1) {
24        exitProgram = 1;
25      }
26      else {
27        if (inputInt <= events.size()) {
28          P->handleEvent(*(events.at(inputInt - 1)));
29        }
30        else {
31          exitProgram = 1;
32        }
33      }
34    }
```

Diese simple Eingabemethode ist prinzipiell bereits ein einfaches Spiel. Die Aktionen des Spielers beschränken sich hierbei jedoch auf die direkte Eingabe des Events, welches sie ausführen wollen. Dieses kleine Beispiel zeigt jedoch sehr gut, wie sich Events mit Aktionen des Spielers verknüpfen lassen, und sich somit der Status des Puzzles ändert.

Ausschnitt 6.13: Die textuelle Ausgabe eines Puzzles, nachdem das Event E_BenutzeSchalter per Benutzereingabe ausgeführt wurde.

```
<<< Puzzle >>>
>> Puzzlenode (Schalter | Benutzt) :: COMPLETED <>

>> Puzzlenode (Schluessel | Aufgehoben) :: ACTIVE <>

$$$ Relation $$$
[Schalter::Benutzt] <<< [Schluessel::Aufgehoben]
<<<<>>>>

<<< Eingabe >>>
>1: E_BenutzeSchalter::Schalter
>2: E_OeffneTuer::Tuer
>3: E_NimmSchluessel::Schluessel
```

Die textuelle Repräsentation des Puzzles, welche in der Konsole ausgegeben wird, stellt somit die Spielwelt dar. Diese ändert sich abhängig vom derzeitigen Stand des Puzzles, und den ausgeführten Events. Die Textausgabe stammt jedoch direkt vom System. Um eine benutzerdefinierte Anbindung an Änderungen im Puzzle zu realisieren, und somit prozedurale Änderungen an der Spielwelt zu ermöglichen, muss der sogenannte *PuzzleUpdateListener* verwendet werden. Dieses Interface ermöglicht eine benutzerdefinierte Implementation von *Callbacks*,

die vom Puzzle aufgerufen werden. Diese dienen als Schnittstelle zwischen dem Puzzle-System und der Spielwelt.

### 6.2.2 Beispielanwendung in Unreal Engine 4

Der in dieser Arbeit entwickelte Puzzle Generator wurde in C++ geschrieben, und ist somit für die Unreal Engine 4 perfekt geeignet. In diesem Abschnitt möchte ich eine Kurze Anleitung zeigen, wie das System in die Unreal Engine 4 eingebunden werden kann.
Zuerst muss ein C++ Projekt angelegt werden. In diesem Beispiel werde ich das von der Engine angebotene First-Person Template verwenden.
Zuerst sollte im Projektordner die benötigte Ordnerstruktur angelegt werden. Im Hauptverzeichnis des Projektes (in welchem auch die Projektdatei mit der Endung .uproject liegt) legen wir einen neuen Ordner für Drittanbietersoftware an. In diesem Beispiel wird dieser *ThirdParty* heißen. Die benötigten Dateien des Systems werden sich in diesem Ordner befinden. Im Ordner *ThirdParty* legen wir einen neuen Ordner mit dem Namen *PuzzGen* an. Weiters legen wir in diesem wiederum die Ordner *Includes* und *Libraries* an. Damit ist die benötigte Ordnerstruktur fertig.
In den *Includes* Ordner werden alle Header-Dateien des Systems eingefügt, wie zum Beispiel *PuzzGen.h* und *PuzzleObject.h*. Natürlich werden an dieser Stelle alle benötigt. In den *Libraries* Ordner müssen wir die Bibliotheksdateien ablegen. Diese heißen *PuzzGen.lib* und *PuzzGen.x64.lib*. Damit ist die erste Vorbereitung erledigt.
Nun müssen wir in der Build-datei des Projektes noch definieren, welche Dateien die Unreal Engine laden sollte. Hierzu öffnen wir die Datei Source/ProjektName/ProjektName.Build.cs.

Am Beginn der Datei sollte folgende Zeile eingefügt werden:

Ausschnitt 6.14:

```
using System.IO;
using UnrealBuildTool;
```

Weiters benötigen wir die folgenden Funktionen:

```
1  private string ModulePath
2  {
3      get { return ModuleDirectory; }
4  }
5
6  private string ThirdPartyPath
7  {
8      get { return Path.GetFullPath(Path.Combine(ModulePath,
           "../../ThirdParty/")); }
9  }
10
```

```
11 public bool LoadPuzzGen(ReadOnlyTargetRules Target)
12 {
13     bool isLibrarySupported = false;
14
15     if ((Target.Platform == UnrealTargetPlatform.Win64) ||
           (Target.Platform == UnrealTargetPlatform.Win32))
16     {
17         isLibrarySupported = true;
18         string PlatformString = (Target.Platform ==
               UnrealTargetPlatform.Win64) ? "x64" : "x86";
19         string LibrariesPath = Path.Combine(ThirdPartyPath,
               "PuzzGen", "Libraries");
20         PublicAdditionalLibraries.Add(Path.Combine(LibrariesPath,
               "PuzzGen." + PlatformString + ".lib"));
21     }
22
23     if (isLibrarySupported)
24     {
25         // Include path
26         PublicIncludePaths.Add(Path.Combine(ThirdPartyPath,
               "PuzzGen", "Includes"));
27     }
28
29     Definitions.Add(string.Format("WITH_PUZZGEN_BINDING={0}",
           isLibrarySupported ? 1 : 0));
30
31     return isLibrarySupported;
32 }
```

Sollte sich die zu Beginn angelegte Ordnerstruktur von dieser Unterscheiden, so müssen die Pfade in diesen Funktionen angepasst werden. Weiters müssen wir in der Konstruktorfunktion (die nach dem Projekt benannt ist) zwei Aufrufe hinzufügen. Die Funktion sollte dann in etwa so aussehen:

```
1
2   public PuzzleGame(ReadOnlyTargetRules Target) :
        base(Target)
3   {
4     PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
5
6     PublicDependencyModuleNames.AddRange(new string[] {
          "Core", "CoreUObject", "Engine", "InputCore",
          "HeadMountedDisplay" });
7
8     LoadPuzzGen(Target);
9   }
```

Nun sollte es möglich sein, in der Unreal Engine die nötigen Funktionen zum generieren von Puzzles zu verwenden.

# 7 Ergebnisse

Die hier präsentierten Ergebnisse basieren auf dem Testuniversum, welches in dem Beispielprogramm erstellt wurde. Das Universum besteht aus folgenden Spielobjekten:

**Schalter** Ein Schalter, der Benutzt oder Unbenutzt sein kann, und auf das Event E_BenutzeSchalter reagiert

**LichtSchalter** Ein Lichtschalter, der Ein- oder Ausgeschalten sein kann, und auf das Event E_SchalteEin reagiert

**Tuer** Eine Tür, die Offen oder Geschlossen sein kann, und auf das Event E_OeffneTuer reagiert

**Schluessel** Ein Schlüssel, der Aufgehoben oder NichtAufgehoben sein kann, und auf das Event E_NimmSchluessel reagiert.

*Die Events sind in der "StateTransition" als Zwei-Weg-Transitionen definiert. Zum Beispiel führt Event E_SchalteEin zum Statuswechsel Eingeschalten ↔ Ausgeschalten.*

Ausschnitt 7.1:

```
<<< Puzzle >>>
>> Puzzlenode (LichtSchalter | Eingeschalten) :: ACTIVE <>

>> Puzzlenode (Schalter | Benutzt) :: INCOMPLETE <>

>> Puzzlenode (LichtSchalter | Ausgeschalten) :: INCOMPLETE
    <>

$$$ Relation $$$
[LichtSchalter::Eingeschalten] <<< [Schalter::Benutzt]
[Schalter::Benutzt] <<< [LichtSchalter::Ausgeschalten]
<<<<>>>>

<<< Eingabe >>>
>1: E_BenutzeSchalter::Schalter
>2: E_OeffneTuer::Tuer
>3: E_NimmSchluessel::Schluessel
>4: E_SchalteEin::LichtSchalter
```

Ausschnitt 7.1 zeigt ein generiertes Puzzle mit 3 Nodes, und eine Relation der Länge 3. Die Lösung für dieses Puzzle ist: E_SchalteEin → E_BenutzeSchalter → E_SchalteEin.

Ausschnitt 7.2:

```
<<< Puzzle >>>
>> Puzzlenode (Schluessel | Aufgehoben) :: ACTIVE <>

>> Puzzlenode (Schluessel | Aufgehoben) :: ACTIVE <>

>> Puzzlenode (LichtSchalter | Eingeschalten) :: INCOMPLETE
    <>

>> Puzzlenode (Tuer | Offen) :: INCOMPLETE <>

$$$ Relation $$$
[Schluessel::Aufgehoben] <<< [Tuer::Offen]
[Schluessel::Aufgehoben] <<< [LichtSchalter::Eingeschalten]
[LichtSchalter::Eingeschalten] <<< [Tuer::Offen]
<<<<>>>>

<<< Eingabe >>>
>1: E_BenutzeSchalter::Schalter
>2: E_OeffneTuer::Tuer
>3: E_NimmSchluessel::Schluessel
>4: E_SchalteEin::LichtSchalter
```

Ausschnitt 7.2 zeigt ein Ergebnis mit 4 Nodes, wobei zwei davon identisch generiert wurden. Die Node mit (Schluessel, Aufgehoben) wurde zweimal generiert. Jedoch bieten beide Nodes verschiedene Abhängigkeiten. In der Relation sieht man, dass zum Öffnen der Tür, zuerst der Schlüssel aufgehoben sein muss. Zusätzlich muss jedoch auch der LichtSchalter eingeschalten sein. Die Lösung für dieses Puzzle wäre somit: E_NimmSchluessel → E_SchalteEin → E_OeffneTuer.

Ausschnitt 7.3:

```
<<< Puzzle >>>
>> Puzzlenode (Tuer | Offen) :: INCOMPLETE <>

>> Puzzlenode (Schalter | Benutzt) :: ACTIVE <>

>> Puzzlenode (Schluessel | Aufgehoben) :: INCOMPLETE <>

>> Puzzlenode (Schalter | Benutzt) :: INCOMPLETE <>

$$$ Relation $$$
[Tuer::Offen] <<< [Schalter::Benutzt]
[Schalter::Benutzt] <<< [Schluessel::Aufgehoben]
[Schluessel::Aufgehoben] <<< [Tuer::Offen]
<<<<>>>>

<<< Eingabe >>>
>1: E_BenutzeSchalter::Schalter
>2: E_OeffneTuer::Tuer
```

```
>3: E_NimmSchluessel::Schluessel
>4: E_SchalteEin::LichtSchalter
```

Ausschnitt 7.3 zeigt ein Ergebnis mit 4 Nodes, wobei die 2 Nodes mit (Schalter, Benutzt) identisch generiert wurden, anders als in Ausschnitt 7.2 jedoch vollkommen verschiedene Positionen in der Relation haben. Die Relation selbst stellt den Eindruck eines Zyklus dar. Tatsächlich ist dieses Puzzle jedoch lösbar, da zwei verschiedene Nodes beinhaltet sind. Die Lösung für dieses Puzzle ist: E_BenutzeSchalter → E_NimmSchluessel → E_OeffneTuer → E_BenutzeSchalter → E_BenutzeSchalter. Der Schalter muss am Ende zwei mal benutzt werden, damit dieser den Status Ausgeschalten, und danach wieder Eingeschalten erhält. Dieses Ergebnis zeigt die Problematik mit der aktuellen Sinnhaftigkeit der Ergebnisse relativ gut. Ein solches Puzzle *kann* Sinn machen, ist jedoch für die direkte Auffassung unlogisch. Diese Ergebnisse könnten, wie in Sektion 8 zur weiterführenden Arbeit näher beschrieben, optimiert werden.

Ausschnitt 7.4:

```
<<< Puzzle >>>
>> Puzzlenode (Schluessel | Aufgehoben) :: INCOMPLETE <>

>> Puzzlenode (LichtSchalter | Eingeschalten) :: ACTIVE <>

>> Puzzlenode (Tuer | Offen) :: INCOMPLETE <>

$$$ Relation $$$
[Schluessel::Aufgehoben] <<< [Tuer::Offen]
[LichtSchalter::Eingeschalten] <<< [Schluessel::Aufgehoben]
<<<<>>>>

<<< Eingabe >>>
>1: E_BenutzeSchalter::Schalter
>2: E_OeffneTuer::Tuer
>3: E_NimmSchluessel::Schluessel
>4: E_SchalteEin::LichtSchalter
```

Ausschnitt 7.4 zeigt ein Ergebnis mit 3 Nodes. Die Lösung für dieses Puzzle wäre wie folgt: E_SchalteEin → E_NimmSchluessel → E_OeffneTuer.

# 8 Mögliche Weiterführung der Arbeit

Die in dieser Arbeit präsentierte Implementation ist noch eine rudimentäre Umsetzung. Jedoch erfüllt es die grundlegenden Voraussetzungen an ein solches System. Dieses Basissystem könnte noch mit einer besseren Benutzbarkeit, mehreren Möglichkeiten für Regeln zur Einschränkung der generierten Puzzles, oder einen besseren Algorithmus zur zufälligen Generation umgesetzt werden. Außerdem kann auf das existierende System weiter aufgebaut werden. Möglichkeiten wären zum Beispiel eine direkte Einbindung in prozedurale Weltgeneratoren.

Weiters könnte noch die Kompatibilität mit weiteren bekannten Spieleengines getestet und umgesetzt werden, wie zum Beispiel mit Unity Engine, CryTek Engine oder GameMaker Studio.

Das System kann um die Möglichkeit eines automatischen Puzzlefortschrittes erweitert werden. Beispielsweise könnte das betätigen eines Schalters gleichzeitig das Öffnen einer Tür hervorrufen, ohne dass der Spieler dafür eine eigene Aktion tätigen muss. Dies würde dem Benutzer dynamischere Lösungen bieten und dem Spieler eine interessantere Entwicklung des Spieles vorlegen.

Der Aufwand zum Einrichten des Systems ist durch das Definieren der Objekte, der Events und Regeln immer noch relativ hoch. Eine Möglichkeit diesen Aufwand zu reduzieren wäre ein erweitertes System, welches aus einer Menge an vordefinierten Objekten eine gewisse Anzahl auswählt, und diese in das Puzzle aufnimmt. Dieses System könnte ebenfalls mit verschiedenen Datenstrukturen für solche Objektmengen, wie zum Beispiel CSV Dateien, sowie einer Benutzeroberfläche zum einfachen Erstellen solcher Listen, erweitert werden.

Weiters können natürlich die Algorithmen, die zur Generierung von den Puzzles derzeit verwendet werden, erweitert und verbessert werden, um die Ergebnisse möglichst zu verbessern um dem Benutzer noch mehr Möglichkeiten zur Anpassung der Lösungsmenge zu geben.

Außerdem zeigt sich die Interpretation des Puzzlesystems in die Spielwelt immer noch als sehr aufwändig und schwierig. Eine passende Interpretation wird über den Nutzen eines prozedural generierten Puzzles entscheiden. Das System könnte somit mit Tools erweitert werden, die eine einfachere Implementation einer Interpretation für den Benutzer ermöglicht und somit bessere Ergebnisse produzieren kann.

# 9 Konklusion

Während das System lösbare Puzzles generiert, ist der Algorithmus selbst zur prozeduralen Generation noch weiter ausbaufähig. Die derzeit gelieferten Ergebnisse folgen nicht immer einer logischen Struktur. Auch wenn die Ergebnisse mit einer passenden Interpretation und Einbindung in eine Spielwelt sinnvoll präsentiert werden können, so ist damit mehr Implementierungsaufwand für den Spieleentwickler nötig. Das System kann jedoch eine Möglichkeit liefern, prozeduralen Inhalt in Spielen diverser zu machen. Wenn die in dieser Arbeit präsentierte Methode mit schon bestehenden Algorithmen zur Generation von Welten und Umgebungen gekoppelt werden kann, so können interessante Ergebnisse erwartet werden. Trotzdem bleibt der Implementierungsaufwand für den Spieleentwickler noch relativ groß. Die Interpretation der Ergebnisse wird ausschlaggebend für die Sinnhaftigkeit und Spielbarkeit der generierten Puzzles sein. Da Puzzles und Rätsel von Spielern hohe Anforderungen erfüllen müssen, könnte dieses System jedoch besser zur Generation von Aufgaben für Spieler verwendet werden. Viele sogenannte "Quests" folgen in Spielen einem Aufbau, der mit diesem System abgebildet werden kann. Da der Fortschritt im Puzzle automatisch verfolgt, und dem Spieleentwickler zur Laufzeit mitgeteilt wird, ist

der Implementierungsaufwand für das System in diesem Kontext kleiner. Zum Beispiel das Sammeln von Gegenständen oder das Besiegen von gegnerischen Einheiten könnte in dem System prozedural in eine Reihenfolge erstellt werden und dem Spieler textuell dargestellt werden. Der Fortschritt der Aufgabe wird dann vom System selbst festgehalten und für den Spieleentwickler bereitgestellt. Wie groß der Nutzen des Systems tatsächlich ist, muss durch genaue Evaluierung und durch Feedback von Spieleentwicklern ermittelt werden.

# Literatur


[1] Katherine E Buckley and Craig A Anderson. A theoretical model of the effects and consequences of playing video games. *Playing video games: Motives, responses, and consequences*, pages 363–378, 2006.

[2] Clara Fernández-Vara and Alec Thomson. Procedural generation of narrative puzzles in adventure games: The puzzle-dice system. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, page 12. ACM, 2012.

[3] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1, 2013.

[4] Wikipedia. No man's sky — wikipedia, the free encyclopedia, 2017. [Online; accessed 25-August-2017 ].

[5] Wikipedia. Rogue (video game) — wikipedia, the free encyclopedia, 2017. [Online; accessed 25-August-2017 ].