# Einleitung

# Vision

Am Anfang des Projekts wurde in Form eines Projektbeschriebs dargelegt, wie die Vorstellung des Projektteams von dem Endprodukt aussieht. Dabei wurde zuerst die Abgrenzung eines Adventure Games gegenüber anderen Spielgenres versucht, damit die Projektvorgabe „Adventure Game“ eingehalten wird. Das Adventure Game, wurde recherchiert, zeichnet sich dadurch aus, dass es von der Story lebt. Der Schwerpunkt liegt nicht auf Aufregung und Geschicklichkeit wie beispielsweise bei einem Action Game, sondern auf Spannung und Erlebnis. Es kann zwar auch andere Elemente zur Abwechslung enthalten und Teile haben, die auch in ein Action Game passen würden, aber die Geschichte oder Puzzles machen den wichtigsten Teil aus. Oft könnte ein Adventure Game mit einem Film verglichen werden, bei dem der Spieler die verschiedenen Schauplätze aufsucht und dort meist nach dem Lösen von Rätseln die Fortsetzung erfährt. Das Spiel muss dabei nicht linear sein, sondern kann sich auch leistungsbedingt entwickeln; schneidet der Spieler bei Rätseln nicht gut ab, oder wählt er bei Dialogoptionen die unfreundlichen Antworten, so kann die Geschichte manchmal „weniger gut“ enden.

Auf dieser Grundlage wurde eine Spielidee entwickelt, die diese Elemente enthält und eine Geschichte wurde formuliert. „Escape“ sollte das Spiel heissen und in etwa den real-life-Spielen „Adventure Room“ oder auch „Mystery Room“ entsprechen. Diese Spiele setzen ein Konzept um, bei dem mehrere Personen zusammen in einem Raum eingesperrt werden und versuchen müssen, innerhalb einer vorgegebenen Zeitspanne den Weg heraus zu finden. Dabei ist meist nicht das Finden des Wegs das Problem, sondern die Hindernisse in Form von Schlössern, die durch das Lösen von Rätseln geöffnet werden können. Dieses Spiel versprach einiges an Spannung und einen Aufbau, der modular ist und dem Projektteam einige Freiheiten erlauben sollte. Zusätzlich wurde die folgende Geschichte formuliert, damit auch das Element der Story nicht zu kurz kommt:

In einer Fabrik werden künstliche Intelligenzen programmiert und in einem Test-Haus überprüft und ihre Qualität getestet. Dafür wird die K.I. auf die Festplatte verschiedener Körper geladen, deren Kontrolle sie damit übernimmt. So kann sie einmal als Kugel und einmal als Roboter die Rätsel lösen. Labyrinthe, Denkaufgaben, typische Frage-Antwort-Spiele und viele andere Aufgaben soll die K.I. im Rahmen des Tests lösen. Je nachdem, wie gut ihr, oder besser gesagt dem Spieler, das gelingt, wird sie bewertet. Die Bewertung basiert je nach Rätsel auf die Schnelligkeit, Vollständigkeit oder Effizienz mit der es gelöst wird. Jedes Level wird also bewertet und aus allen Bewertungen zusammen ergibt sich ein ständig angepasster Index-Wert, der das Ende der Geschichte beeinflusst. Spielern, die nur einen tiefen Index-Wert erreicht haben, wird eröffnet, dass sie höchstens als schlechtes Beispiel in der Theorie auftauchen werden. Mittelgute Ergebnisse führen dazu, dass die K.I. zwar als akzeptabel eingestuft wird, aber nie produktiv eingesetzt, sondern höchstens als Grundlage für verbesserte Weiterentwicklungen verwendet wird. Schneiden Spieler und K.I. besonders gut ab, so wird ihnen eine glorreiche Zukunft als produktiv eingesetztes Programm prophezeit.

# Änderung an der Vision

Anfang des Semesters wurde als Eigenheit eines Adventure Games die Story hervorgehoben, da diese vielleicht das wesentliche Unterscheidungsmerkmal von diesem Genre zu anderen ist. Auch zum Beispiel ein Action Game hat eine Story, aber bei einem Adventure Game ist sie der Mittelpunkt. Es geht darum, dass der Spieler sich in einer Geschichte wiederfindet und durch das Spielen herausfinden kann, wie diese ausgeht. Vielleicht kann sie sich sogar je nach Spielweise oder bestimmten Umständen unterschiedlich entwickeln und gibt dem Spieler damit die Motivation, das Spiel mehrmals zu spielen und dabei unterschiedliche Entwicklungen zu provozieren. Das Projektteam hatte ein solches Spiel vor Augen, weil dies ein normales Adventure Game zu einem qualitativ hochwertigen Spiel machen kann. Es war geplant, dass die Story nach einer anfänglichen Einarbeitungsphase, in welcher das Team ein gutes Verständnis für die Unity-Entwicklungsumgebung erreicht, detailliert umgesetzt wird. Jedoch hat sich gezeigt, dass Unity noch mehr Funktionalitäten und Möglichkeiten bietet als angenommen; die Einarbeitungsphase liesse sich noch einige Monate verlängern. Somit blieb es bei der anfänglich gesketchten Story und sie wurde nicht weiter ausgebaut, sondern um ein mögliches Ende gekürzt. Da es sich um ein Modul handelte, das nebst der Projekterfahrung viele Freiheiten liess, wurde auf das von Entwicklerseite her uninteressantere Thema verzichtet und ein Schwerpunkt auf die Vervollständigung des Spieles gelegt.

Als weiterer Unterhaltungsfaktor war geplant, dass der Spieler seine Abenteuer als K.I. in verschiedenen Körpern absolvieren kann. Die K.I. sollte pro Raum auf die Festplatte von verschiedenen Robotern oder elektrischen Gegenständen gespeichert werden und so einmal rollend und einmal laufend ein Level bestreiten. Dies wurde aus zwei Gründen nicht umgesetzt: erstens gibt es im Unity Asset Store allgemein nicht viele Spielfiguren und wenn, sind es keine elektrischen und sie sind meist nicht animiert. Zweitens ist es sehr aufwändig, eine eigene zu erstellen. Für eine graphisch und designerisch begabte Person wäre es eine Herausforderung gewesen, für das Projektteam eine ganz andere Kunst und ein ganz anderes Gebiet. Ein Teammitglied hat dann auch einige Zeit investiert und beschlossen, dass in diesem Projekt darauf verzichtet werden muss. Es schien, dass Monate investiert werden müssten, damit etwas Gutes dabei herauskommen würde. Jedoch konnte eine eigene Animation umgesetzt werden und so wurde zumindest ein im Unity Asset Store gratis herunterladbarer Charakter mit eigenen Bewegungen belebt.

Ein weiterer Punkt, der während der Entwicklung einige Änderungen erfahren hat, war die globale Einrichtung einer State Machine. Viel diskutiert und in jedem Komplexheitsgrade diskutiert, wurde beschlossen, diese einfach zu halten und erst einmal ein paar Räume zu gestalten, bevor man diverse Abhängigkeiten berücksichtigt. Angedacht war nämlich ein flexibles Spiel, bei dem der Spieler sich frei bewegen könnte und mehrere Räumen parallel gelöst hätte. Beim Erreichen eines Check Points, zum Beispiel beim Umlegen eines Schalters, hätte der aktuelle Raum seinen State gewechselt und damit in einem anderen Raum einen Fortschritt bewirkt. Statt dies zu planen, bevor überhaupt ein einziger Raum beendet ist, hat man die Priorität auf letzteres gelegt. Der grosse Vorteil bestand so in der unabhängigen Entwicklung der Räume; jedes Teammitglied konnte für sich arbeiten und musste sich noch keine Gedanken machen, ob Designentscheidungen für den eigenen Raum eine Konsequenz für einen anderen Raum mit sich ziehen könnte. Nachdem die Räume erste Formen angenommen hatten, wurde ein einfaches State Pattern umgesetzt. Dieses wird im nächsten Kapitel beschrieben.

[Irgendwo einbauen:] Das Ziel, das Spiel modular aufzubauen und beliebig erweiterbar zu machen, wurde jedoch gut erfüllt.

# Globale Designs

## Animation mit State-Maschine

Animationen in Unity funktionieren mit Hilfe von State Maschines. Man kann Animationen erstellen, welche für sich alleine stehen können. Möchte man jedoch, dass ein Objekt je nach seinem Zustand anders animiert wird, muss man vom „Animator-Tab“ in der Unity-Entwicklungsumgebung Gebrauch machen. Dort kann man seine Animationen hinzufügen und mit Status-Übergängen verbinden. Angestossen wird ein Übergang jeweils durch eine sogenannte „Condition“ (Voraussetzung).



Abbildung 1 Beispiel einer State- Maschine in Unity für die Animation des Protagonisten

Conditions verwenden Parameter, welche wiederum fix einem State-Diagramm zugeordnet sind. Diese Parameter sind dann von überall her änderbar. So kann mit der Änderung eines Parameters ein Zustandsübergang angestossen werden.

Ein konkretes Beispiel ist die Geh- und Stillstandanimation einer Spielfigur. Bewegt sich der Spieler, soll die Figur auch gehend aussehen. Steht sie still, soll sie nicht laufen, sondern stehen. Dafür werden also eine IDLE- und eine Gehend-Animation gemacht und einem State-Diagramm hinzugefügt. Anschliessend wird ein Parameter “Walking” definiert, welcher kontrolliert, in welchem Status sich das Diagramm befindet. Standardmässig hat Walking den Wert „false“. Wir definieren nun den Zustand „idle\_bad“ als Standardzustand und erstellen zwei Übergänge: Einer von idle\_bad nach walking2 und einer, der den umgekehrten Weg geht. Beide Übergänge haben eine entsprechende Condition, welche sicherstellt, dass, wenn der Spieler anhält oder losläuft, der Übergang eingeleitet wird.



Abbildung 2 Beispiel einer Übergangsbedingung

Im Verlauf des Projekts wurden drei Animationen erstellt, welche auch verwendet werden. Es war dem Team ein Anliegen, die Animationen der Objekte eigenhändig zu erstellen, da dies ein zentraler Punkt in der Spieleentwicklung ist. Sie mögen nicht immer so rund und perfekt aussehen wie in aktuellen XBOX-, PC oder Playstation-Spielen, doch sie sind mit Liebe und Sorgfalt kreiert und arrangiert worden.

[Michel] Vielleicht auch: Was ist eine Szene? Wie verbindet man mehrere? Etc.

## Spielstand

Wie vorhergehend in diesem Dokument bereits erklärt wurde, wurde auf eine globale State Maschine verzichtet. Dafür haben wir die Animationen genauer angeschaut und jeden Level in seine eigene State Maschine verwandelt. Der Status jedes Levels wird dann zusammen mit der aktuellen Punktzahl als „Spielstand“ bezeichnet. Dieses Paket kann dann auf die Festplatte gesichert und beim Starten des Spiels neu geladen werden.

Jeder Level hat folgende Zustände:

* Gruppierungs-State
  + abstract, nur zur Gruppierung der States nach Level benötigt
  + Erbt von „BaseState“
* NotAllowed-State
  + Level wird in der Levelübersicht nicht angezeigt
  + Level kann nicht besucht werden
* NotStarted-State
  + Level wird in Level-Übersicht gelb angezeigt
  + Level ist spielbar
  + Implementiert das Interface „LevelPlayable“
* Started-State
  + Level wird in Level-Übersicht gelb angezeigt
  + Level ist spielbar
  + Implementiert das Interface „LevelPlayable“
  + Beinhaltet möglicherweise Zusatzinformationen (z.B. Anzahl Versuche)
* Finished-State
  + Level wird in Level-Übersicht grün angezeigt
  + Level ist spielbar
  + Implementiert das Interface „LevelCompleted“

Zusätzlich dazu gibt es die bereits erwähnte abstrakte Klasse „BaseState“, von welcher alle States im Spiel erben. Die beiden Interfaces „LevelPlayable“ und „LevelCompleted“ dienen der Anzeige in der Level-Übersicht. So werden nur spielbare oder abgeschlossene Level überhaupt angezeigt.

Zustandsübergänge werden über statische Methodenaufrufe auf die Klasse „GameMemory“ erreicht.



Abbildung 3 Konkrete Abfolge der Stati innerhalb eines Raumes

# Probleme und Lösungen

Eine Vielzahl an Eigenheiten von Unity haben dem Projektteam Probleme bereitet. Der Kampf mit Kleinigkeiten hat daher verhindert, dass die erwartete grosse Begeisterung für die Entwicklungsumgebung aufkommen konnte. In den folgenden Abschnitten werden einige Probleme kurz erläutert.

## Bauen eines Raumes

Da das geplante Spiel aus mehreren Räumen bestehen würde, liegt es nahe, dass in einem ersten Schritt ein solcher erstellt werden sollte. Doch wie macht man eine Wand? Welches Game Objekt benutzt man für einen Boden? Antwort auf diese Fragen finden sich zwar schon auf Foren, aber zu Unity gibt es sonst hauptsächlich Tutorial-Videos. Die Plane als Boden und Quads als Wände werden anleitungsgemäss erstellt und bei der mühseligen Platzierung wird geflucht. Da die Game Objekte noch keine Collider haben und das Projektteam auch noch nicht genau weiss, wie man diese anwenden würde, muss von allen Seiten geschaut werden, dass die Wand keinen Abstand mehr vom Boden hat. Danach steht ein kahler Raum und der Gedanke lässt sich nicht verscheuchen, dass das auch einfacher hätte sein können.

Ein Raum ist kein Raum ohne eine Tür und allenfalls ein paar Fenster. Die Erwartung bestand, dass dies Standard ist. Denkt man sich eine Game-Entwicklungsumgebung, würde man erwarten, dass Räume leicht errichtet werden können. Einige Recherchen zeigten jedoch, dass nicht einfach Planes als Wände aufgestellt werden können und nachträglich Fenster hineingesetzt. Nein, ein Loch muss konstruiert werden, indem mehrere kleinere Planes rundum platziert werden.

## Importieren von Objekten

Eine weitere kleinere Hürde stellt sich, sobald ein Objekt, eine Textur, oder eben eine Spielfigur für das Game gesucht wird. Standardmässig enthält Unity keine Gegenstände, aber viele können zusätzlich heruntergeladen werden. Dazu muss der Asset Store gefunden und geöffnet werden. Sucht man in der Suche dort nach dem gewünschten Objekt, so findet sich sicherlich etwas Passendes. Allerdings meist nicht gratis. Da das Projektbudget stark beschränkt war, wurde beschlossen, auf die schönen Dinge zu verzichten und mit den kostenlosen zurechtzukommen.

Wird ein Objekt aus dem Asset Store heruntergeladen und dann in das Projekt importiert, kommt dieses nicht als einzelne Datei. Zu dem Modell selber gehören Texturen, Mesh und je nachdem noch zahlreiche andere Dateien. Wenn das Objekt so in die Szene übernommen werden soll, kann die Datei mit dem Play-Symbol (.fbx-Datei) auf das Spielfeld gezogen und dort platziert werden. Dafür wird am besten die Tastenkombination Shift und Control verwendet, damit die Objekte an Wand und Boden snappen und nicht millimetergenau gesetzt werden müssen (Dies funktioniert jedoch nur, wenn Wände und Boden bereits einen Collider zugeteilt erhalten haben. Siehe dazu das nächste Unterkapitel). Das Thema muss mit der Platzierung jedoch nicht abgeschlossen sein: möglicherweise muss nur ein Collider hinzugefügt werden und alles funktioniert, aber oft wurde das Objekt für eine älteren Unity-Version erstellt und es gibt Probleme. Es kann vorkommen, dass das Objekt gar nicht sichtbar ist. Öfter ereignete sich, dass zuerst alles zu klappen schien. Das Objekt hat sogar schon einen Collider und somit auch die Möglichkeit, dass der Charakter damit interagiert; aber es gibt keine Reaktion. Dies liegt daran, dass bei den meisten heruntergeladenen Objekten durch die verschiedenen Unity-Versionen eine Verschiebung der Collider stattfindet. Das Mesh befindet sich zwar auf dem Pult, aber der dazugehörige Collider ist weit hinten irgendwo im Nirgendwo. Sobald das Problem bekannt ist, lässt es sich leicht beheben, indem der Collider an die gleichen Koordinaten wie das dazugehörige Objekt verschoben wird.

## Kamerverfolgung der Spielfigur

In beinahe jedem Spiel ist es essentiell, dass der dargestellte Bildausschnitt auf die Spielfigur des Spielers fokussiert. Insbesondere bei 3D-Spielen ist aber die Kameraverfolgung der Spielfigur nicht immer ganz trivial.

Unity bietet aber für diese Art von Problemen gute Lösungen. Im einfachsten Fall macht man aus der Kamera, die die Spielfigur verfolgen soll ein «Kind»-Objekt, indem man die Kamera in der Unity-Hierarchy zu einem Unterelement der Spielfigur macht. Somit vollführt die Kamera die gleichen Transformationen wie die Spielfigur, ohne dass man eine Zeile Code schreiben muss.

Dieser Ansatz hat aber seine Limitationen. Soll z.B. eine rollende Kugel verfolgt werden, würde die Kamera die Rotation der Kugel ebenfalls durchführen und beim Spieler höchstens für Übelkeit sorgen. In diesem Fall kann ein Script die Kameraführung übernehmen.

public class CameraController : MonoBehaviour

{

public GameObject player;

private Vector3 offset;

void Start()

{

offset = transform.position - player.transform.position;

}

void LateUpdate()

{

transform.position = player.transform.position + offset;

}

}

Dieses Script speichert beim Start des Spiels die relative Position der Kamera zur Spielfigur als 3D-Vektor. Die überschriebene Methode «LateUpdate» holt die neue Position der Spielfigur und positioniert die Kamera relativ zur Spielfigur neu. Die Methode «LateUpdate» wird bewusst verwendet, damit die Kameraneupositionierung möglichst nach der Positionsänderung der Spielfigur durchgeführt wird.

## Navigation

Eine oft auftauchende Aufgabe in Spielen ist, dass man seine Spielfigur an einen bestimmten Ort auf dem Spielfeld hinbewegen will. Bei Computerspielen mit Maussteuerung hat es sich eingebürgert, dass man auf den Ort, wo sich die Spielfigur hinbewegen soll klickt, um die Figur zu veranlassen sich an diesen Ort zu bewegen.

Bei Unity wird diese Art der Maussteuerung standardmässig mit Raytracing und einem Navigation Agent gelöst, beides Komponeten, die Unity von Haus aus beinhaltet.

Der Navigation Agent kann einem beliebigen Game Object in Unity hinzugefügt werden. Die Komponente berechnet den Pfad für das Game Object und kann gegeben falls auch Hindernissen ausweichen, indem jeweils ein neuer Pfad berechnet wird. Um die Pfadberechnungen während der Laufzeit des Spiels zu vereinfachen, wird ein Navigation Mesh erzeugt. Der Navigation Mesh kennt alle statischen (d.h. zur Laufzeit nicht veränderlichen) Objekte im Spiel und beinhaltet alle «begehbaren» Flächen im Spiel. Mithilfe dieses Navigation Mesh können die verwendeten Navigation Agent sich auf dem Spielfeld orientieren.

Damit der Navigation Agent ein Game Object bewegt, muss ihm zuerst ein Zielpunkt angegeben werden. Dies kann mit Raytracing bewerkstelligt werden. Als erstes erstellt man dazu ein Ray-Objekt, das von der Kamera aus Richtung Mauszeiger zeigt. Dieses Ray-Objekt wird der Raytracing-Funktion von Unity mitgegeben. Raytracing verfolgt diesen «Strahl» und kontrolliert, ob es zu einer Kollision mit einem anderen Game Object kommt. Falls es eine solche Kollision gibt, können die Koordinaten ausgelesen werden und dem Navigation Agent übergeben werden. Der Navigation Agent berechnet dann mithilfe des Navigation Mesh den Pfad zum Zielpunkt und bewegt das Game Object dorthin.

## Kollisionen

Eine wichtige Mechanik in Computerspielen ist die Auswertung von Kollisionen zwischen Spielelementen, so sollen z.B. Münzen vom Spieler aufgelesen werden und Berührungen mit dem Gegner zu Schäden führen.

In Unity gibt es zur Auswertung von Kollisionen eine eigene Komponentengruppe, die verschiedene Formen von «Colliders» mitbringt. Diese Collider-Komponenten können den entsprechenden Game Object hinzugefügt werden.

Leider ist die Handhabung dieser Collider-Komponenten in Unity etwas umständlich gelöst worden. Zusätzlich zur Collider-Komponente wird ebenfalls die Rigidbody-Komponente benötigt. Der «Rigidbody» ist eigentlich für die Physiksimulation von Unity zuständig. Wenn man nur seine Fähigkeiten zur Auswertung von Kollisionen benötigt, müssen die Physikeigenschaften am Rigidbody abgeschaltet werden.

## Verwendung mehrerer Scenen

Die Spielwelt in Unity besteht aus einer oder mehrerer sogenannter «Scenes». Es ist naheliegend pro Level in einem Spiel eine eigene Scene zu verwenden. Umso erstaunlicher ist die unintuitive Handhabung des Scene-Wechsels in Unity. So werden die Scenes von Unity über eine Id verwaltet, die nur im «Build Settings»-Menü einsehbar ist. Diese Id kann auch nicht manuell gestzt werden, sondern wird durch die Reihenfolge der Scenes im «Build Settings»-Menü bestimmt!

## Script-Beispiel in eigenes Programm integrieren

Es gibt einen negativen und einen positiven Aspekt bei der Spieleentwicklung mit Unity. Wer in Unity etwas erstellen will, ist sicher nicht der Erste, der etwas in der Art und Weise macht. Auf der anderen Seite kann man davon ausgehen, dass jedes eigene Problem schon von einem Mitglied der grossen Unity-Community gelöst wurde. Das tönt also gut – ob in Stackoverflow oder auf der offiziellen Unity-Hilfe-Seite: Lösungen sind vorhanden. Das Problem ist es nun, diese Lösungen ins eigene Programm zu integrieren. Dabei gibt es verschiedenste Probleme.

### Verschiedene Unity-Versionen

Die Lösungen mögen noch so zahlreich zu finden sein, doch ein Grossteil ist in der aktuellen Unity-Version gar nicht mehr anwendbar. Die Umgebung scheint sich sehr schnell zu ändern und auch Lösungen, welche vor einem Jahr noch funktioniert haben, bringen heute nicht mehr die erwünschte Wirkung. So sucht man in der Regel eher lange in den zahlreichen Antworten.

### Aufbau

Der Aufbau der Applikationen in Unity ist nicht so, wie man es sich von gewöhnlichen Haus- und Büroanwendungen gewohnt ist. Es funktioniert nichts linear: Anfragen, Antworten und sonstige Events bewegen sich zwischen unzähligen Objekten und man fragt sich bei der Lösungsfindung des Öfteren, wo genau man die pfannenfertige Lösung eigentlich implementieren muss.

## Vergleich mit herkömmlichen Applikationen

Das Projektteam hatte vor dieser Applikation durchaus schon Programmiererfahrung, doch eine derartige Umgebung ist uns noch nicht untergekommen. Wir sind es uns gewohnt, dass die Dinge etwas linearer ablaufen als in Unity (spezielles C#). Um auf einem Objekt Methoden aufzurufen, braucht man in unserer Welt eine Referenz auf das Ziel. In Unity sind die Scripte in Gameobjects eingepackt und es reicht vollends, dieses zu kennen und eine Art Nachricht zu senden. Diese Nachricht wird vom Gameobject dann interpretiert und als Methodenaufruf im Skript verarbeitet.

Wir kannten Multithreading und Parallelität. In den bisher bekannten Projekten war aber immer noch eine gewisse Gebundenheit an eine zentrale Ausführungslinie gegeben. Threads werden zum Beispiel normalerweise irgendwo gestartet und laufen einfach ab dem Zeitpunkt selbstständig. Der Ursprung ist aber offensichtlich ein Haupt-Thread, welcher das geordnet in Gang gibt. In Unity macht eigentlich jedes Objekt von Anfang an was es will und lässt sich dabei nicht gross stören. Die meisten Objekte werden regelmässig aufgerufen (Update-Funktion), nur um zu schauen, ob sie gerade etwas machen möchten. Eine Szene in Unity besteht also nicht aus Objekten, welche von einem Hauptobjekt aus erstellt wurden und eventuell noch verwaltet werden. Eine Unity-Szene besteht aus einer wilden Schar unabhängiger Objekte, welche untereinander kommunizieren und deren Lebensspanne nicht von einer zentralen Instanz diktiert wird.

Diese Umstellung war sehr interessant und spannend. Wir haben die Unterschiede nicht gewertet, sondern uns einfach damit abgefunden, dass die Softwareentwicklung in Unity anders funktioniert, als wir es gewohnt waren.

# Schlussfolgerung

# Quellen