Verbesserung von Level Design mittels Player Metrics

Gernot Raudner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2012

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 28. September 2012

Gernot Raudner

Inhaltsverzeichnis

Kurzfassung

Benutzerorientierte Entwicklung kann im Softwaredesignbereich von großem Vorteil sein. So lässt sich das zu entwickelnde Produkt leichter an eine Zielgruppe anpassen. Diese Methode wird im Bereich der Mensch-Computer-Interaktion schon seit geraumer Zeit eingesetzt, hat allerdings erst vor wenigen Jahren die Spielebranche erreicht. Auch wenn es viel Zeit kostet, festzustellen, welche Benutzerdaten aufgezeichnet und wie diese visualisiert werden sollen, stellt diese Methode ein wertvolles Werkzeug zur Spieleentwicklung dar. Neben der Beschreibung des Begriffs *Game Metrics* werden in dieser Arbeit sowohl Fallbeispiele anhand bereits veröffentlichter Spiele vorgestellt, als auch die Entwicklung eines Systems zum Sammeln und Analysieren von *Gameplay Metrics* des Spiels *Minecraft* gezeigt. Außerdem wird gezeigt, wie solche Daten verwendet werden können, um Game- und vor allem Level Design zu optimieren.

Abstract

Finding a way to improve game design is not an easy task. There are many different ways to accomplish this mission. With user-oriented development, software design can easily adapted to the needs of a target audience. It has been an important technique in the human-computer-interaction field for a while, but has reached game developers only a few years ago. Although tracking and visualizing userdata requires a large amount of planning-time and often extensive software, it is a very promising procedure within game development. This thesis explains the term *game metrics*, shows some case-studies of well-known game releases that used *gameplay metrics* and describes the development of a metrics gathering and analyzing tool for *Minecraft* called *MetricsCraft*.

Kapitel 1

Einleitung

1.1 Ziele

Viele moderne Spieleentwickler gehen immer mehr in die Richtung des benutzerorientierten Game Designs [30, 33, 20, 43]. Diese Methode stellt unter anderem ein wichtiges Werkzeug für das Optimieren von Level Design dar.

Diese Arbeit beschäftigt sich mit Metrikdaten, die beim Spielen eines Computerspiels erzeugt werden. Es wird gezeigt, wie sich sogenannte *Player Metrics* dazu verwenden lassen, Game Design (und vor allem Level Design) zu verbessern. Diese Daten werden benutzt, um Spieler zu klassifizieren und deren Bedürfnisse zu erkennen, sodass das betroffene Spiel optimal an die Spieler angepasst werden kann.

1.2 Gliederung

Das folgende Kapitel beschäftigt sich mit der Definition des Begriffs *Game Metrics*, sowie den untergeordneten Bereichen *Performance*, *Process* und *Player Metrics*. Außerdem wird beschrieben, was *Personas* sind und inwiefern sie für die Spielentwicklung interessant sein können. Zuletzt behandelt dieses Kapitel noch die Anwendung von *Player Metrics* in aktuellen Spielen wie *Tomb Raider: Underworld* und *Hitman: Blood Money*. In Kapitel 3 wird ein Projekt vorgestellt, das zeigen soll, ob und wie das Level Design von *Minecraft* mit Hilfe von *Player Metrics* spielerspezifisch optimiert werden kann. Dazu wurde ein Serverplugin entwickelt, das es ermöglicht, verschiedene Metriken aufzuzeichnen und zur Analyse zu visualisieren. Die Beschreibung der mit Hilfe des entwickelten Projekts durchgeführen Analysen erfolgt in Kapitel 4. Hier werden verwendete Techniken und gezogene Schlussfolgerungen gezeigt. Kapitel 5 beschäftigt sich mit möglichen Erweiterungen des Projekts. Außerdem beinhält es eine Schlussfolgerung der Ergebnisse des Projekts und der Analyse.

Kapitel 2

State of the Art

Schon seit geraumer Zeit ist der Begriff *Softwaremetrik* in der IT-Branche gegenwärtig. Der IEEE-Standard beschreibt ihn in [14] folgendermaßen:

> „Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet."

Viele Unternehmen nutzen bei der Entwicklung ihrer Software die Vorteile von Metriken, darunter auch die NASA[1] und akademische Institutionen[2] [36]. Spiele sind zwar auch Software, allerdings hat die Branche erst in den letzten zehn Jahren den Nutzen von Metriken bei der Entwicklung erkannt [8].

In diesem Kapitel werden die Begriffe *Game Metrics* und *Personas* genauer erklärt. Außerdem werden einige Beispiele zur Anwendung beschrieben.

2.1 Game Metrics

Game Metrics lassen sich grob in drei Kategorien unterteilen, wobei hier speziell auf die *Player Metrics* eingegangen wird [8].

2.1.1 Performance Metrics

Wie bei jeder Software können auch bei der Entwicklung eines Spiels u. a. *Performance Metrics* gemessen werden. Sie geben Auskunft darüber, wie flüssig das Spiel auf diverser Hardware läuft.

Seien es nun Spielkonsolen wie *Xbox 360* oder *Playstation 3*, bei denen die Systemkomponenten vordefiniert und demnach immer gleich zu behandeln sind, oder Desktopcomputer, die vollständig vom Benutzer konfigurierbar sind. Die Software kann natürlich gewisse Mindestvoraussetzungen stellen. Je höher diese sind, desto geringer ist allerdings die Zahl der

[1] http://earthdata.nasa.gov/our-community/community-data-system-programs/metrics-planning-group-mpg

[2] http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

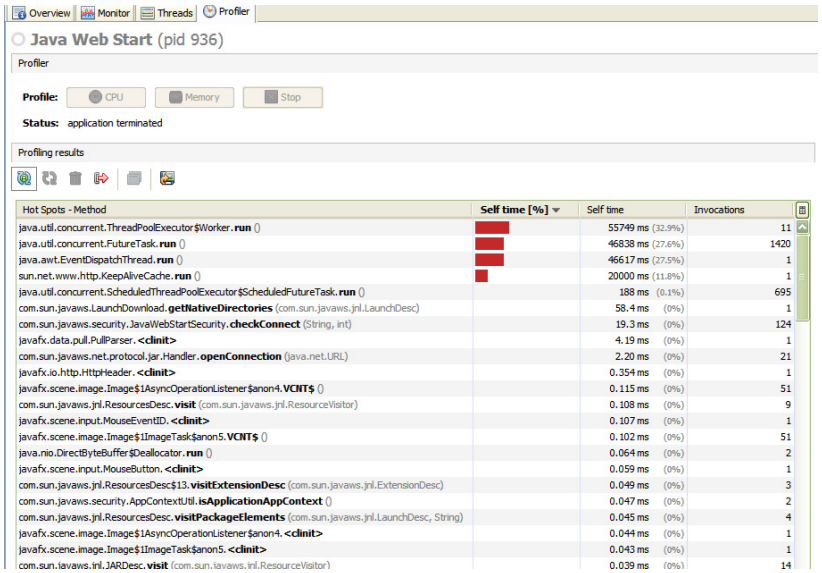


Abbildung 2.1: Screenshot der VisualVM. Aus [42].

Kunden-PCs, die diese Voraussetzungen erfüllen. Aus diesem Grund ist performante Programmierung empfehlenswert. Je nach Entwicklungsumgebung und Programmiersprache gibt es eine Vielzahl an Tools, die genaue Rückschlüsse über die Geschwindigkeit der implementierten Klassen und sogar einzelner Methoden ziehen können. Solche Software wird *Profiler* genannt. Das in Kapitel 3 beschriebene Projekt bedient sich der mit dem *JDK 6* mitgeliefertem *VisualVM*[3]. In Abb. 2.1 wird gezeigt, wieviel CPU-Zeit jede aufgerufene Methode für sich beansprucht. Mit dieser Information kann nach der Prozessanalyse festgestellt werden, wo sich Flaschenhälse in Gameloops befinden und welche Klassen zur Behebung dieses Problems verbessert werden müssen. Von großer Bedeutung sind hierbei oftmals die FPS[4], die direkt Aufschluss darüber geben, wie lange die Berechnung jedes einzelnen Bildes dauert.

Die vor allem bei Indie-Entwicklern beliebte Gameengine *Unity3D*[5] besitzt in der kostenpflichtigen Version ebenfalls einen sehr umfangreichen Profiler. Mehrere unterschliedlich färbige Graphen zeigen die CPU-Zeit der einzelnen Services (Rendering, Scripts, GarbageCollector, Physics, Others) an. Wird einer dieser Services ausgewählt, zeigt das Programm zugehörige Methoden und deren Metriken an. So können neben CPU-Auslastung auch Speicherbedarf und Anzahl der Methodenaufrufe überprüft werden.

[3] http://visualvm.java.net/index.html
[4] Frames Per Second
[5] http://www.unity3d.com

2.1.2 Process Metrics

Der zweite große Bereich der Game Metrics umfasst die Quantifizierung der Produktion selbst, also Entwicklungszyklen, Zeit zwischen Patches, Länge der Entwicklungspipeline und mehr. Larry Mellon hat in [18] folgendes dazu geschrieben:

> „Metrics Driven Development (MDD) is not a process per se, but rather finding a way to quantify your tasks and risks to best focus your team's limited resources on your project biggest problems."

Gemeint ist hierbei folgendes: Ein Team kann wesentlich effizienter eingesetzt werden, wenn jeder einzelne Entwickler immer mit der Durchführung wichtiger Aufgaben betraut ist. Wenn beispielsweise rechtzeitig erkennbar wird, dass eines der benötigten Features in der Entwicklung weiter zurück liegt als andere, so lassen sich die Prioritäten der Aufgaben verschieben und die Ressourcen des Teams optimal nutzen, wenn Entwickler von Aufgaben mit niedrigerer Priorität abgezogen werden.

Durch die Quantifizierung von einzelnen Bereichen können Process Metrics aufgezeichnet werden mit denen es möglich ist, die Entwicklung zu optimieren. Projektmanagement spielt hierbei eine tragende Rolle. Bei *Scrum* (siehe auch [22]) werden beispielsweise User Stories und zugehörige Tasks definiert, die anschließend von den Teammitgliedern mit einer Aufwandsschätzung und verantwortlichen Personen versehen werden. Wird nach einigen Tagen festgestellt, dass diese Schätzung zu gering war, können Entwickler von anderen Tasks abgezogen werden um sich der dringender benötigten Aufgabe zuzuwenden.

Neben Projektmanagement sollte die Überwachung der Pipeline bei der Spieleentwicklung priorisiert werden. Zu Beginn, solange die Codebase noch überschaubar und die Iterationszeiten, bis ein Build getestet werden kann, kurz sind, mag dies noch vernachlässigbar sein. Wenn zu späteren Zeitpunkten allerdings mehrere Minuten pro Build verloren gehen kommt das, je nach Größe des Teams, auf eine beachtliche Anzahl an Stunden. Werden Pipelinezeiten, Fehlerraten pro Pipelineabschnitt und ähnliches aufgezeichnet, kann nach Verbesserung deren oftmals Zeit eingespart werden [18].

2.1.3 Player Metrics

Bei der Entwicklung eines Computerspiels kommen viele verschiedene Faktoren zusammen, die das letztendlich erreichte Resultat beeinflussen können. Je weiter die Veröffentlichung eines solchen Spiels zurück liegt, desto geringer ist die Anzahl dieser Faktoren. Wird beispielsweise das 1972 erschienene *Pong* betrachtet ist anzunehmen, dass der Hauptantrieb bei der Entwicklung der Entwickler selbst war. Er allein gestaltet das Spiel nach seiner Erfahrung, seinen Vorlieben und Erwartungen in die Kunden. Werden hingegen aktuelle Veröffentlichungen wie *Anno 2070* oder *Die Siedler 7* untersucht lässt

Abbildung 2.2: Charakterfenster in *Eye of the Beholder*, aus [37].

sich feststellen, dass der Fokus in der Entwicklung immer stärker auf den Endbenutzern liegt (mehr dazu in [10]).

Dieser Fokus kann recht weit ausgebreitet werden. Angefangen von PR-bezogenen Bereichen wie Foren, Social Network-Seiten oder Communityplattformen, auf denen sich Fans und Spieler austauschen können, über Verkaufszahlen und Kosten-Nutzen-Faktor bis hin zu konkreten Benutzereingaben innerhalb des Spiels kann alles, was sich in Zahlen ausdrücken lässt, in den Bereich der *Player Metrics* eingeordnet werden.

Computerspiele bieten meist eine große Anzahl unterschiedlicher Metriken. Schon die ersten Spiele besaßen zählbare Parameter. Sei es nun die einfache Punkteanzeige in *Pong* oder ein komplexeres System mit vielen verschiedenen Daten wie beispielsweise die Charaktereigenschaften in *Eye of the Beholder* (siehe Abb. 2.2) oder *Final Fantasy*. Je nach Genre werden diese unterschiedlich verwendet, viele davon sind vom Spieler beeinflussbar. Jene messbaren Werte fallen innerhalb der Player Metrics in den Bereich der *Gameplay Metrics*, dazu später mehr.

Customer Metrics

Einen Teil der Player Metrics stellen die *Customer Metrics* dar. Nachdem ein Großteil der vom *Economist* befragten Firmen ihre Kunden als Haupteinkommensquelle über die Aktionäre stellen [11], spielt dieser Bereich vor allem bei Onlinespielen eine große Rolle. Je länger der Kunde mit einem Produkt beschäftigt ist, desto wichtiger wird die korrekte Berechnung, wie hoch die Kosten des Akquirierens von Kunden im Vergleich zur dadurch entstehenden Umsatzsteigerung sind. Vor allem bei Onlinespielen auf *Facebook* wie

z. B. *Farmville*[6], wo Spieler nichts für das Spiel an sich, sondern regelmäßig für Inhalte zahlen, ist eine umfangreiche Analyse dieser Metriken sehr zu empfehlen.

Community Metrics

Die Player Metrics umfassen noch eine dritte Kategorie, die sich ebenfalls zumindest zum Teil mit Marketing beschäftigt. Hierbei wird die Fangemeinschaft anhand der bereits erwähnten Plattformen analysiert. Die Benutzer können hierbei auf (mehr oder weniger) direktem Wege mit den Entwicklern kommunizieren und Verbesserungsvorschläge bringen oder einfach untereinander produktbezogene Themen diskutieren. Der virale Effekt solcher Communities kann für Spiele lebensnotwendig sein. Nicht zuletzt sorgen Hypes häufig für äußerst erfolgreiche Verkaufsstarts, lediglich wegen der Mundpropaganda und des damit verknüpften Onlinemarketings.

Aber auch viele Offlinespiele wie die meisten *Bioware*-Games oder auch *Minecraft* zeugen von einer großen Fangemeinde. Interessant können vor allem Teaser am Ende einer Episode sein, wie es beispielsweise bei dem von *Ubisoft* entwickelten Action-Adventure *Assassin's Creed* der Fall ist. Am Ende des ersten Teils können im Epilog die Wände eines Raums betrachtet werden. Der Spieler kann dabei mysteriöse Symbole erkennen, deren Bedeutung nicht im Spiel erklärt werden (siehe Abb. 2.3). Es müssen also außerhalb des Spiels Nachforschungen getätigt werden um mögliche Rückschlüsse auf die Story des Spiels ziehen zu können. Dadurch entsteht eine Gemeinschaft aus Spielern, die alle auf der Suche nach demselben Ergebnis sind. Je größer sie wird, desto mehr Aufmerksamkeit erregt sie und somit auch das Produkt (hier also *Assassin's Creed*). Außerdem werden Erwartungen in eine mögliche Fortsetzung gesteckt, was im besten Fall zu einem Hype führen kann [26].

Gameplay Metrics

Je größer ein Projekt, desto mehr Daten lassen sich normalerweise sammeln. Im Bereich der *Human-Computer Interaction*-Forschung (HCI) wird das Sammeln dieser Daten zur Verbesserung von Interfacedesigns schon seit geraumer Zeit verwendet [7]. Methoden wie *Paper-Prototyping* oder *Eye-Tracking* werden vor allem in der Webentwicklung genutzt um herauszufinden, ob das Design vom User wirklich wie gewünscht angenommen wird [29, 21]. Mittels Eye-Tracking kann in Labortests festgestellt werden, wo der Aufmerksamkeitsfokus der Testperson liegt. Diese Informationen ermöglichen wiederum Optimierungen, zum Beispiel eine effiziente Positionierung von Werbeflächen indem diese in die Nähe der ermittelten Bereiche, denen hohe Aufmerksamkeit gewidmet wird, platziert werden. Bei solchen Lab-

[6] http://www.farmville.com

Abbildung 2.3: Epilog in *Assassin's Creed*, unerklärte Symbole an den Wänden. Aus [38].

ortests werden üblicherweise Daten wie Geschlecht und Altersgruppe des Testers aufgezeichnet. Oftmals werden auch sogenannte *Personas*, also Userrollen, vordefiniert. Wäre nun ein Webshop zu entwickeln, ließen sich folgende Personas im Vorhinein festlegen:

- Mutter,
- Teenager,
- Arbeiter,
- ...

Anschließend wird versucht, sich in diese Rolle hineinzuversetzen. Wie würde so jemand den zu entwickelnden Webshop benutzen? Was würde diese Person sich wünschen? Schon in frühen Stadien der Entwicklung lässt sich so die Benutzerfreundlichkeit von Applikationen ohne großen Aufwand feststellen. Mehr über Personas in Abschnitt 2.2.

Für einen Webshop könnten außerdem die Anzahl der Klicks pro Menülink, die durchschnittliche Dauer einer Bestellung oder die Absprungrate nach einer gewissen Anzahl von Klicks interessant sein. So lässt sich bei korrekter Analyse der Daten folgern, dass User möglicherweise weniger Produkte bestellen weil sie nicht so oft klicken möchten. Im Gegensatz zu den Eye-Tracking-Daten, die Labortests benötigen, können solche Metriken quantitativ gesammelt werden indem sie in Echtzeit aufgezeichnet und in eine Datenbank geschrieben werden. Die Nutzung von Datamining APIs wie *Google Analytics* vereinfachen diesen Vorgang um ein Vielfaches [12]. Da keine Labortests benötigt werden um die gewünschten Daten zu sammeln, können sowohl Zeit und Geld gespart als auch uneingeschränkte Benutzererfahrungen aufgezeichnet werden. Die Daten, die bei Labortests gesammelt werden,

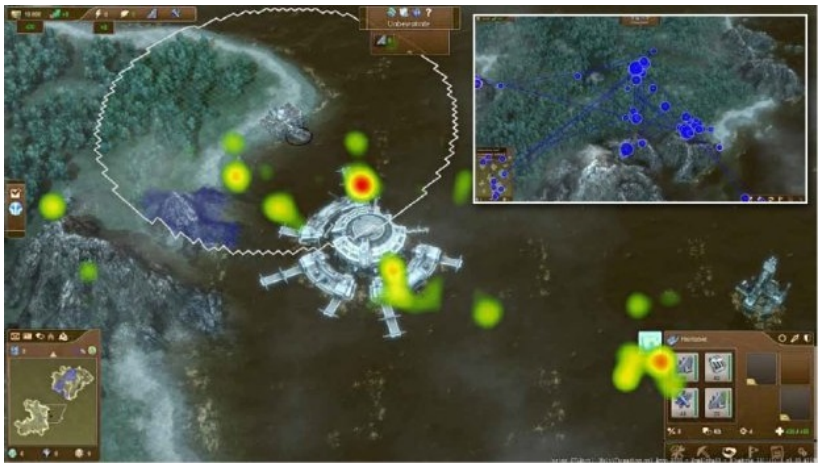


Abbildung 2.4: Heatmaps der Eye-Tracking Ergebnisse, das Handelssystem rechts unten wird neben der Ingame-Ansicht häufig betrachtet. Aus [10].

sind normalerweise immer zumindest teilweise verfälscht, da der User das Produkt nicht in seiner gewohnten Umgebung verwenden kann.

Viele der in der HCI-Forschung verwendeten Methoden lassen sich auch bei der Entwicklung von Computerspielen anwenden. Um das Tutorial von *Anno 2070* zu optimieren, hat das *GamesLab* von *Blue Byte* Labortests mit (immer) unterschiedlichen Spielern, die aus einer Datenbank ausgewählt werden, durchgeführt. Mittels Eye-Tracking-Geräten wurde gemessen, welche Bereiche des Interfaces häufig betrachtet werden und ob und wieviele Spieler die eingeblendeten Hinweise lesen bzw. überhaupt wahrnehmen. Videos von den von Testern gespielten Test Sessions mit darüberliegenden Eye-Tracking-Visualisierungen (siehe Abb.2.4) gaben hierbei Aufschluss darüber, wann Spieler nach Informationen suchen und welche Bereiche des Tutorials noch unklar waren [10]. Labortests wie diese können aber hohe Kosten mit sich tragen. Sobald die Entwicklung in ein Stadium kommt, in der eine größere Gruppe an Spielern das Produkt in ihrer gewohnten Umgebung testen kann (beispielsweise bei einem Beta-Test), können enorm viele Daten über dieses Verhalten aufgezeichnet werden. Das offiziell erste größere Projekt, bei dem die Verwendung von Gameplay Metrics eine tragende Rolle übernahm, war Microsofts *Halo 2*, das 2003 veröffentlicht wurde [8]. Damals wurde eine eigene Plattform entwickelt, um das Verhalten von Spielern während der Betaversion aufzuzeichnen, sodass die Entwickler die dabei gesammelten Metriken später analysieren und Probleme ermitteln konnten [16]. Mehr dazu in Abschnitt 2.3.

Hagen et. al haben in [8] die Kategorie der Gameplay Metrics noch in drei weitere Gebiete unterteilt:

- *Interface Metrics* beziehen sich auf sämtliche Interaktionen des Users

mit dem *GUI*[7]. Dazu zählen das Einstellen von Systemvariablen wie Auflösung oder Steuerung, Klicken von Buttons und anderweitige Verwendung von Menüelementen.

- *In-game Metrics* sind sämtliche Daten, die Entstehen, sobald der Spieler mit dem Spiel interagiert, also sich bewegt, Fähigkeiten verwendet, Schalter drückt oder anderweitig mit Elementen der Spielwelt interagiert.
- *System Metrics* geben Informationen darüber, welche Aktionen von der Game Engine in Reaktion auf das Spielerverhalten durchgeführt werden. Wenn ein Spieler beispielsweise ein neues Gebiet betritt, wird aufgezeichnet, welche Daten geladen werden, welche Events auftreten usw.

Bei dem mit dieser Arbeit einhergehenden Projekt, dessen Entwicklung in Kapitel 3 näher beschrieben wird, wird zum größten Teil von den *In-game Metrics* Gebrauch gemacht.

Meistens sind ein Großteil der Möglichkeiten, wie der Spieler mit der Welt interagieren kann, ohne gröbere Probleme quantifizierbar. Was es bei der Verwendung von Metrics zu beachten gilt, ist, welche spezifischen Werte dieser oft äußerst großen Menge an Daten bei der Verbesserung von Game Design interessant sind. Eine solche Auswahl kann normalerweise nicht auf jedes Spiel angewandt werden. Natürlich ist es möglich, Werte wie Erfahrung, Trefferpunkte, Positionswerte u. ä. genreübergreifend zur Analyse heranzuziehen. Diese Werte allein geben jedoch meistens nicht genug Aufschluss. Wichtig ist vor allem, in welchem Kontext die Werte stehen. Ein simples Beispiel: Angenommen, es wird aufgezeichnet, wie lange Spieler für ein Level brauchen. Es resultieren Zeiteinheiten, die nicht viel Aussagekraft haben. Werden diese Werte allerdings in den Kontext „Zeit pro Level“ gestellt, so lassen sich zumindest problematische Levels hervorheben. Um das Level Design zu verbessern werden zudem noch weitere Daten der Spieler benötigt um festzustellen, warum sie so lange brauchen. In einem Platformer wie *Replica Island*[8] werden dazu Position und Ursache von sterbenden Charakteren aufgezeichnet [41].

2.1.4 Visualisierung

Die gesammelten Daten können im Textformat bei größeren Projekten sehr schnell unübersichtlich und kaum nutzbar werden. Visualisierung von Metriken spielt somit eine zentrale Rolle bei der Analyse. Im Folgenden werden einige bewährte Methoden aufgezeigt.

[7] Graphical User Interface
[8] http://replicaisland.net/

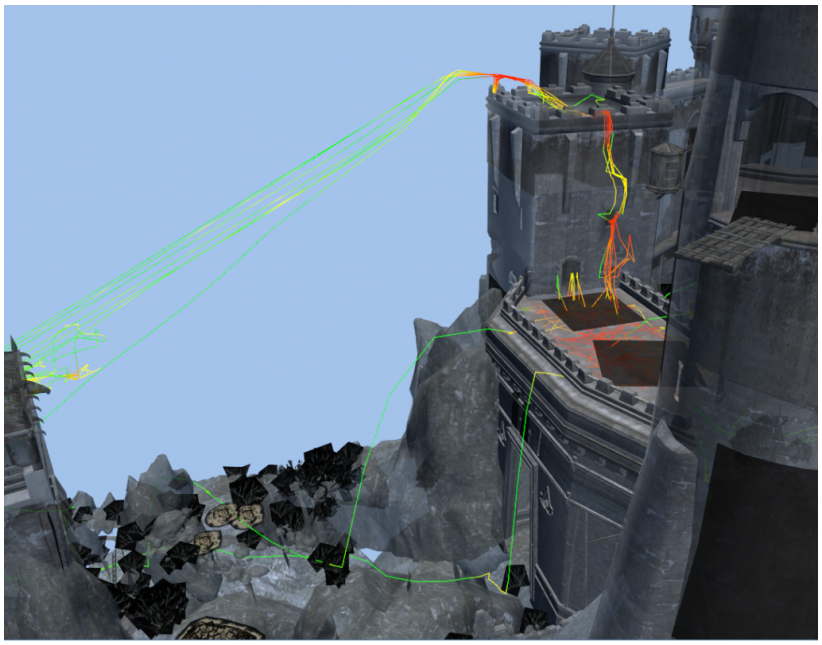


Abbildung 2.5: 3D-Heatmap aus *Assassin's Creed: Brotherhood*, die Bewegung von acht Spielern wurde aufgezeichnet. Aus [31].

Heatmaps

Steht eine große Sammlung an Datensätzen zur Verfügung, sind Heatmaps eine gute Wahl, diese darzustellen. Benötigt werden sogenannte *Spatial Gameplay Metrics*, also positionsbezogene, vom Spieler erzeugte Daten und eine Visualisierung der Umgebung, in der diese Daten aufgezeichnet wurden. Dafür wird bei 3D-Spielen oft eine Top-Down-Ansicht der Spielwelt oder des zu analysierenden Levels verwendet. Äußerst praktisch ist natürlich eine Einbindung in die verwendete 3D-Engine (siehe Abb. 2.5). Bei Sidescrollern genügen oft, wie in [41] ersichtlich, Screenshots. Stehen diese Daten zur Verfügung, so werden die Einträge nach Position sortiert. Dabei sollte (im Fall eines 2D-Spiels) eine Tabelle in folgender Form zustande kommen:

Position	#
(0/0)	13
(0/4)	8
(3/0)	1
(-7/6)	42
(-7/2)	0
(1/5)	0
...	...

Sie gibt Aufschluss darüber, wieviele Aktionen an welchen Positionen durchgeführt wurden. Der höchste Wert wird nun als Maximum festgelegt, der niedrigste als Minimum, dazwischen wird interpoliert. Anschließend können mit Hilfe der Positionsdaten gefüllte rote Kreise mit unterschiedlicher Transparenz (die eben berechnet wurde) auf die Umgebungskarte gezeichnet werden. Wird der Kreisdurchmesser nun beispielsweise auf drei Einheiten und die Maximaldeckkraft auf 50% gesetzt, so entstehen an Stellen, wo die zu untersuchende Aktion häufig durchgeführt wurde, tiefrote Flecken. Nun kann anstatt der Transparenz auch der Farbwert geändert werden um wie in Abbildung 2.5 einen von grün zu rot verlaufenden Übergang zu erreichen. Eine sinnvolle Anwendung wäre beispielsweise das Speichern der Position, an der die Spieler sterben. Dadurch kann auf einen Blick festgestellt werden, ob einzelne Passagen für die Spieler schwerer sind als vom Game Designer geplant.

Pfade

Das Aufzeichnen von Pfaden kann vor allem bei Spielen mit einer offenen Welt, aber auch bei einzelnen Abschnitten levelbasierter Spiele hilfreich sein. Hierbei werden ebenfalls Umgebungskarten zu Hilfe genommen. Gridbasierte Spiele wie *Minecraft* vereinfachen die Aufzeichnung dieser Positionsdaten: die Spielerposition wird bei jedem Betreten einer Zelle in die Datenbank geschrieben. Bei anderen Spielen werden diese Daten (da sie theoretisch 60 mal pro Sekunde erzeugt werden) oft per Stream bereitgestellt und in diskreten Abständen (z. B. 0,5 Sekunden) angezeigt. Natürlich muss zu jedem Positionswert ein zugehöriger Zeitwert existieren, um den Pfad nachverfolgen zu können. Mit Hilfe dieser gezeichneten Pfade können selten besuchte Orte oder missverstandene Wegweiser identifiziert werden. In Abb. 2.5 ist beispielsweise zu erkennen, dass zwei von acht Spielern nicht den Pfad wählen, der vom Game Designer beabsichtigt war. Für die betroffenen Spieler kann dieser Abschnitt des Levels frustrierend sein, da das Ziel nur über einen enormen Umweg erreichbar ist, auf dem sie möglicherweise von vielen Feinden aufgehalten werden.

Pfade können auch sehr gut von Rennspielen zur Verbesserung von Balancing und Level Design verwendet werden. So kann auf einer Top-Down-Umgebungskarte abgelesen werden, ob und warum Spieler die perfekte Linie häufig verfehlen.

Die Entwickler von *Kane & Lynch: Dog Days* verwendeten nicht nur die Bewegungspfade der Spieler, sondern auch die Pfade von verschossenen Kugeln. Bei letalen Schüssen erhielt der Datenserver die Position von Spieler und Opfer, sowie die verwendete Waffe [8]. Dadurch konnte das Verhalten der Spieler genauer analysiert werden. Es könnte beispielsweise in Erfahrung gebracht werden, ob und wie Spieler Deckungsmöglichkeiten nutzen bzw. ob hierbei Verbesserungsbedarf im Level Design besteht.

Graphen

Eine simple Methode, um Datensätze darzustellen, sind Funktionsgraphen. Zeit/Wert-Funktionen sind im Bereich der Gameplay Metrics oft von großem Vorteil, vor allem bei Werten, deren Balancing eine wichtige Rolle im zu entwickelnden Produkt spielt [3]. In *Split/Second* werden die Fahrkünste der K.I. durch einen sogenannten DCB[9]-Wert bestimmt. Mittels eines Funktionsgrafen dieses Werts (in Verbindung mit der Gesamtspielzeit der Tester) konnte herausgefunden werden, dass das Spiel zu einem gewissen Zeitpunkt für einige Spieler zu schwer wurde, sodass sie immer mehr Rennen verloren. Dies war durch plötzliches Abfallen des DCB-Werts erkennbar. Den Entwicklern wurde somit bewusst, dass die Berechnung des DCB-Werts verändert werden musste um die Spieler nicht zu frustrieren. Bei der Analyse kurz vor Release des Spiels zeigte derselbe Graph eine konstante Steigerung des Werts, was von den Game Designern auch gewünscht war.

In [30] wird das Verhältnis zwischen Ausgaben und Einnahmen in *Assassin's Creed Brotherhood* beschrieben und an einem Funktionsgraphen gezeigt. Auch hier spielt das Balancing dieser Werte eine wichtige Rolle und Fehler können anhand von Visualisierungen der Gameplay Metrics erkannt werden. Viele andere Werte, darunter auch die Gründe für Levelneustarts (Neustart, Tod, Fehlschlag) wurden aufgezeichnet. Zu einem späteren Zeitpunkt der Entwicklung wurden sogenannte *Constraints* hinzugefügt, mit deren optionaler Erfüllung die Spieler ein Level zu 100% abschließen können. Bei der Einführung dieses Features kam es allerdings zu einer ungewöhnlichen Veränderung bei oben erwähnten Werten: die Zahl der Neustarts pro Level stieg unerwartet. Durch eine Grafik, die sowohl die Anzahl der Neustarts als auch die prozentuelle Erfüllung der *Constraints* visualisiert, konnte in Erfahrung gebracht werden, woher die Steigerung des Neustart-Werts kam (siehe Abb. 2.6). Hier wird wieder deutlich, wie wichtig der Kontext ist, in dem Game Metrics zu verstehen sind.

Abgesehen von bereits erwähnten Heatmaps sind bei Analysen von Daten, die nicht unbedingt Positionswerte benötigen, Balken- und Tortendiagramme eine gute Wahl. Vor allem Balkendiagramme lassen sich gut mit Funktionsgraphen in Verbindung bringen (siehe Abb. 2.6). Hierbei können nicht nur, wie bei oben beschriebenem Beispiel, unerwartete Relationen sichtbar gemacht, sondern auch individuelles Spielerverhalten analysiert werden.

Tortendiagramme bieten eine einfache Form der Darstellung von Daten an. Eine simple Anwendung wäre beispielsweise eine prozentuelle Angabe, welche Ursachen für den Tod von Spielern verantwortlich sind. Nun kann auch noch in unterschiedliche Detailgrade abgestuft werden, beispielsweise Ursachen für Tode in jedem Level. Auf einfache Weise ist es so möglich, einen groben Überblick über einen bestimmten Bereich der gesammelten Datensätze zu geben.

[9] Dynamic Competition Balancing

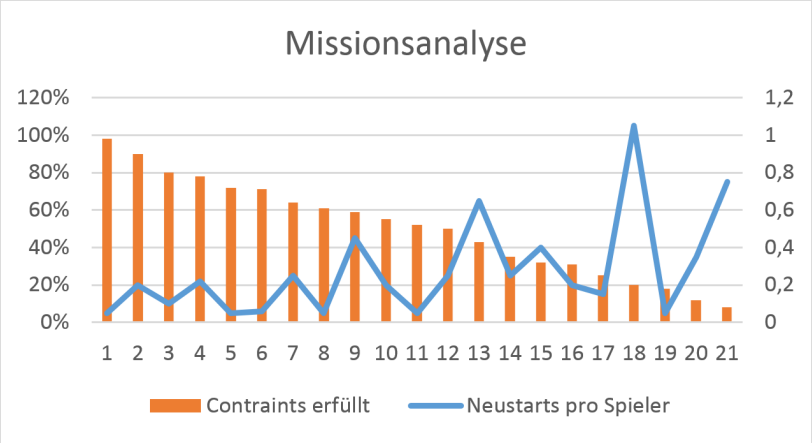


Abbildung 2.6: Der Graph visualisiert sowohl die prozentuelle Erfüllung der *Constraints* als auch die Anzahl der Neustarts pro Level. Aus [32].

Ein Gameplay Metrics-System, das die Vorteile vieler verschiedener Methoden zur Visualisierung von Daten nutzen kann, kann in einem Entwicklungsprozess sehr hilfreich sein. In [16] wird ersichtlich, dass unterschiedlich detaillierte Ansichten der gesammelten Daten es ermöglichen, zwischen verschiedenen Ebenen der Visualisierung zu navigieren und dadurch Fehlerquellen und problematische Stellen im Spiel zu erkennen. Folgender Aufbau eines solchen Systems wäre vorstellbar:

- 1. Ebene: Übersicht – Sterberate, gewonnene Rennen, getötete Monster, gesammelte Gegenstände ...
- 2. Ebene: Detailstufe 1 – Ursachen für Tode, Tode pro Level, Tode pro tötendem Monster, ...
- 3. Ebene: Detailstufe 2 – Tode durch Monster x in Level y an den Positionen z1, z2, z3, ... (Heatmaps)

Durch einen solchen Aufbau lassen sich in der 2. Ebene mehrere Kategorien selektieren, die anschließend in Ebene 3 analysiert werden können.

Verwendung

Einige Verwendungsmöglichkeiten von Gameplay Metrics wurden bereits aufgezeigt. Doch welche Möglichkeiten gibt es, das Gameplay eines Spiels für möglichst viele Spieler zu optimieren? Angenommen in einem Platformer existieren drei verschiedene Varianten, eine höhere Plattform zu erreichen. Nun könnte aufgezeichnet werden, wie hoch der Anteil der einzelnen Varianten ist, die die Spieler benutzen. Wären diese Varianten beispielsweise „fliegen“, „klettern“ und „springen“ kann leicht festgestellt werden, welche dieser Möglichkeiten jeder einzelne Spieler bevorzugt. Das Speichern von Zeitwert

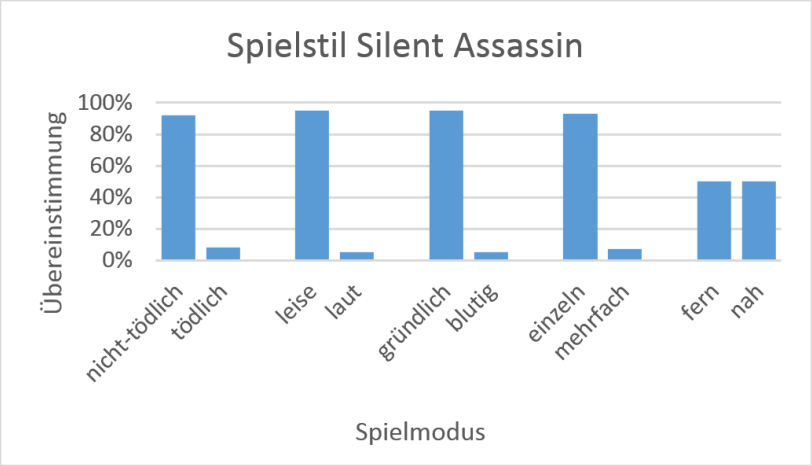


Abbildung 2.7: In *Hitman: Blood Money* hat der Spieler mehrere Möglichkeiten, seine Gegner auszuschalten. Hier wird das Verhalten eines Spielers gezeigt, der bevorzugt aus den Schatten angreift. Aus [25].

zu jeder aufgezeichneten Fähigkeit ermöglicht es, zu ermitteln, ob sich das Verhalten der Spieler über die Zeit verändert. Möglicherweise erkennt der Spieler den Vorteil von bestimmten Fähigkeiten erst, nachdem er mehr Erfahrung im Spiel gesammelt hat.

Sind diese Daten gesammelt, stehen individualisierte Aufzeichnungen jedes einzelnen Spielers zur Verfügung. Diese können Verwendet werden, um dessen Vorlieben zu erkennen (siehe Abb. 2.7). Oftmals stellt sich dabei auch heraus, dass gewisse Fähigkeiten kaum oder garnicht benutzt werden. Dann kann gezielt Fokus auf die Verbesserung dieses Teils des Spiel gelegt werden, oder die betroffene Fähigkeit wird komplett gestrichen. Aus diesen Daten lässt sich auch ablesen, ob möglicherweise mehr Features eingebaut werden sollten, die der beliebtesten Fähigkeit ähneln. Auf obiges Beispiel bezogen hieße das folgendes: Würde ein Großteil der Spieler bevorzugt die Fähigkeit „klettern“ benutzen, sollte in Betracht gezogen werden, das Verhalten der anderen beiden Fähigkeiten zu verbessern oder mehrere Spielpassagen zu designen, in denen der Spieler klettern kann.

Die Spieler, die bevorzugt eine der anderen Fähigkeiten verwenden, sollten allerdings nicht außer Acht gelassen werden. Es ist möglich, aufgrund der gesammelten Daten Spieler in Kategorien einzuordnen. Ein Spiel lässt sich so besser für jede Kategorie individuell designen, sodass die Wünsche einer großen Spielerschaft erfüllt werden können. Mehr dazu im folgenden Abschnitt.

2.2 Personas

Alan Cooper gilt als einer der Vorreiter in Bezug auf *Personas* [9]. Er war an der Entwicklung von über 15 Businessapplikationen beteiligt bis er darauf aufmerksam wurde, dass das Erschaffen benutzerfreundlicher Software ein kompliziertes Unterfangen war [27]. Zu dieser Zeit arbeitete er an einem Projektmanagementtool.

2.2.1 Definition von Rollen

Um die Software benutzerfreundlich zu gestalten, interviewte Cooper mehrere Kollegen und Bekannte, die zur Zielgruppe gehörten. Eine dieser Personen war Kathy.

Kathys Aufgabenbereich bestand darin, sicherzustellen, dass laufende Projekte korrekt besetzt waren und das gesamte Potenzial des Personals genutzt wurde. Sie hatte also die typische Aufgabe eines Projektmanagers und war damit ein passendes Vorbild. Cooper definierte sie als die Basis für seine erste Persona. Er ging daraufhin im Kopf mehrere Szenarien durch, wie Kathy seine Software nutzen und welche Features sie sich wünschen würde.

Später arbeitete er als Software Consultant an einer Businessinformationsapplikation. Nachdem die Entwickler ihm keine praktischen Beispiele geben konnten, wie ihre Software genutzt werden würde, interviewte Cooper mehrere Kunden des Unternehmens. Es stellte sich heraus, dass sich drei Gruppen in Bezug auf Ziele, Aufgaben und Fähigkeiten voneinander unterschieden. Er stellte diese Gruppen dem Entwicklerteam als Chuck, Cynthia und Rob vor. Chuck war ein Börsenanalytiker und nutzte fertige Templates und Berichte. Cynthia ebenso, doch sie erstellte auch ihre eigenen Templates, die sie an Chuck weitergab. Rob war als IT Manager verantwortlich für die Unterstützung der beiden und konnte Cynthias Templates optimieren.

In den darauf folgenden Meetings bekräftigte er seine Designvorschläge immer damit, dass mindestens einer der Drei die Software so optimal nutzen konnte. Daraufhin wurden in der Entwicklung oft Phrasen wie „was würde Cynthia damit machen“ oder „würde Chuck das verstehen“ genutzt. Der Nutzen dieser Personas hatte großen Einfluss auf das Interface Design. Um die Software für alle drei Charaktere zu optimieren, wurden drei verschiedene Interaktionsstrukturen maßgeblich verbessert:

- Verwendung von vordefinierten Templates und Berichten,
- Erzeugung und Verbreitung neuer Templates,
- Optimierung von Templates ohne Veränderung von Inhalt und Verhalten.

Das entwickelte Produkt profitierte von der Nutzung dieser Personas, was der Erfolg des Unternehmens beweist [27]. Mehr über Personas im Bereich Softwareentwicklung in [5].

2.2.2 Nutzen in Game Design

Im Zentrum der Entwicklung jedes digitalen Spiels sollte die *Player Experience* stehen, also wie der Spieler mit dem Produkt interagiert und welche Emotionen dabei in ihm geweckt werden sollen. Im Vordergrund sollte hierbei das Vergnügen stehen. Viele der von Hassenzahl et al. in [13] beschriebenen Eigenschaften von *User Experience Design* können auch auf Game Design angewandt werden, allerdings müssen einige Aspekte anders betrachtet werden. Die Studie gibt einen Überblick über Userexperience im Allgemeinen, wobei sie diese als normalerweise ergebnisorientiert betrachtet. Player Experience ist allerdings prozessorientiert, dadurch verschieben sich auch die Prioritäten bei der Entwicklung. Genauer beschreiben Nacke und Drachen dies in [19] folgendermaßen:

Player Experience: Die Optimierung der Player Experience sollte auf jeden Fall wichtiger sein als Usability und Funktionalität. Dies wird auch in [2] hervorgehoben.

Komplexität: Die Handhabung von Spielen ist größtenteils wesentlich komplexer als die Verwendung von anderer Software wie z. B. Officeanwendungen. Werden beispielsweise *MMORPGs*[10] wie *World of Warcraft* oder *Guild Wars 2* betrachtet, so lässt sich feststellen, dass das betreffende Spiel ohne geeignete Tutorialsequenzen wesentlich schlechtere Player Experience mit sich bringt. Es gibt zu viele Interaktionsmöglichkeiten. Spieler können die Anzahl der Möglichkeiten, im Fall von *World of Warcraft*, sogar noch vergrößern, indem sie selbstentwickelte *Mods* in das Spiel integrieren. Eine Komplexität dieser Größe bedarf oft einer ausführlichen Einführung, damit Spieler das volle Potential eines Spiels ausnutzen können.

Zeitperiode: Der dritte zentrale Punkt beim Design von Player Experience wird in [19] als *temporal dimension* definiert. Zeitwerte spielen vor allem in Verbindung mit dem von Csíkszentmihályi entwickelten *Flow Channel* eine Rolle [6]. Dieser beschreibt, wie Herausforderung des betroffenen Spiels und Können des Spielers dessen Spielerfahrung beeinflussen können. Es kann davon ausgegangen werden, dass der Spieler über die Dauer, die er ein Spiel spielt, dazulernt und sein Können erweitert [17]. Beim Game Design muss dieser Faktor also (im Gegensatz zu ergebnisorientierter Software) ebenfalls berücksichtigt werden. Ein dynamisch anpassbaren Schwierigkeitsalgorithmus kann dies beispielsweise herbeigeführen (siehe [3] und Abschnitt 2.3).

Oftmals präferieren Spieler Genres und sammeln so unterschiedliche Erfahrungen. In [15] haben Joorabchi und El-Nasr anhand einer Studie mit 35

[10] Massively Multiplayer Online Role Playing Games

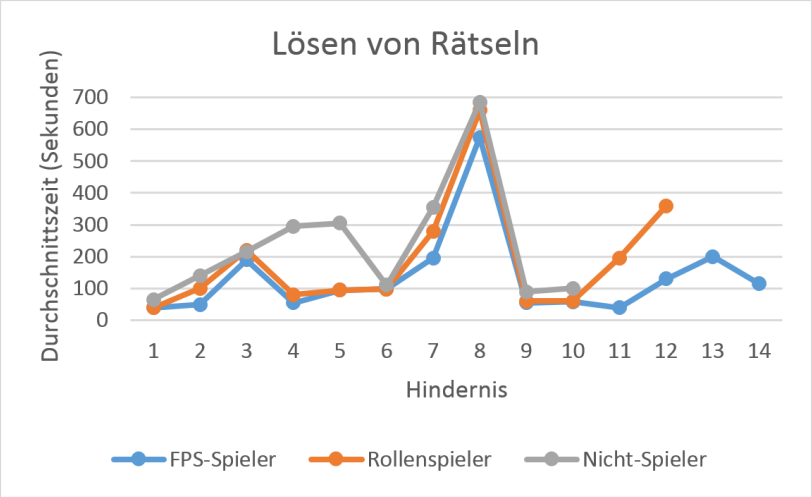


Abbildung 2.8: Durchschnittsdauer, wie lange Spieler der unterschiedlichen Kategorien für Rätsel in *Tomb Raider: Underworld* brauchen. Aus [15].

Teilnehmern gezeigt, dass sich beim Lösen von navigationsbezogenen Rätseln in 3D-Spielen Unterschiede zwischen RPG- und FPS[11]-Spielern bemerkbar machen. Als Testbeispiel wurde *Tomb Raider: Underworld* gewählt, da sich das Spiel weder in das Genre der Rollenspiele, noch in das der Shooter einordnen lässt. Im Vorhinein mussten die Teilnehmer einen Fragebogen ausfüllen nach dessen Auswertung sie sich in die Kategorien FPS-, RPG-Spieler und Nicht-Spieler einteilen ließen. Spieler mit 3D-Platformer-Erfahrung wurden nicht in die Studie miteinbezogen. Während der Test Sessions wurden verschiedene Gameplay Metrics aufgezeichnet, die zu dem Resultat führten, dass FPS-Spieler durchschnittlich schneller Rätsel lösen als die anderen beiden Gruppen (siehe Abb. 2.8).

Anhand der erwähnten Aspekte lässt sich erkennen, dass Spieler, obwohl sie innerhalb eines Spiels dieselben Möglichkeiten haben, sehr unterschiedliches Verhalten aufweisen können. Je mehr verschiedene Vorgehensweisen ein Spiel ermöglicht, desto mehr Spieler werden erreicht. Hier soll nicht nur auf die Vielzahl der Eingabemöglichkeiten (Touchdisplay, Controller, Maus, ...) hingewiesen werden, sondern vor allem auf die unterschiedlichen Präferenzen der Spieler. Wichtiger ist nun, festzulegen, auf welche Weise ein Spiel gespielt werden kann. Das Definieren dieser Spielstile kann sich allerdings als sehr komplex erweisen. Qualitative Befragung von Testkandidaten ist sehr zeit- und kostenintensiv und führt oft zu wenig brauchbaren Ergebnissen. Auch die manuelle Vorgabe von Spielstilen ist nur schwer korrekt durchführbar. Die Verwendung von Gameplay Metrics stellt beim Finden von Verhaltensmustern eine Methode dar, die es ermöglicht, unverfälschte Daten über die

[11] First Person Shooter

Interaktion zwischen Spieler und Spiel zu sammeln und damit Personas zu definieren. Diese Daten haben zwar keine Aussagekraft über das Maß an Vergnügen, das ein Spieler empfindet, können jedoch Aufschluss darüber geben, ob das Verhalten der Spieler stark von dem von Game Designern geplantem abweicht.

Vor allem auf Spiele, bei denen der Spieler einen einzelnen Charakter steuert, lassen sich Personas anwenden. Bezogen auf solche Spiele teilen Tychsen und Canossa in [25] die für die Definition von Personas relevanten Player Metrics in folgende vier Bereiche ein:

Navigation Metrics beschreiben sämtliche Bewegungsaktionen, die ein Spieler durchführen kann. Dazu zählen sowohl vom Charakter direkt ausgeführte Bewegungen, als auch durch das Spiel ausgelöste (wie beispielsweise das Fahren in einem Aufzug). Normalerweise werden diese Werte als *frequency metrics*, also Häufigkeitsmetriken, aufgezeichnet. Zusätzlich sollte, je nach Fähigkeiten des Charakters, mit den gesammelten Daten ein Bewegungsmodifikator mitgespeichert werden. Diese können beispielsweise wie oben „klettern“ und „fliegen“ oder auch „laufen“, „gehen“ und „kriechen“ sein. Wird in einem 3D-Spiel auch noch der Kamerablickwinkel abgebildet, so lässt sich feststellen, in welchen Bereichen eines Levels die Aufmerksamkeit des Spielers liegt. Zusammengefasst bedeutet dies also, dass

1. Position (x, y, z) und Rotation w° des Charakters,
2. Horizontale λ und vertikale ϕ Kamerarotation sowie
3. Bewegungsmodifikatoren (klettern, fliegen, laufen, gehen, kriechen)

in den meisten Fällen relevante Gameplay Metrics darstellen [25].

Navigation Metrics ermöglichen es, Heatmaps und Zeitfunktionen zu erstellen, an denen sich oft unterschiedliche Spielstile bemerkbar machen. So zeigen Chittaro und Ieronutti in [4] anhand von Heatmaps, welche Bereiche eines Levels von Spielern unbeachtet und welche intensiv untersucht werden (siehe Abb. 2.9). Wird der Raum im Fischstil erkundet so bedeutet das, dass diese Spieler sich für kein Objekt speziell interessieren. Im Gegensatz dazu steht der Ameisenstil, wo sämtliche Objekte eingehend betrachtet werden. Grashüpfer- und Schmetterlingstil sind schwerer zu kategorisieren, da sie stark abhängig von den persönlichen Interessen eines Spielers sind [4]. Erkennbar wird dies durch die schärferen Kanten der dunklen Flächen. Der Grashüpfer, obwohl er den gesamten Raum inspiziert, sieht sich interessante Objekte länger an, während er an uninteressanten nur wenig Zeit verbringt.

Interaction Metrics: Diese Gruppe von Metriken bezeichnet jene Daten, die der Spieler bei der Interaktion mit der Spielwelt oder Objekten in dieser erzeugt. Darunter fallen sowohl Gesprächsauslöser mit *NPCs*[12] (also Interak-

[12]Non-Player Characters

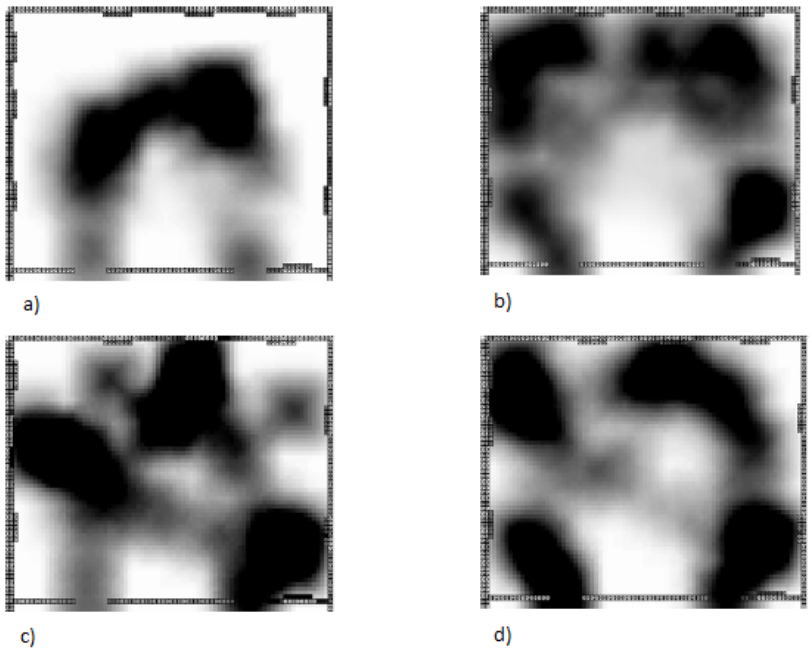


Abbildung 2.9: Bewegungsverhalten unterschiedlicher Spieler beim Untersuchen eines Raums. a) Fischstil, b) Ameisenstil, c) Grashüpferstil, d) Schmetterlingstil. Aus [4].

tion mit Entitäten) als auch das Feuern einer Waffe oder, in Bezug auf Spiele wie *Minecraft*, das Setzen eines Blocks (was unter Interaktion mit Objekten oder der Spielwelt selbst eingliederbar ist). Es kann hilfreich sein, zu diesen Datensätzen Positionen und Zeitwerte zu speichern. Für einen schnellen Überblick auf beispielsweise Tortendiagrammen reicht aber ein quantitativer Wert für jede Interaktionsmöglichkeit. Diese Art von Aufzeichnung kam auch beim Definieren von Personas in dem Projekt, das in Kapitel 3 beschrieben wird, zur Anwendung.

Narrative Metrics: Vor allem in Rollenspielen, in denen Spieler häufig mit *NPCs* interagieren, kommen *Narrative Metrics* zum Tragen. Sie beschreiben, welche Entscheidungen ein Spieler innerhalb von Dialogen trifft, ob er optionale Missionen annimmt und mehr. Actionrollenspiele wie *Mass Effect* profitieren von diesen Daten besonders: Das Spiel bedient sich nichtlinearem Storytelling. Anhand der Informationen, die durch die Analyse von *Narrative Metrics* gesammelt werden können, lässt sich ermitteln, ob Teile des Entscheidungsbaums verbesserungswürdig sind.

Angenommen, ein Spieler muss sich dafür entscheiden, eine von einer globalen Krankheit geplagte Rasse entweder völlig auszurotten, um die Krankheit vollständig zu vernichten, oder die Rasse ins Exil zu schicken, wobei nicht

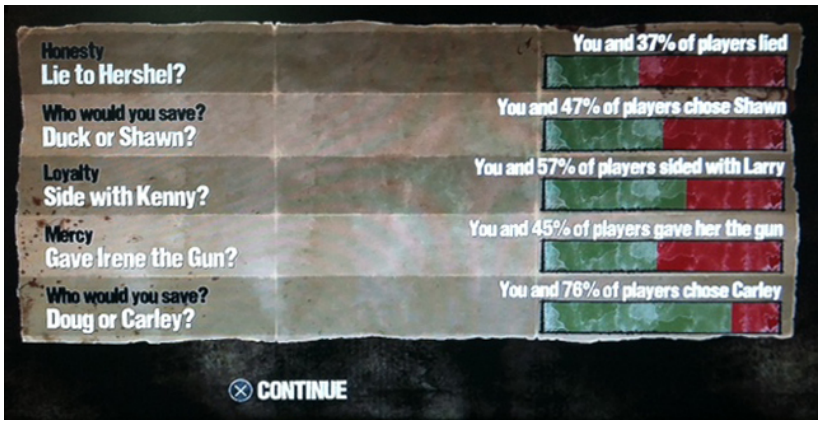


Abbildung 2.10: Der am Ende eines Kapitels angezeigte Entscheidungsbildschirm. Im betroffenen Kapitel standen fünf Entscheidungen zur Auswahl. Der Spieler kann sehen, wieviele Personen sich bisher genau wie er bzw. gegensätzlich entschieden haben. Aus [34].

klar ist, ob die Krankheit sich nicht trotzdem auf andere Rassen ausbreiten kann. Der Spieler steht somit vor einer Entscheidung, die die darauffolgende Geschichte massiv beeinflussen kann. Wenn nun aufgezeichnet wird, dass sich nur 1% der Spieler für die erste Variante entscheiden, so gilt es in Erwägung zu ziehen, die Auswirkungen der Entscheidung zu verdeutlichen oder diese Entscheidung völlig aus dem Spiel zu nehmen. Dadurch kann die zu 99% gewählte Antwort mehr Entwirklungszeit in Anspruch nehmen und somit optimiert werden.

Das 2012 erschienene *The Walking Dead*, bei dem es ebenfalls hauptsächlich um eine nonlineare Geschichte geht, verwendet diese Methode sogar dazu, die Wiederspielbarkeit zu erhöhen. Am Ende jedes Kapitels wird ein Bildschirm angezeigt, an dem die Spieler erkennen können, wie andere bei den verschiedenen Entscheidungen reagiert haben (siehe Abb. 2.10). Das kann den Spieler dazu anregen, einzelne Kapitel noch einmal zu spielen und andere Entscheidungen zu treffen.

Zu den *Narrative Metrics* zählen außerdem Missions- bzw. Aufgabenabschlusszeiten. Sie geben ebenfalls Aufschluss über Spielstile der verschiedenen Personas.

Interface Metrics: Diese speziellen *Gameplay Metrics* beschreiben die Interaktion zwischen Spieler und *GUI*. Um einen Spieler einer Persona zuordnen zu können sind aber hierbei nur Metriken von Relevanz, die die Entwicklung des Charakters angehen (beispielsweise die Navigation in einem *Level-Up*-Bildschirm).

Diese vier Kategorien werden von Tychsen und Canossa als *character-bound metrics*, also charaktergebundene Metriken definiert [25].

Durch die Analyse einzelner Spielstile lassen sich Muster erkennen. Wird nun festgestellt, dass gewisse Muster in vielen Spielern erkennbar sind, so können diese Muster zusammengefasst werden, um Personas zu erstellen. Tychsen und Canossa beschreiben diese folgendermaßen [25]:

> „Personas developed for use in early-phase game design [...] take the form of coherent patterns of play, deconstructed in terms of specific metrics and expressed quantitatively."

Mit Hilfe von Personas ist es also möglich, Gruppen von Spielern als eine einzelne Entität anzusehen. Im folgenden Abschnitt wird am Beispiel von *Hitman: Blood Money* gezeigt, dass viele Spieler die Spielmodi „leise", „nicht-tödlich" und „gründlich" häufiger als ihre Pendants verwenden. Dabei stehen die Spielstile „Intelligenz" und „Heimlichkeit" im Vordergrund. Dieses Spielmuster wird verwendet, um die Persona „stiller Attentäter" zu definieren.

Mit dieser Methode kann das Game Design so angepasst werden, dass die Bedürfnisse aller relevanten Personas zu jederzeit erfüllt werden. Sind beispielsweise drei unterschiedliche Personas bei der Entwicklung eines Spiels relevant, so können diese die Implementierung neuer Features beeinflussen, indem Rücksicht auf ihre bevorzugten Spielmodi und -stile genommen wird. Im nächsten Abschnitt wird dies anhand einiger Fallbeispiele näher beschrieben.

2.3 Fallbeispiele

Viele aktuelle Entwicklerstudios benutzen Gameplay Metrics, um ihr Produkt so erfolgreich wie möglich zu machen. Im Folgenden sollen einige Methoden solcher Entwickler aufgezeigt werden.

2.3.1 Halo 2

Als First Person Shooter hat *Microsofts Halo* den Bereich der *Xbox*-Spiele geprägt [45]. Die Nachfolger und Nebenprodukte waren sogar noch erfolgreicher [44]. Möglicherweise war einer der Gründe für diesen Erfolg der Einsatz von *Player Metrics* zur Verbesserung des Game Designs.

Das Spiel ist in zwölf Missionen aufgeteilt, bei der jede bis zu einer Stunde dauert. Jede Mission besteht aus mehreren Feindkontakten, bei denen unterschiedliche Gegnertypen in Wellen auf den Spieler zukommen. Dieser Aufbau ermöglicht eine hierarchische Visualisierung der *Player Metrics*.

Kim, Gunn et al. stellen in [16] einen Teil des von ihnen für *Halo 2* entwickelten Metriksystems vor. Hierbei machen sie sich die *Skinner Box* [23] und Thorndikes *Puzzle Boxes* zunutze [24]. Skinners Maschine zeichnete automatisch das Verhalten der Tiere auf, die er studierte. Folgende drei Vorteile waren ausreichend, um das Projekt durchzuführen:

- Der Leiter des Experiments musste während den Testsessions nicht anwesend sein,
- Die Zuverlässigkeit der Daten war dadurch erhöht, dass kein Eingreifen des Experimentierenden nötig war und
- Sich wiederholendes Verhalten konnte über längere Zeit beobachtet werden, wodurch es möglich war, neue Erkenntnisse in der Tierforschung zu gewinnen.

Microsofts Entwickler bilden dieses System auf moderne Softwarearchitektur ab. *TRUE*[13] wurde entwickelt um sogenannte *UIEs*[14] aufzuzeichnen. Hierbei standen vorerst primär Officeanwendungen wie *Word* und *Excel* im Vordergrund. Das Programm ermöglicht es, gewisse Eigenschaften vorzugeben, die während der Softwareentwicklung aufgezeichnet werden sollen. Dies hat den Vorteil, dass überschüssige, zur Analyse nicht unbedingt notwendige Daten vorsortiert und nur relevante Informationen aufgezeichnet und angezeigt werden.

Bei der Aufzeichnung dieser *Events* werden nicht nur die Häufigkeit des Auftritts sondern auch kontextbezogene Daten gespeichert. Angenommen, man verwende *TRUE* an einem FPS, so würden bei jedem Abschuss die verwendete Waffe, Position von Täter und Opfer, Kollisionsposition der Kugel, möglicherweise auch Umweltbedingungen (Wind, verzogene Projektilbahn)

[13] Tracking Real-Time User Experience
[14] User initiated events

und ähnliche *Gameplay Metrics* aufgezeichnet. Solche Gruppen von Metriken werden von Kim et al. als *Event Sets* bezeichnet.

Abgesehen von den *Gameplay Metrics* können mit *TRUE* auch qualitative Daten in Form von Fragestellungen an Benutzer gesammelt werden. So kann beispielweise nach jedem gespielten Level oder Spielabschnitt ein Dialogfenster erscheinen, an dem die Spieler zwischen den Antwortmöglichkeiten „zu leicht“, „gerade richtig, ich mache Fortschritt“, „zu schwer, ich weiß nicht, was ich tun soll“, „zu schwer, ich weiß nicht, wo ich hin soll“ und „zu schwer, ich sterbe ständig“ auswählen. Durch die Kombination der in der Testsession gesammelten Daten und der Umfrageergebnisse können wichtige Rückschlüsse gezogen werden.

TRUE enthält ein zusätzliches Feature, das in vielen anderen Metriksystemen eher rar gesät ist: Die Software ermöglicht es, Gameplayvideos aufzuzeichnen und diese mittels Zeitwerten mit gesammelten *Gameplay Metrics* zu kombinieren. Soll also beispielsweise ein ganz spezifischer Feindkontakt oder gar eine einzelne Welle genauer untersucht werden, so besteht die Möglichkeit, relevante Gameplayvideos manuell zu analysieren.

Wie in Abschnitt 2.1 bereits beschrieben kann es hilfreich sein, verschiedene Detailgrade bei der Visualisierung der *Player Metrics* bereitzustellen. In *TRUE* wird diese Methode ebenfalls verwendet. Hierbei sind die einzelnen Visualisierungsebenen interaktiv nutzbar. Wird beispielsweise auf einen Abschnitt innerhalb eines Balkendiagramms (beispielsweise Anzahl Spielertode in Level x) geklickt, so erfolgt eine Weiterleitung auf eine detailliertere Ansicht (wie z. B. Spielertode pro Levelabschnitt oder Ursachen der Tode).

Ziele

Bei der Analyse von *Halo 2* galt herauszufinden, ob es unbeabsichtigte Erhöhungen der Schwierigkeit im Einzelspielermodus gab. Neben den aufgezeichneten Daten sollte auch die Einschätzung der Tester, vor allem im Vergleich zur vorhergehenden Testsession, in Betracht gezogen werden (unter Verwendung oben erwähnten Fragebogens).

Event Sets

Um eine Analyse der Schwierigkeit zu ermöglichen wurde bei den Testsessions ein *Event Set* für das *UIE* „Spielertod“ definiert. Es enthält u. a. folgende *Gameplay Metrics*:

- Zeitpunkt des Todes,
- Auslöser („Mörder“),
- Ursache (z. B. Waffe oder Sturz).

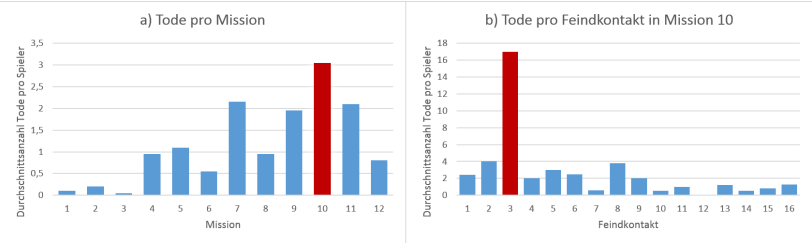


Abbildung 2.11: Anzahl der Spielertode in Halo 2 a) pro Mission und b) pro Feindkontakt. Aus [16].

Testsessions

Die erste Testsession sollte klarstellen, in welchen Missionen und zu welchen Zeitpunkten signifikante Veränderungen des Schwierigkeitsgrades bestehen. Wie in Abbildung 2.11 ersichtlich ist, starben die Teilnehmer in Mission 10 durchschnittlich drei Mal, was wesentlich höher ist als in den anderen Levels. Näher betrachtet kann festgestellt werden, dass der dritte Feindkontakt von unerwartet vielen Spielertoden geprägt war. Eine Erkenntnis, warum die Spieler gerade an dieser Stelle so oft starben, lässt sich jedoch nur schwer durch diese Beobachtung allein gewinnen. *TRUE* ermöglicht es aufgrund des hierarchischen Aufbaus der Ansichten, noch detailliertere Daten darzustellen. So konnte ermittelt werden, dass 85% aller Spielertode in diesem Abschnitt durch *Brutes* (einem granatenwerfenden Gegnertyp) verursacht wurden. Der Detailgrad wurde um noch eine Stufe erhöht, was folgendes Ergebnis brachte: 27% der Spieler staben durch Nahkampfangriffe und insgesamt 41% durch feindliche Granaten.

Trotz dieser Erkenntnis war es zu jenem Zeitpunkt nicht klar, warum Granaten die Hauptursache für Tode an der betroffenen Stelle waren. Da in *TRUE* aber jedes aufgezeichnete *UIE* mit einem Gameplayvideo verknüpft ist, konnte jeder Datenbankeintrag einzeln analysiert werden. Dabei stellte sich heraus, dass die Wurfbahn der Granaten in dem betroffenen Abschnitt wesentlich kürzer war als bei den anderen Feindkontakten, sodass die Spieler weniger Zeit hatten um zu reagieren.

Aufgrund dieser Erkenntnisse wurden nach der ersten Test Session einige Veränderungen in diesem Spielabschnitt vorgenommen. *Brutes* werfen nun in diesem Spielabschnitt gar keine Granaten mehr. Zusätzlich wurde auch die Anzahl der Nahkampftode verringert. Deren Ursprung lag in der Tatsache, dass einige *Brutes* oftmals hinter dem Spieler auftauchten. Hierbei wurde einfach der Bereich, in dem Gegner erscheinen konnten, verschoben.

Die zweite Test Session sollte die Änderungen im Game Design verifizieren, sowohl mit dem Verhalten der Spieler, was in niedrigeren Sterbebalken in den entsprechenden Diagrammen resultieren würde, als auch mit den nach jedem Feindkontakt erscheinenden Fragebögen. Die Ergebnisse zeigten, dass

die Anzahl der Spielertode in diesem Abschnitt um ca. 90% verringert werden konnten. Von den Entwicklern befürchtet wurde allerdings, dass diese Stelle des Spiels nun zu einfach war. Mittels der entsprechenden Fragebögen konnte dies aber widerlegt werden. In Test Session 1 wählten 43% der Tester die Antwort „zu schwer, ich sterbe ständig“. Im Vergleich dazu wählten nach der Veränderung des Abschnitts nur 9% diese Antwort. 74% sind der Meinung, die Schwierigkeit wäre „gerade richtig“.

Fazit

Dieser in [16] beschriebene Fall zeigt also, dass es mithilfe der von *Microsoft* entwickelten Metriksoftware *TRUE* möglich ist, Probleme im Game Design effizient ausfindig zu machen. In Verbindung mit qualitativen Fragebögen kann das mittels des Metriksystem identifizierte potenzielle Problem als solches bestätigt werden.

2.3.2 Tomb Raider: Underworld

Die *Tomb Raider*-Reihe ist eine seit mehr als 15 Jahren existierende Action-Adventure-Serie bei der der Spieler in Third-Person-Ansicht durch Levels manövrieren, Rätsel lösen und Feinde besiegen muss. Drachen und Canossa zeigen in [7], dass es möglich ist, Problemstellen und oftmals auch deren Ursachen mithilfe von *GIS*[15] zu identifizieren.

Ziele

Bereits bekannte Techniken zur Analyse von *Gameplay Metrics* sollten geprüft und validiert werden. Im Vordergrund steht hierbei, wie diese Daten verwendet werden können um Game Design zu verbessern. Außerdem werden *GIS* vorgestellt, die die Visualisierung von positionsbezogenen Metriken beliebiger Spielsituationen ermöglichen. Drachen und Canossa beschreiben *GIS* folgendermaßen [7]:

> „A GIS is a computerized data management system used to capture, store, manage, retrieve, analyze, query, interpret and display spatial information in the form of e.g. maps, reports and charts.“

Die im vorhergehenden Abschnitt erwähnte Software *TRUE* zählt also laut dieser Definition ebenfalls zur Gruppe der *GIS*. In einem *GIS* werden positionsbezogene *Gameplay Metrics* oftmals mit *Geospatial Metrics* referenziert. Diese beinhalten sowohl einen geographischen als auch einen thematischen Teil. Somit wird einerseits die Methodik der geografischen Visualisierung (z. B. Punkt, Linie oder Bereich) sowie andererseits das zugehörige Ereignis

[15] Geographic Information Systems

(beispielsweise der Tod eines Spielers) in einem Datensatz gespeichert. Ein solches System ermöglicht es, die beschriebenen *Geospatial Metrics* auf Levelkarten o. ä. zu visualisieren und zu analysieren. So werden in [7] alle fünf Sekunden die Positionen von Spieltestern aufgezeichnet um festzustellen, ob diese sich wie erwartet durch die verschiedenen Levels bewegen, und wenn nicht, an welchen Situationen welche Probleme aufgetreten sind.

Im Vergleich zu *Halo 2* wurden bei *Tomb Raider: Underworld* keine interaktiven Visualisierungen verwendet. Die Informationen ließen sich lediglich als Ebenen über die betroffenen Levelkarten legen und analysieren. Als Analysesoftware diente bei der Entwicklung die *EIDOS Metrics Suite*, die speziell für von *EIDOS* produzierte Spiele erschaffen wurde. Die *Suite* besteht nicht nur aus mehreren Softwareschichten (SQL-Server, XML-Parser, ...), sie berücksichtigt bei der Analyse auch manuell erstellte Testdaten (beispielsweise in Form von Fragebögen). Das verwendete *GIS* trägt den Namen *ArcGIS* und ist mit den von der *EIDOS Metrics Suite* bereitgestellten Daten kompatibel.

Test und Analyse

Die in [7] gezeigten Daten entstammen von etwa 28000 Spielern, die *Tomb Raider: Underworld* kurz nach Veröffentlichung gespielt haben. Das Spiel beinhaltet sieben Levels, wobei jedes in mehrere Abschnitte unterteilt ist, von denen die *Metrics Suite* Daten sammelt. Einer der komplexeren Levelabschnitte – *Valaskjalf* – soll hier genauer beschrieben werden. Zunächst wurde ein Gitternetz über die Karte des Levelabschnitts gelegt. Dies ermöglichte es, eine Heatmap über die Position der Spielertode zu erzeugen (siehe Abbildung 2.12 a). *ArcGIS* ermöglicht auch die in Abbildung 2.12 b) gezeigte Visualisierung von Todesursachen pro Zelle. So konnten tödliche Stellen schnell identifiziert werden. Neben den beiden gezeigten Grafiken wurde auch noch die Möglichkeit genutzt, eine Heatmap über die Anzahl unterschiedlicher Todesarten pro Zelle zu betrachten.

Die Software hat, wie in Abbildung 2.12 a) ersichtlich, vier möglicherweise verbesserungswürdige Levelbereiche hervorgehoben. Alle vier sind von einer hohen Sterberate geprägt. Der mit der höchsten Sterberate markierte Bereich 1 ließ sich für die Entwickler schnell als Problemstelle identifizieren. Dank zuletzt erwähnter Heatmap konnte festgestellt werden, dass es in dieser Zelle lediglich zwei Ursachen für den Tod der Spieler gab: Die Kombination aus einem feindlichen Angriff und einer schwierigen Sprungpassage brachte die Spieler in schwierige Situationen, die oft zum Tod führten. Bereich 2 zeigt ebenfalls nur zwei verschiedene Todesarten, nämlich der Tod durch Taranteln und Fallen. Da die gemessene Gesamtanzahl an Toden durch Taranteln in diesem Abschnitt allerdings sehr gering war, wurde vorgeschlagen, den verursachten Schaden durch die betroffenen Fallen zu verringern. In Bereich 3 überschneidet sich die Heatmap über die Anzahl der verschiedenen Todesarten mit der, die die Gesamtanzahl der Tode zeigt. Hier ist es mög-

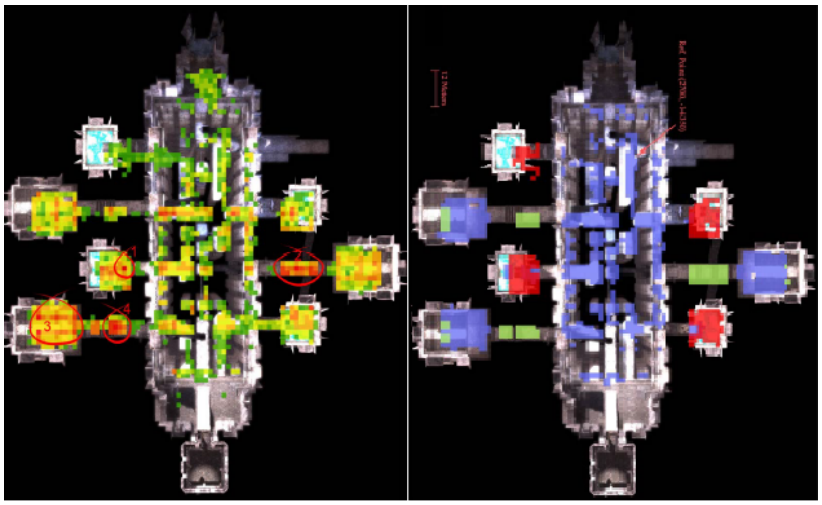


Abbildung 2.12: a) Anzahl der Spielertode im Levelabschnitt *Valaskjalf*, eine rote Färbung entspricht 3050 Tode in einer einzelnen Zelle. b) Auslöser der Spielertode: Absturz (blau), Falle (grün) und Ertrinken (rot). Aus [7].

lich, durch eine Vielzahl an Einflüssen zu sterben. Wie in Bereich 1 wurde auch hier vorgeschlagen, die Schwierigkeit nochmals genauer zu analysieren und gegebenfalls anzupassen. Dieser Raum stellt allerdings den Höhepunkt dieses Levels da, es ist also durchaus möglich, dass die Game Designer mit einer so hohen Sterberate gerechnet haben. Die Analyse von Bereich 4 bringt ähnliche Ergebnisse wie die von Bereich 2.

Fazit

Die Studie zeigt als, dass mögliche Fehlerquellen im Level Design von *Tomb Raider: Underworld* (in Form von zu komplexen Plattformrätseln oder zu starken Feinden) mithilfe von *ArcGIS* ermittelt werden können. Diese Problemanalyse lässt sich mit qualitativen Daten (wie beispielsweise Fragebögen) zusätzlich validieren. Über die Definition von Spielstilen wird beim Einsatz der *EIDOS Metrics Suite*, zumindest bei der Entwicklung von *Tomb Raider: Underworld*, hinweggesehen. Außerdem wird der Zeitfaktor ignoriert, der vor allem bei der Analyse des Verhaltens der Spieler von Relevanz ist. Die Studie von Drachen und Canossa zeigt, dass positionsbezogene *Gameplay Metrics* oftmals mehr wert sind als die simple Anzeige, wieviele Spieler insgesamt gestorben sind. Vor allem auch die Möglichkeit der Betrachtung mehrerer Parameter auf einmal macht *ArcGIS* zu einer mächtigen Software im Bereich der *Gameplay Metrics*.

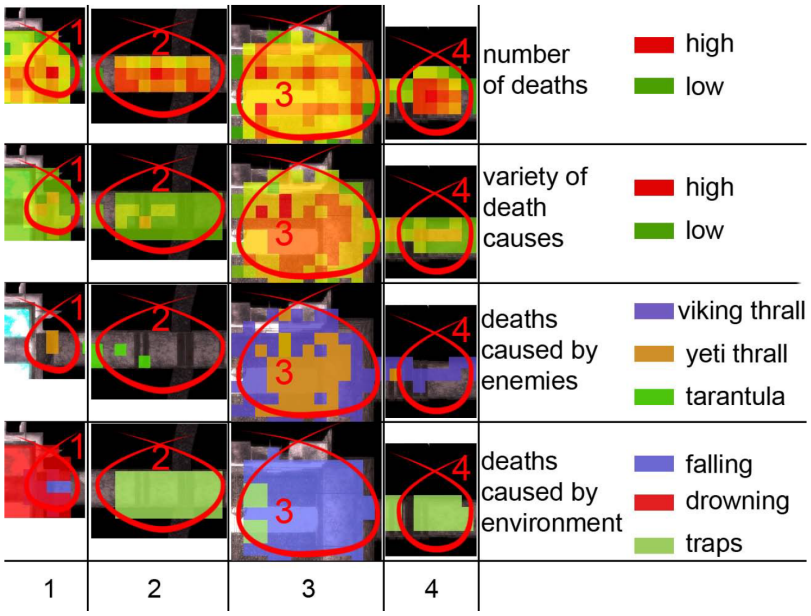


Abbildung 2.13: Die in Abbildung 2.12 a) hervorgehobenen Gebiete im Detail betrachtet. Aus [7].

2.3.3 Split/Second

Das von den *Disney Interactive Studios* 2010 veröffentlichte Arcade-Rennspiel *Split/Second: Velocity* definiert sich durch eine auf Knopfdruck zerstörbare Umgebung. Durch Manöver wie Windschattenfahrten, lange Drifts u. ä. lädt sich während des Rennens eine *Powerplay*-Anzeige. Erreicht diese einen gewissen Wert, können Objekte auf der Strecke zerstört werden um alternative Routen zu erzeugen. So kann beispielsweise ein Kran zerstört werden der einerseits einen Weg blockiert, andererseits eine Rampe erzeugt, die es ermöglicht, auf einem anderen Teil des Levels zu fahren. Durch die rechtzeitige Aktivierung dieser Objekte werden vom Einsturz betroffene Gegner kurzzeitig außer Gefecht gesetzt. In [3] präsentieren Chapresto und Mitchell das von den *Black Rock Studios*, dem Entwickler von *Split/Second*, verwendete Metriksystem *Tracktivity*.

Ziele

Chapresto und Mitchell setzen sich zum Ziel, *Tracktivity* vorzustellen, sowie Visualisierungstechniken aufzuzeigen an denen sich erkennen lässt, ob die Spielerfahrung gut genug ist um den Spielern Spaß zu machen. Hierbei wird vor allem auf die Anpassung der dynamischen Schwierigkeitsberechnung eingegangen. Dabei wird als Maßstab nicht wie in den bisherigen Beispielen das

Level Design, sondern das Können der K.I.[16] verwendet. Außerdem soll gezeigt werden, wie die *Powerplays*, *Split/Seconds*' *USP*[17], optimiert werden können.

Test und Analyse

Das System *Tracktivity* wurde speziell für kompetitive Spiele mit Bestenlisten geschaffen. Es funktioniert auf Client-Server-Basis. Um eine Auslastung der Netzwerkbandbreite zu verhindern werden die gesammelten *Gameplay Metrics* offline vom Client gesammelt und erst nach Abschluss eines Rennens zum Server gesendet. Dieser Server erhält die Daten mittels einer *POST*-Anfrage. Durch die Anfrage wird ein Prozess gestartet, der die Daten analysiert und kategorisiert.

Während der Entwicklung wurden 32 Test Sessions mit je 7 bis 14 Teilnehmern abgehalten. Jeder Tester sollte acht bis zehn Stunden im Einzelspielermodus verbringen. Die erste Verwendung des Metriksystems bestand in einem *Powerplay Report* der, in Bezug auf oben erwähntes *Powerplay*, aussagen sollte, ob diese *USP* ausreichend genutzt wird und genug Einfluss auf das Spiel nimmt. Dazu wurden zu jedem *Powerplay* u. a. die *Gameplay Metrics* „auslösender Spieler" und „zerstörte Spieler" aufgezeichnet. Die Aufzeichnung dieser Daten ermöglicht es, mehrere Aspekte zu analysieren.

Zunächst wurde das Verhältnis zwischen möglichen und durchgeführten *Powerplays* geprüft. Dabei ergaben sich teils unerwartete Ergebnisse. Oftmals wurden spektakuläre Ereignisse nicht ausgelöst, weniger interessante allerdings schon. Vor allem wegen der hohen Entwicklungszeit eben dieser *Powerplays* war diese Tatsache nicht wünschenswert. Die Spieler konnten nicht genug Power-Ups sammeln, um dieses Event zu aktivieren. Nach anschließendem Interview mit den Testern konnte festgestellt werden, dass die Spieler sich des großen *Powerplays* nicht bewusst waren und deswegen nicht genug Energie dafür sparten. Als Lösung für dieses Problem dienten zwei Änderungen: ein vorhergehendes, kleineres *Powerplay* wurde entfernt und die Chance, dass die K.I. das größere auslöst, erhöht. Der oben erwähnte Verhältniswert veränderte sich in die von den Entwicklern gewünschte Richtung.

Auch ein zweiter Aspekt kann mittels der oben erwähnten *Gameplay Metrics* behandelt werden. So wurde das Verhältnis zwischen der Anzahl an Spielern, die durch ein *Powerplay* zerstört wurden, zu der Zahl, wie oft dieses ausgelöst wurde, aufgezeichnet. Diese Daten ermöglichten es, wenig benutzte *Powerplays* oder schlecht definierte Gefahrenbereiche zu identifizieren. Als Beispiel wird in [3] eine Strecke erwähnt, bei der an einer Stelle zwei *Powerplays* auslösbar sind. Hier stellte sich heraus, dass eines der beiden keine einzige Spielerzerstörung auslöste, das andere jedoch durchschnittlich mehr

[16] Künstlichen Intelligenz
[17] unique selling proposition

als einen Spieler gleichzeitig vernichtete. Als Reaktion auf diese Ergebnisse wurden bei den beiden *Powerplays* Aktivierungs- und Schadensbereiche angepasst.

Als wichtig stellte sich auch der Faktor, wie oft Spieler sich mit einem *Powerplay* selbst zerstören, heraus. Es besteht nicht nur die Möglichkeit, dass der Spieler, kommt er in mehrere dieser Situationen, frustriert wird. Auch erleichtern diese Daten das Finden von Fehlern. So gab es beispielsweise einige Stellen, andem dieser Faktor mehr als 100% betrug. Bei genauerer Inspektion stellte sich heraus, dass der Algorithmus, der die Spieler nach einer Zerstörung wieder ins Spiel bringt, fehlerhaft war, wodurch Spieler mehrmals durch dasselbe *Powerplay* zerstört wurden.

Die oben erwähnten Daten wurden nach dem *Powerplay Report* auch streckenweise und global erstellt, sodass Strecken mit höheren Selbstzerstörungswerten identifiziert werden konnten. Diese Strecken zählen zu den schwierigeren. Um die Lernkurve abzuflachen sind diese Strecken erst gegen Ende des Spiels befahrbar.

Neben der Analyse der *Powerplays* stellte auch die Optimierung der K.I. ein zentrales Thema bei der Verwendung von *Gameplay Metrics* dar. In *Split/Second* zählt die künstliche Intelligenz zu den Hauptfaktoren bei der Definition des Herausforderungsgrads. Sind die Gegner zu schwach, kann das Spiel schnell langweilig werden. Liefern sie hingegen nur perfekte Rennergebnisse, wird der Spieler nicht belohnt und verliert so den Spaß am Spiel. Abgesehen davon ist auch noch von einer Steigerung der Spielerleistung auszugehen, was ein zusätzliches Problem darstellen kann. Die Entwickler von *Split/Second* bedienen sich des in Abschnitt 2.1 erwähnten *DCB*-Werts. Dieser Wert entspricht der Fahrleistung der computergesteuerten Gegner. Jeder Spieler beginnt mit einem *DCB* von 0, was der eine geringe Leistung der K.I. mit sich bringt. Anhand der Platzierung, die ein Spieler nach einem Rennen erreicht, wird ein neuer Wert berechnet. Dazu werden mehrere Faktoren miteinkalkuliert, darunter auch vordefinierte Konstanten für Episode und Strecke. Die Schwierigkeitsberechnung vor der ersten absolvierten Strecke wird in Gleichung 2.1 aus [3] gezeigt. Die beiden K-Werte stellen Konstanten dar, D_{Rennen} gibt den finalen Schwierigkeitswert an:

$$D_{Rennen} = DCB + K_{Episode} + K_{Strecke}. \tag{2.1}$$

Für die Berechnung des *DCB*-Werts müssen zwei Parameter definiert werden. Einerseits muss der bisherige *DCB*-Wert kompensiert werden. Sollte der bisherige *DCB*-Wert über $0,7$ liegen, so nimmt $Komp(DCB_i)$ $-0,01$ bzw. (bei einem Wert über $0,9$) $-0,02$ an. Diese Kompensierung verlangsamt den Anstieg der Schwierigkeit bei höheren *DCB*-Werten, also bei konstant guter Spielerleistung. Weiters wird der Faktor berechnet, wie groß die Schwierig-

keitsdifferenz zum nächsten Rennen sein soll:

$$K_{\text{diff}} = \max(\sum_{j=i}^{i-N} f(Pos_j), 1), \tag{2.2}$$

wobei Pos_j hierbei die Ranglistenposition des Spielers darstellt [3]. Wie hoch $f(Pos_j)$ letztendlich ist wird anhand einer Tabelle vordefiniert. Die Differenzberechnung ermöglicht es, die Schwierigkeit für erfahrenere Spieler schneller anzupassen, für Anfänger jedoch trotzdem eine konstante, möglicherweise auch nur geringfügige Erhöhung des Schwierigkeitgrades sicherzustellen.

Sind diese Werte ermittelt, kann die Berechnung des aktuellen *DCB*-Werts anhand folgender Formel durchgeführt werden:

$$DCB_{i+1} = DCB_i + K_{\text{diff}} \cdot (\text{Offset}_{Pos} + \text{Komp}(DCB_i)), \tag{2.3}$$

wobei Offset_{Pos} ebenfalls ein konstanter, von der Ranglistenplatzierung abhängiger Wert ist [3].

Rang	Offset_{Pos}	$f(Pos)$
1	$+0,030$	$+0,50$
2	$+0,015$	$+0,25$
3	$0,000$	$0,00$
4	$-0,015$	$-1,00$
5	$-0,030$	$-2,00$
6	$-0,045$	$-4,00$
7	$-0,060$	$-4,00$
8	$-0,075$	$-4,00$

Tabelle 2.1: Offset_{Pos} gibt einen Verschiebungswert zum vorhergehenden *DCB*-Wert an, $f(Pos)$ einen zu Balancingzwecken verwendeten Wert. Beide werden anhand der Spielerleistung gewählt.

Fazit

In Abbildung 2.14 a) wird ersichtlich, dass das Spiel für zwei bestimmte Spieler ab einem gewissen Zeitpunkt zu schwer wurde. Viele Spieler kamen während der gesamten Spielzeit auch nicht über einen *DCB* von 0 hinaus, sie mussten also regelmäßig Rennen wiederholen, um weiterzukommen. Nach einigen Anpassungen der oben erwähnten konstanten bzw. ranglistenabhängigen Werte konnten die in Abbildung 2.14 b) gezeigten Ergebnisse erreicht werden. Kein Spieler beendete das Spiel mit werden unter $0,2$ und in den meisten Fällen ist ein kontinuierliches Wachstum des *DCB*-Werts zu sehen.

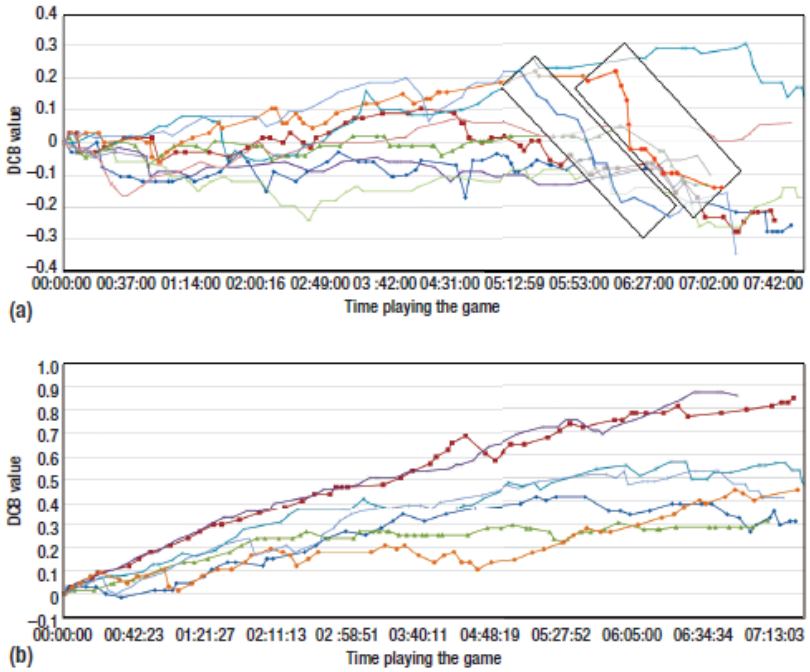


Abbildung 2.14: Gezeigt werden die *DCB*-Werte mehrerer Spieler über Zeit. a) zeigt Ergebnisse einer frühen Version von *Split/Second*, b) hingegen die Ergebnisse des verbesserten Algorithmus' kurz vor Veröffentlichung. Aus [3].

Daraus lässt sich schließen, dass die Spieler stetig dazulernten und auch die Schwierigkeit des Spiels sich diesem Lernprozess anpasste. Anhand dieser Grafik kann die Vermutung aufgestellt werden, dass sich die meisten Spieler während einem Großteil des Spiels im bereits erwähnten *Flow Channel*, also in gutem Verhältnis zwischen Herausforderung und Können, waren [6].

2.3.4 Hitman: Blood Money

Tychsen und Canossa behandeln in [25] den First-Person-Shooter *Hitman: Blood Money*. Der Spieler nimmt hierbei die Rolle eines Auftragsmörders an, der seine Missionen auf vielfältige Weise zu erfüllen.

Ziele

Diese Studie soll hervorbringen, ob sich mit dem Nutzen von *Gameplay Metrics* herausstellen kann, auf welche Weise Spieler Aufträge in *Hitman: Blood Money* absolvieren. Dadurch sollen Spielstile definiert werden anhand derer festgestellt werden kann, ob die implementierten Möglichkeiten, das Spiel zu

spielen, ausreichend genutzt werden.

Test und Analyse

Für eine Analyse der von den Testkandidaten gespielten Sessions musste vorerst entschieden werden, welche Daten gesammelt werden sollten. Dies geschah in drei Schritten.

Modus: Der Spielmodus beschreibt die Erzeugung einer gewissen Metrik innerhalb einer Gruppe von Metriken, deren Aufzeichnung vom Spieler ausgelöst wird. Als Beispiel kann hierbei Methodik beim Absolvieren der Aufträge dienen: Wurden tödliche oder nicht-tödliche Waffen benutzt, konnten Gegner leise ausgeschaltet werden oder erregte der Spieler Aufmerksamkeit? Detailliertere Informationen wie die verwendete Waffe oder der Ort des Geschehens sind hierbei nicht relevant. Diese Daten ermöglichen die Definition von *Interaction Sets*. Ähnlich der bereits beschriebenen *Event Sets* werden diese *Gameplay Metrics* gruppiert und lassen sich so in *Interaktionsmodi* oder, speziell im Fall von Videospielen, Spielmodi einteilen.

Stil: Zur Definition von Spielstilen werden, anders als bei den Spielmodi, viele verschiedene *Gameplay Metrics* benötigt. Ein Spielstil bedient sich einer Gruppe von Spielmodi, *Navigation* sowie *Interaction Metrics*. Er beschreibt lediglich einen situationsspezifischen Stil, den der Spieler in verschiedenen Gegebenheiten anwendet. Vor allem in Spielen mit größeren Communities werden verschiedene Spielstile bekannt. In *Guild Wars* gibt es beispielsweise mehrere *Characterbuilds* wie den *Trapper*, der darauf spezialisiert ist, Fallen zu stellen oder den *Protter*, der einen Mönch darstellt, der keine Heilfähigkeiten, sondern nur Verteidigungsfertigkeiten (wie z. B. Schadensreduktionszauber) besitzt. Welchen Spielstil ein Spieler bevorzugt, ist ihm überlassen. Vor allem in Strategiespielen wie *Starcraft* kann es entscheidend sein, den Spielstil während einer Partie zu ändern. Angenommen, ein Spieler beginnt mit einem *Zergling Rush*, bei dem es darauf ankommt, zu Beginn einer Partie völlig auf Technologie zu verzichten um eine Vielzahl billiger Einheiten produzieren zu können. Hat er damit keinen Erfolg, so muss er so schnell wie möglich seine Strategie wechseln, um dem technologischen Vorteil des Gegners gewachsen zu sein.

Als weiteres Beispiel dient das *Bunny-Hopping*, das von *Quake* geprägt wurde[18]. Der Spieler versucht hierbei, so oft wie möglich zu springen um sich schneller fortzubewegen. Moderne *FPS* wie *Battlefield 3* oder *Call of Duty: Modern Warfare* verhindern dies mit detaillierten Animationen und kurzen Verzögerungen in der Bewegung beim landen.

[18] http://www.esreality.com/?a=longpost&id=528333&page=2

Aufzuzeichnen, ob ein Spieler im *Bunny-Hopping*-Stil spielt, fällt nicht weiter schwer. So kann beispielsweise die Anzahl der durchgeführten Sprünge pro bewegtem Meter gemessen werden. Einen *Zergling Rush* zu erkennen, kann schwerer fallen. Angefangen bei der Zeit, wann der erste *Brutschleimpool* im Bau ist über die Anzahl an Arbeitern vor der ersten Versorgungseinheit und den Zeitpunkt des ersten Feindkontakts bis zur Anzahl produzierter *Zerglinge* pro Zeiteinheit ist es hilfreich, die Bandbreite der aufzuzeichnenden Daten groß zu halten. Das Erfassen von Spielstilen, die in First- oder Third-Person-Spielen zur Anwendung kommen, erfordert oftmals auch positionsbezogene Daten, manchmal sogar Ergebnisse einer *Eye-Tracking*-Analyse.

Wenn nun von den Entwicklern festgestellt werden will, welche Spielstile in ihrem Spiel überhaupt möglich sind, so müssen sehr viele unterschiedliche Metriken aufgezeichnet werden. Es können allerdings Einschränkungen getroffen werden. Sollen beispielsweise lediglich rollenspielbezogene Spielstile erkannt werden (z. B. „Durchklicken“ durch Dialoge, ohne sie zu lesen oder bei Dialogoptionen immer die erste Wahl treffen) so ist das Aufzeichnen von Kampfmetriken nicht notwendig.

Persona: Nutzt ein Spieler ein oder mehrere Spielstile konsequent während des Spiels, lassen mit diesem Verhalten *Personas* definieren. Am Beispiel von *Hitman: Blood Money* kann dies folgendermaßen veranschaulicht werden: Ein Spieler, der Feinde im Nahkampf mit den Parametern „unauffällig“ und „gründlich“ beseitigt kann dies je nach Situation auch von der Ferne aus durchführen, entspricht aber trotzdem der *Persona* „Ninja“ (oder, wie in [25], „stiller Attentäter“). Solche *Personas* können auf unterschiedliche Weise definiert werden, bevor (*Metapher*) oder nachdem (*Linse*) eine spielbare Version des zu entwickelnden Spiels existiert:

- *Designermetapher:* Werden gewisse Spielstile vorgegeben, zwischen denen Spieler wählen können (wie beispielsweise die Parameter „gründlich“ oder „blutig“), so ist es öft möglich, daraus folgernd *Personas* zu erstellen. Während des Game Designs können diese verwendet werden, um bei der Entwicklung von Gameplayfeatures auf ein solches Verhalten einzugehen und die Möglichkeiten so für die verschiedenen Spielstile zu optimieren.
- *Designerlinse:* Ist ein Spiel allerdings so komplex, dass es schwer ist, im vorhinein Spielstile zu erkennen empfiehlt es sich, diese im Nachhinein zu analysieren und mittels Updates auf die dadurch erkannten *Personas* einzugehen. Stellt sich im Beispiel von *Starcraft* heraus, dass es Spieler gibt, die hauptsächlich in diversen *Rush*[19]-Stilen spielen, so könnten billige, frühzeitig ausbildbare Einheiten entsprechend angepasst werden. Vor allem bei *MMOG* stellt die Designerlinse ein wichtiges Entwicklerwerkzeug dar.

[19] Schnelle Einheitenproduktion auf Kosten technologischen Fortschritts

- *Spielermetapher: Personas* können auch direkt an die Spieler gebunden werden, wie beispielsweise bei *Hitman: Silent Assassin*. Der vom Spieler übernommene Charakter wird hierbei schon im Vorhinein vorgestellt. Seine Vorlieben und Methoden können in einer Einleitung gezeigt werden, sodass der Spieler erfährt, welche Spielstile möglich sind.
- *Spielerlinse:* Wenn sich Spieler mit einer Situation auseinandersetzen, in denen sie einen Charakter steuern, so können sie eine geistige Vorstellung dieser Figur entwickeln. Dadurch assoziieren Spieler den Sinn ihrer Interaktionen mit einem gewissen Charakterbild.

Bei der Entwicklung von *Hitman: Blood Money* wurde jede dieser Methoden benutzt. Die Definition der *Persona* „stiller Attentäter“ erfolgte bereits vor der Entwicklung, da sie die vorherrschende Rolle unter den Spielern darstellen sollte. Nicht nur, weil die Hauptperson als professioneller Auftragsmörder auftritt, sondern auch, weil der von dieser *Persona* genutzte Spielstil nach Missionsabschlüssen belohnt wird. Um eine größere Zielgruppe zu treffen entschieden sich die Entwickler, auch andere Spielstile zu entwickeln und entsprechende *Personas* zu definieren. Dazu dienten ihnen Navigations- und Interaktionsmöglichkeitsfelder.

In Möglichkeitsfeldern wird jede Art von Bewegung bzw. Interaktion abgebildet, die ein Spiel ermöglicht. Mithilfe verschiedener, vorher bereitgestellter Spielmodi können so Spielstile und zugehörige Metriken festgelegt werden. In *Hitman: Blood Money* wurden vier Spielstile im Navigations- und sechs im Interaktionsmöglichkeitsfeld definiert (siehe Tabelle 2.2).

Tabelle 2.2: Navigations- und Interaktionsmöglichkeiten in *Hitman: Blood Money* laut [25].

Navigation	**Interaktion**
Muskelkraft	nicht-tödlich
Intelligenz	tödlich
Rollenspiel	gründlich
Heimlichkeit	blutig
	leise
	laut

„Heimlichkeit“ bedient sich beispielsweise der Spielmodi „Schatten nutzen“, „in Schränken verstecken“, „Nahkampf“, „stehen“, „hocken“ und „kriechen“. Zu den zugehörigen *Gameplay Metrics* zählen „Zeit in Schatten und Schränken verbracht“, „Zeit kriechend verbracht“ und „Anzahl Nah-/leiser Kampfbewegungen“. Der oben erwähnte „stille Attentäter“ macht Gebrauch von den Spielstilen „Heimlichkeit“ und „Intelligenz“. Unterschiedliche Kombinationen dieser Stile können zu verschiedenen *Personas* führen, wie Tychsen und Canossa in [25] zeigen.

Das Interaktionsmöglichkeitsfeld bestand aus den drei Spielstilpaaren nicht-tödlich/tödlich, leise/laut und gründlich/blutig. Jedem Spielstil wurde eine Gruppe von *Interaction Metrics* zugeordnet, unter „tödlich“ fielen beispielsweise die meisten Waffenanwendungen, während zu „nicht-tödlich“ u. a. das Nutzen von Lichtschaltern zählte.

Der Spieler kann in *Hitman: Blood Money* anhand einer Grafik, an deren Achsen Werte für Heimlichkeit und Aggression widergespiegelt werden, seine eigenen Spielstile nachverfolgen. Dazu werden ihm die Titel „Massenmörder“, „stiller Attentäter“, „verrückter Schlächter“ und „Saubermacher“ verliehen (frei übersetzt aus [25]). Diese Titel sind eng mit den von den Entwicklern definierten *Personas* verwoben.

Fazit

Gezeigt wird, dass die Definition von *Personas* mit verschiendenen *Gameplay Metrics* ermöglicht werden kann. Je mehr Spielertypen mittels so beschrieben werden können, desto größer ist die Zielgruppe. Wird beim Game Design darauf geachtet, dass zu jeder Zeit sämtliche *Personas* ihre eigenen Spielstile nachverfolgen können, kann die Player Experience für Spieler, auf die diese *Personas* zutreffen, optimiert werden.

Kapitel 3

MetricsCraft

Viele der in Kapitel 2 erwähnten Beispiele unterstützen die Entwicklung eines Spiels. Das in diesem Kapitel beschriebene Projekt bedient sich verschiedener Ansatzweisen. Das Konzept der *Personas* wird bis zu einem gewissen Grad übernommen. Außerdem wird eine der automatischen Balancingtechnik aus [3] ähnliche Methode genutzt, um Leveldesign spielerspezifisch anzupassen. Das Konzept der verschiedenen Heatmap-Ebenen wurde aus [7] übernommen, um manuelle Analysen zu erleichtern.

Für jede Auswertung werden Daten benötigt. In diesem Fall sollen *Player Metrics* von *Minecraft* gesammelt werden. Dafür sprechen einige Gründe: *Minecraft* unterscheidet sich in vielen Bereichen von den meisten anderen Spielen, sogar die Eingliederung in ein oder mehrere Genres fällt nicht leicht. Gerade diese Tatsache macht die Analyse von Spielerverhalten umso interessanter. Außerdem ist der Quellcode für Käufer insofern offen, dass es erlaubt ist, ihn zu dekompilieren und somit tief ins Spiel einzugreifen. Mittlerweile existieren sogar ganze Frameworks, was die Pluginentwicklung ungemein vereinfacht. Hinzu kommt die verwendete Programmiersprache: *Java*.

Ziel des in diesem Kapitel vorgestellten Projektes ist es, herauszufinden, ob sich die Level-Generierung von *Minecraft* mittels *Player Metrics* optimieren lässt. Dazu wird ein mit dem *Minecraft*-Server kompatibles System entwickelt das es ermöglicht, Metriken zu sammeln und zu visualisieren. Bis zu einem gewissen Grad sollen auch Spielstile erkannt werden. Anhand dieser ermittelten Daten soll der Algorithmus zur Erschaffung neuer Gebiete abgeändert und an die Bedürfnisse Spieler angepasst werden.

3.1 Minecraft

Der Entwickler *Mojang*[1], gegründet von *Markus „Notch“ Persson*, hat mit *Minecraft* mehr Erfolg als viele Kritiker erwartet hätten. Insgesamt über 5

[1] http://www.mojang.com/

Abbildung 3.1: Eine neue Welt in Version 1.2.4.

Millionen verkaufte Exemplare (siehe auch [28],wobei hier *XBLA*-Versionen[2] nicht inkludiert sind) beweisen, dass auch Indie-Entwickler erfolgreich sein können. *Minecraft*, Perssons erstes großes Projekt, begann mit einem seit 2009 stillgelegten Spiel namens *Infimier*, dessen Grundprinzipien denen von *Minecraft* sehr ähnlich waren. Während der Alphaphase, die am 17. Mai 2009 startete [39], ließ sich das Spiel bereits auf der offiziellen Website für 10$ erwerben. Ende 2010 erschien die Betaversion, die dann für 15$ verkauft wurde. *Minecraft* wurde so berühmt, dass Persson sich 2011 ein kleines Team leisten konnte, aus dem schließlich *Mojang* wurde. Der Erscheinungstermin der finalen Version war der 18. November 2011. Seither gibt es etwa jedes Monat einen Patch, der neue Features bringt.

Minecraft ist ein Spiel, das mit der Kreativität seiner Spieler lebt und untergeht. Es ist gut mit dem Bausteinspiel *Lego* vergleichbar. Der Spieler hat keine Vorgaben, wie er zu spielen hat. Es gibt lediglich einige Richtlinien in Form von Achievements, die eine grobe Orientierungsmöglichkeit bieten.

Wenn ein Server zum ersten Mal betreten wird, kann die dynamisch erzeugte Welt in First-Person-Ansicht erkundet werden. Diese Welt besteht aus einer großen Menge an Blöcken. Es gibt mehr als achtzig verschiedene dieser Blöcke, wobei sich alle unterschiedlich verhalten. Eines haben sie jedoch gemeinsam: sie können von den Spielern abgebaut werden, um Ressourcen zu erlangen. Mit diesen Ressourcen können wiederum Blöcke platziert oder Werkzeuge erschaffen werden. Die Welt ist gefüllt mit Sümpfen, Wüsten, Gebirgsketten, Bäumen, Höhlen, mit Schätzen bestückten Dungeons, Nutztieren wie Schafen, Kühen, Hühnern und nicht zuletzt Monstern, die sich während der Nacht ihren Weg zur Oberfläche bahnen. Zu Beginn können sich die Spieler lediglich vor den Zombies und Skeletten schützen, indem sie sich Unterschlüpfe bauen. Später ist es möglich, mit den richtigen Ressourcen

[2]Xbox Live Arcade

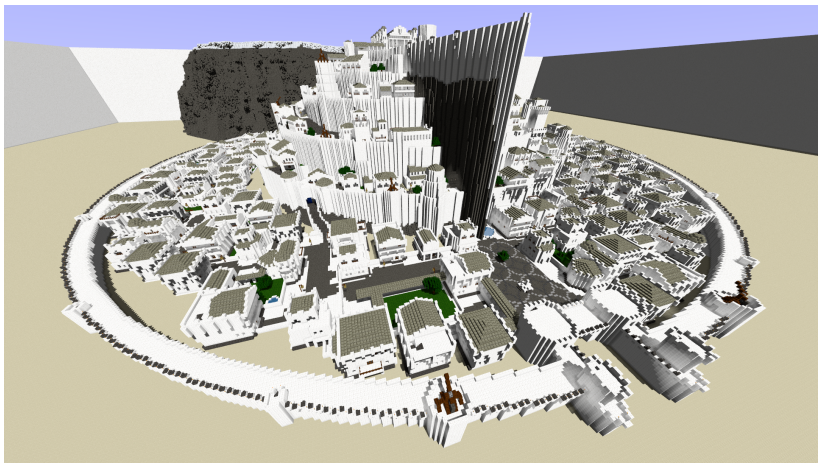


Abbildung 3.2: Minas Tirith, mit Minecraft-Blocks von mehreren Spielern nachgebaut. Aus [40].

und Werkbänken auch Waffen und Rüstungen herzustellen, mit deren Hilfe dann auch Dungeons mit wertvolleren Materialien, aber auch gefährlicheren Monstern erkundet werden können.

Es kann, sollten Kämpfe für den Benutzer uninteressant sein, im *Creative Mode* gespielt werden, bei dem den Spielern sämtliche Werkzeuge und Materialien in unbegrenzter Menge zur Verfügung stehen. Dieser Modus wird häufig verwendet um unglaublich große und beeindruckende Gebilde zu erschaffen (siehe Abb. 3.2), die anschließend von der großen Community bestaunt und vielleicht sogar auf der von *Mojang* veranstalteten Spielemesse *MineCon* ausgestellt werden können.

3.2 Entwicklung

Minecraft bietet sich vor allem wegen des teilweise offenen Quellcodes an. Für die Dekompilierung existieren mittlerweile Tools im Internet (wie z. B. das *Minecraft Coder Pack*[3]), die zum Teil auch das Labeling der Klassen, Methoden und Variablen übernehmen. Allerdings führt dies zu einigen Problemen wie der fehlenden Eingliederung in Packages und der (durch die fehlenden Labels) unübersichtlichen Klassenstruktur.

Wie in [7] beschrieben gibt es viele hilfreiche Tools, um das Sammeln und Analysieren von *Player Metrics* zu vereinfachen. *MetricsCraft* sollte sowohl Informationen über die Spieler sammeln als auch dem Entwickler diese in geeigneter Form darstellen können. Das Projekt wurde als *Bukkit*-Serverplugin entwickelt, sodass die Spieler keine zusätzliche Software herunterladen müs-

[3] http://mcp.ocean-labs.de/index.phpA/MCP_ReleasesA

sen, um auf dem Server zu spielen und somit Daten zu erzeugen.

Zu Beginn der Entwicklung sollen einige Restriktionen eingehalten werden: Es muss möglich sein, dass das Plugin neben dem Server läuft, ohne seine Performance einzuschränken. Somit bleibt das Spielerlebnis für die Spieler gleich und etwaigen Verfälschungen der Daten, die bei Performanceeinbrüchen entstehen können, wird entgegengewirkt. Die Daten müssen trotzdem vollständig gespeichert werden. Quantität ist hier zwar sehr wichtig, jedoch können, vor allem beim Analysieren von Positionsdaten, kaum Fehler in Form von fehlende Spielerdaten toleriert werden. Optimalerweise soll auch der mit dem Spiel interagierende Client ohne grobe Verzögerungen neben dem Server ausgeführt werden können.

Bukkit

Bukkit[4] und das zugehörige Serverplugin *Craftbukkit* erleichtern die Pluginentwicklung ungemein, indem der dekompilierte Quellcode von *Minecraft* gekapselt und per Adapter in eine sinnvolle und durchschaubare Klassenstruktur umgewandelt wird. Um ein Plugin zu entwickeln muss lediglich von der Klasse `JavaPlugin`, die von der *Bukkit*-Bibliothek bereitgestellt wird, abgeleitet werden. Damit das Plugin vom Server geladen wird, muss eine .jar-Bibliothek daraus erzeugt werden, die im Root-Verzeichnis eine Datei namens „plugin.yml“ beinhaltet. Sofern dort der Pfad zur Pluginklasse angegeben ist, lädt der Server das Plugin beim Start oder nach Eingabe des Administratorbefehls „reload“.

Level-Generierung

Jede *Minecraft*-Welt erweitert sich ständig selbst, sobald sich ein Spieler in einen bisher unentdeckten Bereich bewegt. Ist ein Spieler nur noch 128 Blocks vom aktuellen Ende der Welt entfernt, werden acht sogenannte *Chunks* erzeugt, jeder mit einer Größe von 16×16 Blöcken in Z- und X-Richtung. Diese Chunks definieren bei der Erzeugung, welchem Biom jeder einzelne X-Z-Block zugehört. Zu den Biomen zählen Berge, Wüsten, Meere, Flüsse, Sumpfgebiete und mehr.

Nach dieser Definition werden die einzelnen Biome „bevölkert“, d. h. es werden mehrere `ChunkPopulators` angewandt. Diese erzeugen je nach Biom unterschiedliche Landschaftsformationen, lassen Tiere erscheinen, verwandeln Gestein in die unterschiedlichsten Arten von Erzen und erzeugen Strukturen wie Dungeons und Dörfer. Jeder `ChunkPopulator` besitzt seinen eigenen Algorithmus und erhält als Parameter lediglich die aktuelle Welt, ein Zufallsobjekt (damit die Welt theoretisch kopiert werden kann, sofern der selbe *Seed* für diese Variable verwendet wird) und den aktuell zu bevölkernden

[4] http://bukkit.org/

Chunk. Daraus werden dann die benötigten Daten (wie Biome, Blockpositionen u. ä.) gewählt und die dazupassenden Populationen erzeugt. An dieser Stelle lässt sich die Levelerzeugung am besten anpassen, indem z. B. die Zufallsvariablen verändert oder völlig neue `ChunkPopulators` erstellt werden.

3.2.1 Personas und Metrics

Minecraft unterscheidet sich in vielerlei Hinsicht von den meisten anderen Spielen. Daher gibt es auch keine Standardmethode zum Sammeln von *Gameplay Metrics*, wie es beispielsweise bei simplen Platformern oder Rennspielen der Fall wäre. Zu definieren, welche Daten gesammelt werden sollten, ohne das Spiel jemals gespielt zu haben oder genau zu wissen, wie es funktioniert, scheint unmöglich. Selbst die zu sammelnden Daten der unterschiedlichen *Minecraft*-Versionen variieren durch die neuen Möglichkeiten, mit der Welt zu interagieren, stark. Diese Arbeit beschäftigt sich mit der Version 1.0.0.

Um herauszufinden, wie die *Player Experience* der Spieler positiv beeinflusst werden kann, müssen vorerst einige verschiedene Arten von Spielern definiert werden. Dieser Vorgang ist ein andauernder Prozess, der während der Beobachtung von Spielern in der Entwicklungszeit ständig angepasst werden muss. Je nachdem wie die einzelnen Spieler eines Servers spielen, soll sich dieser an deren Wünsche und Vorlieben anpassen.

Wie bereits in Kapitel 2.3 erwähnt, erklären Tychsen und Canossa in [25] ihre Vorgangsweise beim Definieren von Personas in *Hitman: Blood Money*. Bei der Entwicklung von *MetricsCraft* wurde ein ähnlicher Weg gewählt. Die *Designermetaphern* dienten dazu, ein Möglichkeitsfeld für die Spielerinteraktionen zu erzeugen. Im Vergleich zu dem von Tychsen und Canossa beschriebenen Beispiel gibt es in *Minecraft* keine Interaktionspaare wie z. B. „leise/laut“, trotzdem konnte ein Möglichkeitsfeld skizziert werden (siehe Abb. 3.3). Metapher A beschreibt hier einen Spieler, der hauptsächlich isst, Gegner tötet und Blöcke zerschlägt, Metapher B jemanden, der Ressourcen sammelt, Gebäude oder ähnliches errichtet und logische Schaltungen entwickelt. Hierbei werden Spielstile definiert, die die Möglichkeiten innerhalb von *Minecraft* repräsentieren.

Der erste Spieler hat möglicherweise Spaß daran, durch Dungeons zu laufen und Monster zu töten. Dementsprechend sollen die von ihm verwendeten Spielerfähigkeiten aufgezeichnet werden. In diesem Fall verwendet er vermutlich die Fähigkeiten laufen, springen, Waffe benutzen und Ressourcen sammeln (oder Blöcke zerstören). Die dazupassenden relevanten Daten wären, wie es vorerst scheint, folgende:

- Anzahl Kills
- Anzahl seltene Ressourcen gesammelt
- Anzahl Lebenspunkte verloren/wiederhergestellt

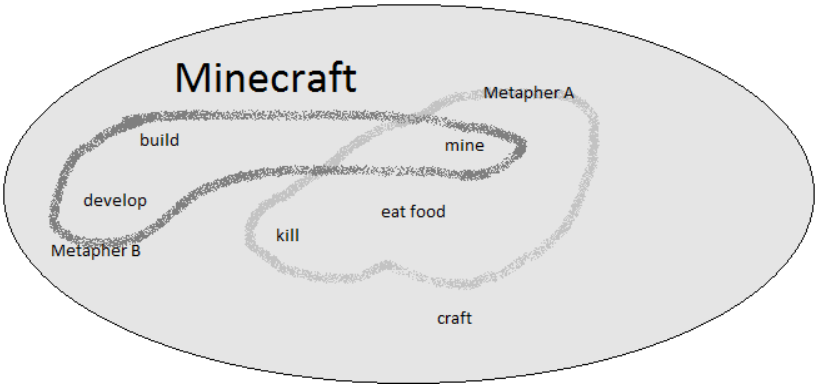


Abbildung 3.3: Möglichkeitsfeld nach [25], auf *Minecraft* angewandt.

- Anzahl Tode durch Hungern/Gegner/Lava
- Zeit in Minen verbracht (relativ)

Auf diese Weise wurden vier verschiedene Spielstile entwickelt, die jeden Spieler zum Teil beschreiben können: *Erkundung*, *Abenteuerlust*, *Errichtung*, *Erschaffung* (siehe Kapitel 4).

Nach dieser Definition kann festgelegt werden, welche Daten relevant sind, um die Level-Generierung individuell anzupassen.

3.2.2 Plugin

Zunächst wird eine von `JavaPlugin` abgeleitete Klasse `MetricsCraft` angelegt, die die Methoden `onDisable` und `onEnable` implementiert. Diese Methoden werden von *Bukkit* beim Serverstart bzw.-stop aufgerufen. Darauf folgend muss auf diverse Spieleraktionen reagiert werden. Mit Hilfe von `Events`, die von *Bukkit* zur Verfügung gestellt werden, fällt dies recht einfach aus.

```
1 private void registerEvents() {
2     new MetricsBlockListener(this);
3     new MetricsWorldListener(this);
4     new MetricsPlayerListener(this);
5     new MetricsEntityListener(this);
6 }
```

Die Methode `registerEvents` wird in `onEnable` aufgerufen und sorgt dafür, dass die jeweiligen `Listener` auf diverse Veränderungen der Welt reagieren. Neben dem Registrieren der `Events` wird die Klasse `InformationGatherer` instanziert. Diese Klasse leitet von `Thread` ab und wird gleich nach dem Instanzieren gestartet.

Programm 3.1: `run`-Methode des `InformationGatherers`

```
1  @Override
2  public void run() {
3    while(true){
4      if (!processInformations.isEmpty()){
5        try{
6          processInformations.pop().process(this);
7          if (processInformations.isEmpty()){
8            writeXml();
9          }
10       }catch (NullPointerException e){
11         Logger.getLogger("Minecraft").info("[InformationGatherer][run]
       Error getting information: " + e.getMessage());
12       }
13     }else
14       synchronized(this){
15         try {
16           this.wait();
17         } catch (InterruptedException e) {
18           e.printStackTrace();
19         }
20       }
21   }
22 }
```

Informationen sammeln

Um die Performance des Servers nicht einzuschränken, läuft das Sammeln von Informationen über andere Threads. Einer davon ist der bereits erwähnte `InformationGatherer`. Zunächst wird geprüft, ob im Hauptverzeichnis des Servers bereits die Dateien „MetricsCraft_biomes.xml“, „MetricsCraft_player.xml“ und „MetricsCraft_movement.xml“ existieren. Ist dies nicht der Fall, werden sie mit Hilfe der `DocumentBuilderFactory`, die im *javax*-Paket enthalten ist, angelegt. Andernfalls werden sie ausgelesen und als `org.w3c.dom.Document` gespeichert. Die Klasse `Thread` besitzt eine `run`-Methode, die beim Start aufgerufen wird. Implementiert wird diese Methode vom `InformationGatherer` wie in Codebeispiel 3.1.

Die Variable `processInformations` ist ein `Stack` vom Typ `Information`, der ständig abgearbeitet wird. Das Interface `Information` besitzt lediglich die Methode `process(InformationGatherer ig)`. Es bestehen drei Klassen, die es implementieren. Jede XML-Datei wird von einer dieser Klassen verwaltet: `MovementInformation` für „movement.xml“, `PlayerInformation` für „player.xml“ und `ChunkInformation` für „biomes.xml“. Jede XML-Datei ist anders aufgebaut, deshalb werden die jeweiligen `process`-Methoden unterschiedlich implementiert. Mehr dazu weiter unten.

Neben dem `InformationGatherer` bestehen noch zwei Threads, die par-

allel zum dem Server laufen und für das Sammeln und Verarbeiten von Informationen verantwortlich sind. Der `MetricsWorldListener` erzeugt eine Instanz des `ChunkAnalyzers` und startet den Thread. Er erhält die Informationen der bisherigen „biomes.xml“ und sammelt `ChunkSnapshots`, die bei der Level-Generierung von *Minecraft* erzeugt werden. Sobald der `Listener` Informationen sendet, werden diese mit Hilfe eines vereinfachten Regionlabeling (ähnlich dessen in [1]) in entsprechende Biome eingeteilt (mehr über Biome in Abschnitt 3.2). Die Einteilung in Biome dient dazu festzustellen, welche Arten von Gebieten die Spieler bevorzugen und welche sie eher meiden. Ist dieser Vorgang abgeschlossen, wird eine neue `ChunkInformation` angelegt:

```
1 MetricsCraft.infoGatherer.putInformation(
2   new ChunkInformation(
3     sn.getBiome(i, j).name(),
4     idToWrite,
5     Material.getMaterial(sn.getBlockTypeId(i, y, j)).name(),
6     new Vector(x, y, z)
7   )
8 );
```

Die Parameter geben die Art des Bioms, die ID der zugewiesenen Region, das Material des obersten Blocks (für die 2D-Ansicht der Karte, siehe Abschnitt 3.2.3) und die Position des aktuellen Blocks an. Hierbei beschreibt die Variable `sn` den `ChunkSnapshot`, der vom `MetricsWorldListener` erzeugt und an den `ChunkAnalyzer` gesendet wird.

Der letzte Thread, der `MovementHandler`, beschäftigt sich mit dem Verarbeiten der Bewegungsinformationen der Spieler. Drachen und Canossa beschäftigen sich in [7] mit sogenanntem *Spatial User Behaviour*, also (frei übersetzt) dem Verhalten von Spielern im Raum. Als Fallbeispiel diente, wie in Kapitel 2 beschrieben, *Tomb Raider: Underworld*. Auch Chris Pruett arbeitet in [41] mit diesem System. In Verbindung mit oben erwähnter Definition der unterschiedlichen Gebiete lässt sich mittels *Heatmaps* gut feststellen, wo Spieler diverse Aktionen am öftesten durchführen. Mehr dazu in Abschnitt 3.2.3.

Wie schon der `ChunkAnalyzer` erhält auch der `MovementHandler` Informationen von mehreren `Listenern`, die dann in eine `MovementInformation` gepackt und an den `InformationGatherer` weitergeleitet werden.

```
1 MovementStruct ms = movements.pop();
2 MetricsCraft.infoGatherer.putInformation(
3   new MovementInformation(
4     ms.player.getDisplayName(),
5     ms.position,
6     ms.lifeTime,
7     ms.action
8   )
9 );
```

Da keine weiteren Aktionen mit den erhaltenen Bewegungsdaten durchgeführt werden müssen, wäre dies auch ohne den zuletzt erwähnten Thread möglich, indem die entsprechenden `MovementInformations` von den jeweiligen `Listeners` selbst gesendet werden. Allerdings wäre es dann schwer, möglicherweise nötiges Preprocessing der erhaltenen Daten nachträglich zu implementieren.

XML

Zur Speicherung der gesammelten Daten wird die Klasse `XMLWriter` implementiert, die das schreiben in und lesen von XML-Dateien kapselt. Zunächst wird innerhalb dieser Klasse ein Struct mit den Parametern `attribute`, `value` und `text` erzeugt, das sich zum Speichern von XML-Elementen in `ArrayLists` eignet.

```
1  private static boolean writeToNode(Document doc, ArrayList<XMLData>
     elementAttributeList, Node nl, String elementName) {
2    Element currentElement = (Element)nl;
3    boolean returnValue = false;
4    for(XMLData entry: elementAttributeList){
5      if (!entry.attribute.equals("") && !entry.value.equals(""))
6        currentElement.setAttribute(entry.attribute, entry.value);
7      if (!entry.text.equals(""))
8        currentElement.setTextContent(entry.text);
9    }
10   return returnValue;
11 }
```

Mit der Methode `writeToNode` kann eine Liste aus Daten vom Typ `XMLData` einem einzelnen XML-Node hinzugefügt werden, wie beispielsweise eine `MovementInformation` an einen Spieler-Node angehängt wird. Die Klasse ermöglicht auch das Suchen von einzelnen Elementen nach *Tags* oder Elementnamen, was beim Wiederaufnehmen der Datensammlung von Vorteil ist (z. B. wenn ein Spieler öfter am selben Server spielt). Das Herzstück der `XMLWriter`-Klasse ist die `write`-Methode. Sie ist verantwortlich dafür, dass die Daten ins XML-Format gebracht werden. Diese Methode kann, obwohl sich die drei zu schreibenden Elemente voneinander unterscheiden, für alle drei Dateien verwendet werden.

Positionsinformationen werden, wie in Codebeispiel 3.2 beschrieben, in XML-Format gebracht. Abrufbar sind somit die Aktion des zu betrachtenden Spielers, dessen Position und der Zeitpunkt, an dem die Aktion durchgeführt wurde. Da mit Threads gearbeitet wird, ist diese Information besonders wichtig, die Aktionen könnten sonst (wie bei dem `BUILD`-Element in Codebeispiel 3.2) in falscher Reihenfolge gespeichert werden, weil ein Thread möglicherweise schneller beim Verarbeiten der Informationen ist als ein anderer.

Mit diesen Informationen kann auf einer Karte der *Minecraft*-Welt eingezeichnet werden, wo sich welcher Spieler befunden bzw. welche Wege er

Programm 3.2: „movement.xml“

```
<movement>
  <player name="DerGernTod">
    <MOVE lifetime="2" x="-173" y="65" z="65"/>
    <MOVE lifetime="6" x="-173" y="65" z="65"/>
  </player>
  <player name="knittl">
    <MOVE lifetime="2" x="-165" y="68" z="67"/>
    <MOVE lifetime="12" x="-165" y="65" z="67"/>
    <TAKE_DAMAGE lifetime="13" x="-165" y="64" z="67"/>
    <TAKE_DAMAGE lifetime="13" x="-165" y="64" z="67"/>
    <BUILD lifetime="12" x="-165" y="64" z="67"/>
  </player>
</movement>
```

Programm 3.3: „bioms.xml“

```
<BIOMS>
<SWAMPLAND id="106">
  <topblock type="GRASS" x="-116" y="63" z="-15"/>
  <topblock type="GRASS" x="-117" y="62" z="-14"/>
  <topblock type="STATIONARY_WATER" x="-117" y="62" z="-17"/>
  [...]
</SWAMPLAND>
<RIVER id="99">
  <topblock type="GRASS" x="-119" y="62" z="-15"/>
  <topblock type="STATIONARY_WATER" x="-119" y="62" z="-16"/>
  <topblock type="LEAVES" x="-120" y="67" z="-8"/>
  <topblock type="STATIONARY_WATER" x="-120" y="62" z="-16"/>
  [...]
</RIVER>
<TAIGA id="102">
  <topblock type="GRASS" x="-121" y="65" z="-3"/>
  <topblock type="LEAVES" x="-121" y="67" z="-4"/>
  <topblock type="GRASS" x="-122" y="65" z="-1"/>
  [...]
</TAIGA>
</BIOMS>
```

gewählt hat. Außerdem lässt sich feststellen, in welchen Bereichen welche Aktionen häufiger benutzt wurden.

Um der obigen XML-Datei Sinn zu geben, wird eine Sammlung aus Blockdaten der Welt benötigt. Der Aufbau der entsprechenden Datei ist in Codebeispiel 3.3 zu sehen. Mit diesen Informationen können einzelne Regionen hervorgehoben werden. Gemeinsam mit den Positionsdaten der Spieler ist es möglich, die bevorzugten Gebiete festzustellen und die Level-

Programm 3.4: „player.xml“

```
<playerlist>
  <player name="DerGernTod">
    <DAMAGE_TAKEN_DROWNING>22</DAMAGE_TAKEN_DROWNING>
    <BUTTONS_PRESSED>19</BUTTONS_PRESSED>
    <CHUNKS_DISCOVERED>4146</CHUNKS_DISCOVERED>
    <MINED>51</MINED>
    <LIFETIME>41864</LIFETIME>
    <MININGTIME>3570</MININGTIME>
    <DEVELOP>11</DEVELOP>
    <BUILT>86</BUILT>
    <UNDERGROUNDTIME>2729</UNDERGROUNDTIME>
    <SPRINTTIME>42</SPRINTTIME>
    <JUMPS>232</JUMPS>
    <RUNTIME>12800</RUNTIME>
  </player>
</playerlist>
```

Generierung dahingehend anzupassen.

Die letzte XML-Datei speichert Häufigkeit und Dauer der von den Spielern durchgeführten Aktionen. Sie wird benötigt, um Spieler in Kategorien einzuteilen und festzustellen, welche Präferenzen sie haben.

Letztendlich müssen die zu schreibenden Daten natürlich auch gespeichert werden.

Listener

Die vier in der Beschreibung der Methode `registerEvents` erwähnten Klassen leiten von der *Bukkit*-Klasse `Listener` ab und implementieren einige Callbackmethoden, die mit der Annotation `@EventHandler` markiert werden. Anhand des Parameters der Methoden (wie z. B. `BlockBreakEvent`) wird festgelegt, welche und wann diese aufgerufen werden sollen.

Wie bereits erwähnt, werden diverse Daten aus unterschiedlichen Kategorien zum Klassifizieren der Spieler benötigt.

MetricsBlockListener	`onBlockBreak`	`onBlockPlace`
MetricsWorldListener	`onWorldInit`	`onChunkPopulate`
MetricsPlayerListener	`onPlayerJoin`	`onPlayerQuit`
	`onPlayerInteract`	`onPlayerAnimation`
	`onPlayerMove`	
MetricsEntityListener	`onEntityDeath`	`onEntityGainHealth`
	`onEntityDamage`	`onEntityTarget`

Die Registrierung eines Spielers beginnt mit dem `onPlayerJoin`-Event. Hier wird zunächst geprüft, ob für diesen User bereits ein Eintrag existiert.

Mit Hilfe des `InformationGatherers` werden die Dateien „movement.xml“ und „player.xml“ nach den Spielernamen durchsucht und anschließend gegebenenfalls zu ergänzt. Außerdem wird der Timer, der die Zeit misst, die der jeweilige Spieler auf dem Server verbracht hat, wieder aufgenommen.

Mit den Callbackmethoden `onPlayerInteract`, `onPlayerAnimation` und `onPlayerMove` werden folgende Spielerdaten gespeichert:

- `CHESTS_OPENED`
- `BUTTONS_PRESSED`
- `JUMPS`
- `MININGTIME`
- `UNDERGROUNDTIME`
- `BOATTIME`
- `SWIMTIME`
- `CROUCHTIME`
- `SPRINTTIME`
- `RUNTIME`

Außerdem erhält der `MovementHandler` die Positionsdaten, indem die Aktion `MoveActions.MOVE` gesendet wird. Der Parameter der `onPlayerMove`-Methode vom Typ `PlayerMoveEvent` beinhaltet alle nötigen Informationen, um eine `MovementInformation` zu erstellen.

Im `MetricsEntityListener` befinden sich einige weitere Callbackmethoden, die Spielerdaten wie `HEALTH_REGAINED` oder `DAMAGE_TAKEN` speichern. In *Minecraft* gibt es, wie bereits erwähnt, sehr viele unterschliedliche Arten von Blöcken. Jeder Block erhält im `MetricsBlockListener` eine Kategorie und bestimmte Wertigkeit. So kann zwischen den Materialtypen Normal, Entwicklung, Aufbau und Hochwertig unterschieden werden. Mit diesen Eigenschaften wird die aktuelle Level-Generierung mit dem Verhalten der Spieler verglichen. Damit ist es möglich festzustellen, ob die Spieler genug Ressourcen für ihren Spielstil erhalten. Zudem wird in diesem `Listener` auch festgelegt, welche Werkzeuge für den optimalen Abbau von welchem Material nötig sind. Die Methoden `onBlockBreak` und `onBlockPlace` speichern mit diesen Informationen die folgenden Daten:

- `FINE_MINED`
- `MINED`
- `DEVELOP_REMOVE`
- `DEVELOP`
- `BUILT`
- `FITTING_ITEMS_USED`

Der letzte benötigte `Listener` beschäftigt sich mit der vom Levelalgorithmus generierten Welt. Hier wird das Event `onChunkPopulate` abgefangen das erzeugt wird, sobald ein Spieler in die Nähe eines neu generierten

Chunks kommt. Das `ChunkPopulateEvent` beinhaltet aber nicht den Spieler, von dem es ausgelöst wurde. Dementsprechend muss (um festzustellen, ob ein Spieler den Spielstil „Erforschung“ anwendet) berechnet werden, welcher Spieler dem *Chunk* am nächsten ist. Anschließend wird dem `ChunkAnalyzer` ein `ChunkSnapshot` des aktuell bevölkerten *Chunks* übergeben.

Um Fehler in der Erzeugung der Top-Down-Karte anhand der XML-Datei zu vermeiden, sollte das Plugin vom ersten Start des Servers weg ständig laufen. So können keine *Chunks* verloren gehen, was in Löchern bzw. fehlenden Teilen auf der Karte resultieren würde.

3.2.3 Client

Um die gesammelten Daten bequem analysieren zu können ist eine passende Visualisierung notwendig. In *MetricsCraft* wurde hierfür die in *Java* integrierte UI-Bibliothek *Swing* gewählt. Sie bietet die benötigten Grafiken und lässt sich mit externen Bibliotheken leicht erweitern.

Beim Start des Plugins wird eine Instanz des von `JFrame` abgeleiteten `MetricsCraftClients` erzeugt. Neben der Ableitung wird auch noch das Interface `WindowListener` implementiert, um auf das `windowClosed`-Event zu hören. Im Konstruktor wird die Größe des Fensters eingestellt. Außerdem erhält die `ContentPane` des Frames die beiden `JPanels LoadingPanel` und `SwitchPanel`. Diese beiden Panels stehen etwa im Verhältnis 1:2 (siehe Abb. 3.5).

LoadingPanel

Dieser Teil des Clients befindet sich auf der linken Seite des Fensters. Er ist für das Laden und Verwalten der unterschiedlichen XML-Dateien verantwortlich. Per Klick auf den Ordner-Button kann eine XML-Datei ausgewählt werden. Mit dem Aktualisieren-Button wird diese anschließend geladen. Das Laden der XML-Dateien muss asynchron erfolgen, da diese relativ schnell auf eine Größe von über $50MB$ wachsen. Hierzu wird ein entsprechender Thread erzeugt, der von der abstrakten generischen Klasse `XMLLoader` ableitet. Je nachdem, welche XML-Datei geladen wird, verändert sich die generische Klasse.

```
1 public class PlayerXMLLoader extends XMLLoader<HashMap<String, ArrayList
      <PlayerXMLLoader.PlayerStatsStruct>>>
2 public class SpatialXMLLoader extends XMLLoader<HashMap<String,
      ArrayList<SpatialXMLLoader.LocationStruct>>>
3 public class BiomsXMLLoader extends XMLLoader<ArrayList2D<String, String
      >>
```

Mit dem `PlayerXMLLoader` wird eine `HashMap` erzeugt, die für jeden Spieler eine Liste aus Daten enthält. Ähnlich dazu speichert eine andere `HashMap`, die vom `SpatialXMLLoader` instanziert wird, Spielerdaten mit Positionsinformationen. Mittels einer vom `BiomsXMLLoader` erzeugten zweidimensiona-

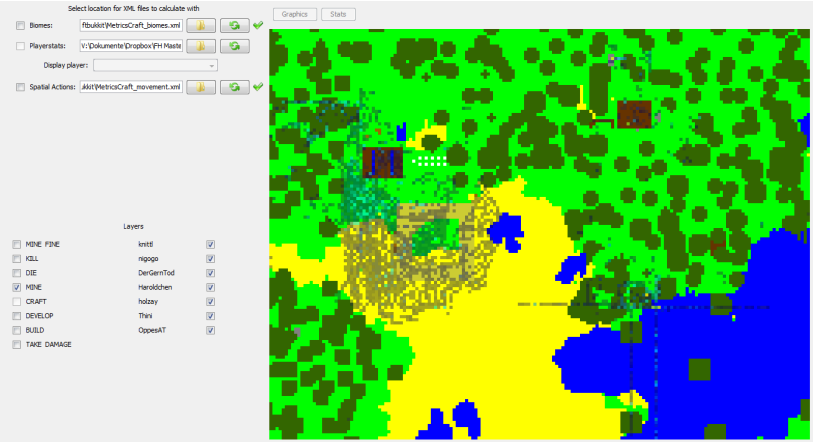


Abbildung 3.4: Der *MetricsCraft*-Client bei geöffnetem `GraphicsPanel`. Die Regionsanzeige ist deaktiviert, hervorgehoben werden die `MINE`-Aktionen der Spieler.

len `ArrayList` ist es nun möglich, eine Karte der Welt in Vogelperspektive zeichnen zu lassen.

Nachdem die jeweiligen XML-Dateien fertig geladen wurden, wird die Methode `dispatch` aufgerufen. Sie ermöglicht dem `LoadingPanel` auf die erzeugten Daten zuzugreifen. Je nach Typ des `XMLLoaders` wird unterschiedlich verfahren.

Auf das Laden einer XML-Datei im Biomeformat folgt beispielsweise die Aktivierung des `GraphicsPanels` innerhalb des `SwitchPanels`. Das *GraphicsPanel* ist für das Zeichnen, Scrollen und Zoomen der Weltkarte verantwortlich. Mit der Methode `reloadList` ist es möglich, dem `GraphicsPanel` den entsprechenden `XMLLoader` zu übergeben, dem wiederum die benötigten Daten entnommen werden können. Näheres dazu weiter unten. Nach dem Übergeben wird mittels der Methode `releaseXml` der benötigte Speicher freigegeben. Wird diese nicht aufgerufen kann es aufgrund der vielen zu erzeugenden `BufferedImages` zu Speicherproblemen kommen.

Der Abschnitt zum Laden von XML-Dateien, die Positionsdaten enthalten, wird erst freigegeben, nachdem das Erzeugen der Karte erfolgreich war. Anschließend ist es möglich, den Pfad der Datei anzugeben und den Aktualisierungsbutton zu drücken. Dadurch erhält der bisher noch nicht erwähnte `MovementLayer` die Daten, die von der Klasse `SpatialXMLLoader` erzeugt wurden, im richtigen Format um sie ins `GraphicsPanel` zu zeichnen. Das Laden dieser Datei schaltet im `LoadingPanel` weitere Checkboxen frei, mit deren Hilfe einzelne `MovementLayer` für unterschiedliche Spieler oder Aktionen umgeschaltet werden können. Siehe dazu Abb. 3.4.

Zuletzt besteht noch die Möglichkeit, die Datei „player.xml“ zu laden. Dieser Abschnitt der `finishedLoading`-Methode aktiviert das `StatsPanel`

innerhalb des `SwitchPanels`. Es erhält außerdem die von der Datei gelesenen Daten und errechnet Statistiken für den ersten Spieler in der Datenbank.

GraphicsPanel

Im `GraphicsPanel` werden einige grafische Eigenschaften der zu zeichnenden Weltkarte festgelegt, darunter Zoomstufe, Startposition und Farbdefinitionen der verschiedenen Blocks und Regionen. Für die Navigation werden außerdem die von *Swing* bereitgestellten Interfaces `MouseMotionListener`, `MouseListener` und `MouseWheelListener` implementiert.

Die Klasse beginnt nach dem Laden der XML-Daten mit der Methode `paintImage`. Zuerst werden Minima und Maxima der beiden Achsen berechnet. Mit diesen Informationen werden zwei `BufferedImages` in passender Größe instanziert, wobei ein Pixel einem *Minecraft*-Block entspricht. Codeabschnitt 3.5 zeigt, wie die vom `XMLLoader` erzeugten Daten verwendet werden um das `Graphics`-Objekt der beiden `BufferedImages` zu befüllen. Die Variable `g2` und das entsprechende Bild stellen einen Layer dar, der dem Benutzer per Klick auf die entsprechende Checkbox neben dem `BiomsXMLLoader`-Abschnitt im `LoadingPanel` die zugehörigen Biome anzeigt.

Neben der oben beschriebenen Methode bestehen noch zwei weitere wichtige Methoden: `loadLayers` und `activateLayer`. Erstere wird aufgerufen, sobald der Benutzer die XML-Datei „movement.xml“ lädt. Sie instanziert einen `SpatialPlayerLayer` für jede von den Spielern vorher getätigte Aktion. Diese Layer erzeugen, nicht unähnlich dem Codeabschnitt 3.5, deaktivierbare `BufferedImages`. Die Farbe wird jedoch anders bestimmt. Pro Blockposition wird berechnet, wieviele Aktionen an dieser Stelle von den aktuell betrachteten Spielern getätigt wurden. Auf diese Weise wird eine *Heatmap* wie in Kapitel 2 erzeugt, die die am häufigsten besuchten Orte und meistbenutzten Aktionen zeigt.

Die Navigation innerhalb des `GraphicsPanels` erfolgt mittels Drag & Drop, mit dem Mausrad kann die Zoomstufe bis zu einem gewissen Grad verändert werden. Umgesetzt wurde dieses Verhalten mit Hilfe der `drawImage`-Methode des `Graphics`-Objekts des `JPanels`:

```
1 this.getGraphics().drawImage(bimg, 0, 0, PANELSIZE[0], PANELSIZE[1],
2   canvasPosition.x, canvasPosition.y,
3   (int)((canvasPosition.x + PANELSIZE[0]*scaleFactor)),
4   (int)((canvasPosition.y + PANELSIZE[1]*scaleFactor)),
5   this.getParent());
```

Auf diese Weise wird immer nur ein Teil des `BufferedImages` gezeichnet, abhängig von Zoomstufe und Position des Panels im Bild. Diese beiden Informationen werden auch von verschiedenen `Layers` benutzt um sie korrekt über die Weltkarte zu legen.

Programm 3.5: Codeabschnitt innerhalb der `paintImage`-Methode, der für das Zeichnen der Karte verantwortlich ist.

```
1 [...]
2 for (int i = 0; i < imageSize[0]; i++){
3   for (int j = 0; j < imageSize[1]; j++){
4     String type = list.get((i + lowest.x) + "", (j + lowest.y) + "");
5     String biome = biomesList.get((i + lowest.x) + "", (j + lowest.y) +
      "");
6     if (biome == null)
7       g2.setColor(Color.black);
8     else
9       for(BiomeType b : BiomeType.values())
10        if (biome.equalsIgnoreCase(b.name()))
11          g2.setColor(b.getColor());
12    g2.fillRect(i,j,1,1);
13    if (type == null)
14      g.setColor(Color.black);
15    else
16      for(BlockType b : BlockType.values())
17        if (type.equalsIgnoreCase(b.name()))
18          g.setColor(b.color);
19    g.fillRect(i * BLOCKSIZE[0], j * BLOCKSIZE[1], BLOCKSIZE[0],
      BLOCKSIZE[1]);
20  }
21 }
22 [...]
```

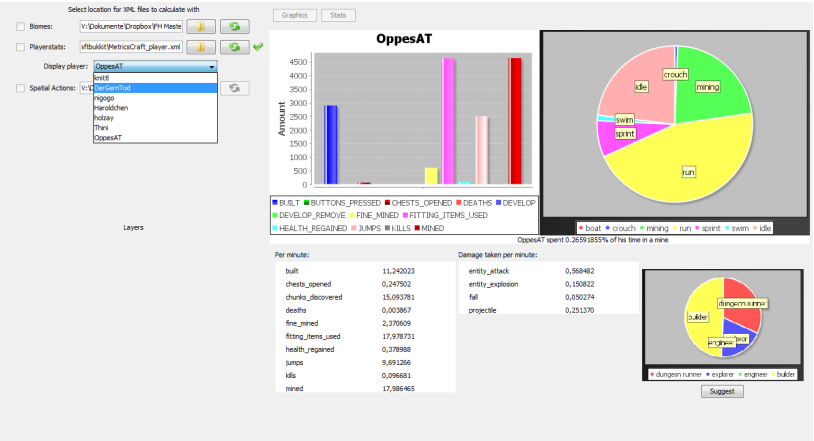


Abbildung 3.5: Das `StatsPanel` des *MetricsCraft*-Clients. Die Ansicht zeigt daten über den Spieler *OppesAT*.

StatsPanel

Das `StatsPanel` bedient sich der *jfree*-Bibliothek *jfreechart*, die es ermöglicht, Diagramme in ein `Graphics`-Objekt zu zeichnen. So können einige Spielerdaten übersichtlich visualisiert werden. Das Herzstück dieses Panels ist jedoch die Anzeige der Abweichungswerte (in Abb. 3.5 rechts unten). Die Berechnung hierfür übernimmt die Klasse `Classifier`. Beim Aufbau des Panels wird deren `calculate`-Methode aufgerufen. Sie berechnet, wie sehr ein Spieler von verschiedenen vordefinierten Verhaltensmustern abweicht. Es werden vier unterschiedliche Muster implementiert, die im Laufe des Praxistests ständig erweitert und angepasst wurden. Die Muster werden in einer `HashMap` folgendermaßen gespeichert:

```
1 list.put(new PlayerStatsStruct(PlayerDataType.RUNTIME, 0.8f),0.37f);
```

Zuerst wird der zu betrachtende Datensatz festgelegt, hier `RUNTIME`. An zweiter Stelle der Wert, den der Spieler haben sollte, um dem Muster zu entsprechen. Der letzte Parameter entspricht einer Gewichtung, nämlich wieviel Prozent der Bewertung diese Aktion auf das Endergebnis haben soll. Die vier Muster entsprechen den in Kapitel 2.3 erwähnten Spielstile, die in der Implementierung `EXPLORER`, `DUNGEON_RUNNER`, `BUILDER`, und `ENGINEER` genannt werden (mehr dazu in Kapitel 4).

Unterhalb des Tortendiagramms befindet sich ein Button, der dem Benutzer anhand des ermittelten Spielstile des Spielers Empfehlungen zur Anpassung des Level Designs gibt. Dieser Button ist stellt die Schnittstelle zwischen Client und Plugin dar. Wird er benutzt, werden einige `Populators` des Servers verändert, indem beispielsweise mehr Monster erzeugt, andere Regionen erschaffen und neue Dungeons platziert werden.

Kapitel 4

Analyse

Die Verwendung von *Gameplay Metrics* zur Analyse von Game Design ist kein triviales Thema. Oft bedarf es umfangreicher Planung um herauszufinden, welche Daten überhaupt benötigt werden. Und selbst wenn diese Auswahl gründlich durchdacht ist kann es vorkommen, dass sie nach einigen Tests mit Spielern dennoch verändert werden muss.

Diese Arbeit soll anhand von *Minecraft* die Frage beantworten, ob und wie *Player Metrics* dazu dienen, Leveldesign zu verbessern. Zu Beginn der Entwicklung des in Kapitel 3 beschriebenen Plugins *MetricsCraft* musste definiert werden, welche Werkzeuge zur Analyse von *Gameplay Metrics* nötig sind. Dazu stellen sich einige Fragen, mit deren Beantwortung festgelegt werden kann, welche Daten aufgezeichnet werden sollen:

- Welche Spielmodi werden bereitgestellt?
- Welche Interaktions- und Navigationsmöglichkeiten bestehen?
- Welche Spielstile sind möglich?

4.1 Spielstile

In Kapitel 2.3 wurde bereits ein Einblick in Möglichkeitsfelder gegeben, deren Nutzung bei der Definition von Spielstilen und Personas sehr hilfreich sein kann. Diese wurden auch hier zur Definition von Personas herangezogen. Hierzu lassen sich in *Minecraft* mögliche Spielmodi und zugehörige Metrics zusammengefassen.

Das Spiel bietet allerdings ein äußerst umfangreiches Repertoire an Möglichkeiten, so erweist sich das Erkennen der wichtigsten als sehr komplex. Herauszufinden, welche Spielstile und -modi in einem Spiel möglich sind, erfordert eine genaue Analyse. Da sich *Minecraft*, wie in Kapitel 3 beschrieben, mit jeder Version verändert wurde entschieden, die Untersuchung mit Version 1.0.0 durchzuführen. Tabelle 4.1 zeigt einige in dieser Version mögliche Spielmodi und dazupassende *Gameplay Metrics*, wobei Zeitwerte sich auf den Prozentsatz des vom Spieler benutzten Modus' in Relation zur Gesamtzeit,

Modus	**Relevante Metriken**
Laufen	Zeit laufend verbracht
Sprinten	Zeit sprintend verbracht
Schwimmen	Zeit schwimmend verbracht Zeit in Booten verbracht
Kriechen	Zeit kriechend verbracht
Springen	Häufigkeit von Sprüngen
Waffe benutzen	Verursachter Schaden
Blöcke abbauen	Zeit mit Abbau verbracht abgebaute Blöcke
Blöcke setzen	platzierte Blöcke
Blöcke benutzen	benutzte Blöcke
Gegenstände anfertigen	erzeugte Gegenstände
Werkzeuge benutzen	benutzte Werkzeuge
Werkzeuge zerstören	zerstörte Werkzeuge
Schalter auslösen	bewegte Hebel gedrückte Buttons ausgelöste Bodenplatten ausgelöste Falltüren

Tabelle 4.1: Einige der in *Minecraft* möglichen Spielmodi und zugehörigen *Gameplay Metrics*.

die der Spieler am Server verbracht hat, beziehen. Die übrigen Werte werden als Häufigkeitsdaten pro Zeiteinheit betrachtet.

Anhand der in Tabelle 4.1 gezeigten Definition einiger relevanter Spielmodi können Spielstile vordefiniert werden. Diese Spielstile stellen eine Kombination verschiedener Modi dar, mit deren Hilfe sich feststellen lässt, welche Metriken aufgezeichnet werden müssen um in Erfahrung zu bringen, wie sich Spieler verhalten. Dabei kamen mehrere Spielstile zum Vorschein, deren Aufzeichnung mittels verschiedener Spielmodi (und einiger zusätzlicher Interaktionsmetriken) ermöglicht wird:

Erkundung: Der Spieler ist neugierig, sucht ständig neue Gebiete und erzeugt dabei neue Bereiche des Levels. Dabei überquert er u. a. Meere und Gebirgsketten. **Spielmodi:** Laufen, Sprinten, Schwimmen. **Relevante Metriken:** Zeit laufend, sprintend, schwimmend und in Booten verbracht, Anzahl erzeugter *Chunks* pro Minute, Trefferpunkte pro Minute durch Fallschaden und Hunger verloren.

Abenteuerlust: Der Spieler ist auf der Suche nach Schätzen, also wertvollen Materialien und Truhen, die tief unter der Oberfläche versteckt sind. Er

Abbildung 4.1: Mit *Minecraft*-Blocks gebautes Abbild von *Darth Vader*, bei dem mehrere Nutzer gemeinsam am selben Projekt gearbeitet haben. Aus [35].

muss dabei Monster bekämpfen und darauf achten, keinen falschen Block zu entfernen, um nicht an plötzlichem Tod durch herabströmende Lava zu sterben. **Spielmodi:** Laufen, Springen, Waffe benutzen, Blöcke abbauen. **Relevante Metriken:** Anzahl abgebauter wertvoller Blöcke (im Vergleich zur Gesamtanzahl abgebauter Blöcke), pro Minute verursachter Schaden, Zeit unter der Oberfläche verbracht, pro Minute wiederhergestellte Trefferpunkte, Trefferpunkte pro Minute durch Feinde, Lava und Explosionen verloren.

Errichtung Dem Spieler geht es hauptsächlich um den Aufbau von Objekten, sei es eine ganze Stadt bestehend aus einer Vielzahl von Häusern, Tempeln, Plätzen und Palästen oder eine abstrakte Konstruktion (siehe Abb. 4.1). **Spielmodi:** Kriechen, Springen, Laufen, Blöcke platzieren, Kisten benutzen. **Relevante Metriken:** Zeit kriechend und laufend verbracht, Anzahl platzierter Blöcke, erzeugter *Chunks*, Kistennutzungen und Sprünge pro Minute, Trefferpunkte pro Minute durch Fallschaden verloren.

Erschaffung Der Spieler entwickelt mit Hilfe bestimmter Gegenstände und Blocks komplexe logische Schaltungen, die durch Interaktion mit Hebel, Falltüren und Buttons aktiviert und deaktiviert werden können. **Spielmodi:** Laufen, Hebel, Falltüren und Buttons benutzen, Entwicklerblöcke zerstören und platzieren, Gegenstände erschaffen. **Relevante Metriken:** Anzahl benutzter Hebel, Falltüren und Buttons, platzierter und zerstörter Entwicklerblöcke sowie erzeugter Gegenstände pro Minute, Zeit laufend verbracht.

Stil	Metriken	Ziel	Einfluss
Erkundung	Zeit laufend verbracht	80%	37%
	Zeit sprintend verbracht	5%	2%
	Zeit schwimmend verbracht	10%	15%
	Entdeckte *Chunks*	100/Min.	40%
	Fallschaden	0,1/Min.	2%
	Schaden durch Hunger	0,1/Min.	4%
Abenteuerlust	Zeit laufend verbracht	40%	18%
	Zeit im Untergrund verbracht	50%	22%
	Sprünge	5/Min.	4%
	Zeit mit Abbau verbracht	50%	14%
	Materialien abgebaut	10/Min.	14%
	Wertvolles gesammelt	2/Min.	8%
	Trefferpunkte wiederhergestellt	1/Min.	2%
	Schaden durch Angriffe	0,4/Min.	3%
	Schaden durch Explosionen	0,4/Min.	3%
	Schaden durch Lava	0,4/Min.	4%
	Schaden verursacht	8/Min.	8%
Errichtung	Zeit laufend verbracht	10%	5%
	Zeit kriechend verbracht	2%	5%
	Sprünge	4/Min.	5%
	Kisten geöffnet	0,2/Min.	5%
	Fallschaden	0,05/Min.	5%
	Blöcke platziert	6/Min.	75%
Erschaffung	Zeit laufend verbracht	10%	5%
	Auslöser benutzt	1/Min.	20%
	Blöcke platziert	5/Min.	55%
	Blöcke zerstört	0,1/Min.	10%
	Gegenstände erzeugt	0,05/Min.	10%

Tabelle 4.2: Balancingwerte der einzelnen *Gameplay Metrics*, die für die Definition der Spielstile relevant sind. Bei dem Erschaffungsstil werden (im Gegensatz zum Errichtungsstil) nur Entwicklerblöcke berücksichtigt. Prozentwerte sind auf die Gesamtspielzeit des Spielers am Server bezogen.

Einige dieser Entscheidungen mögen auf den ersten Blick nicht schlüssig wirken (wie beispielsweise das Aufzeichnen der Zeit, die ein im Erschaffungsstil spielender Spieler laufend verbringt). Grund für das Speichern solcher Werte ist folgender: *Minecraft* bietet im Vergleich zu *Hitman: Blood Money* (siehe Kapitel 2.3) keine Spielstilpaare wie beispielsweise „leise“/“laut“. Um in Spielerdaten dennoch Spielstile zu erkennen werden Werte benötigt, die auf den jeweiligen Spielstil zutreffen. So kann es von Bedeutung sein, dass bei dem Spielstil „Erschaffung“ der Wert „Zeit mit Laufen verbracht“ gering

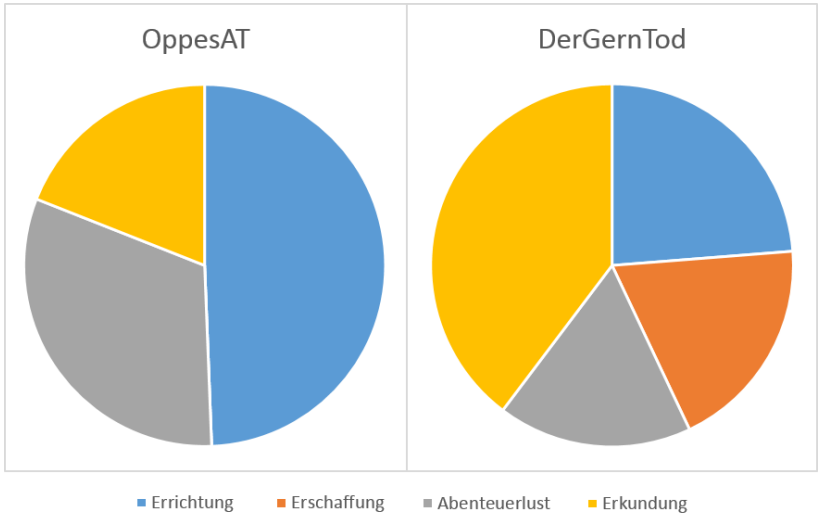


Abbildung 4.2: Ein Tortendiagramm, das die mit Hilfe der gesammelten *Gameplay Metrics* ermittelten Spielstile der Spieler *OppesAT* und *DerGernTod* zeigt.

sein soll. Es stellte sich als äußerst kompliziert heraus, diese Werte zu definieren. Nach vielen Testsessions war es jedoch möglich, annähernd präzise Definitionen für die erwähnten Spielstile zu finden.

Sämtliche aufgezeichnete *Gameplay Metrics* besitzen somit einen Zielwert, den es zu erreichen gilt. Wird nun dieser Zielwert mit den von einem Spieler erzeugten Metriken verglichen, stellt die Differenz einen Abweichungswert dar. Dieser kann Verwendet werden um festzustellen, ob das Verhalten eines Spielers einem vordefinierten Spielstil entspricht. Neben dem Zielwert werden die einzelnen Metriken auch mit einer Gewichtung versehen. Diese beiden Balancingwerte wurden aufgrund einiger Tests mit *Minecraft*-Spielern in einem iterativen Prozess immer wieder neu angepasst. Ob Spielstile korrekt erkannt wurden, ließ sich nur per Interview mit den Testpersonen ermitteln. Die in Tabelle 4.2 gezeigten Werte bewährten sich bei den Tests.

4.2 Visualisierung

Durch diverse Tests ließ sich feststellen, dass sich die Spielstile der einzelnen Spieler oft stark voneinander unterscheiden. Mithilfe von Diagrammen konnte diese Tatsache genauer untersucht werden. Wie in Abbildung 4.2 ersichtlich verwendet der Spieler *OppesAT* primär den Spielstil „Errichtung“. Der hohe Anteil an „Abenteuerlust“ kömmt möglicherweise daher, dass für die Errichtung von Bauten Ressourcen notwendig sind, die Teilweise nur in Höhlen auffindbar sind. *DerGernTod* verbringt offensichtlich einen Großteil

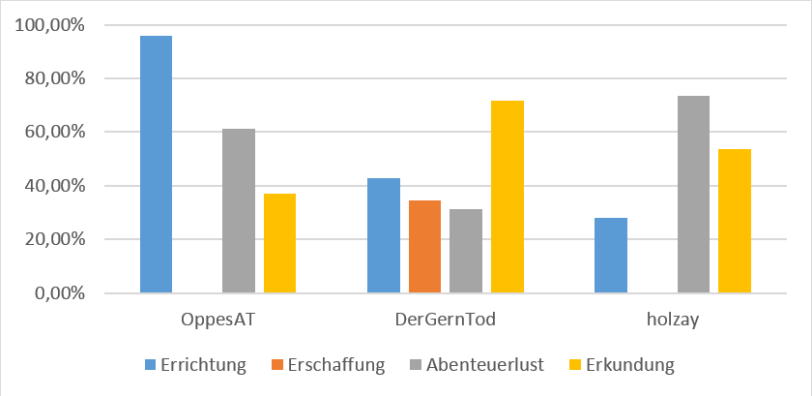


Abbildung 4.3: Das Balkendiagramm zeigt, wie sehr die Spielstile der Spieler *OppesAT*, *DerGernTod* und *holzay* den entworfenen, mit Hilfe der in Tabelle 4.2 definierten Stile entsprechen.

seiner Zeit damit, neue Gebiete zu erkunden. Abbildung 4.3 zeigt, wie sehr verschiedene Spieler einem Spielstil entsprechen. So lässt sich anhand dieser Grafik feststellen, dass der Spieler *OppesAT* den Spielstil „Errichtung“ bevorzugt. Bei *DerGernTod* ist hingegen nicht ganz sicher, ob dieser Spieler überhaupt einen Stil bevorzugt, da keiner der berechneten Abweichungswerte bei unter 10% liegt, wie es bei *OppesAT* und „Errichtung“ der Fall ist. Allerdings lässt sich feststellen, dass sowohl *OppesAt* als auch *holzay* keinen Gebrauch des Spielstils „Erschaffung“ machen.

Heatmaps

Wie in Kapitel 3 beschrieben, werden in *Minecraft* jedes Mal neue *Chunks* erzeugt, wenn ein Spieler sich dem Levelrand nähert. Somit lässt sich die Welt theoretisch unendlich oft erweitern. Bei der Erzeugung dieser *Chunks* wird in *MetricsCraft* eine Karte gezeichnet, die der Client anzeigen kann.

Viele der oben erwähnten Daten werden positionsbezogen gespeichert, sodass sich Heatmaps erzeugen lassen. *MetricsCraft* ermöglicht es, mehrere Heatmaps simultan anzeigen zu lassen, sodass häufig besuchte Positionen herausstechen. Abbildung 4.4 zeigt eine auf die erzeugte Karte projizierte Heatmap über die Häufigkeit der Aktion „Abbauen“.

4.3 Fazit

Es stellt sich heraus, dass der Nutzen von Heatmaps am Beispiel *Minecraft* kaum sinnvolle Ergebnisse bringt. Mittels dieser Heatmaps lässt sich zwar feststellen, in welchen Gebieten sich Spieler hauptsächlich aufhalten, allerdings entstehen dadurch keine Vorteile bei der Verbesserung von Level De-

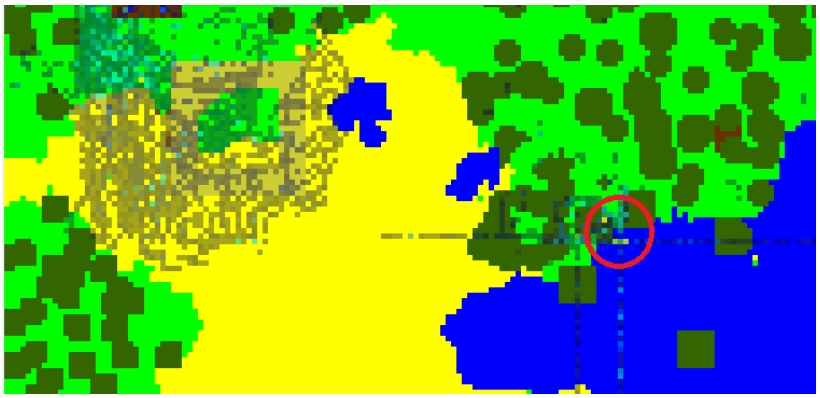


Abbildung 4.4: Top-Down-Ansicht eines Ausschnitts einer *Minecraft*-Welt. Die Heatmap „Abbauen“ ist Aktiviert. Hervorgehoben werden die Stellen, an denen am meisten Blöcke abgebaut wurden.

sign. Dies kommt vor allem daher, dass Spieler sich nach dem ersten Login auf einem Server normalerweise nicht sehr weit bewegen. Die Tests ergaben, dass selbst die Spieler mit dem Spielstil „Erkundung“ immer wieder zurück zur Ursprungsposition (oder in die Nähe dieser) kommen. Nicht nur, weil sie nach dem Tod dorthin zurück teleportiert werden, sondern auch freiwillig, um zu sehen, was andere Spieler in der zwischenzeit errichtet haben. Da jede Welt außerdem per Zufall generiert wird und nicht mehr als 20 Spieler gleichzeitig auf einem Server spielen können, ist die Quantität der Daten, die für Heatmaps normalerweise sehr hoch sein sollte, zu gering, um allgemeine Rückschlüsse ziehen zu können.

Um abzuschätzen, inwiefern der Algorithmus zum Generieren der *Chunks* verändert werden sollte, ist es Hilfreich, Spielstile zu definieren. Anhand dieser lässt sich beispielsweise vermuten, welche Ressourcen gerade benötigt werden. Im Falle des Stils „Erkundung“ ist es von Vorteil, festzustellen, ob gewisse Biome seltener erzeugt werden. Dann kann dem Spieler eine völlig neue Welt geboten werden, was sein Bedürfnis nach Erkundung stillen kann. Im vorhergehenden Abschnitt ist ersichtlich, dass die Spieler *OppesAT* und *holzay* nicht von dem Spielstil „Erschaffung“ Gebrauch machen. Für diesen Spielstil wird vor allem die Ressource *Redstone* benötigt. Die Anschaffung dieser Ressource stellt kein großes Problem dar, ihr Vorkommen in Höhlen ist relativ hoch. Wenn allerdings kein *Redstone* gebraucht wird, so kann es aufgrund der langen Zeit, die für den Abbau benötigt wird, frustrierend sein, ständig auf dieses Material zu stoßen. Wird wie in diesem Beispiel erkannt, dass kein *Redstone* benötigt wird, kann der Algorithmus für die Level-Generierung insofern abgeändert werden, sodass weniger dieser Materialien beim Graben zum Vorschein kommen.

Das große Problem, das am Beispiel *Minecraft* allerdings vorliegt, ist die

Tatsache, dass der Algorithmus nur bei neu erkundeten Bereichen zum Tragen kommt. Wird dieser also auf die am Server befindlichen Spieler angepasst so hilft es nur jenen Spielern, die sich dazu entschließen, in neue Gebiete vorzudringen. Wenn die Spielerstatistiken allerdings gespeichert werden und ein Server anhand dieser *Gameplay Metrics* eine neue Welt generiert, so kann diese voll und ganz auf die Spieler abgestimmt werden.

Kapitel 5

Schlussbemerkungen

Diese Arbeit zeigt einige Methoden, wie die Verwendung von *Player Metrics* beim Entwickeln eines Spiels von Vorteil sein können. Neben verschiedenen Visualisierungstechniken wie z. B. Heatmaps werden auch Gebiete der benutzerorientierten Entwicklung beschrieben.

Es stellt sich heraus, dass *Minecraft* kein optimales Beispiel bietet, um automatische Levelgenerierung für Spieler anzupassen. Bei Spielen, in denen Spieler dazu getrieben werden, immer mehr Gebiete zu erforschen und alte hinter sich zurückzulassen, bestünde mehr Potential. *MetricsCraft*, das mit dieser Arbeit einhergehende Projekt, stellt einige Möglichkeiten zur Analyse von Metriken zur Verfügung und ermöglicht es, anhand der ermittelten Spielstile den Algorithmus zum Generieren von Levels während der Laufzeit zu verändern.

5.1 Projekterweiterungen

Eine mögliche Erweiterung zum System wäre die automatische Anpassung des Levelalgorithmus. Dazu müssten auch sämtliche Spieler in Betracht gezogen werden, sodass durch die Optimierung für einen Spielstil nicht ein anderer benachteiligt wird. Einige weitere Features von *MetricsCraft* standen in der Planung, konnten aus zeitlichen Gründen aber nicht mehr umgesetzt werden. Aktuell werden per Klick auf den „Suggest“-Button Vorschläge zur Veränderung des Leveldesigns gegeben, die dann vom Serveradministrator angenommen oder verworfen werden können. Dabei werden bereits existente `ChunkPopulators` manipuliert. Geplant war, verschiedene `ChunkPopulators` zu entwickeln, die zu den definierten Spielstilen passen. Diese *ChunkPopulators* sollten automatisch anhand des analysierten Spielerverhaltens mit den von *Minecraft* bereitgestellten ausgetauscht werden. Per Interview könnte anschließend festgestellt werden, ob die Spieler Veränderungen bemerkt haben und ob diese zu ihrer Zufriedenheit sind.

Da der Client mit dem Plugin und damit mit dem Server verbunden ist,

könnten die Daten auch in Echtzeit aktualisiert werden, anstatt den Zwischenweg über XML-Dateien zu gehen. Es müsste dann allerdings sichergestellt werden, dass die Daten bei einem Serverabsturz nicht verloren gehen und nach einem solchen Vorfall die Aufzeichnung wieder aufgenommen wird. Mit der synchronen Verwendung der Analysedaten kann das oben erwähnte automatische Verfahren in Echtzeit (also ohne den Klick auf den „Suggest"-Button mit anschließender Bestätigung) durchgeführt werden.

Ein weiteres Feature, dessen Implementierung sich anbieten würde, wäre folgendes: Bei der Erzeugung der *Chunks* könnte eine `playerWasHere`-Variable mitgespeichert werden, die bei der Spielerbewegung verändert werden würde. Wenn sich bei oben erwähntem Streamingverfahren herausstellen würde, dass zukünftige *Chunks* anders „bevölkert" werden sollten, so könnten manipulierte *ChunkPopulators* nicht nur auf neue, sondern auch auf noch nicht besuchte Bereiche angewandt werden. Dies könnte vor allem für Spieler von Vorteil sein, die bevorzugt nahe an ihrem ursprünglichen Einstiegspunkt bleiben.

Um die Validität der definierten Spielstile zu sichern (oder die Balancingwerte zu optimieren) könnten mehrere *Minecraft*-Server separat laufen, sodass eine größere Zahl an Spielerdaten gesammelt und analysiert werden kann. Diese Technik könnte noch erweitert werden, indem für alle Server derselbe Zufallswert (*Seed*) gewählt würde. Dadurch werden idente Welten erzeugt, sodass die gesammelten Metriken serverübergreifend verwendbar sind.

Als größeres Projekt bietet sich auch an, ein von verschiedenen Spielen verwendbares *GIS* (siehe Abschnitt 2.3) zu entwickeln. *MetricsCraft* wurde speziell für *Minecraft 1.0.0* entwickelt, was eine Adaption zu einem anderen Spiel unmöglich macht. Selbst die Verwendung in neueren Versionen bringt einige benötigte Anpassungen mit sich. So muss jeder neu eingeführte Block kategorisiert (ist er wertvoll, zählt er zu Entwicklerblocks, ...) und neue Features bei der Definition der Spielstile berücksichtigt werden. Neue Versionen ermöglichen beispielsweise die Viehzucht oder das Halten eines Wolfsrudels.

5.2 Conclusio

Immer mehr Entwicklerstudios nutzen mittlerweile Metriken, um das Game Design ihrer Produkte zu verbessern [30, 33, 43]. Es ist eine anfangs zwar aufwendige Methode, die sich im Laufe der Entwicklung aber meist bezahlt macht. Sie bietet die Möglichkeit, eine Vielzahl an Daten zu sammeln und korrekte Analysen durchzuführen. Durch die Objektivität von Spielern ist eine solche Genauigkeit bei Interviews und Fragebögen oft nicht gegeben. Diese Arbeit zeigt, dass die Kombination dieser beiden Techniken (also qualitative, als auch quantitative Analyse) beim Game- und vor allem Level Design von Vorteil sein kann. Es empfiehlt sich also, die Arbeit im vorhinein auf sich zu nehmen, um eine optimalen *Player Experience* zu erschaffen.

Quellenverzeichnis

Literatur

[1] Wilhelm Burger und Mark Burge. *Digital Image Processing: An Algorithmic Introduction using Java*. Springer, Nov. 2007.

[2] Eduardo H. Calvillo-Gomez, Paul Cairns und Anna L. Cox. „Assessing the Core Elements of the Gaming Experience“. In: *Evaluating User Experience in Games*. Hrsg. von Regina Bernhaupt. Human Computer Interaction Series. Springer London, 2010, S. 47–71.

[3] Eduardo J. Chapresto, Kenny Mitchell und Francisco J. Serón. *Capture and Analysis of Racing-Gameplay Metrics*. Techn. Ber. Disney Interactive, University of Zaragoza, 2011.

[4] Luca Chittaro und Lucio Ieronutti. „A visual tool for tracing users' behavior in Virtual Environments“. In: *Proceedings of the working conference on Advanced visual interfaces*. AVI '04. Gallipoli, Italy: ACM, 2004, S. 40–47.

[5] Alan Cooper. *The Inmates Are Running the Asylum*. Sams, Apr. 1999.

[6] Mihaly Csikszentmihalyi. *Flow - The Psychology of Optimal Experience*. New York, NY, USA: Harper & Row, 1990.

[7] Anders Drachen und Alessandro Canossa. „Analyzing spatial user behavior in computer games using geographic information systems“. In: *Proceedings of the 13th International MindTrek Conference: Everyday Life in the Ubiquitous Era*. MindTrek '09. New York, NY, USA: ACM, 2009, S. 182–189.

[8] Anders Drachen u. a. *Introduction to Game Metrics*. Website. Copenhagen, Denmark, 2010. URL: http://www.agorainformatics.com.

[9] Paul Freiberger und Michael Swaine. *Fire in the Valley: The Making of the Personal Computer*. McGraw-Hill Professional, 1984.

[10] Benedikt Grindel. „Spielspaß erforschen“. In: *Making Games Magazin* 01/2012 (Jan. 2012), S. 14–19.

[11] Sunil Gupta und Valarie Zeithaml. „Customer Metrics and Their Impact on Financial Performance“. In: *Marketing Science Vol. 25, No. 6*. INFORMS, 2006, S. 718–739.

[12] Layla Hasan, Anne Morris und Steve Probets. „Using Google Analytics to Evaluate the Usability of E-Commerce Sites“. In: *Human Centered Design*. Hrsg. von Masaaki Kurosu. Bd. 5619. Lecture Notes in Computer Science. Springer, 2009, S. 697–706.

[13] Marc Hassenzahl und Noam Tractinsky. „User experience - a research agenda“. In: *Behaviour & Information Technology* 25.2 (2006), S. 91–97.

[14] *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Standard 1061–1992. Institute of Electrical und Electronics Engineers, 1992.

[15] Mona Joorabchi und Magy El-Nasr. „Measuring the Impact of Knowledge Gained from Playing FPS and RPG Games on Gameplay Performance“. In: *Entertainment Computing ICEC 2011*. Hrsg. von Junia Anacleto u. a. Bd. 6972. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, S. 300–306.

[16] Jun H. Kim u. a. „Tracking real-time user experience (TRUE): a comprehensive instrumentation solution for complex systems“. In: *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*. CHI '08. New York, NY, USA: ACM, 2008, S. 443–452.

[17] Christoph Klimmt u. a. „Player Performance, Satisfaction, and Video Game Enjoyment“. In: *Entertainment Computing ICEC 2009*. Bd. 5709. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, S. 1–12.

[18] Larry Mellon. *Applying Metrics Driven Development to MMO Costs and Risks*. Techn. Ber. Redwood City, CA, USA: Versant Corporation, 2009.

[19] Lennart Nacke und Anders Drachen. „Towards a Framework of Player Experience Research“. In: *Proceedings of EPEX'11*. EPEX '11. Bordeaux, France: ACM, Juni 2011, S. 6.

[20] Randy J. Pagulayan u. a. „The human-computer interaction handbook“. In: Hrsg. von Julie A. Jacko und Andrew Sears. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 2003. Kap. User-centered design in games, S. 883–906.

[21] Michael Schiessl u. a. *Eye tracking and its application in usability and media research*. Techn. Ber. Berlin, Germany: Eye Square Berlin, Humboldt University Berlin, 2003.

[22] Ken Schwaber. *Agile Project Management with Scrum*. Prentice Hall, 2004.

[23] B. F. Skinner. *The behavior of organisms: An experimental analysis*. New York, NY, USA: Appleton-Century, 1938.

[24] E. L. Thorndike. „Animal Intelligence: An Experimental Study of the Associate Processes in Animals“. In: *The American psychologist*. USA: American Psychological Association, 1998, S. 1125–1127.

[25] Anders Tychsen und Alessandro Canossa. „Defining personas in games using metrics“. In: *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*. Future Play '08. New York, NY, USA: ACM, 2008, S. 73–80.

Online-Quellen

[26] *Assassin's Creed 2: Rekordverdächtig: Eintrag ins Guinness Buch*. 2010. URL: http://www.spieleradar.de/news/assassins-creed-2/0_14149_8107/rekordverdaechtig-eintrag-ins-guinness-buch.html.

[27] Alan Cooper. *The origin of personas*. Mai 2008. URL: http://www.cooper.com/journal/2003/08/the_origin_of_personas.html.

[28] Tom Curtis. *Minecraft pulled in more than 80M in Mojangs first 15 months*. 2012. URL: http://gamasutra.com/view/news/167199/Minecraft_pulled_in_more_than_80M_in_Mojangs_first_15_months.php.

[29] Nicky Danino. *Human-Computer Interaction and Your Site*. Nov. 2001. URL: http://www.sitepoint.com/computer-interaction-site/.

[30] Jonathan Dankoff. *Game Telemetry with Playtest DNA on Assassin's Creed*. Sep. 2011. URL: http://engineroom.ubi.com/game-telemetry-with-playtest-dna-on-assassin%E2%80%99s-creed-part-2/.

[31] Jonathan Dankoff. *Game Telemetry with Playtest DNA on Assassin's Creed*. Mai 2012. URL: http://engineroom.ubi.com/game-telemetry-with-playtest-dna-on-assassins-creed-part-3.

[32] Jonathan Dankoff. *Game Telemetry with Playtest DNA on Assassin's Creed*. Feb. 2012. URL: http://engineroom.ubi.com/game-telemetry-with-playtest-dna-on-assassin%E2%80%99s-creed-part-2/.

[33] Anders Drachen. *10 Great Game Telemetry Reads*. Feb. 2012. URL: http://blog.gameanalytics.com/blog/10-great-game-telemetry-reads.html.

[34] Gabe Habash. *The Pleasure of Decision in Reading*. Juni 2012. URL: http://blogs.publishersweekly.com/blogs/PWxyz/2012/06/14/the-pleasure-of-choice-in-reading/.

[35] Jonin600. 2011. URL: http://www.fanpop.com/spots/minecraft/images/19669248/title/minecraft-photo.

[36] Sam Miller. *Areas Of User For Software Metrics*. Jan. 2008. URL: http://www.articlesbase.com/management-articles/areas-of-use-for-software-metrics-306127.html.

[37] Pirate Neilsouth. *Rage3D Forum Beitrag über Amigapackages*. 2010. URL: http://www.rage3d.com/board/showpost.php?p=1336244460&postcount=2.

[38] Matthew Pellett. *Assassin's Creed Ending Explained*. 2007. URL: http://www.computerandvideogames.com/175552/blog/assassins-creed-ending-explained/.

[39] Markus Persson. *Minecraft (alpha)*. 2009. URL: http://forums.tigsource.com/index.php?topic=6273.0.

[40] PiranhaGreg. *Minas Tirith in Minecraft*. 2012. URL: http://www.minecraftforum.net/topic/90844-mcexplorer-world-renderer/page__view__findpost__p__12404132.

[41] Chris Pruett. *Hot Failure: Tuning Gameplay With Simple Player Metrics*. 2010. URL: http://www.gamasutra.com/view/feature/134526/hot_failure_tuning_gameplay_with_.php.

[42] Tarandeep Singh. *Java VisualVM - Developer's Nightmare is Over*. 2009. URL: http://www.spieleradar.de/news/assassins-creed-2/0_14149_8107/rekordverdaechtig-eintrag-ins-guinness-buch.html.

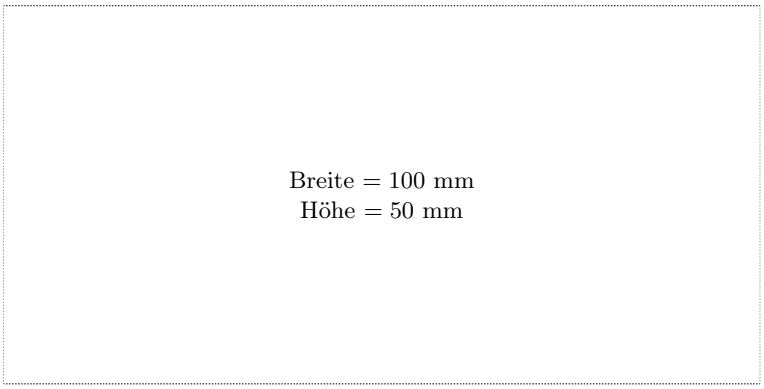
[43] Clive Thompson. *Halo3: How Microsoft Labs Invented a New Science of Play*. Aug. 2007. URL: http://www.wired.com/gaming/virtualworlds/magazine/15-09/ff_halo.

[44] *VGChartz der Haloserie*. URL: http://www.vgchartz.com/gamedb/?page=1&results=200&name=halo&platform=&minSales=0&publisher=&genre=&sort=GL.

[45] *VGChartz von Halo: Combat Evolved*. URL: http://www.vgchartz.com/game/939/halo-combat-evolved/.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —