Höhere Technische Bundeslehranstalt Wien 3, Rennweg

IT & Mechatronik

HTL Rennweg: Rennweg 89b

A-1030 Wien, Tel +43 1 24215-10, Fax DW 18

Diplomarbeit

eventuell KURZTITEL  
Ausgeschriebener Titel der Diplomarbeit

ausgeführt an der  
Höheren Abteilung für  
Informationstechnologie/Ausbildungsschwerpunk  
der Höheren Technischen Lehranstalt Wien 3 Rennweg

im Schuljahr 20??/20??

durch  
Mitarbeiter Eins (alphabetisch)  
Mitarbeiter Zwei  
Mitarbeiter Drei  
Mitarbeiter Vier

unter der Anleitung von

Hauptbetreuer  
eventuell Nebenbetreuer

Wien, Oktober 2014

Kurzfassung

Darum geht es.

Auf der Registerkarte 'Einfügen' enthalten die Kataloge Elemente, die mit dem generellen Layout des Dokuments koordiniert werden sollten. Mithilfe dieser Kataloge können Sie Tabellen, Kopfzeilen, Fußzeilen, Listen, Deckblätter und sonstige Dokumentbausteine einfügen. Wenn Sie Bilder, Tabellen oder Diagramme erstellen, werden diese auch mit dem aktuellen Dokumentlayout koordiniert. Die Formatierung von markiertem Text im Dokumenttext kann auf einfache Weise geändert werden, indem Sie im Schnellformatvorlagen-Katalog auf der Registerkarte 'Start' ein Layout für den markierten Text auswählen.

Text können Sie auch direkt mithilfe der anderen Steuerelemente auf der Registerkarte 'Start' formatieren. Die meisten Steuerelemente ermöglichen die Auswahl zwischen dem Layout des aktuellen Designs oder der direkten Angabe eines Formats. Wählen Sie neue Designelemente auf der Registerkarte 'Seitenlayout' aus, um das generelle Layout des Dokument s zu ändern. Verwenden Sie den Befehl zum Ändern des aktuellen Schnellformatvorlagen-Satzes, um die im Schnellformatvorlagen-Katalog verfügbaren Formatvorlagen zu ändern.

Abstract

That‘s why.

Ensure that the entire document's proofing language is set to English. Remember that proofing language is set for each word, even each individual character, not for the document as a whole, so even a single character in the wrong place can throw it off.

Open the document and hit Ctrl+A to select all text and objects in the document. Then go to Review | Language | Set Proofing Language and ensure it's set to "English

Ehrenwörtliche Erklärung

Ich versichere,

* dass ich meinen Anteil an dieser Diplomarbeit selbstständig verfasst habe,
* dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe
* und mich auch sonst keiner unerlaubten Hilfe bzw. Hilfsmittel bedient habe.

Wien, am

<eigenhändige Unterschriften aller Teammitglieder>

Präambel

Die Inhalte dieser Diplomarbeit entsprechen den Qualitätsnormen für „Ingenieurprojekte“ gemäß § 29 der Verordnung des Bundesministers für Unterricht und kulturelle Angelegenheiten über die Reife- und Diplomprüfung in den berufsbildenden höheren Schulen, BGBl. Nr. 847/1992, in der Fassung der Verordnungen BGBl. Nr. 269/1993, Nr. 467/1996 und BGBl. II Nr. 123/97.

Liste der betreuenden Lehrer

<[Dir|AV|Prof], akad. Grad, Vorname Name Hauptbetreuer>

<[Dir|AV|Prof], akad. Grad, Vorname Name Hauptbetreuer Stellvertreter>

<[Dir|AV|Prof], akad. Grad, Vorname Name Betreuer> ... (in alphabetischer Reihenfolge des Nachnamens)

<[Dir|AV|Prof], akad. Grad, Vorname Name Betreuer>

Liste der Kooperationspartner:

Inhaltsverzeichnis

[Kurzfassung 2](#_Toc401212867)

[Abstract 3](#_Toc401212868)

[Ehrenwörtliche Erklärung 4](#_Toc401212869)

[Präambel 5](#_Toc401212870)

[1 Beschreibung der Formatierung 7](#_Toc401212871)

[1.1 Vorlagen 7](#_Toc401212872)

[1.1.1 Formatvorlagen 7](#_Toc401212873)

[1.1.2 Schriften und Absätze 7](#_Toc401212874)

[1.1.3 Bilder 8](#_Toc401212875)

[1.1.4 Tabellen 9](#_Toc401212876)

[1.1.5 Formel 10](#_Toc401212877)

[1.1.6 Sourcecode 11](#_Toc401212878)

[1.1.7 Fachbegriffe 12](#_Toc401212879)

[1.1.8 Zitieren 12](#_Toc401212880)

[1.1.9 Quellenverzeichnis 13](#_Toc401212881)

[1.1.10 Rechtliches zum Zitieren 13](#_Toc401212882)

[1.2 Bad Practice 14](#_Toc401212883)

[1.3 Beispiele 16](#_Toc401212884)

[1.3.1 Zitieren mit Latex 16](#_Toc401212885)

[1.3.2 Formatierungen 16](#_Toc401212886)

[Tabellenverzeichnis 18](#_Toc401212887)

[Abbildungsverzeichnis 19](#_Toc401212888)

[Stichwortverzeichnis 20](#_Toc401212889)

# Software

Alle Softwarekomponente des Spiels wurden in C# innerhalb des Monogame Frameworks geschrieben.

## Monogame Framework

Abbildung 1. Monogame Logo  
Quelle: https://upload.wikimedia.org/wikipedia/commons/e/e6/MonoGame\_Logo.svg

Monogame ist eine Open-Source Weiterführung vom von Microsoft entwickelten XNA’ Framework. Es basiert auf der Programmiersprache C# und unterstützt DirectX als auch OpenGL.

* DirectX ist eine von Microsoft entwickelte Graphik API. Sie wird von Windows Systemen und Xbox Konsolen unterstützt.
* OpenGL ist eine Open-Source Graphik API. Sie ist Plattform übergreifend und kann für Windows, macOS und Linux Systeme verwendet geben.
* Xamarin?

Diese Bibliotheken werden vom Monogame Framework zum kommunizieren mit der Graphikkarte verwendet. Beim Erstellen eines Projektes muss man sich für eines dieser Zwei entscheiden.

Für dieses Projekt bietet sich OpenGL am besten an, da wir von keinen relevanten Einschränkungen betroffen sind und wir dann mit unserem Produkt mehrere Plattformen gleichzeitig ansprechen können.

Eine neu erstellte Monogame Vorlage sieht folgendermaßen aus:

|  |
| --- |
| using Microsoft.Xna.Framework;  using Microsoft.Xna.Framework.Graphics;  using Microsoft.Xna.Framework.Input;  namespace Game1  {  public class Game1 : Game  {  GraphicsDeviceManager graphics;  SpriteBatch spriteBatch;    public Game1()  {  graphics = new GraphicsDeviceManager(this);  Content.RootDirectory = "Content";  }  protected override void Initialize()  {  base.Initialize();  }  protected override void LoadContent()  {  spriteBatch = new SpriteBatch(GraphicsDevice);  }  protected override void UnloadContent()  {  }  protected override void Update(GameTime gameTime)  {  if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape)) Exit();  base.Update(gameTime);  }  protected override void Draw(GameTime gameTime)  {  GraphicsDevice.Clear(Color.CornflowerBlue);  base.Draw(gameTime);  }  }  } |

|  |  |
| --- | --- |
| Initialize() | wird für das Laden aller Elemente verwendet, welche nicht von der Pipeline geladen werden. (Services, Spielstände, Datenbanken) |
| LoadContent() | wird für das Laden aller Elemente verwendet, welche von der Pipeline geladen werden. (Graphiken, Sounds, Schriften) |
| UnloadContent() | wird für das Entladen aller Elemente verwendet, welche von der Pipeline geladen wurden. Dies befreit die verbrauchte RAM. |
| Update() | wird für das Aktualisieren der Spiellogik verwendet. |
| Draw() | wird für das Zeichnen mit der Graphikkarte verwendet. |

Initialize() und LoadContent() werden immer beim Programmstart aufgerufen.

UnloadContent() wird beim Programmende aufgerufen.

Update() und Draw() werden ständig aufgerufen und bilden die GameUpdateLoop.

Beim ausführen dieses Programms wird ein Bildschirm mit einem hellblauen Hintergrund erstellt. Beim Drücken der ESCTaste oder dem Menüknopf eines Controllers wir das Fenster geschlossen. Diese Vorlage ist alles was Monogame von uns an Arbeit entnimmt, alles Andere muss selbst entwickelt werden.

### UpdateLoop

Die Funktionen Update() und Draw() bilden die UpdateLoop, welche jede Bildwiederholung aufgerufen wird. Logik Code ist hier vom Graphik Code getrennt. Dies hat best practice, als auch technische Gründe (da der Aufruf dieser zwei Methoden sich unter Extremfällen doch verschieden verhält, ist für uns aber nicht relevant).

Update() {

In der Update Funktion wird der Logik Code ausgeführt. Dies beinhaltet so ziemlich alles, was nicht direkt auf den Bildschirm gezeichnet werden muss. Jeder zeitabhängige Code verwendet die DeltaTime zu der Berechnung von Zeitdifferenzen. Der Großteil des Codes wird sich hier befinden.

}

Draw() {

In der Draw Funktion wird der Code zum Zeichnen ausgeführt. Dies wird mithilfe eines Spritebatches erledigt. Ein Spritebatch ist eine Ansammlung von Befehlen die an die Graphikkarte am ende eines Updateloops geschickt werden. Diese werden innerhalb eines Batches organisiert, um die Optimierung von z.B. sich wiederholenden Graphiken zu ermöglichen.

}

### DeltaTime

Wenn man z.B. Bewegung simulieren möchte, benötigt man einen Bezug zu der vergangenen Zeit. Diese Funktionalität vergibt uns das GameTime Objekt, welches von Monogame bereitgestellt und verwendet wird.

Das Objekt GameTime besteht aus folgenden Attributen:

* TimeSpan TotalGameTime Die vergangene Zeit seit Spielstart
* TimeSpan ElapsedGameTime Die vergangene Zeit seit letztem Update()
* Bool IsRunningSlowly Ob das Spiel unter der festgelegten Frequenz rennt

Bei der Berechnung von Zeit gibt es zwei Herangehensweisen:

**Fixe Wiederholrate**

Bei der fixen Wiederholrate wird die Frequenz, mit der sich das Spiel selbst aufruft limitiert.

Es erlaubt für einfachere Kalkulation von Zeitdifferenzen, da diese im Idealfall immer gleich sind. Also wenn man sein Spiel auf 60fps’ limitiert, wäre ein Zeitsprung zwischen zwei Bildern immer 0.0166s (1/60s) lang. Damit umgeht man Berechnungen mit Hilfe von ElapsedGameTime. Wenn es dazu kommt, dass die Wiederholrate durch Performanceprobleme unter diese Schwelle fällt, verhält sich die Spiellogik gleich, da die Zeitdifferenz statisch definiert wurde.

Die Vor- und Nachteile diese Methode sind die Folgende:

**+** Zeitberechnung statisch und Logik Code ist vorhersehbarer

**+** Performance Schwankungen oberhalb der Schwelle werden abgeschnitten

**+** Das System überarbeitet sich nicht, was sinnvoll bei Laptops und Mobilgeräten ist, oder bei limitierter Stromversorgung

**-** Monitore mit Bildfrequenzen über der Schwelle werden nicht ausgenutzt, kommt negativ bei der Kundschaft an

**-** Unbrauchbar für Online Multiplayer – durch verschiedener Hardware würde es zu Desynchronisierungen kommen

Diese Methode ist mittlerweile veraltet, wird aber dennoch bei kleineren Projekten gerne verwendet.

**Variable Wiederholrate**

Bei der variablen Wiederholrate wird die Frequenz nicht limitiert und hat somit kein Maximum, das sie erreichen kann. Für die Berechnung der Zeit wird die GameTime verwendet. Die seit dem letzten Frame vergangene Zeit kann mit (float)gameTime.ElapsedGameTime.TotalSeconds abgefragt werden. Diese ist in Sekunden angegeben werden und muss als FActor mit anderen Zahlen multipliziert werden.

Die Vor- und Nachteile diese Methode sind die Folgende:

**+** Wird als State of the Art von Konsumenten gesehen

**+** Nutzt die Hardware vollkommen aus

**-** Bei extremen Performance Schwankungen können Logikprobleme entstehen (siehe Movement)

## Architektur

Da keine etablierte Engine für die Entwicklung des Spiels verwendet wurde, musste diese auch selber gemacht werden. Dies ist normal für Entwickler die nur ein Framework verwenden.

Unter einer Game Engine versteht man ein wiederverwendbares Stück Software, welches auf der Basis von Daten agiert. Dadurch kann eine Engine bei mehreren Projekten eingesetzt werden, da sich diese nur in den Daten unterscheiden und die Logik gleich bleibt.

Kommerzielle Game Engines beinhalten meist Entwickler Tools mit graphischer Oberfläche. Bei größeren Teams erlaubt das für erweiterte Zusammenarbeit, ohne dass sich jeder mit der Programmiersprache auskennen muss.

Da das Wraithknight Team aber nur einen Programmierer hat, sind diese Funktionalitäten nicht nötig.

### ECS

//hat Ähnlichkeiten mit Model-View-Controller

Die **E**ntity**C**omponent**S**ystem Design Pattern setzt Komposition über Vererbung.

Vom Konzept her ist sie das polare Gegenteil vom klassischen **O**bjekt**O**rientierten**P**rogrammieren. ECS wird meist in der Spieleentwicklung verwendet, da gerade dort die Schwächen von OOP sehr früh zu Problemen führen.

Die ECS Architektur befindet sich aber nur innerhalb eines vom Programmierer definierten ECS Environments. Im Falle von Wraithknight wäre das das tatsächliche Spiel. Das ECS Environment übernimmt Aufgaben wie das Entity-Management (das Löschen/Erstellen von Entitäten), das System-Management, sowie die verschiedenen Boot-Routinen[[1]](#footnote-1).

Einige Spielelemente, wie z.B. das Menu, benötigen das ECS nicht, und werden daher “klassisch” programmiert.

Als Beispiel:

//Soll ich das noch machen? Beispiel mit Vererbung, Deadly Diamond, Logik Konflikten

#### Entity

Eine Entity ist ein Container für eine beliebige Anzahl an Komponenten. In Wraithknight beinhalten Entities noch Metadaten, wie etwa:

* eine ID
* ein Boolean ob sie noch am “Leben” ist
* das Team (Freundlich, Feindlich, Neutral)
* den Typen der Entität, der bei der Erstellung verwendet wurde.

Diese Metadaten sind für eine funktionelle ECS Architektur nicht notwendig, erleichtern aber das Debugging.

//Code?

#### Component

Eine Komponente ist ein Datensatz der das Verhalten einer Entity beschreibt. Komponenten beinhalten KEINE SPIELLOGIK und sind mit einer Tabelle einer Datenbank zu vergleichen.

Eine MovementComponent hätte z.B. eine X/Y Position und eine Geschwindigkeit womit die Bewegung ausgedrückt wird. Wenn eine Entity eine aktivierte MovementComponent besitzt, ist sie dazu fähig sich zu bewegen und bewegt zu werden.

#### System

Systeme existieren meistens parallel zu einer Komponentenklasse. Sie führen eine Collection mit allen nötigen Komponenten, die sie überwachen. Innerhalb eines Systems findet man die Logik von den zugehörigen Komponenten.

Ein MovementSystem hätte eine Liste mit allen MovementComponents die gerade im Spiel vorhanden sind. Jedes Update wird über diese Liste iteriert und die Komponente aktualisiert. In unserem Beispielfall würde die Geschwindigkeit mit der vergangenen Zeit multipliziert werden und dann zu den Positionskoordinaten addiert werden.

### Gebundene Komponente

Manchmal ist es nötig, dass eine Komponente auf die Informationen einer anderen Komponente verfügen muss. In diesem Fall werden gebundene Komponente verwendet.

Die DrawComponent Klasse beinhaltet Informationen über das Zeichnen einer Graphik.

Das DrawSystem ist derzeit nur dazu verantwortlich die DrawComponent zu zeichnen.

Wenn sich eine Entity aber bewegt, also es wird die Position in der MovementComponent geändert, kriegt die DrawComponent davon nichts mit und wird immer noch auf der alten Position gezeichnet. In dem Fall binden wir die MovementComponent an die DrawComponent. Dann hat das DrawSystem Zugriff auf die tatsächliche Position einer Entity, nur indem es eine DrawComponent nach ihrer Bindings fragt.

//Codebeispiel?

## Assetmanagement

Die als PNG exportierten Graphiken müssen in die Spiellogik eingebunden werden. Dabei hilft der Monogame Content-Manager, welcher jedes Asset in eine .XNA Datei umwandelt. Innerhalb des Programmes unterscheiden sich die Ressourcen aber in der Klasse.

### Monogame Content-Manager

Der Monogame Content-Manager greift auf Dateien innerhalb des „Content“ Ordners zu und konvertiert diese auf .XNA Dateien um. Diese Konvertierung erfolgt aber nur bei der Kompilierung des Programms. Ressourcen, welche vom Content-Manager geladen wurden befinden sich nun in der Pipeline.

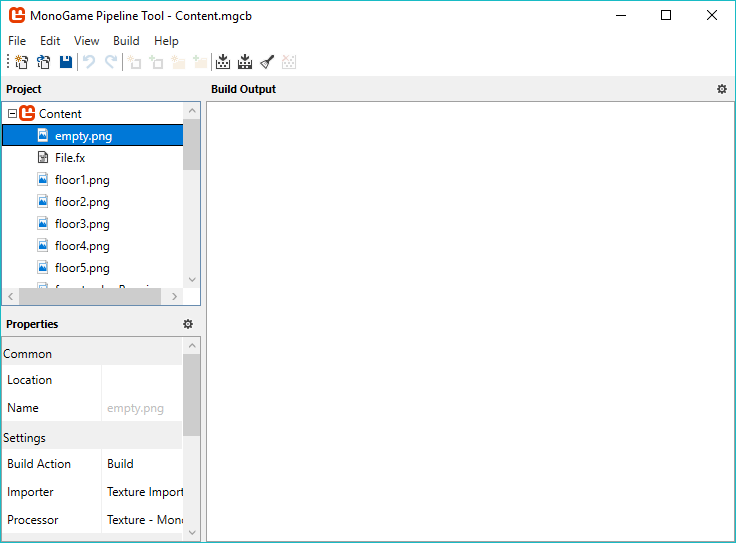
Es können optional noch Properties gesetzt werden, diese sind für das jetzige Projekt aber nicht relevant.

Abbildung 2. Monogame Content-Manager

Auf von der Pipeline geladene Objekte kann folgendermaßen zugegriffen werden:

//CODE

### Dynamische Asset-Bibliothek

Bei größeren Projekten verliert man schnell die Übersicht über eingebundene Ressourcen, wenn man keine ordentliche Verwaltung betreibt.

Die Bibliothek ist eine global erreichbare statische Klasse. Sie führt jeweils eine Collection für eine Art von Ressource. Auf die Bibliothek kann mit Asset.GetTexture() zugegriffen werden.

Eine weitere Funktion der Bibliothek, ist das automatische Entladen von nicht mehr verwendeten Assets. Diese Funktionalität ist vor allem bei Spielen mit hohen Graphikauflösungen, somit auch hohen Dateigrößen, relevant. In diesem Projekt sind die verwendeten Graphiken von sehr geringer Größe, weshalb das automatische Entladen keinen technisch relevanten Nutzen hat, sondern nur dank der Automatisierung dem Entwickler etwas Arbeit abnimmt.

## Entity-Erstellung

Entitäten bestehen aus einer Liste von Komponenten und einigen Metadaten, welche die Verwaltung vereinfachen.

Die Entitäten eines bestimmten Typen bestehen aber immer aus den gleichen (oder zumindest ähnelnden) Komponenten. Ein Baum besteht immer aus einer DrawComponent und einer CollisionComponent. Bei der Generierung einer Entität wird der Typ und ein paar Startvariablen bekannt gegeben.

Nachdem die Entität mit den Komponenten gefüllt wurde, werden die Metadaten in Relation zu den Inhalten gesetzt. Dies finalisiert die Generierung. Jetzt müssen die Komponenten noch den Systemen bekannt gegeben werden. Jedem System wird eine Liste von zu registrierenden Komponenten gegeben und sie suchen sich die jeweilig relevanten Komponenten aus und verarbeiten sie.

//Entity wird erstellt, per typ sortiert, komponente eingefügt und in systemen registriert

//CODE AUSSCHNITT ZEIGEN

## Input

Unter Input versteht man jedes vom User erstelltes Signal, welches der Computer verarbeiten soll. In der Spieleentwicklung umfasst das hauptsächlich die Maus, Tastatur und gegebenenfalls den Kontroller. In letzter Zeit tauchten auch andere Input Möglichkeiten auf, wie zum Beispiel Sprachsteuerung oder Eyetracking, diese befinden sich aber noch in einer Niche oder helfen behinderten Spielern die Erfahrung zugänglicher zu machen.

In diesem Projekt wird Userinput nur per Keyboard und Maus umgesetzt.

Um das Verwalten von Userinput sorgt sich die InputReader Klasse.

### Keyboard

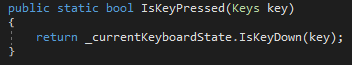
Monogame hat ein eingebautes enum namens Keys für alle verwendbare Tastenanschläge.

Abbildung 3. Auslesen von Keyboard-Input

Mit der obigen Funktion kann man abfragen, ob eine Taste auf dem Keyboard gedrückt ist.

Parallel zum \_currentKeyboardState wird auch \_lastKeyboardState abgespeichert. Mithilfe davon kann man Abfragen wie

* Release: \_lastKeyboardState == true && \_currentKeyboardState == false
* Trigger: \_lastKeyboardState == false && \_currentKeyboardState == true
* Active: \_currentKeyboardState == true

durchführen.

Mit diesen drei Input Abfragen hat man jeden relevanten Keyboardinput abgedeckt.

### Maus

Die Position vom Mauszeiger wird als 2-dimensionaler Vektor abgespeichert.

Auf die Maustasten wird, wie bei dem Keyboard, mit einem enum zugegriffen.

Genauso wie bei dem Keyboard, gibt es auch hier die \_currentMouseState und \_lastMouseState Variablen.

Die Abfragen funktionieren gleich wie beim Keyboard Schema:

* Release: \_lastMouseState == true && \_currentMouseState == false
* Trigger: \_lastMouseState == false && \_currentMouseState == true
* Active: \_currentMouseState == true

Auch hier hat man mit diesen drei Inputabfragen jeden relevanten Mouseinput abgedeckt.

## Graphiken

Wenn die Graphiken erfolgreich von der Monogame Pipeline geladen wurden, kann auf sie mithilfe der Asset-Bibliothek zugegriffen werden. Eine Entität bekommt graphische Funktionalitäten mit Hilfe von einer DrawComponent, welche die geladene Textur speichert. Soll diese dann animiert werden, bindet man eine AnimationComponent an die DrawComponent. Eine DrawComponent alleine weiß nichts über die eigentliche Position der zugehörigen Entität. Möchte man, dass diese sich mit der Entität mitbewegt, muss eine MovementComponent an die DrawComponent gebunden werden.

### Simple Graphiken

Simple Graphiken bestehen nur aus einer einzigen Textur und einer einzigen DrawComponent.

Sie werden hauptsächlich bei stationären Entitäten, wie Kulissen oder Bäumen eingesetzt.

### Komplexe Graphiken

Komplexe Graphiken bestehen aus mehreren Texturen mit mehreren DrawComponents. Bei einer Komplexen Graphik wäre z.B. der Oberkörper und die Beine in jeweils separaten DrawComponents. Dies ermöglicht auch separate Animation von den einzelnen DrawComponents.

Sie finden derzeit keinen Einsatzbereich in der Diplomarbeit, aber die Systeme dazu sind funktionsfähig.

### Simple Animationen

Animationen sind eine Abfolge von Frames (Texturen). Jedes Frame hat seine selbst definierte Anzeigezeit. Die Anzeigezeit gibt an, wie lange ein Frame in der Animation sichtbar ist. Ist die Anzeigezeit abgelaufen, wird die nächste Frame der Animation geladen. Ist der letzte Frame abgelaufen, gilt die Animation als beendet.

Animationen generell werden durch bestimmte Trigger ausgelöst. Die Trigger werden mithilfe der StateComponent zu einer AnimationComponent kommuniziert. Jedes System kann potentiell einen Trigger setzen, weshalb die StateComponent eine besondere Komponente ist, denn jede andere Komponente hat direkten Zugriff auf sie, wenn sie in einer Entität definiert ist.

Eine Animation hat auch die „nächste“ Animation abgespeichert. Diese kann auf sich selbst verweisen (womit die Animation sich wiederholt) oder auf eine andere Animation verweisen (womit man Animationen „aneinanderketten“ kann, um komplexe Animationen zu bilden).

### Komplexe Animationen

Komplexe Animationen sind mehrere aneinander gekettete simple Animationen. Eine komplexe Animation kommt bei der Laufanimation des Helden vor. Wenn er zum laufen beginnt, wird ein Trigger gesetzt, welcher den Übergang in die Laufanimation auslöst. Ist der Übergang fertig, wird der Run-Cycle geloopt. Hört der Held wieder auf zu laufen, wird wieder ein Trigger gesetzt, welcher die Animation für die Abbremsung auslöst.

//BILDER

## Bewegung

Entities, die sich bewegen können, besitzen eine MovementComponent. Diese ist auch im MovementSystem registriert und wird von diesem verwaltet.

Bewegung in Spielen kann sehr komplex werden, mit vielen Nuancen um sie möglichst realistisch oder gut anfühlend zu machen. Der Spaß von Platformern (nicht unser Genre) kommt hauptsächlich von der Bewegung und Physik. Es fühlt sich gut an, ein physikalisches Objekt mit Gewicht und Momentum durch die Welt zu bewegen.

Da das Spiel 2-dimensional ist, wird die Bewegung auch nur in 2 Dimensionen simuliert, in der X- und Y-Achse.

### Vektoren

Das Monogame Framework stellt eine Vector2 Struct zur Verfügung, welche eine kartesische, 2-dimensionale Koordinate darstellt. Diese eignet sich gut zu der Repräsentation einer Position, aber nicht so gut für Geschwindigkeiten.

Würde man bei Geschwindigkeiten eine kartesische Koordinate verwenden, würde es einem schwer fallen die Geschwindigkeit mit der Zeit abnehmen zu lassen. Im Spiel ist das aber ein gewünschter Effekt. Der Boden, auf dem sich eine Entität bewegt, bremst sie durch Reibung ab, bis sie zum Stillstand kommt. Bei kartesischen Koordinaten wäre das leicht umzusetzen: Jedes Update() wird die X- und Y-Koordinate mit der Abbremsung subtrahiert. Die zwei Achsen sind hier aber voneinander unabhängig, was heißt, dass eine Achse vor der anderen auf 0 abgebremst werden kann. Dieser Ansatz funktioniert bei 2D-Sidescrollern, wie dem klassischen Super Mario. Hier schaut man mit einem 90 Grad Winkel auf eine waagrechte Spielfläche. Die Y-Achse beschreibt die vertikale Höhe über dem Boden.

Bei Top-down Spielen, wie unserem, definieren die Achsen positionelle Koordinaten, wie bei dem Spiel „Schiffe versenken“. Um hier eine realistische Abbremsung zu verwirklichen, muss man die Kartesische Koordinaten zuerst in Polare umwandeln, und dann die Länge davon mit der Abbremsung verkürzen. Hierfür wurde Coord2 Struct selbst erstellt, welche aus einem kartesischen Vector2 und einem polaren Polar2 (auch selbst erstellt) besteht. Die Achsen von Coord2 können nur durch die Funktionen manipuliert werden, welche automatisch zwischen polaren und kartesischen Koordinaten umwandeln.

#### Kartesisch

Kartesische Koordinaten bestehen aus einer X- und Y-Achse. Das von Monogame vorgegebene Vector2 Struct bietet auch einige arithmetische Funktionen, welche die spiel relevanten Berechnungen vereinfachen.

Kartesische Koordinaten eignen sich für die Darstellung von Positionen.

#### Polar

Die Polare Implementierung ist selbst gemacht und wurde nicht vom Framework vorgegeben. Sie besteht aus einem Winkel und einer Länge.

Polare Koordinaten eignen sich für die Darstellung von Bewegungen.

### Beschleunigung und Abbremsung

Die größte Herausforderung bei dem Entwurf eines guten Bewegungssystem besteht darin, dass sich die Bewegung realitätsnah und befriedigend anfühlt.

Man kann den Userinput direkt auf die Geschwindigkeit des vom Spieler kontrollierten Charakter binden. Dies ist der simpelste Ansatz und wird gerne bei Arcade-Style Spielen, wie Pacman, verwendet.

Doch in der realen Welt beschleunigen Objekte nicht sofort auf die gewünschte Geschwindigkeit, denn sie besitzen eine Trägheit. Um dieses Verhalten in der virtuellen Welt zu simulieren bindet man den Userinput stattdessen auf die Beschleunigung des Charakters, welche dann auf die Geschwindigkeit wirkt. Hier muss man auf die Feinheiten der Bewegung achten, damit der Spieler nicht ein „schwammiges“ Feedback wahrnimmt.

Ein gut umgesetztes Moveset trägt eine Menge zum *Gamefeel* bei.

## Kollision

Wenn zwei physikalische Objekte sich berühren kommt es zu einer Kollision. Wie damit dann umgegangen wird bezeichnet man „Collision-Response“.

Das Objekt, welches sich in Bewegung befindet und die Kollision auslöst nennt man „Actor“. Das Objekt, welches von der Kollision betroffen wird nennt man „Target“.

Im üblichen Fall von physischen Objekten kommt es zu einer Blockade. Dabei behält das stillstehende Objekt, das Target, seine Position und das Bewegende Objekt, der Actor, muss seine Position und Bewegung aktualisieren.

Logische Objekte können auch miteinander kollidieren, wie zum Beispiel ein Projektil mit einem Gegner. Hier wird eine physische Response alleine nicht ausreichen, denn der Gegner muss vom Projektil schaden nehmen. In diesem Fall muss eine logische Response ausgelöst werden, mit dem Projektil als dem Actor, und dem Gegner als Target. Bei Bedarf kann immer noch eine physische Response durchgeführt werden.

### AABB

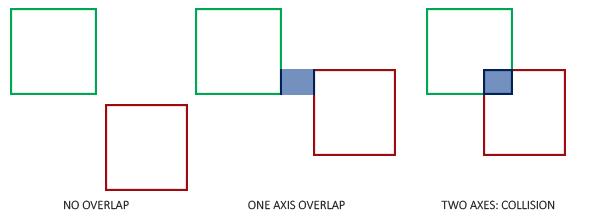
Die **A**xis-**A**ligned-**B**ounding-**B**oxes Technik ist das simpelste Prinzip zu der Berechnung von Kollisionen. Physische Objekte werden als Boxen dargestellt. Eine Box besteht aus einer Position und den Dimensionen. Die Boxen sind nicht rotationsfähig, da, wie der Name sagt, sie an die Achsen ausgerichtet ist.

Abbildung 4. AABB Kollision  
Quelle: https://learnopengl.com/img/in-practice/breakout/collisions\_overlap.png

Ob zwei Boxen mit einander kollidieren kann man mit folgender Formel berechnen:

if(A.Left < B.Right && A.Right > B.Left && A.Top < B.Bottom && A.Bottom > B.Top)

Table 1 – Bezeichnung der AABB Kollisions Formel

|  |  |
| --- | --- |
| A = | *AABB* |
| B = | *AABB* |
| Left = | *X* |
| Right = | *X + Width* |
| Top = | *Y* |
| Bottom = | *Y + Height* |

Diese Formel stellt nur fest, ob zwei AABBs einander schneiden. Wie damit umgegangen wird, wird nicht beschrieben.

### Minkowski Summe

Die Minkowski Summe ist eine mathematische Formel, welche bei der Berechnung von Geometrischen Körpern verwendet wird. Dabei werden zwei Formen miteinander addiert, um die Summenform zu bilden. Dies ist einer der einfachsten Algorithmen, um den Penetrations Vektor einer Kollision zu berechnen. Die Summenform kann gegen den Punkt (0/0) getestet werden. Falls sich der Punkt in der Form befindet, besteht eine Kollision.

Falls eine Kollision besteht, kann man auch die kürzeste Distanz nach außen, den Penetrations Vektor berechnen. Der Penetrations Vektor ist die relative Distanz von (0/0) bis zu der nächsten Seite.

Wenn man den Penetrations Vektor berechnet hat, kann dieser zu der Position des Actors addiert werden, um den Actor aus der Kollision herauszubewegen.

https://blog.hamaluik.ca/posts/simple-aabb-collision-using-minkowski-difference/

### Collision-Response

Wenn eine Kollision erkannt wurde, muss sie irgendwie gelöst werden. Eine Kollision zählt als gelöst, wenn nach der Lösung die Kollision nicht mehr besteht. Auch wenn eine Kollision mathematisch korrekt gelöst wurde, könnte sie logisch Fehler beinhalten. In Wraithknight bestehen die Bedingungen dazu aber nicht, weshalb dies kein Problem ist. //siehe swept cd

#### Physische

Wenn ein physisches Objekt mit einem anderen physischen Objekt kollidiert entsteht eine physische Kollision, welche eine physische Response beansprucht.

Bei physischen Responses unterscheidet man zwischen den folgenden:

//BILDER

##### **Block**

Die üblichste Art von Response. Bei einer Kollision wird der Actor an die Grenzen vom Target gesetzt, und die Bewegung auf der Achse, auf welcher kollidiert wurde, wird auf 0 gesetzt.

##### **Bounce**

Bei einer Kollision wird der Actor an die Grenzen vom Target gesetzt, und die Bewegung auf der Achse, auf welcher kollidiert wurde, wird negiert.

##### **Stick**

Bei einer Kollision wird der Actor an die Grenzen vom Target gesetzt. Der Actor verliert dann die Kollisions Komponente und bindet sich an die Bewegung vom Target. Sinnvoll bei Wurfmessern, welche in z.B. Gegnern stecken bleiben sollen.

##### **Disappear**

Bei einer Kollision wird der Actor verschwinden. Ohne einem Actor kann es keine Kollision mehr geben, somit gilt sie als gelöst. Sinnvoll bei Partik Eleffekten.

#### Logische

Wenn ein logisches Objekt mit einem anderen logischen Objekt kollidiert entsteht eine logische Kollision, welche eine logische Response beansprucht. Logische Kollisionen müssen nicht unbedingt gelöst werden, es muss nur auf sie reagiert werden.

Im Umfang der Diplomarbeit werden nur Projektile als logische Kollisionsfähige Objekte verwendet. Im groberen Umfeld der Spieleentwicklung findet man auch noch Triggers (Bereiche, welche auf Kontakt ein Event auslösen).

Um den folgenden Abschnitt zu verstehen, muss man sich zuerst die ProjectileComponent genauer anschauen. Für Kollisionen hat die ProjectileComponent folgenden relevante Attributen:

|  |  |
| --- | --- |
| public int Power; | Ist die “Lebensanzahl” eines Projektils. |
| public int Damage; | Ist der Schaden, welcher auf Kontakt verursacht wird. |
| public bool IsPhasing; | Besagt, ob ein Projektil durch andere logische Objekte “durchschwebt”. Ein Schwertschlag kann mehrere Gegner gleichzeitig treffen, ein Pfeil nicht. |

Bei logischen Responses unterscheidet man zwischen den folgenden:

//HIER UNBEDINGT BILDER

##### **Das Schadensystem**

//Hier die Ideen hinter den berechnungen erklären

##### **Projektil auf Projektil**

Kollidiert ein Projektil mit einem anderen Projektil, sollen diese sich gegenseitig beeinflussen.

##### **Projektil auf Leben**

Kollidiert ein Projektil mit einem “lebendem” Objekt (Held, Gegner), soll dieses Schaden nehmen.

Das Projektil versucht so viel Schaden zu machen wie möglich:

Wenn Damage kleiner ist als Power, hat das Projektil genug Kraft um den vollen Schaden zu machen. Dabei nimmt die Target HealthComponent den vollen Damage vom Actor als Schaden.

Ist Power kleiner als Damage, hat das Projektil nicht genug Kraft um den vollen Schaden zu machen. Deswegen nimmt das Target nur die übrige Power vom Actor als Schaden.

Das Projektil verliert so viel an Power, wie es Schaden gemacht hat. Dabei kann es nicht mehr Schaden machen, als das Target an Leben übrig hat.

//Berechnungen zum erklären?

## Levelgeneration

Ein Markenzeichen vom Roguelike Genre ist die procedurale Levelgeneration. Es gibt viele Algorithmen für das generieren von zufälligen Leveln. Eines der einfachsten ist Cellular Automata, was sehr anschauliche Ergebnisse liefert.

### Cellular Automata

In Cellular Automata agiert jede Zelle selbstständig, abhängig von ihrer Umgebung. Die Zellen befinden sich in einem Gitter und beginnen in einem zufälligen Startzustand. Das System durchgeht dann mehrere Generationen, wo die Regeln angewendet werden.

Eines der bekanntesten Systeme ist „Conway’s Game of Life“.

Eine Zelle bestimmt ihren Inhalt laut ihrer Nachbarn. Die Regeln, wodurch eine Zelle den Inhalt bestimmt, werden vom Entwickler vorgegeben. Der Vorteil dabei ist, dass man Presets erstellen kann. Man kann problemlos zwischen den Presets wechseln, um, z.B., die Schwierigkeit eines Levels zu erhöhen.

//BILDER

### Finaler Algorithmus

## Künstliche Intelligenz

Das Ziel von künstlicher Intelligenz ist es realistisches Verhalten zu illusionieren. Simulieren ist in den meisten Fällen nicht notwendig, da vorallem in Computerspielen der Spieler mit anderen Sachen abgelenkt ist. Um intelligentes Verhalten zu simulieren, braucht man auch die notwendige Rechenleistung dafür, weshalb man sehr sparsam mit künstlicher Intelligenz umgehen sollte.

### Generelle Intelligenz

C:\Users\Fillip\Documents\ShareX\Screenshots\2019-03\devenv_2019-03-28_12-51-37.pngC:\Users\Fillip\Documents\ShareX\Screenshots\2019-03\devenv_2019-03-28_12-45-05.pngJedes, von einer Intelligenz wahrnehmbare, Objekt wird mit einer IntelligenceNode Komponente markiert. Eine Intelligenz fähige Entität besitzt eine IntelligenceComponent Komponente, in welcher sich Anweisungen für das Verhalten befinden.

Abbildung 5. Intelligenz Anweisungen

Die “Gegner Ritter” Entität besitzt folgende Anweisungen:

* Wenn der Held 100 Einheiten von dir entfernt ist, attackiere ihn und aktiviere einen Cooldown von 1000 Millisekunden

*Wenn dies nicht der Fall ist, springe zur nächsten Anweisung*

* Wenn der Held 300 Einheiten von dir entfernt ist, laufe auf ihn zu und aktiviere einen Cooldown von 250 Millisekunden

Der updateCooldownMilliseconds Parameter bezieht sich nur auf die Intelligenz. Während dieser Cooldown abläuft ist die Entität immer noch funktionsfähig, nur die Intelligenz Aufrufe werden limitiert und die Entität trifft für diese Zeit keine Entscheidungen.

## UI

//Warte auf graphiken

### Menu

### HUD

## Gamescreens

Gamescreens sind, wie der Name sagt, “Spiel Bildschirme”. Es ist ein abstraktes Konzept, um Spielstadien zu beschreiben. Das Hauptmenü, Pausemenü und das eigentliche Spiel sind Gamescreens. Diese sind auf einem Stack organisiert, wo nur der oberste Screen aktualisiert wird. So kann man ein Pausemenü über ein bereits laufendes Spiel legen. Das Spiel erscheint immer noch im Hintergrund des Pausemenüs, doch es ist eingefrohren und übernimmt keinen Input. Stattdessen ist das Pausemenu das aktive Screen, bis es wieder geschlossen wird.

## Performance

Performance war schon immer ein wichtiges Thema bei der Entwicklung von Spielen. Mit der Zeit wurden immer mächtigere Spieleplattformen veröffentlicht, weshalb man bei der Entwicklung von kleineren Spielen jetzt mehr Toleranzraum hat.

Bei der Entwicklung auf die PC Plattform bleibt es aber dennoch wichtig das Produkt möglichst gut zu optimieren, da 10 Jahre alte Laptops auch noch als PC gelten und somit auch Teil der potentiellen Kundschaft sind.

Die überwiegende Mehrheit von Performanceprobleme sind Logikfehler. Darunter fällt nicht geeignete Software-Architektur, schlechte Organisierung von Daten oder ein ineffizienter Algorithmus. Natürlich hat jede Zeile Code einen Einfluss auf die letztendliche Performance des Programms, aber wenn der Einfluss eines “Problemes” so klein ist, dass man ihn zwischen Compiler-Optimierungen nicht erkennt, gilt er als irrelevant.

Da Performance Testing ein ständiger Vorgang ist, könnte man alleine darüber ein ganzes Diplomarbeits Buch schreiben. Aus Übersichtsgründen wurden daher die 3 Gebiete der Spiel-Optimierung hervorgehoben, mit je einem angewendetem Beispiel.

### CPU – WallCollision

Im Spiel müssen ständig Kollisionen geprüft werden. Dies ist ein Aufwand von O(n²) und steigert sich exponential mit der Anzahl an Kollisionsboxen. Vor allem die Kollisionsüberprüfung ist ein massiver Leistungsfresser, weshalb jede Optimierung in diesem Bereich langlebig die Performance verbessert. Eine Optimierungsmöglichkeit wäre ein Collision Tree.

Ein Collision Tree unterteilt den nach Kollisionen geprüften Bereich in mehrere kleinere Bereiche. Der Actor befindet sich in einem dieser Unterbereiche und wird nur mit anderen Kollisionsboxen innerhalb dieses Unterbereiches geprüft. Um es anhand eines realen Beispiels zu erklären: Ein Auto in Wien muss nicht mit einer Wand in Graz geprüft werden. Die zwei Orte sind so weit voneinander entfernt, dass eine Kollision unmöglich wäre. Dieses Spiel kann man dann weiter und weiter spielen, und die verkleinerten Unterbereiche in noch mehr, noch kleinere Unterbereiche aufteilen.

Im Fall von Wraithknight sind alle Wände und Kulissen gleich groß, und an einem Grid angelegt. Daher kann man sie ganz einfach in ein Array einspeichern. Durch die Position des Actors kann man die Position im Array berechnen und dann nur noch die acht Nachbarzellen überprüfen.

Die oben erwähnte Implementierung vermindert den Rechenaufwand auf O(1).

Diese Optimierung hat die Wiederholfrequenz um ~40fps gesteigert.

### GPU – SpriteBatching

Das Zeichnen auf dem Bildschirm wird von der Graphikkarte umgesetzt. Damit die Graphikkarte eine Textur zeichnen kann, muss sie zuerst in den Speicher geladen werden. Dies verbraucht Zeit und muss für jede Graphik, jedes Update durchgeführt werden. Im schlimmsten Fall muss jedes mal, wenn eine Textur gezeichnet wird, die alte entladen und die neue geladen werden.

Diesen Prozess kann man optimieren, indem man alle Graphiken mit derselben Textur direkt hintereinander zeichnet. Wenn die Textur gewechselt wird, werden alle Graphiken nacheinander gezeichnet, welche die neue Textur verwenden. Hiermit minimiert man das Laden und Entladen der Graphikkarte und erlaubt ihr sich mehr auf ihre eigentliche Aufgabe zu konzentrieren, nämlich Graphiken auf den Bildschirm zu zeichnen.

Diese Optimierung hat die Wiederholfrequenz um ~5fps gesteigert.

Der Effekt ist größer bei schlechteren Graphikkarten.

### Memory – Referenz GC

Quelle: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/

Eine Fähigkeit von C++ ist die manuelle Verwaltung vom Arbeitsspeicher. In C# kümmert sich der Garbage Collector darum. Auch wenn es seine Vorteile mit sich bringt, damit der Garbage Collector seine Arbeit verrichten kann, muss er zuerst den gesamten Thread stoppen. Desto ineffizienter die interne Datenstruktur des Spieles ist, desto länger dauert dieser Freeze.

Man soll versuchen so wenige Pointer wie möglich zu verwenden. Auch Redundanz sollte in oft vorkommenden Objekten vermieden werden. Systeme sollten so wenig wie möglich übereinander wissen und keine direkten Pointer aufeinander haben. Architekturen sollten Pyramidenförmig sein, mit den Pointern nach unten zeigend.

Diese Optimierungen haben die GC unbemerkbar gemacht.

1. Eine Abfolge an Anforderungen die beim Spielstart erfüllt sein müssen (z.B. Levelgeneration) [↑](#footnote-ref-1)