Inhaltsverzeichnis

1.Eidesstattliche Versicherung....3
2.Projektmappe STAR....4
2.1.Allgemeine Beschreibung....4
2.2.Technische Beschreibung....4
2.3.Aufbau....4
3.S.T.A.R....5
3.1.Hierarchischer Aufbau der „Verwaltungsebene“....5
3.1.1.Aufbau....5
3.1.2.Ablauf....5
3.2.IGameState....6
3.3.Game State: Game (SGame)....6
3.3.1.Der Spieler....7
A.Allgemein....7
B.Aufbau....7
C.Ablauf....7
3.3.2.Das Level....7
A.Level Tiles....7
B.Level Variables....7
C.Quadtree....8
C.I.Aufbau....8
C.II.Funktionsweise....8
3.3.3.Die Interactive Objects....9
A.Allgemein....9
3.3.4.GraphXPack....9
3.3.5.Layer....10
A.Allgemein....10
B.Layer Object....10
C.FXData....10
D.Parallax Layer....10
E.Cloud Layer....10
F.Deco Layer....10
3.3.6.Camera....11
A.Allgemein....11
B.Funktionsweise....11
3.3.7.Game Logic....12
A.Kill Event....12
B.Exit Game Event....12
C.Start Level Countdown....12
3.3.8.Enemy....13
A.Allgemein....13
B.Aufbau....13
C.Enemy Manager....13
D.Animation....13
3.4.Game State: Menu (SMenu)....14
3.4.1.Aufbau....14
3.4.2.Main Menu....14
3.4.3.Options Menu....14
3.4.4.CustMapMenu....14

3.5.Game State: Loading Screen(SLoadingScreen).....15
3.5.1.Allgemein.....15
3.5.2.Ablauf.....15
3.6.Effekte.....15
3.6.1.Allgemein.....15
3.6.2.Aufbau.....15
4.Projekt StarEdit.....16
4.1.Allgemeine Beschreibung.....16
4.2.Form StarEdit.....16
4.3.Form Level Editor.....16
4.3.1.Allgemein.....16
4.3.2.Das LevelControl.....16
4.3.3.Die Steuerelemente.....16
4.4.Form Enemy Editor.....17
4.4.1.Allgemein.....17
4.4.2.Enemy Control.....17
4.4.3.Steuerelemente.....17
4.4.4.Tools.....17
5.Projekt AStar.....18
5.1.Allgemein.....18
5.2.Aufbau.....18
5.2.1.Map.....18
5.2.2.A* Wegfindung.....18
6.Projekt XNAParticleSystem.....19
6.1.Allgemein.....19
7.Projekt CUDAParticleEngine.....20
7.1.Allgemein.....20
7.2.Interop-Services.....20
8.Development Log S.T.A.R.....21
9.Literaturverzeichnis.....32

1. Eidesstattliche Versicherung

Hiermit versichere ich, dass alles was Ihnen vorliegt, von mir selbstständig erarbeitet wurde. Hilfen und Quellen wurden im Abschlussbericht ausreichend angegeben.

....

Name Unterschrift

Datum: .....

2. Projektmappe STAR

2.1. Allgemeine Beschreibung

S.T.A.R. („Stewie The Awesome Runner“) ist ein Jump 'n' Run Spiel. Es basiert auf einem Tile-Based-Map Prinzip, d.h. das Level besteht aus vielen kleinen Rechtecken. Ziel ist es an das Ende (Exit) des Levels zu gelangen ohne zu Sterben. Um dieses zu erreichen, muss der Spieler Fallen und Gegnern ausweichen und das Ende des Levels innerhalb der Zeit erreichen. Steuern kann man den Spieler wahlweise über die Tastatur oder den Xbox 360 Controller. Durch den Editor wird es jedem ermöglicht seine eigenen Levels und Gegner zu erstellen. Die Level kann man in vieler Weise an seine Wünsche anpassen, wie z.B. Hintergrund, grundsätzlicher Grafikstil, Farbe, Dekorationen usw.. Der Gegner Editor ermöglicht die Erstellung der Animationen eines Gegners, welche sich durch eine Anzahl von Tools relativ bequem erstellen lässt.

2.2. Technische Beschreibung

Programmiert wurde das Spiel ursprünglich in der Programmiersprache C# in der .Net Version 3.5 mit dem Erweiterungsframework XNA 3.1. Die aktuelle Version liegt in der .Net Version 4.0 mit XNA 4.0 vor. Zur Programmierung wurde zuerst Microsoft Visual Studio 2008 Team System und dann Microsoft Visual Studio 2010 Ultimate verwendet. Das Unterprojekt CUDAParticleEngine wurde in CUDA C/C++ in der Version 3.2 geschrieben. Um CUDA in der Entwicklung nutzen zu können, benötigt man Parallels Nsight von nVidia als Add-In für Visual Studio. Generell benötigt man für CUDA eine nVidia GeForce der Reihe 8xxx oder höher.

Die Texturen für das Spiel wurden mit Adobe Photoshop CS 5 Student and Teachers Edition und Adobe Illustrator CS 5 Student and Teachers Edition erstellt.

2.3. Aufbau

Zur Zeit der Erstellung dieser Dokumentation (Juni 2011) besteht das gesamte Projekt aus 216 Typen die sich wie folgt aufteilen:

- 110 Klassen
- 54 Enumerationen
- 13 Strukturen
- 34 Delegaten
- 5 Schnittstellen

welche insgesamt ca. 28243 C#, 625 CUDA C/C++ und 607 HLSL Codezeilen beinhalten.

Zu dem ist die Projektmappe in 5 Unterprojekte (und 4 Content Projekte) unterteilt:

- Star
- StarEdit
- AStar
- cudaParticleEngine
- XNAParticleSystem

3. S.T.A.R.

3.1. Hierarchischer Aufbau der „Verwaltungsebene“

3.1.1. Aufbau

Die hier genannte Verwaltungsebene verwaltet, wie der Name schon sagt, den hauptsächlichen Ablauf des Spiels. Spezieller gesagt verwaltet diese Ebene mithilfe der GameManager Klasse in welchem Status bzw. GameState sich das Spiel gerade befindet. Außerdem organisiert diese Klasse die Initialisierung und Verwerfung der einzelnen GameStates. Weiterhin beinhaltet die Klasse die Konfiguration (Options) des Spiels. Diese Klasse wird von der Game1 Klasse instanziiert. Die Abb. 3.1 macht dies deutlich:

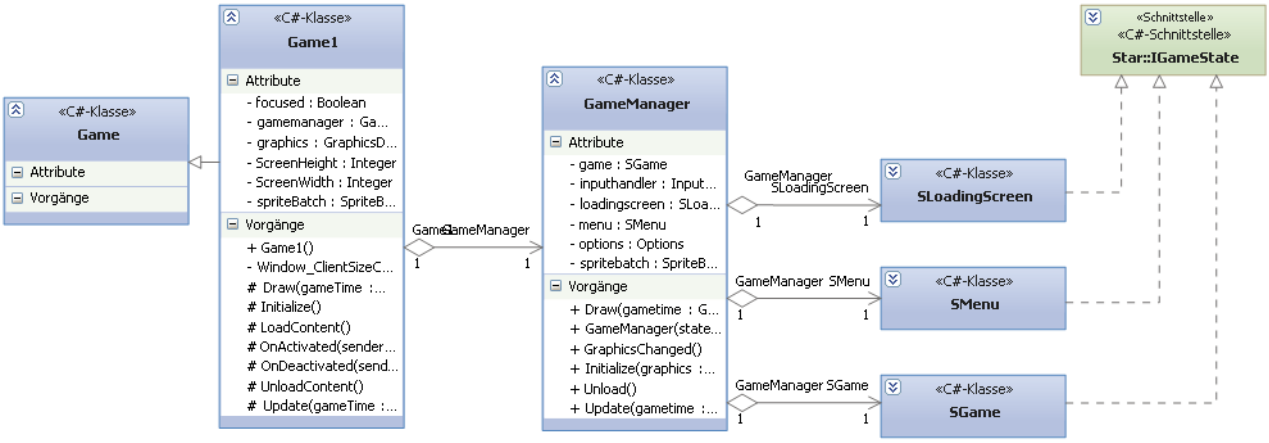

Abb. 3.1: Verwaltungsebene

3.1.2. Ablauf

Zu Beginn des Spiels werden alle GameStates über die Methode Initialize() initialisiert, die Konfiguration, falls vorhanden, aus der Datei geladen oder eine standardmäßige Konfiguration erstellt. Nach der Initialisierungsphase wird von der Game1 Klasse aus regelmäßig, solange das Spiel fokussiert ist, die Update() Methode aufgerufen. In dieser Methode() wird der aktuelle GameState durch die IGameState.Update() upgedatet. Diese Methode gibt solange den eigenen GameState zurück, bis ein Wechsel (z.B. von Menu zum Laden) auftritt. Der GameManager überprüft, ob dies der Fall ist und aktiviert zum einen den neuen GameState und zum anderen setzt er die Überblendungsanimation zurück. Falls der GameState auf Exit wechselt, gibt der GameManager via return an die Game1 Klasse zurück, die dann das Spiel beendet.

Die Draw Methode (siehe Abb. 3.2) vom GameManager zeichnet zuerst, falls noch nicht geschehen, genau 1 Mal den alten GameState auf ein spezielles RenderTarget2D für den Überblendungseffekt. Anschließend wird der aktuelle GameState auf ein separates RenderTarget2D gezeichnet. Zuletzt werden diese beiden RenderTargets auf den Back Buffer der Grafikkarte gezeichnet.

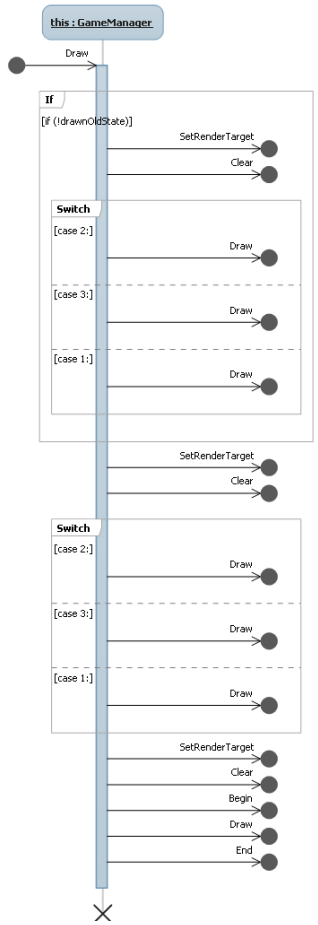

Abb. 3.2: Draw()[0]

3.2. IGameState

Die Schnittstelle IGameState definiert die Standard Schnittstelle für alle Game States. Sie sorgt dafür, dass alle Game States mit den entsprechenden Parametern versehen werden, und Methoden bereitstellen um sich selber zu initialisieren und zu entladen. Außerdem kann bei der Update Methode über den return Wert ein Game State Wechsel erzeugt werden.

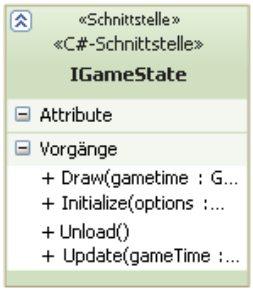

Abb. 3.3: IGameState

3.3. Game State: Game (SGame)

Der SGame Zustand des Spiels ist der primäre (dort läuft das eigentliche Spiel) Zustand des Spiels. Er beinhaltet die Schnittstelle IGameState. Er steuert alle Abläufe des Spiels z.B.:

- Spieler:
  - Bewegung
  - Kollision
- Gegner:
  - Wegfindung
  - Animation
- Level
- Layer
  - Hintergrundanimation bzw. Bewegung
- Spielevents
  - Tod
  - Sieg
  - Start

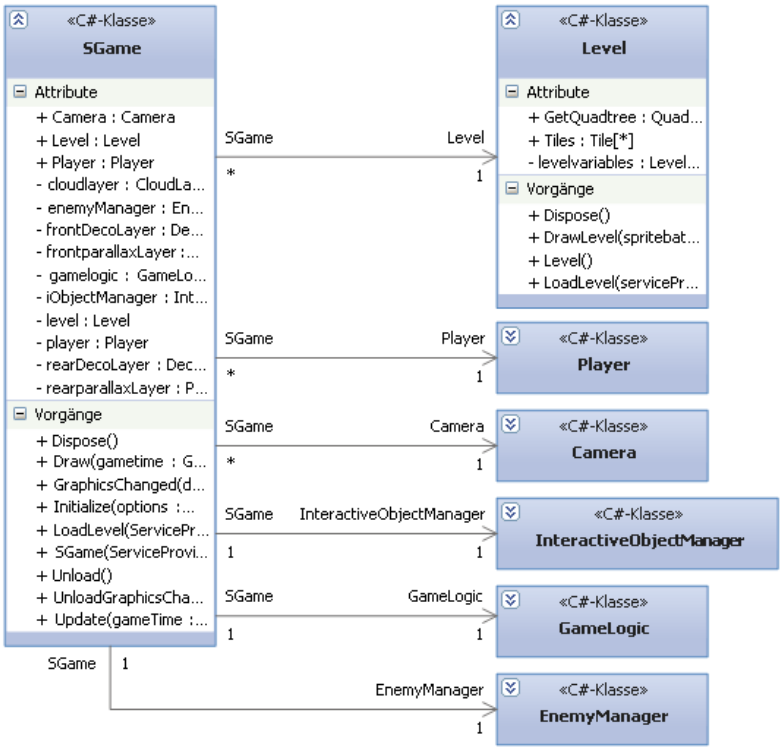

Abb. 3.4: SGame und die wichtigsten Member

3.3.1. Der Spieler

A. Allgemein

Eine zentrale Klasse des GameStates ist der Spieler bzw. Player. Diese Klasse stellt den Spieler, den der Benutzer steuert dar und behandelt alle Bewegungen, Animationen und Kollisionen.

B. Aufbau

Der Spieler besteht aus mehreren Rechtecken, die sein Körper, Füße, Hände und Kopf darstellen. Die einzelnen Teile sind in der Enumeration PPI (Player Piece Indicator) enthalten:

```
public enum PPI
{
    Hand2,
    Body,
    Head,
    Foot1,
    Foot2,
    Hand1,
}
```

[1]

Die Reihenfolge der Enumeration bestimmt außerdem auch die Reihenfolge, wie der Spieler gezeichnet wird. Daher sind die Hände ganz vorne und hinten, da sie vor und hinter dem Spieler gezeichnet werden sollen.

C. Ablauf

In der Update(....) Methode werden als Erstes die Kräfte, die auf den Spieler einwirken berechnet. Anschließend wird die Kollision mit dem Level mithilfe des Quadtrees (siehe 3.3.2.C. Quadtree) berechnet. Währenddessen wird auch die Animation des Spielers anhand der Bewegung des Spieler upgedatet. Schließlich gibt die Update Methode zurück welche Kollisionen vorliegen.

3.3.2. Das Level

A. Level Tiles

Das Level besteht grundsätzlich aus einem zweidimensionalen Array aus sogenannten Tiles (Englisch = Kacheln). Diese Tiles stellen jeweils ein Rechteck des Levels dar. Ein Tile besitzt einen Typ der angibt wie das Tile dargestellt werden soll (Ob Wall, Leer usw...). Weiterhin hat es eine Eigenschaft, die angibt wie sich dieses Tile in der Kollision verhalten soll. Für diese beiden Eigenschaften wird zum Einen die Enumeration TileType und zum Anderen die Enumeration TileCollision verwendet. Ein spezieller Fall stellt das Tile dar, wenn es vom Typ Wall ist, da es dann als Dekoration je nach GraphXPack (siehe 3.3.4. GraphXPack) noch Gras enthalten kann.

B. Level Variables

Die Level Variables enthalten hauptsächlich Information über die Darstellung des Levels z.B. welche Hintergründe und Verfärbungen das Level hat.

C. Quadtree

C.I. Aufbau

Der Quadtree teilt pro Ebene das Level in 4 Quadranten auf. Jeder so erzeugte Quadrant wird von der nächsten Ebene wieder in 4 Quadranten aufgeteilt. Dies wird so oft wiederholt (in diesem Fall 6 mal), bis man eine annehmbare Größe von Quadranten erhält. Der letzte Layer beinhaltet dann schlussendlich eine Liste von Tiles mit der man kollidieren kann.

C.II. Funktionsweise

Aber wie funktioniert nun so die Leistungsoptimierung? Der Zweck dahinter ist ganz einfach. Angenommen wir haben ein Level mit 100 Tiles in X-Richtung und 100 Tiles in die Y-Richtung. Ohne Quadtree müssten wir also mit $100\,Tiles * 100\,Tiles = 10000\,Tiles$ die Kollision pro Spieler und Gegner berechnen. Angenommen unser Level hat jetzt noch 30 Gegner. Dann ergibt sich daraus $100*100*(30+1)=310000$ Kollisionsberechnungen pro Bild.

Um diese gigantische Zahl drastisch zu verkleinern teilt man das Level in die oben genannten Quadranten auf.

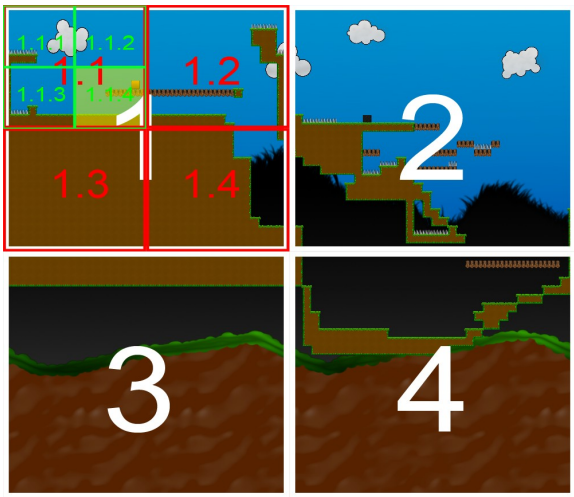

Abb. 3.5: Quadtree mit 3 Ebenen

Ein vereinfachtes Beispiel dazu findet sich auf Abb. 3.5. Mithilfe des Quadtrees wird zunächst die ungefähre Position der Spielerfigur oder des Gegners über die Quadranten bestimmt. Dabei geht man Layer pro Layer vor. Angenommen unser Spieler befindet sich in dem Gelb markierten Bereich. Beginnend mit dem ersten Layer (hier Weiß markiert) werden über eine einfache Rechteckskollision die vier Quadranten darauf überprüft, ob sich der Spieler ganz oder auch nur teilweise in ihnen befindet. Ist dies der Fall, wird dieser Quadrant zwischengespeichert. Als nächstes werden von allen Quadranten in denen sich der Spieler befindet (in unserem Fall nur der Quadrant 1) die Sub-Quadranten (1. X) genauer angeschaut. Dort werden ebenfalls die 4 Quadranten (hier 1.1-1.4) auf die Kollision mit dem Spieler oder Gegner überprüft. Dies wird solange wiederholt, bis man an der untersten Ebene des Quadtrees angelangt ist. Dem Einen oder Anderen wird inzwischen aufgefallen sein, dass wir ja jetzt schon eine ganze Stange von Kollisionsabfragen haben, ohne überhaupt mit der eigentlichen Kollision des Levels begonnen zu haben; aber im Vergleich zu der obigen Rechnung sind wir jetzt im optimalen Fall (und bei einer Quadtree Tiefe von 3) $4*3=12$ Kollisionsabfragen, was ja bis jetzt sehr gut zu verkraften ist. Da wir jetzt eine Liste von Quadranten der untersten Ebene haben, nehmen wir von jedem Quadranten, in dem wir uns befinden (im optimalen Fall einen) die Level Tiles und fügen sie alle zusammen in eine Liste. Wie groß ist aber jetzt so eine Liste? Pro untersten Quadranten sind bei einer Levelgröße von $X*Y$ und einer Quadtree Tiefe von n also

$Tiles\,pro\,Quadrant = (X*Y)/4^n$, in unserem Beispiel $(100*100)/4^3 \approx 156$ Tiles.
Zusammengefasst haben wir also pro Spieler oder Gegner $4*n+(X*Y)/4^n$ Kollisionsberechnungen pro Bild. Auf unseren Fall angewendet ca.
$4*3+(100*100)/4^3 \approx 168$ Kollisionsberechnungen pro Spieler oder Gegner pro Bild.

Bei 30 Gegnern: $(30+1)*(4*3+(100*100)/4^3)\approx 5208$ Kollisionsberechnungen pro Bild. Diese Optimierung spart also Kollisionsberechnungen um den Faktor 60! Bei einer Quadtree Tiefe von 5 sogar um den Faktor 335 ($(30+1)*(4*5+(100*100)/4^5)\approx 923$)!

Natürlich muss dieser Quadtree auch erst erstellt werden. Dies passiert sobald das Level geladen wurde. Dabei wird zunächst Ebene für Ebene erstellt. Der letzte Layer überprüft dann für jeden Quadranten, welche Tiles des Levels in diesem Quadranten liegen und fügt diese gegebenenfalls zu der Tile-Liste des Quadranten hinzu.

3.3.3. Die Interactive Objects

A. Allgemein

Interactive Objects (kurz IObjects) sind Objekte mit dem der Spieler interagieren kann (bisher ein „Sprungbrett“). Sie sind im Gegensatz zu den LevelTiles vollständige Klassen, die sich von der abstrakten Klasse InteractiveObject ableiten (siehe Abb. 3.6).
Alle IObjects zusammen werden von dem InteractiveObjectManager verwaltet, der diese initialisiert, updatet und zeichnet. Da der Spieler mit diesen Objekten interagieren kann, beinhaltet der Manager die GetPlayerInfluence() Methode, die alle Spielereinflüsse aller Objekte zusammenzählt und zurück gibt. Ein IObject stellt dafür die virtuelle Methode GetPlayerInfluence() bereit, die standardmäßig ein Vektor mit $\begin{pmatrix}0\\0\end{pmatrix}$ zurück gibt. Diese Methode kann von den abgeleiteten Klassen überschrieben werden, um ggf. den Spieler zu beeinflussen.

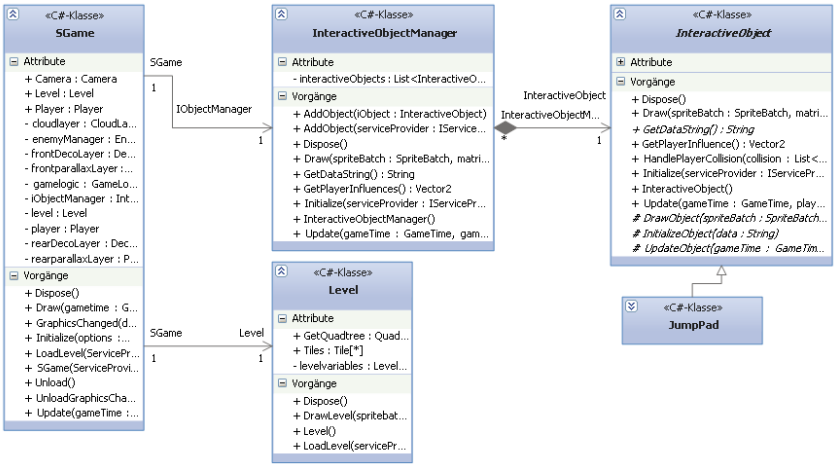

Abb. 3.6: InteractiveObjects und ihre Beziehungen zu SGame

3.3.4. GraphXPack

Ein GraphXPack stellt eine Umgebung in einem Level dar. Es ist eine Ordnerstruktur die bestimmte Leveltexturen bereitstellt. Das bisherige Standardpack hat den Namen „Grassy“ (Abb. 3.7) und stellt eine Super Mario ähnliche Gras-Wald-Landschaft dar. Ein Pack MUSS mindestens die Graphiken für die LevelTiles (Wall, Platform, Spike, usw...) bereitstellen, sowie eine Hintergrundgraphik (idR ein Verlauf). Optionale Grafiken sind Dekorationen und Parallax Layer (siehe D Parallax Layer) Hintergründe. Installierte GraphXPacks werden in der Datei installed.packs gespeichert.

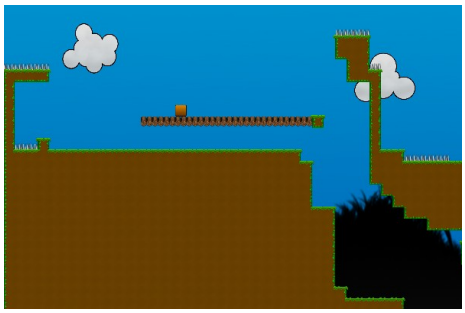

Abb. 3.7: GraphXPack „Grassy“

3.3.5. Layer

A. Allgemein

Layer stellen den Hintergrund und Umgebung von einem Level dar. Die Basis für alle Layer stellt die abstrakte Klasse Layer (Abb. 3.8) dar, die die Schnittstelle IGraphicsChange implementiert. Diese Klasse stellt grundsätzlich den Code für die Initialisierung eines Layers, sowie Fehlerbehandlung und Effekte bereit.

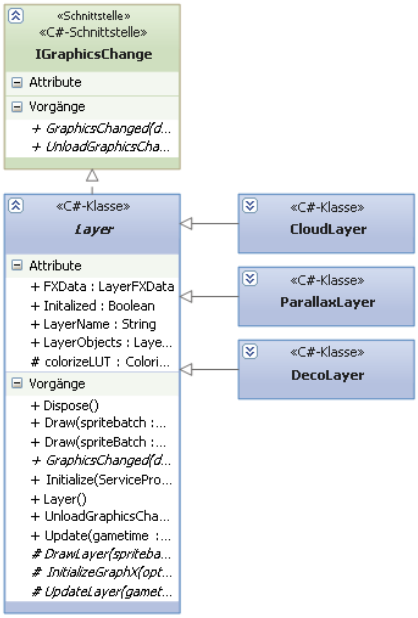

Abb. 3.8: Layer

B. Layer Object

Ein Layer Object ist ein spezielles Struct, dass von Layern für die entsprechenden Animationen und Darstellungen genutzt wird. Es beinhaltet zum Einen einen Verweis auf die zugehörige Textur und zum Anderen ein ExtendedRectangle was die Rotation um einen bestimmten Punkt, sowie eine rotationsunabhängige Verschiebung ermöglicht.

C. FXData

Jeder Layer kann durch die FXData eingefärbt werden. Einstellungen sind: Farbwert, Sättigung, Helligkeit und Kontrast.

D. Parallax Layer

Jedes Level besitzt zwei Parallax Layer, einen hinteren und vorderen. Ein Parallax Layer besteht aus 2 gleichen Texturen, die sich nacheinander endlos wiederholen. Dadurch entsteht der Effekt durch eine unendlich große Welt zu laufen, ohne riesige Texturen zu laden. Der hintere und vordere Parallax Layer sind unabhängig voneinander und können somit verschiedene Texturen und Geschwindigkeiten haben. Die Berechnungen der Verschiebung geschieht auf Basis der Kamerabewegung.

E. Cloud Layer

Der Cloud Layer stellt Wolken in den oberen 15% der Bildschirmfläche dar. Dazu hat ein Cloud Layer n viele LayerObjects, die er mit Hilfe von Sinus und Kosinus Funktionen animiert. Sobald sie das Level links verlassen, kommen sie rechts wieder herein.

F. Deco Layer

Um Dekorationen wie Büsche, Zäune usw... darzustellen wird der DecoLayer verwendet. Dieser enthält ebenfalls n viele LayerObjects, die sich anders als bei den vorherigen Layern an einer festen Position befinden und nicht verschoben werden. Das Level besitzt 2 Deco Layer, einen vor dem Level und einen dahinter. Das GraphXPack (siehe 3.3.4) stellt diese Dekorationen bereit. Die Dekorationen werden, da sie feste Positionen besitzen, anders als bei den anderen Layern, mit der Camera Matrix verschoben um immer an der richtigen Position im Level zu sein.

3.3.6. Camera

A. Allgemein

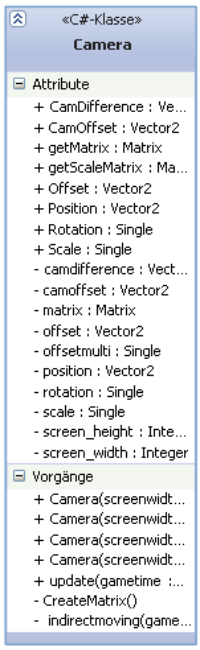

Abb. 3.9: Camera

Die Kamera (Klasse Camera, siehe Abb. 3.9) ist die Funktion des Spiels um die Darstellung des Levels zu verschieben, zu rotieren und zu skalieren. Die Verschiebung kann speziell direkt oder indirekt geschehen, wobei ersteres im Editor und letzteres im Spiel verwendet wird. Die indirekte Verschiebung sorgt dafür, dass die Kamera nicht direkt dem Spieler folgt, sondern sich nur verschiebt wenn der Spieler sich dem Rand nähert. Durch diesen Effekt wirkt das Spiel dynamischer, weil man an Rätseln oder schwierigen Stellen eher den Überblick behalten kann.

B. Funktionsweise

Die Camera besteht grundsätzlich auch einer 4*4 Matrix im „DirectX“ Format. Das „DirectX“ Format bedeutet, dass man um „korrekte, logische“ Matrixmultiplikationen zu erhalten Rechts-assoziativ rechnen muss. Diese besondere Regelung ist zu beachten, da gilt:
$Matrix1 * Matrix2 \neq Matrix2 * Matrix1$. Die Matrix der Camera wird folgendermaßen initialisiert:

```
private void CreateMatrix()
{
    matrix =
        Matrix.CreateTranslation(new Vector3(offset,0)) *
        Matrix.CreateTranslation(new Vector3(position.X, position.Y, 0)) *
        Matrix.CreateRotationZ(rotation) *
        Matrix.CreateScale(new Vector3(scale, scale, 0)) *
        Matrix.CreateTranslation(new Vector3(screen_width / 2, screen_height / 2, 0));
}
```

[2]
Zuerst wird das Zentrum des Bildschirms auf den Nullpunkt verschoben:

```
Matrix.CreateTranslation(new Vector3(screen_width / 2, screen_height / 2, 0))
```

Anschließend wird der Bildausschnitt um den entsprechenden Faktor skaliert:

```
Matrix.CreateScale(new Vector3(scale, scale, 0))
```

Als nächstes wird der Bildausschnitt gedreht und an die entsprechende Position verschoben:
Rotation:

```
Matrix.CreateRotationZ(rotation)
```

Verschiebung:

```
Matrix.CreateTranslation(new Vector3(position.X, position.Y, 0))
```

Der Offset wird hauptsächlich von Effekten genutzt. Dies hat den Vorteil, dass die eigentliche Position der Kamera nicht verloren geht:

```
Matrix.CreateTranslation(new Vector3(offset,0))
```

3.3.7. Game Logic

Die Logik des Spiels übernimmt die sogenannte Game Logic. Die zurzeit wichtigsten GameEvents sind das ExitGameEvent und das KillEvent. Die GameLogic Klasse verwaltet alle Game Events. Die abstrakte Game Event Klasse besitzt die abstrakte Methode CheckActivation(...) mit der ein Game Event überprüfen kann ob es aktiviert wird. Aktionen, die ausgeführt werden sollen, müssen über das return in der Update Methode in der Form der Enumeration EventAction zurückgegeben werden.

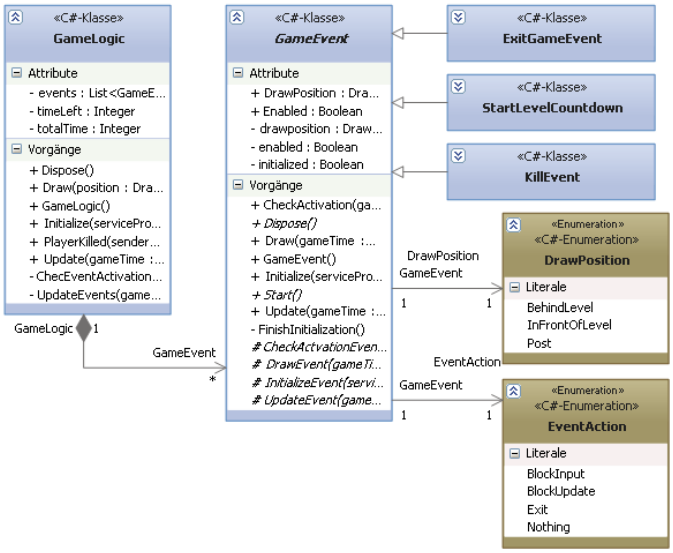

Abb. 3.10: Game Logic und Game Events

Da solche Events idR auch grafisch dargestellt werden, setzt die GameEvent Klasse die abstrakte Methode DrawEvent() voraus. Events können an verschiedenen Positionen gezeichnet werden. Diese Positionen werden in der Enumeration DrawPosition angegeben.

A. Kill Event

Das Kill Event behandelt alle Aktionen, die mit dem Tod des Spielers in Verbindung stehen. Zum Einen das Kollidieren mit den Spikes und zum Anderen das Kollidieren mit dem entsprechenden Rechteck des Gegners. Falls dies der Fall ist wird das Event aktiv und blockiert die Eingabe. Außerdem wird eine spezielle Ausblendeanimation gestartet. Wenn diese zu Ende ist, wird das Spiel in den Menu State zurückgesetzt.

B. Exit Game Event

Wie der Name schon sagt, behandelt dieses Event das Ende eines Levels. Dabei wird lediglich abgeprüft ob der Spieler mit dem Exit kollidiert. Wenn dies der Fall ist, wird das Level Schwarz ausgeblendet und das Spiel wird in den Menu State zurückgesetzt.

C. Start Level Countdown

Dieses Event tritt als erstes beim Levelstart ein. Es blockiert zunächst den hauptsächlichen Update des Spiels, bis ein Timer abgelaufen ist und gibt dann den Update des Spiels frei. Das Event wird danach in diesem Level nicht mehr Aktiv.

3.3.8. Enemy

A. Allgemein

Ein Gegner stellt einen feindlichen NPC (Non-Player Character) im Spiel dar. Seine Mission ist es, den Spieler daran zu hindern das Ziel zu erreichen.

B. Aufbau

Der Gegner besteht aus der Klasse Enemy, die die grundsätzlichen Aufgaben des Gegners übernehmen wie z.B. die künstliche Intelligenz. Der Gegner ist dabei abhängig (siehe Abb. 3.11) von einem Gegner Manager, von dem er die Grafiken zum Zeichnen erhält. Weiterhin verwaltet der Enemy Manager auch die Kollision mit dem Spieler und falls Gegner getötet werden, sorgt der Enemy Manager, dass diese dann auch korrekt aus dem Spiel entfernt werden.

Der Gegner selber besteht aus grafischer Sicht aus mehreren Rechtecken, die rotiert, skaliert, verschoben und gefärbt werden können. Neben den grafischen Elementen besitzt der Gegner auch ein Rechteck, das für die Kollision mit dem Level genutzt wird. Diese „Bounding Box“ ist unabhängig von jeder Animation.

Zur Wegfindung besitzt jeder Gegner ein Objekt vom Typ AStar (siehe 5. Projekt AStar)

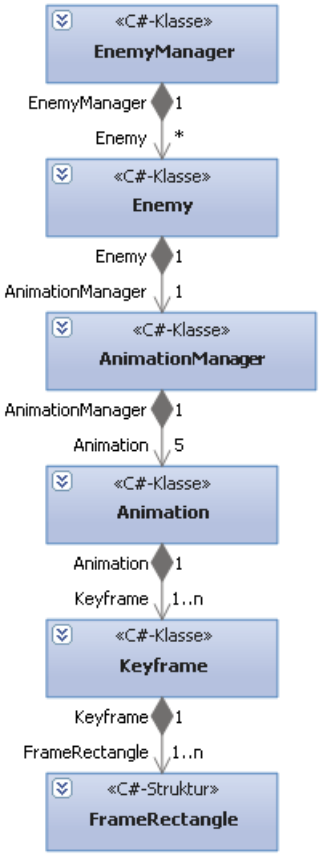

Abb. 3.11: Enemy Manager, Enemy und seine Animationen und ihre Abhängigkeiten

C. Enemy Manager

Der Enemy Manager (nicht zu verwechseln mit dem Single Enemy Manager aus dem Editor) verwaltet alle im Level befindlichen Gegner. Um die maximale Anzahl von Gegnern im Level zu erhöhen, werden die Gegner in mehreren Threads upgedatet.

Eine der Hauptaufgaben des Managers ist die Verwaltung der Gegner im Bezug auf Tod und Leben. Falls ein Gegner durch den Spieler getötet wird, wird dieser in einer speziellen Liste gespeichert. Der Manager setzt darauf hin den Gegner auf den Status tot. Da der Gegner nicht sofort entfernt werden soll, weil die Todesanimation noch abgespielt werden soll, wird diese Liste weiterhin upgedatet, aber nicht in die Kollisionsberechnung mit dem Spieler einbezogen. Nachdem die Todesanimation abgelaufen ist, wird der Gegner zum löschen freigegeben.

D. Animation

Wie man in Abb. 3.11 sehen kann, besitzt jeder Gegner einen AnimationManager. Dieser Manager verwaltet alle Animationen des Gegners. Ein Gegner hat zurzeit 5 verschiedene Animationen. Die Wichtigsten davon sind Walk und Jump. Jede Animation besitzt mehrere Keyframes, die nacheinander abgespielt werden. Ein Keyframe ist dabei ein grafischer Zustand des Gegners, der dargestellt werden kann. Die oben genannten Rechtecke sind in einem Keyframe als FrameRectangle abgespeichert. Jedes FrameRectangle beinhaltet Informationen über die relative Position des Rectangles zur Position des Gegners, Rotation, Rotationspunkt und Textur (=Name) des Rechteckes. Zugriff auf den aktuellen Keyframe ermöglicht der AnimationManager. Er speichert die aktuelle Animation und diese speichert den aktuellen Keyframe, so kann der Gegner ganz leicht auf seinen aktuellen Status zugreifen. Der Gegner enthält zusätzlich auch ähnlich wie der Spieler Code über die Verwaltung der Animation, bzw. welche Animation gerade abgespielt werden soll.

3.4. Game State: Menu (SMenu)

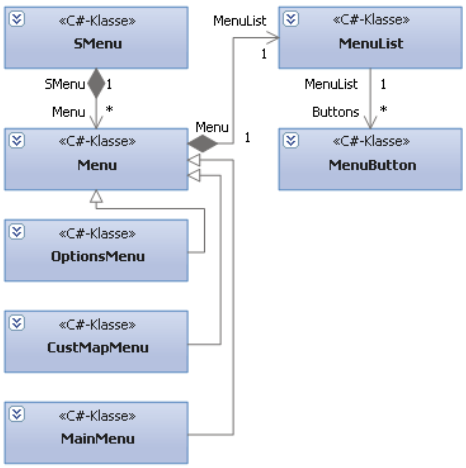

Abb. 3.12: SMenu und Menu

Das Menu dient dem Benutzer zur Einstellung und Levelauswahl des Spiels. Dazu ist der Menu State in mehrere Menüs aufgeteilt, wobei das Main Menu das primäre Menü darstellt.

3.4.1. Aufbau

Die Basis stellt die Klasse Menu dar. Diese stellt grundsätzliche Funktionen zur Navigation im Menu dar. Außerdem übernimmt sie standardmäßig die Zeichnung der Menülisten. Diese Basis Methoden sind virtuell und können, wenn benötigt, von den abgeleiteten Klassen überschrieben werden. Die Auswahl und Zeichnung des aktuellen Menüs wird in dem GameState als switch-case erreicht.

3.4.2. Main Menu

Das Hauptmenü stellt die Klasse MainMenu dar. Der Menu State beginnt immer mit dem Hauptmenü. Von diesem Menü aus kommt man in die Auswahl der eigenen Levels (CustMapMenu), dem GameState Credits und aus dem Spiel heraus (Exit).

3.4.3. Options Menu

Das Options Menu gibt dem Benutzer die Möglichkeit das Spiel auf seinen Computer anzupassen und die Eingabemethode auszuwählen. Außerdem kann man auch hier die Lautstärke der Musik einstellen. Um dies zu ermöglichen hat das Options Menu Zugriff auf die Options und kann diese entsprechend verändern.

3.4.4. CustMapMenu

Dieses Menü stellt die Auswahl der eigenen Levels zur Verfügung. Dabei werden die Levels aus dem Custom Levels Verzeichnis geladen und zur Liste hinzugefügt. Falls ein Level ausgewählt wird springt das Spiel in den GameState LoadingScreen, indem dann das Level geladen wird.

3.5. Game State: Loading Screen(SLoadingScreen)

3.5.1. Allgemein

Der Loading Screen State ist für das Laden des Levels zuständig. Er wird durch den MenuState initialisiert und nach ihm folgt entweder der GameState Game oder das Menü.

3.5.2. Ablauf

Zu Beginn des States wird ein neuer Thread gestartet, der dann in dem Objekt von SGame die LoadLevel() Methode aufruft. Dieser Thread ruft dann nach und nach Events auf, die den Fortschritt des Ladens aktualisieren. Durch die Nutzung eines Threads hängt das Spiel nicht während das Level geladen wird. Außerdem kann man direkt den aktuellen Fortschritt beobachten. Nachdem das Laden des Level abgeschlossen ist, wartet der Loading Screen auf den Druck einer beliebigen Taste. Sobald dieser erfolgt, wechselt das Spiel in den GameState Game.

3.6. Effekte

3.6.1. Allgemein

Ein zentrales Element in diesem Projekt stellen die Effekte dar. Sie sorgen für Abwechslung im Spiel und setzen einige grafische Akzente. Neben dem Partikel System gibt es eine ganze Reihe von Shader Effekten die in HLSL (High Level Shader Language) geschrieben wurden.

3.6.2. Aufbau

Als Basis dieser Shader Effekte dient die Klasse GraphicEffect, die die grundsätzliche Verwaltung von Effekten übernimmt wie z.B. das Laden und Zeichnen dieser Effekte. Da jeder Effekt andere Parameter hat, wird für jeden Effekt eine Klasse benötigt.

Gezeichnet werden die Effekte durch den SpriteBatch, der Effekte (als Klasse Effect) als Parameter akzeptiert. Falls mehrere Effekte auf die gleiche Ebene gezeichnet werden sollen, muss man einen ständigen Wechsel zwischen RenderTargets durchführen, auf die abwechselnd das Element gezeichnet wird, dass mit dem Effekt versehen werden soll. Dies ist notwendig, da die Textur eines RenderTargets nur verwendet werden kann, wenn er nicht das Ziel der Zeichnung ist.

4. Projekt StarEdit

4.1. Allgemeine Beschreibung

Das Projekt StarEdit beinhaltet den Editor von S.T.A.R.. Der Editor nutzt Windows Forms mit C# 4.0. Die Darstellung von XNA Inhalten wurde auf Basis von Microsoft Beispielcode [3][4] zur Implementierung von XNA in Windows Forms realisiert. Dabei wird ein Benutzersteuerelement erstellt (GraphicsDeviceControl, siehe Abb. 4.1), in dem wie gewohnt mit z.B. dem SpriteBatch gezeichnet werden kann. Dieser Code wurde von mir noch um einen Timer und abstrakte Methoden erweitert, so dass er im Ungefähren der von XNA bekannten Game Klasse entspricht. Diese Klasse initialisiert das GraphicsDevice.

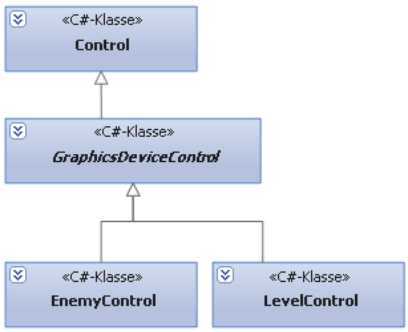

Abb. 4.1: GraphicsDeviceControl

4.2. Form StarEdit

Die Basis des ganzen Editors stellt die Form StarEdit dar. Durch sie kann man ein neues Level erstellen, ein Level laden, einen neuen Gegner erstellen und einen vorhandenen Gegner laden. Der Editor ist durch diese Form in einem MDI (Multiple Document Interface) System aufgebaut (siehe Abb. 4.2), d.h. alle weiteren Fenster befinden innerhalb dieser Form als „Dokument“.

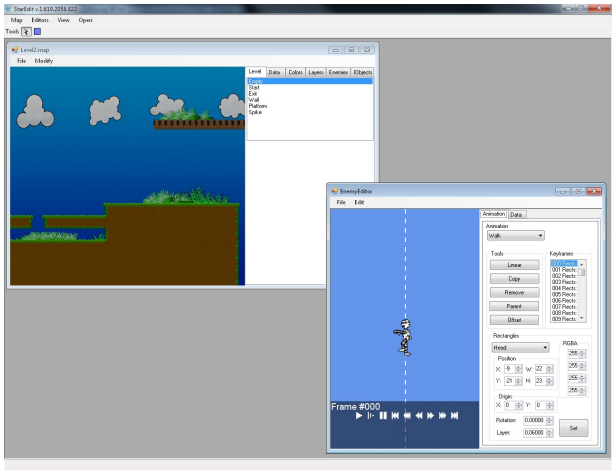

Abb. 4.2: StarEdit und ein Level und ein Gegner als Multiple Documents

4.3. Form Level Editor

4.3.1. Allgemein

Über diese Form wird dem Benutzer ermöglicht Levels zu erstellen und zu bearbeiten. In den linken zwei Drittel befindet sich das LevelControl, welches das Level darstellt. Im rechten Drittel befinden sich Steuerelemente zur Auswahl und Einstellung des Levels.

4.3.2. Das LevelControl

Das LevelControl leitet sich von der oben genannten GraphicsDeviceControl Klasse ab und stellt die primäre Arbeitsfläche dar. Dieses Benutzersteuerelement arbeitet wie die Klasse SGame. Es beinhaltet somit ein Level, alle Layer, sowie InteractiveObjectManager und EnemyManager. Der grundsätzliche Unterschied besteht darin, dass kein Spieler enthalten ist, keine Spiellogik und die Gegner nicht upgedatet werden. Mit der Maus platziert man hier die ausgewählten Levelelemente. Mit Rechtsklick kann man die Kamera verschieben und mit dem Mausrad Zoomen. Die Verschiebung und Skalierung wird auch hier durch eine Matrix realisiert. Zur Erkennung von MouseEvents werden die entsprechenden Events überschrieben wie z.B. OnMouseClick().

4.3.3. Die Steuerelemente

Die Steuerelemente auf dem rechten Drittel des Level Editors dienen zur Auswahl der Komponenten, die man in dem Level platzieren, ändern oder löschen will. Zur Übersichtlichkeit sind die Tools in mehrere Kategorien unterteilt. Angefangen mit der Auswahl der Tiles, die man platzieren kann. Sobald man in einen anderen Tab wechselt wird eine Methode in dem LevelControl aufgerufen, dass das aktuelle Tool wechselt. Parameter des aktuellen Tools werden dann anschließend beim Anklicken oder Auswählen andere Elemente übergeben.

4.4. Form Enemy Editor

4.4.1. Allgemein

Mit dem Enemy Editor kann man Gegner erstellen und vorhandene ändern. In der aktuellen Version ist es nur möglich Animationen und die Kollisionsrechtecke zu bearbeiten.

4.4.2. Enemy Control

Das Enemy Control arbeitet ähnlich wie das Level Control, dabei simuliert es einen EnemyManager. Da hier immer nur ein Gegner gezeichnet werden soll und der Gegner von einem EnemyManager abhängig ist, beinhaltet das EnemyControl einen speziellen SingleEnemyManager. Dieser stellt genau einen Gegner dar. Zur Steuerung der Wiedergabe der Animation dient die Klasse PlayControl. Innerhalb des EnemyControl, kann man mithilfe der Maus die Rechtecke des Gegners verschieben und rotieren.

4.4.3. Steuerelemente

Genau wie bei dem Level Editor dienen auch hier die Steuerelemente im rechten Drittel des Editors zur Verwaltung der Eigenschaften der Tools. Die hauptsächliche Kategorie hier ist die Animation, die Eigenschaften über die aktuelle Animation, Auswahl von keyframeübergreifenden Tools ermöglicht und Informationen über den aktuellen Keyframe und das aktuelle Rechteck darstellt.

4.4.4. Tools

Der Enemy Editor stellt einige keyframeübergreifende Tools für die Animationen zur Verfügung. Das mitunter wichtigste Tool ist wahrscheinlich das Linear Tool mit dem Linear Interpoliert werden kann. Dieses Tool kann allerdings auch zum Setzen von Standardwerten benützt werden. Ein weiteres wichtiges Tool ist das Parent Tool mit dem man Bewegungen von übergeordneten Rechtecken übernehmen und auf Wunsch die Rotation dabei beachten kann. Dieses Tool ist besonders bei der Animation von Extremitäten des Gegners hilfreich. So kann z.B. der Unterarm mit dem Oberarm zusammen bewegt werden.

5. Projekt AStar

5.1. Allgemein

Das Projekt AStar definiert den A* (gesprochen A-Star) Algorithmus, der häufig in Strategiespielen zur Wegfindung benutzt wird. Dieser Algorithmus vereint die Vorteile des Dijkstra-Algorithmus und dem Greedy-Algorithmus, d.h. er berechnet die Wegkosten aus der Luftlinie zum Ziel als Heuristik und die Wegkosten anhand der bisherigen Wegpunkte. Weitere Informationen zur Vorgehensweise des A*-Algorithmus findet sich auf der Wikipedia Seite dazu. [5][6]

5.2. Aufbau

5.2.1. Map

Die AStarMap Klasse wird von allen Instanzen der AStar Klasse geteilt. Sie beinhaltet die aktuelle berechnete Wegfindungs-Map des Levels. Sie wird beim Laden des Levels initialisiert und steht ab dann zur Verfügung.

Diese Map besteht aus Nodes, die das Rechteck, die X und Y Position im Level und die A* Definition Walkable beinhalten:

```
public enum Walkable
{
	Walkable,
	Blocked,
	Platform,
	NotReachable
}
```

[7]

Die Enumeration Walkable definiert, wie das Rechteck in der Wegfindung genutzt werden kann. Ein Beispiel davon sieht man auf Abb. 5.1. Dort werden begehbare Felder Weiß dargestellt, blockierte Felder Rot, Platformen Grün und NotReachable blau. Als Anpassung an ein Jump 'n' Run habe ich hier noch die speziellen States Platform und NotReachable definiert:

Abb. 5.1: A Map Rechtecke entsprechend ihres States farbig markiert*

Platform definiert einfach, dass diese Rechteck von links, unten und rechts durchlaufen bzw. durchsprungen werden können und man von oben darauf stehen kann.

NotReachable zeigt, dass dieses Rechteck auf normalem Weg nicht erreicht werden kann (ein Gegner kann keine 5 Felder hochspringen). Allerdings kann er durch sie von oben durchfallen.

5.2.2. A* Wegfindung

Die eigentliche Wegfindung findet in der Klasse AStar statt. Da die Wegfindung sehr aufwändig ist, läuft sie in einem eigenen Thread. In diesem Thread läuft dann der oben beschriebene A*-Algorithmus ab. Das Ergebnis wird über ein Event zurückgegeben. Als Parameter des Events wird auch die Zeit übergeben, die gleichzeitig auch als Fehlerübergabe dient. Falls kein Weg gefunden wurde, wird als Zeit der Wert -2 zurückgegeben. Falls kein Weg gefunden werden kann, da der Start oder das Ziel nicht begehbar ist, wird -1 zurückgegeben. Wird einer der Fehlermeldungen zurückgegeben, wird nach kurzer Zeit, damit der Computern nicht vollständig ausgelastet wird, eine neue Wegfindung begonnen. Bei erfolgreicher Wegfindung wird versucht, dem Pfad zu folgen und falls der Gegner ihn verloren hat, wird ein neuer Weg gesucht.

6. Projekt XNAParticleSystem

6.1. Allgemein

Das XNAParticleSystem ist ein effizientes Partikel System, das auf XNA basiert. Es wurde mehrfach optimiert und kann auf aktuellen Computersystem bis zu 250'000 Partikel darstellen. Die primäre Klasse ist der ParticleManager, der die Verwaltung des ganzen Systems übernimmt. Aus Leistungsgründen bestehen die Partikel lediglich aus mehreren Arrays, die sich innerhalb des ParticleManagers befinden. Die Initialisierung ist anfänglich etwas komplex, da das Partikel System sehr dynamisch ist und viele Parameter berücksichtigt. So können z.B. mehrere verschiedene Gravitationsarten auf die Partikel einwirken. Weiterhin kann ein Partikel System eine dauerhafte Fontäne oder eine schlagartige Explosion darstellen. Unabhängig davon kann man auch die Startgeschwindigkeit einstellen, die sich in einem bestimmtem Rahmen befinden kann. Parameter:

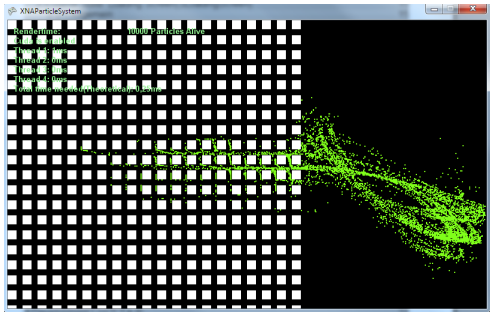

Abb. 6.1: XNAParticleSystem

- Startrichtung
  - So wie Geschwindigkeit
  - In beide X – Richtungen
  - In beide Y – Richtungen
  - In alle Richtungen
  - Innerhalb eines bestimmten Radius eines Winkels
- Gravitationsarten:
  - Schwerkraft in eine bestimmte Richtung
  - Newtonsche Gravitation
  - Abgewandelte Punktgravitation
- Spawn-Art
  - Fontäne
    - Zeit am Leben
  - Explosion
- Spawn-Position
  - Punkt
  - Rechteck
- Geschwindigkeit
  - Mindestgeschwindigkeit
- Kollisionsart
  - Kollision
  - Keine Kollision
- Alphawert
  - Auf Zeitbasis
  - Fest
- Kollisionsreibung
- Luftreibung
- Größe
- Textur

7. Projekt CUDAParticleEngine

7.1. Allgemein

Die CUDAParticleEngine stellt eine Spezialisierung des XNAParticleSystems dar. Es nutzt das bei nVidia vorhanden CUDA (Compute Unified Device Architecture)[8], welches in die Kategorie GPGPU[9] fällt. Es beinhaltet die gleiche Logik wie das XNAParticleSystem, jedoch ist es in C/C++/“C for CUDA“ geschrieben. Der massive Vorteil darin liegt, dass viele Aufgaben gleicher Art gleichzeitig ausgeführt werden können, was einen enormen Leistungsschub ermöglicht.

7.2. Interop-Services

Um diesen Code in C# zugänglich zu machen, wird beim Erstellen diese Projektes eine Programmbibliothek erstellt, die dann über InteropServices in C# implementiert werden kann. Diese Implementierung übernimmt die Klasse ParticleCuda. Um die gängigen Typen aus XNA in „C for CUDA“ zugänglich zu machen, habe ich die particleSystemTypes.h und xnaTypes.h Header-Dateien geschrieben, die die Typen deklariert und die Operanden wie +, +=,* usw... definiert.

8. Tätigkeitsnachweis

- Juli 2010
  - 15.

  Erste Idee zu Jump & Run Spiel
  - 20-23.
    - Programmierung der ersten Spielgrundlagen
      - Entwurf und Aufbau der Spielfigur (Player Klasse)
      - Animation der Spielfigur
      - Anfang eines Tile-Based-Map Aufbaus (Level Klasse)
      - Steuerung über Xbox 360 Wireless Controller
  - 24.-26.
    - Erste Programmierung der Spielphysik (Collision)
      - Kollision mit Boden, Wand, Decke (über Sprung)
      - Sehr fehlerhaft (kann durch manche Wände Laufen, fällt durch Boden)
  - 26.-30.
    - Überarbeitung der Kollision
      - Behebung der Fehler
      - Verbesserung des Sprunges
    - Allgemeine Spielsteuerung
      - Keyboardunterstützung eingefügt
      - Allgemeine Spielsteuerung (Inputhandler verwaltet Eingabe)
- August 2010
  - 01.-08
    - Erweiterung der Tiles (Plattform(-Kollision))
    - Allgemeine Fehlerbehebung
    - Verbesserung der Performance (Behebung von massivem Ruckeln bei großen Maps)

- 09.
  - Einbau eines Gamestates
    - GameManager Klasse verwaltet den Spielstatus
      - Intro
      - Menü
      - InGame
      - ….
    - Interface für Gamestates (IGameState)
    - Erster Entwurf eines Einfachen Menüs
- 10.-12.
  - Erweiterung des Menüs
    - Abstrakte Klasse für Allgemeines Menü
- 12.-15.
  - Laden von CustomMaps(selbst erstelle Maps) über Menü möglich
- 15.-20.
  - Einbau von Hintergrund Layern
    - Abstrakte Klasse Layer in Interface Ilayer
    - CloudLayer zur Wolkendarstellung
- 24.
  - Erstellung von Parallax Layer (unendlich langer, wiederholender Hintergrund)
  - Reine Arbeitszeit bisher: 2d 1h
- 29.
  - Optimierung der Kollisionsperformance durch

  Erstellung von einem Quadtree (Aufteilung der Map in gleich große Rectangles)
    - Erstellung der Quadtree und QuadTreeNode Klasse
  - Verbesserung der Parallax Layer
    - Lücken zwischen Texturen geschlossen
    - Ruckelhafte Bewegung behoben

- September 2010
  - 01.
    - Optimierung des QuadTree
      - Möglichkeit gleichzeitig in mehreren Rechtecken zu sein
        - behebt durchfallen bei Rechteckwechsel
  - 05.
    - Einbau von sog. GraphXPacks
      - Komplette Grafikpakete für Umgebung (z.B. Wiesenlandschaft, Höhle)
    - Weitere Animierung der CloudLayer
    - Kamera Offset → Spieler ist nun im unteren Bereich des Spielfenster, egal welche Größe dieses Besitzt.
    - Bisherige Zeit: 2d 9h
  - 07.
    - Anfang der Editorentwicklung
      - Erlernung von XNA als Windows Forms Control
      - Implementierung von XNA in Windows Forms
      - Verlinkung der Projekte STAR und StarEdit durch Verweise
      - Bau der StarWindow Form und des Controls LevelControl.
      - Einfache Maussteuerung des geladenen Levels (Bewegung,Zoom)
      - Dateien GraphicsDeviceControlMicrosoft.cs, GraphicsDeviceService.cs und ServiceContainer.cs in Projekt StarEdit stammen von Microsoft
        - http://creators.xna.com/en-US/sample/winforms_series1
      - Modifizierung von GraphicsDeviceControl für eigene Zwecke
        - Einfügen und Steuern einer GameTime
        - Eigene Methoden befinden sich in der partial class GraphicsDeviceControl in der Datei GraphicsDeviceControlMarvinPohl.cs
      - Laden von Levels in Editor möglich
    - Bisherige reine Arbeitszeit 2d 14h

- 10.
  - Einbau einer Spiellogik (GameLogic) Klasse
    - Verwaltung von Ereignissen während des Spiels
      - Tod
      - Ziel erreicht
    - Möglichkeit einzelner GameEvents z.B. ExitGameEvent (GameEvents werden immer von der GameEvents klasse abgeleitet.
  - PixelShader Effekte
    - Regeneffekt
    - MotionBlur
  - Bisher reine Arbeitszeit 2D 19h
- 11.-17.
  - Anfang von Erstellung der Enemy Klasse (Gegner)
    - Komplexer Aufbau durch Dynamische Gegnermodelle (werden aus Dateien geladen)
    - Kollision
  - Anfang von Erstellung des Enemy Editors
    - EnemyEditorForm
    - EnemyControl.cs
    - Grafiken können mithilfe des MSBuild von .PNG in .xnb live konvertiert werden. Dafür wurden die Klassen ContentBuilder.cs und ErrorLogger.cs und app.config aus dem Beispiel Code WinFormsContentLoading benutzt:
      - http://creators.xna.com/en-US/sample/winforms_series2
  - Generische Funktionen die spezielle Dateien laden
    - Schreiben oder lesen Listen im Format: „Key=Value;“ aus/in Dateien wobei Key in der Enumeration ist → Generisch.
    - public static class FileManager
      - public static Dictionary<T,string> GetFileDict<T>(string filepath)
      - public static void EnumDictToFile<T>(Dictionary<T,string>,string filename)
  - Bisher reine Arbeitszeit 3D 6h

- 18.-20.
  - Erweiterung des Editors zu MDI (Multi Document Interface)
  - weitere Pixelshader (KillEffect, GraphicEffect,Wave)
  - KillEvent und Spike Tile (Spieler stirbt wenn er Spikes berührt)
  - 83 Mann stunden
- 23.-24.
  - Erweiterung des Editors
    - Erstellung beliebig großer Level möglich
    - Einrahmung des veränderbaren Feldes
- 26.-27.
  - Erweiterung des Editors
    - Verändern des Hintergrunds Möglich
  - Verbesserung der Levelspeicherung
    - TileType wird jetzt als Zahl und nicht als String gespeichert

- Oktober 2010
  - 01.-03.
    - Erweiterung des Editors
      - Rechteckstool
  - 04.-06.
    - Verbesserung der InGame Kamera
      - scrollt nichtmehr linear hinter dem spiele hinterher
    - Bau einer ersten PhysikEngine/Partikelsystem
      - Partikel erkennen Kollision, haben Schwerkraft,Reibung, Lebenszeit, Textur
  - 08.-10.
    - Verbesserung/Erweiterung des Partikelsystems
      - Performanceverbesserung (Multithreading; ThreadSafe Content)
      - Erweiterung der physikalischen Möglichkeiten
  - 12.-14.
    - Grundklassen der Gegneranimation
    - Einbau der Gegneranimation in Editor

- 15.-31.
  - Erweiterung des Gegner Editors
    - Erstellung von neuen Gegnern, Laden von existierenden Gegnern
- November
  - 11.
    - Einbindung der Gegner in Spiel
    - Erweiterung des Gegner Editors
    - Reine Arbeitszeit 5d 14h (134h)

  - 15.-23.
    - Kollision des Gegners
      - Interaktion mit Spieler
    - Korrekte Spiegelung des Gegners bei Richtungswechsel
- Dezember
  - 5.
    - Platzierung eines Gegners im Editor Möglich,
    - Speichern und Laden von Gegnern im Level möglich
  - 10.
    - Erstellung und Testen eines Kontrastpixelshaders
  - 12.
    - Einbau eines Ladebildschirms für das Laden der Level
      - Anzeige des aktuelle geladenen
      - Level wird in extra Thread geladen
  - 20.-31.
    - Einbau von einem Färbungspixelshader
      - Optimierung mithilfe der Nvidia Shader Library auf Pixel Shader 2.0
      - Färbung und Sättigung separat möglich.
- 2011 Januar
  - 8.
    - Erweiterung des Färbungspixelshaders um Helligkeit und Kontrast
  - 9.
    - Speichern und Laden des Färbungspixelshaders
    - Einbau des Pixelshaders ins Spiel

        - Bisherige Zeit: 6 Tage 7 h
    - 10.
        - Programmierung des A* Algorithmus nach Beschreibung der Funktionalität von Wikipedia
            - Benötigte Zeit: 8h
    - 14. - 15.
        - Erste Einbindung des A* Algorithmus und Implementierung in Gegner Klasse
    - 16.
        - Erweiterung der Gegner KI
            - Gegner verfolgt Spieler
            - Gegner springt Spieler hinterher
        - Bisherige Zeit 6d 19h
    - 21.-23.
        - Diverse Fehlerbehebungen im Enemy Editor
        - Erstellung von Mummy - Gegner Grafiken
    - 27.
        - Erste Animierung des Mummy Gegners
        - Erweiterung der Enemy Editor funktionalität.
- Februar
    - 10.
        - Umstrukturierung des Menüs
            - Einbau einer Übergangsanimation
            - Generalisierung der Menu Klasse
            - Hauptmenü als eigene Klasse
        - Behebung eines Deadlocks durch debugForm
        - Weitere Todesanimation bei Gegnern (Welleneffekt)
    - 18.
        - Umstrukturierung des Editors für geplante Erweiterung um DecoLayer Editierbarkeit
    - 19.
        - Erster Einbau des DecoLayers in Editor

- 23.-24.
  - Grobe Erlernung von CUDA
    - Definition CUDA siehe: http://de.wikipedia.org/wiki/Compute_Unified_Device_Architecture
    - Erlernung der Funktionen:
      - http://developer.nvidia.com/object/cuda_training.html
      - Sample Code aus CUDA Toolkit 3.2 Beispielen
        - http://developer.nvidia.com/object/cuda_3_2_downloads.html
  - Erstellung einer DLL für die Particle Engine in CUDA
    - Grundsätzlicher interop Code aus
      - http://www.andyhcoates.com/Blog/post/How-to-use-CUDA-from-XNA.aspx
      - http://www.andyhcoates.com/Blog/post/blah.aspx
- 25.-26.
  - Portierung des vorhandenen Partikelcodes von C# auf „C for CUDA“ / C / C++
  - Einbindung der DLL in C#
    - Stundenlange versuche der Fehlerbehebung in DLL (keine – spärliche Debugmöglichkeit)
      - Kleine Hilfe durch NVIDIA Compute Visual Profiler (Programmanalyse während der Ausführung) → Anzeige von Fehlermeldungen bei Funktionsaufrufen innerhalb des C Codes (nicht in CUDA Code)
    - Erste Lauffähige Version der Engine auf CUDA-Basis

- März 2011
  - 7.
    - Erweiterung der Cuda Enginge
      - Mehrere Cuda Particle Engines gleichzeitig möglich
        - Erstellung eines structs der Alle relevanten Pointer enthält
        - Speicherung des structs in C# als IntPtr
      - Behebung des nichtfunktionierends der Fontänen Funktion
  - 11.
    - Erweiterung des Editors
      - Anzeige und löschen von gesetzten Dekorationen möglich
      - DecoLayer besitzen jetzt gleiche FXData wie Level.

- 15.
  - Erweiterung des Editors
    - Größe und Rotation und Angelpunkt für Rotation sind jetzt für Dekorationen definierbar.
- 26.-29.
  - Erstellung des Trace-Effekts (Simple Motion Blur)
  - Einbindung des Trace-Effekts als Kill Effekt
  - Abänderung des Kill Events
    - Einbau des Trace Effekts
    - Ausblendung zu Schwarz und Einblendung von „GameOver“

- April 2011
  - 3.-6.
    - Erweiterung des Gegner Editors
      - Rechtecke können jetzt gelöscht werden
      - Workflow verbessert
      - Parent Tool verbessert
  - 7.-9.
    - Versuche Gegner zu spiegeln.
      - Sehr problematisch durch viele rotierte Rechtecke.
  - 11.
    - Testen des Editors auf fremd PC's (Nils Baur)
    - Anfrage an Nils Baur für die Erstellung von Gegnern.
  - 15.
    - Erfolgreiche Spiegelung von Gegner
      - Zeichnen des kompletten Gegners auf extra RenderTarget
      - Spiegelung des kompletten Targets20.
    - Erstellung der InteractiveObjects Klasse als Oberklasse für Objekte, die den Spieler direkt Beeinflussen
    - Erstellung eines JumpPads als InteractiveObject
      - Gegner fliegt bei Berührung in gewünschte Richtung
      - Partikel als Effekte
  - 29.
    - Erweiterung des JumpPads
    - Anpassung der Steuerung

- Mai 2011
  - 01.
    - Erstellung der InteractiveObjectManager Klasse
  - 06.
    - Erklärung des Editors für Nils Baur zur Gegnererstellung
    - Erstellung des Plakats für Sonntag 8.Mai
  - 07.
    - InteraktiveObjekte (bisjetzt JumpPad) im Editor einfügbar
    - Speichern und laden von Interaktiven Objekten möglich.
  - 10.-12.
    - Erstellung eines neuen Spielers (Grafik)
    - Leistungsoptimierungen der JumpPads
  - 13.
    - Erklärung an Herrn Heidenreich, wie C/C++ Programmbibliotheken (DLLs) zu erstellen und in C# zu implementieren sind.
  - 15.
    - Generalisierung der Eingabe (Interface IgameController)
      - Vorbereitungen auf konfigurierbare Eingabe via Options-Menu
    - Bisherige reine Arbeitszeit: 243h (10d 3h)
  - 28.-30.
    - Umschreiben auf XNA 4.0
      - Einbüßung mancher Effekte (Behebung folgt)
      - Umschreiben der CUDAParticleEngine auf VS 2010.

- Juni 2011
  - 01.
    - Erstellung des ToonShader Effekts mit Abwandlungen auf Basis von http://digitalerr0r.wordpress.com/2009/03/22/xna-shader-programming-tutorial-7-toon-shading/
  - 05.
    - Erstellung von Skalierungseffekt zwischen GameStates
    - Behebung eines Bugs an dem Wellen Effekt
  - 12.-13.
    - Implementierung von Musik, welche unter Creative-Commons 3.0 BY-NC-SA stehen.
    - Löschen von Gegnern im Editor jetzt möglich.
    - Behebung einiger Fehler im Editor, die dazu führten, dass sich der Prozess StarEdit.exe nicht schloss (jetzt korrekte Zerstörung der laufenden Threads und keine Verwaisung von Threads).
  - 18.
    - Einbau des Game States Credits
  - 19.
    - Verbesserung der Menu GUI
      - Custom Levels Liste scrollt jetzt.
      - Menukeys werden jetzt nach einer bestimmten Zeit, die sich je länger man drückt verringert, automatisch erneut gedrückt.
    - Verbesserung einiger Lags, die auf leistungsschwächeren Computern (Laptop) aufgetreten sind

9. Literaturverzeichnis

0: Pohl, Marvin, GameManager.cs , Zeilen: 183-253 , 12-06-2011
1: Pohl,Marvin, Player.cs , Zeilen: 14-22 , 17-06-2011
2: Pohl, Marvin, Camera.cs , Zeilen: 82-90 , 17-06-2011
3: Microsoft , http://create.msdn.com/en-US/education/catalog/sample/winforms_series_1 , 19-06-2011
4: Microsoft , http://create.msdn.com/en-US/education/catalog/sample/winforms_series_2 , 19-06-2011
5: Wikipedia , http://de.wikipedia.org/wiki/A*-Algorithmus , 22-06-2011
6: Wikipedia , http://en.wikipedia.org/wiki/A*_search_algorithm , 22-06-2011
7: Pohl, Marvin, Enums.cs , Zeilen: 8-14 , 22-06-2011
8: Wikipedia , http://de.wikipedia.org/wiki/Compute_Unified_Device_Architecture , 23-06-2011
9: Wikipedia , http://de.wikipedia.org/wiki/GPGPU , 23-06-2011