

# Implementierung einer Komponenten basierten Game-Engine unter Verwendung des XNA Frameworks (4.0)

Independent Coursework, Christoph Kapffer

## Inhalt

Einleitung.....	1
Architektur.....	2
Entity .....	2
Component.....	2
Scene .....	3
System .....	3
Implementierung.....	3
Database.....	3
Threading.....	4
Serialisierung .....	4
Schwachstellen und Verbesserungsvorschläge.....	5
Notification events .....	5
Scene loading.....	6
Performance-Einbrüche .....	6
Multiplattform .....	6
Fazit .....	7

## Einleitung

Die Arbeit befasst sich mit der Implementierung einer Komponenten-basierten Game-Engine unter Verwendung des XNA Frameworks. Die grundlegenden Architektonischen Konzepte wurden aus dem Artikel [Entity Systems are the future of MMOG development](#) (geschrieben von Adam Martin) abgeleitet. Um ein generelles Verständnis über die hier dargestellten Themen zu bekommen, ist es äußerst empfehlenswert sich zunächst den oben genannten Artikel komplett durchzulesen.

## Architektur

Die wichtigsten Elemente der Engine sind Entity, Component, Scene und System.

### Entity

Eine Entity repräsentiert ein Gameobjekt. Um unterschiedliche Gameobjekte zu erzeugen, werden allerdings keine Unterklassen mit verschiedenen Spezialisierungen von Entity gebildet, wie im herkömmlichen OOP-Stiel. Stattdessen wird jedes Gameobjekt lediglich durch diese eine Entity-Klasse repräsentiert. Entity wird also nie abgeleitet.

Um dennoch verschieden geartete Gameobjekte zu erhalten wird eine Entity mit einer Reihe von Komponenten (Components) dekoriert. Durch Kombination unterschiedlicher Komponenten wird die Erscheinung und das Verhalten einer Entity geprägt. Seine Konfiguration wird also über die mit ihm assoziierten Komponenten bestimmt. Auf diese Art und Weise lassen sich auch unterschiedliche Typen von Gameobjekten erzeugen.

Der naive Implementierungsansatz würde vorsehen, eine Entity als Array oder Liste von Komponenten zu implementieren. Stattdessen werden die Komponenten zentral in einer Database verwaltet. Das hat den Vorteil, dass, wenn man z.B. eine Operation auf alle Komponenten eines bestimmten Typs ausführen möchte, nicht erst mal über alle Entities iterieren muss, um zu gucken ob sie eine Komponente dieses Typs besitzen und wenn ja dann erst die Operation darauf ausführen kann. Die Zuordnung der Komponenten zu den jeweiligen Entities wird allein über die Datenstruktur gesteuert. Näheres hierzu unter [Implementierung](#).

### Component

Eine Komponente ist eine kleine, atomare Einheit, die nichts weiter besitzt als eine geringe Anzahl an Properties. Zum Beispiel besitzt die Komponente „Transform“ lediglich die drei Eigenschaften, Position, Orientierung und Skalierung. Die Komponente „Movement“ besitzt die Eigenschaften lineare und angulare Geschwindigkeit, und linearer und angularer Impuls.

Eine Komponente enthält keinerlei Methoden zur Verarbeitung oder Datenmanipulation (Transform bildet hier eine kleine Ausnahme, wobei auch dessen Methoden sich nur auf mathematische Umformungen beschränken). Stattdessen wird die Handhabung der Komponenten in sogenannten [Systemen](#) durchgeführt. Desweiteren dienen Komponenten auch als Label, in dem Sinne, dass sie verraten: Wenn eine Entity die Komponente Transform besitzt, hat sie folglich eine Position im dreidimensionalen Raum, so wie eine Orientierung und eine Größe. Wenn dem so ist, und es sich bei der Entity um einen Gegenstand handelt, der auch sichtbar ist und gezeichnet werden soll, hat die Entity wahrscheinlich auch die Komponenten Mesh und Appearance.

Wichtig ist außerdem, dass man Komponenten so klein wie möglich hält, um nicht Informationen doppelt zu speichern. Dabei können nämlich auch Probleme bei der Synchronisierung der gleichen Daten entstehen, die in zwei unterschiedlichen Komponenten vorhanden sind. Sobald man dieses Phänomen in der eigenen Implementierung entdeckt ist das ein Zeichen dafür, dass die doppelt vorhandenen Daten der zwei Komponenten in eine dritte extrahiert werden müssen. So hatte eine erste Implementierung die Komponenten Mesh und Appearance vereint, da ja beide zur Anzeige benötigt werden. Da aber auch die Physiksimulation Informationen über das Mesh braucht um die Kollisionsberechnungen durchzuführen, musste Mesh extrahiert werden.

## Scene

Ein Objekt, welches in dem oben genannten Artikel nicht vorkommt. Eine Szene repräsentiert zum Beispiel ein Level in einem Spiel, oder aber auch nur einen Ladebildschirm. Sie enthält eine Reihe von Entities, die in einem Level (oder ähnlichem) vorkommen sollen. Das Wort „enthält“ ist hier wieder relativ, da auch die Scene keine Liste implementiert sondern lediglich eine Id und ein paar Zugriffsmethoden besitzt. Der Zusammenhang zwischen einer Szene und die in ihr vorkommenden Entities wird ebenfalls über die Database geregelt (mehr dazu im Punkt [Implementierung](#)).

## System

Ein System ist für die Manipulation der Daten der Komponenten zuständig. Die Gamelogic findet immer in einem System statt. Vorstellen kann man sich das so, dass ein System alle Methoden besitzt die beim normalen OOP in den Komponenten vorhanden wären. So beinhaltet die Komponente Appearance nicht die Draw-Methode um sich selbst zu zeichnen, sondern das GraphicsSystem. Genauso wird das PhysicsSystem die Positionen der Transform-Komponenten verändern. Für jedes Gebiet der Spieleprogrammierung gibt es ein eigenes System. Im trivialsten Fall sind dies Rendering, Physik, Input und Audio.

## Implementierung

### Database

Hier werden Entities, Komponenten und Szenen, so wie deren Verhältnisse untereinander verwaltet. Für jede dieser drei Kategorien gibt es ein Dictionary, welches die Instanzen über die ihnen zugeordneten IDs in konstanter Zeit  $O(1)$  zugreifbar macht.

```
private IDictionary<int, Scene> _scenes;  
private IDictionary<int, Entity> _entities;  
private IDictionary<int, Component> _components;
```

Die Verhältnisse der Objekte untereinander werden durch drei weitere Dictionaries abgebildet. Prinzipiell hätte ein geschachteltes Dictionary für die Abbildung der Verhältnisse gereicht, aber um den Zugriff auf die Daten zu beschleunigen, wurden mehrere verschachtelte Dictionaries angelegt, welche jeweils verschiedene Keys besitzen und somit unterschiedliche Sortierungen der Komponenten aufweisen. Je nachdem welche Information in der Abfrage vorhanden ist kann so das entsprechende Dictionary zum schnellen Lookup verwendet werden. Hier die drei Dictionaries im Detail:

```
private IDictionary<int, IDictionary<Type, ISet<int>>>> _componentIdsBySceneAndType;
```

Bildet ein Set (Liste in der jedes Element nur einmal vorkommen kann) von ComponentIds auf ihren entsprechenden Typ und anschließend auf die Szene, die den Entities, die diese Komponenten verwenden, zugeordnet ist ab. So kann man z.B. alle Komponenten eines bestimmten Typs für eine beliebige Szene in  $O(2)$  bekommen. Das Durchsuchen des Sets geht ebenfalls schnell, wenn es als HashSet implementiert ist.

```
private IDictionary<int, IDictionary<Type, int>> _componentIdsByEntityAndType;
```

Bildet eine ComponentId auf ihren entsprechenden Typ und anschließend auf die Entity, die dieser Komponente zugeordnet ist, ab. So kann man z.B. zu einer gegebenen Entity sofort auf eine ihm zugeordneten Komponente anhand ihres Typs in  $O(2)$  zugreifen. Dies wird sehr oft benutzt, z.B. in Form von `myEntity.Get<Transform>().Position = new Vector3(1, 2, 3);`

```
private IDictionary<int, ISet<int>> _entityIdsByScene;
```

Bildet ein Set (Liste in der jedes Element nur einmal vorkommen kann) von EntityIds auf die Szene ab, die dem Set zugeordnet ist. So kann man z.B. für eine gegebene Szene eine Liste aller ihr zugeordneten Entities in  $O(1)$  bekommen.

Da diese drei beschriebenen Dictionaries nur auf Basis der Ids von Entities und Komponenten arbeiten um Speicherplatz zu sparen, muss zusätzlich immer noch ein Call z.B. an `Entity.GetInstance(entityId)` bzw. `_entities[entityId]` erfolgen, was die Performancewerte von oben jeweils um  $O(1)$  erhöht. Dennoch handelt es sich bei einem jeden von diesen Aufrufen um Operationen mit konstanter Zeit.

## Threading

Die Klasse Database ist als Thread-sicherer Singleton implementiert. Auf dieser Abstraktionsebene ist Multithreading schon anhand der Zugriffsmethoden immer gegeben. Problematisch wird es allerdings dann, wenn einige DomainSysteme Entityressourcen selbst verwalten müssen. So werden zum Beispiel Xna Models im Graphics durch ein Dictionary mit der zugehörigen Entity als Key und dem Model als Value gespeichert. Sobald eine Entity entfernt wird, oder seine ModelMesh-Komponente, muss das Dictionary im GraphicsSystem angepasst werden. Da die Draw-Methode an sich schon asynchron ausgeführt wird, kann es passieren, dass bei der Erstellung der Liste der aktuell sichtbaren Elemente eine Entity noch existiert, bis zum eigentlichen Draw-Call aber schon wieder entfernt wurde. Dies führt dann zu einer CollectionChanged-Exception. Um dem entgegen zu wirken wird hier mit einer Kopie der ursprünglichen Liste gearbeitet. Dieser Trick kann allerdings nicht in jeder Situation angewandt werden. Im Gegensatz zum passiven GraphicsSystem, welches die Komponenten und Entities ja nicht verändert, werden die SoundInstanzen im Audiosystem und die Bepu-Bodies im PhysicsSystem durch ihre interne Verarbeitung manipuliert. Wenn zu einem ungünstigen Augenblick die zugehörigen Entities entfernt werden gibt es ebenfalls Zugriffsprobleme innerhalb der Listen. Diese können aber glücklicher Weise auch als `ConcurrentDictionary` implementiert werden, wo durch eine gewisse Threadsicherheit besteht.

## Serialisierung

Um Szenen speichern und laden zu können müssen alle Elemente einer Szene serialisierbar sein. Hierfür gibt es die Klasse `EntityState`, welche zum einen die Id der Entity und zum anderen den Bytestream der serialisierten Komponenten dieser Entity enthält. Desweiteren gibt es die Klasse `SceneState`, welche die Id der Szene enthält, so wie den Bytestream aller serialisierten EntityState-Objekte der Entities, die in der Szene vorkommen. Dadurch ergibt sich ein Snapshot der aktuellen Szenensituation, der durch Deserialisierung direkt wieder hergestellt werden kann. Für Quicksaves werden die SceneStates im Speicher der Anwendung abgelegt. Typischerweise gibt es nur einen Quicksave Slot. Zusätzlich wird aber nach dem Betreten einer neuen Szene ein Snapshot angelegt, damit man sie jederzeit wieder neu starten kann. Demnach gibt es im `SceneManager` neben den

Quicksave Slot noch eine Liste an SceneStates, die so groß ist wie die Anzahl der existierenden Szenen (also nur ein Slot pro Szene).

Um ein Spielstand auf die Festplatte zu speichern, wird einfach der erstellte SceneState selbst serialisiert und in eine Datei geschrieben.

Als Konsequenz aus der Serialisierung der Szenen und Entities, müssen auch alle Komponenten serialisierbar sein. Auch die, die vom Nutzer der Engine erstellt werden. Besonders gilt dies für die Komponente `ScriptCollection` und die darin abgelegten `ScriptCollectionItems`. Da ein `ScriptCollectionItem` üblicherweise ein Stück ausführbaren Code enthält, wird dieser als anonymer Delegate in die `Action`-Klasse abgelegt. Die Klasse Action an sich ist immer serialisierbar, wenn allerdings der Code des anonymen Delegates fremde Variablen aus dem Programmkontext benutzt, schlägt die Serialisierung fehl. Um das zu vermeiden, muss für jedes Script von `ScriptCollectionItem` abgeleitet werden.

## Schwachstellen und Verbesserungsvorschläge

Natürlich gibt es bei einem Projekt diesen Umfangs diverse Schwachstellen, vor allem dann, wenn der zur Verfügung stehende Zeitrahmen für eine erfolgreiche Umsetzung kaum ausreicht.

Nachfolgend sind einige Schwachstellen der Engine aufgeführt, die aber theoretisch alle ausgebessert werden können.

### Notification Events

Bei der Erweiterung der Engine durch den User, wird dieser neue Komponenten ins Spiel bringen. Dabei muss jede Property der Komponente nach dem gleichen Muster implementiert werden.

```
private Vector3 _linearVelocity;
public Vector3 LinearVelocity
{
    get
    {
        return _linearVelocity;
    }
    set
    {
        if (value != _linearVelocity)
        {
            _linearVelocity = value;
            ComponentChangedEvent<Movement>.Invoke(this, "LinearVelocity");
        }
    }
}
```

Das ist aufwändig, hinderlich und vor allem eine potenzielle Fehlerquelle. Wesentlich einfacher wäre es, wenn man das für alle Properties gleiche Verhalten abstrahieren kann. In `Sol2E.Core` gibt es eine Klasse `ComponentProperty`. Diese versucht genau das zu gewährleisten, kann es mit der aktuellen Implementierung leider nicht (siehe Inline-Dokumentation). Kurz vor Projektabschluss habe ich von dem `INotifyPropertyChanged` Interface erfahren. Dieses scheint genau diese Funktionalität abzubilden, jedoch ist es bei dem Projekt stand zu spät gewesen, alles darauf hin umzurüsten, um eventuell festzustellen, dass es nicht funktioniert, weil ich zum Beispiel nicht weiß, wie es um seine Serialisierungsmöglichkeiten steht. Da waren das Risiko und der Zeitaufwand zu hoch, da es noch dringendere Aufgaben gab, die noch umgesetzt werden mussten.

## Scene Loading

Für den Spieler störend ist die Tatsache, dass bei einem Szenenwechsel das Spiel kurz einfriert. Dies kann man verhindern, indem der Szenenwechsel in einem Background Thread stattfindet. Das Löschen von Entities kann bei gleichzeitiger Anzeige der Szenen immer noch zu Problemen führen (wie weiter oben unter Threading beschrieben). Alternativ könnte hier eine „ZwischenSzene“ angezeigt werden, die zu Spielbeginn erzeugt wird und fortwährend im Speicher liegt (sie sollte deshalb nicht sehr groß und z.B. nur einen Ladebalken oder ähnliches enthalten). So können die Daten der alten Szene entfernt und der neuen Szene erzeugt werden, ohne, dass Sound, Physics oder Graphics da mit rein funken. Die Implementierung dieses Ansatzes wurde aus Zeitmangel nicht umgesetzt.

## Performance-Einbrüche

Schon bei der Implementierung der zwei Beispielszenen (die zwar an sich souverän laufen) lässt sich ein Trend erkennen, dass je mehr Collisionscripts auf ihre Bedingung getestet werden, die Dauer des Physikupdates zu nimmt und die Framerate herunter geht. Wenn man z.B. in der Lagerhalle jeder Kiste ein eigenes Collisionscript zuweist, sinkt die Framerate bis auf 15 FPS herab. Dies liegt daran, dass etwa 90 Elemente permanent Collisionevents hervorrufen. Es ist zwar grundsätzlich ungeschickt, solche abfragen in die Kisten hinein zu legen, aber es zeigt den Usecase für größere Level.

Diesen Performance-Einbruch könnte man allerdings dadurch stark verringern, indem das CollisionScript nicht auf alle drei Kollisionsarten (CollisionBegan, CollisionEnded, IsColliding) reagiert, sondern für jede Kollisionsart ein eigenes Script implementiert. Im Augenblick wird dem Bepu-Objekt immer alle drei Eventlistener angehängt, wenn einer Entity ein Collisionscript hinzugefügt wird. Die Trennung, je nach gewünschter Kollisionserkennung hätte hier deutliche Vorteile, weil der User in der Regel eh nur am CollisionBegan-Event interessiert ist. Im Kistenbeispiel würden nach der aktuellen Implementierung im jeden Updatecall für alle Kisten, die mit irgendetwas kollidieren das IsColliding Event ausgelöst, obwohl wir eigentlich nur an CollisionBegan interessiert sind, was nur bei neuen Kollisionen auftritt.

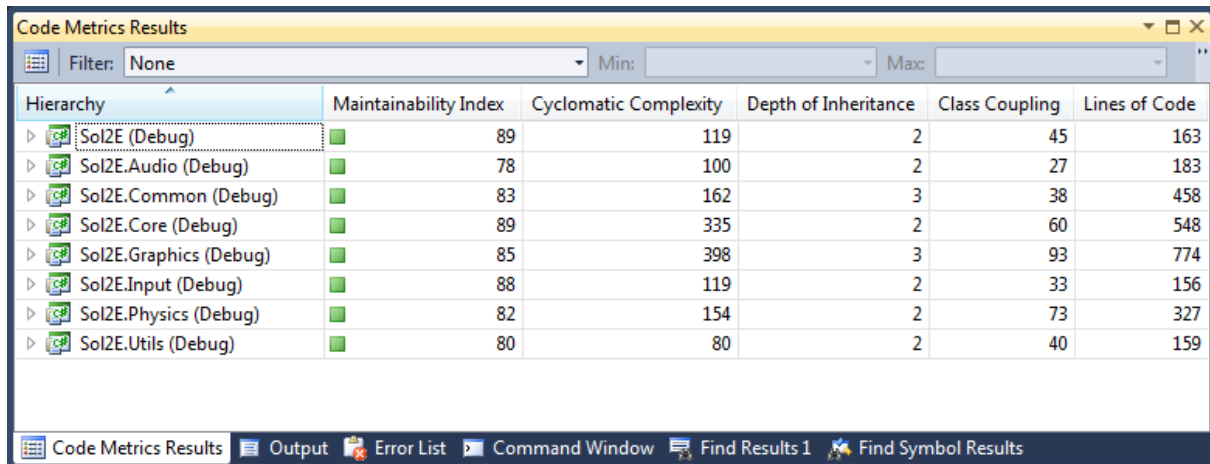
## Multiplattform

Die Unterstützung für andere Plattformen wie Windows Phone und XBOX ist zwar durch XNA gegeben, aber leider nicht durch Sol2E. Im Fall des Windows Phones, liegt es einfach nur daran, dass das Inputsystem derzeit keine Implementierung für TouchInput aufweist. Dies könnte sicher ohne größere Probleme nachgeholt werden, aber nicht im Rahmen der regulären Projektzeit.

XBOX-Unterstützung ist deswegen ausgeschlossen, weil die XBOX keine binäre Serialisierung unterstützt, sondern nur XML-Serialisierung. Der Grund, warum ich dennoch binäre Serialisierung verwende, liegt darin, dass die Dictionary-Klasse standardmäßig keine XML-Serialisierung unterstützt. Es gibt Implementierungen, von diversen Drittanbietern, die genau das ermöglichen, aber um diese genauer zu evaluieren, bevor sie in der Engine eingesetzt werden war ebenfalls nicht genug Zeit vorhanden, beziehungsweise lag der Schwerpunkt nicht auf dem Multiplattform-Aspekt, so dass es an sich genug andere Baustellen gab, die dringender waren.

## Fazit

Insgesamt bin ich mit der Umsetzung und dem entstandenen Produkt zufrieden. Die Architektur ist flexibel und beinhaltet nur flache Vererbungshierarchien. Dies wird durch die Code-Metriken belegt:



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
▶ Sol2E (Debug)	89	119	2	45	163
▶ Sol2E.Audio (Debug)	78	100	2	27	183
▶ Sol2E.Common (Debug)	83	162	3	38	458
▶ Sol2E.Core (Debug)	89	335	2	60	548
▶ Sol2E.Graphics (Debug)	85	398	3	93	774
▶ Sol2E.Input (Debug)	88	119	2	33	156
▶ Sol2E.Physics (Debug)	82	154	2	73	327
▶ Sol2E.Utils (Debug)	80	80	2	40	159

Der Maintainability Index bewegt sich stets zwischen ca. 80 und 90, wohin gegen die Komplexität relativ hoch ist. Die tiefste Vererbungshierarchie umfasst drei Stufen. Dabei handelt es sich um die Mesh-Komponente, die im Vergleich zu den anderen Komponenten eine Ausnahme darstellt, weil ModelMesh und SimpleMesh von ihr abgeleitet sind.

Sehr zufrieden bin ich mit der Einfachheit der Benutzung der Engine. Die Implementierung der Beispiellevel beschränkt sich auf weniger Initialisierungsfunktionen und einige Skripte. Besonders positiv daran ist die Tatsache, dass der Setup-Code einer Szene auf einfache Art und Weise durch einen externen Leveleditor erzeugt werden könnte. Zu jedem Game-Objekt / Entity könnte es ein PropertyWindow geben, in dem über Drag and Drop Komponenten an- und abgehängt werden könnten. Die einzelnen Parameter der Komponenten könnten durch Reflection herausgefunden werden und in dem Fenster aufgelistet und manipuliert werden. Zusätzlich müsste der Editor noch über ein Fenster verfügen, indem die Code-Snippets der Skripte eingetragen werden können.

Um die Engine um weitere Subsysteme zu erweitern, muss der User lediglich eine Klasse erstellen, die von `AbstractDomainSystem` ableitet. Diese muss vor dem `LoadContent()` Call über `AddDomainSystem(IDomainSystem system)` in den Gameloop aufgenommen werden. Anschließend kann innerhalb der Klasse alle eigens erstellten Komponenten nach Belieben manipuliert werden.

Ein weiteres Hauptziel des Projekts, ein Template, welches die dynamisch gelinkten Bibliotheken der Engine benutzt, zu entwickeln und es in Visual Studio zu integrieren erwies sich ebenfalls als deutlich komplizierter als gedacht. Der Code zur Erstellung eines Installers war trivial, jedoch wurde bei dem Export des Templates stets alle Dateien ignoriert, die nicht unmittelbar in das Projekt integriert waren. Die korrekte Lösung dieses Problem liegt in der Verwendung des `IWizard` Interfaces, welches in den Prozess der Template-Erstellung eingreifen kann (siehe [hier](#) und [hier](#)). Da ich davon aber absolut keine Ahnung habe und es schwer war nützliche Informationen über die Implementierung des Interfaces zu finden, wurden die externen Dateien einfach mit ins Projekt aufgenommen, was zwar nicht schön ist, aber funktioniert.

Mit insgesamt 2768 Codezeilen (ohne Kommentare, Leerzeilen, öffnende und schließende Klammern) reinem Engine-Code und 392 Codezeilen aus dem Template ist der Umfang des Projekts sehr hoch und der Benötigte Zeitaufwand von geschätzt ~280 Stunden (nur dieses Semester) deutlich höher als für ein IC vorgesehen.

Mehrere Refactoring Iterationen, so wie zwei (nicht komplette, aber tiefgreifende) neu Implementierungen (davon nur eine in diesem Semester) zeigen darüber hinaus, dass man schnell in Sackgassen geraten kann, wenn man das Grundkonzept nicht ausreichend genug durchdacht hat. Als Beispiel sei hier die Tatsache genannt, dass ich zunächst XNA-Komponenten wie Texturen und Soundeffekte direkt als Properties der Komponenten implementiert habe. Dies ist schlicht wegen der Serialisierung nicht möglich und folgt auch sonst nicht den Leitsätzen aus dem oben genannten [Artikel](#).

Die Orientierung an eben diesen Leitsätzen hat sehr geholfen, wobei das Szenenmanagement und die Serialisierungsprozesse selbst erarbeitet werden mussten. Größere Probleme bei der Verwendung des XNA Frameworks gab es nicht. Die Bepu-Physics Engine hingegen hatte schon einige Stolperfallen in petto. Beispielsweise dauert das Erzeugen neuer Objekte ohne das Prototyping (vgl. den Code in **CreateBodyFromMesh**) schlicht viel zu lang. Besonders auffällig wurde das erst bei der Implementierung des Beispiellevels, in dem das Feuern einer neuen Kugel das Spiel jedes Mal für etwa eine viertel Sekunde hat einfrieren lassen.

Ein eher nicht gelungenes Produkt mit dem hohen Aufwand und der hohen Komplexität zu recht zu fertigen ist in meinen Augen kein Argument, zumal man an der Fehleinschätzung des Aufwandes zumeist nicht unbeteiligt ist. Aus diesem Grunde habe ich so lange daran gearbeitet, bis ich mit dem Resultat zufrieden war, wodurch sich allerdings der Abgabezeitpunkt so stark verzögert hat. Auch lassen sich ein leichter Hang zum Perfektionismus und ein gewisser Grad an Detailverliebtheit nicht verleugnen.