

h_da

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbi

FACHBEREICH INFORMATIK

ABSCHLUSSARBEIT ZUR ERLANGUNG DES AKADEMISCHEN GRADES
BACHELOR OF SCIENCE (B.Sc.)

Entwicklung einer erweiterbaren 2D Game Engine mit visuellem Editor

vorgelegt von: Fedja Adam

Studiengang: Informatik

Referent: Prof. Dr. Wolf-Dieter Groch

Korreferent: Prof. Dr. Thomas Horsch

21. Januar 2013

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 21. Januar 2013

Fedja Adam

Abstract

Diese Arbeit befasst sich mit dem Entwurf und der Umsetzung einer Game Engine für zweidimensionale Computerspiele. Es werden wesentliche Teilbereiche des Gesamtsystems erläutert und verschiedene Ansätze diskutiert, diese zu implementieren. Dies geschieht exemplarisch am Beispiel der von mir entwickelten *Duality Engine*, welche in der Programmiersprache C# geschrieben wurde und auf dem *.Net Framework* basiert.

Im ersten ihrer zwei Kernbereiche befasst sich diese Arbeit mit der Entwicklung einer flexiblen und erweiterbaren Infrastruktur zur Simulation, Verwaltung und Darstellung einer Spielwelt sowie der dafür notwendigen Ressourcen. Dabei wird die grundlegende Software Architektur des Projekts erörtert sowie die Handhabung von Laufzeitobjekten und Spielinhalten näher thematisiert. Weiterhin werden mehrere Verfahren zur persistenten Datenspeicherung untersucht sowie verschiedene Designentwürfe einer einfachen Rendering Schnittstelle gegenübergestellt.

Anschließend werden im zweiten Kernbereich der Arbeit verschiedene Aspekte der Editorentwicklung hervorgehoben und im Kontext von Erweiterbarkeit und Nutzerfreundlichkeit betrachtet. Es wird ein modulares System skizziert, in das einzelne Elemente einer Benutzeroberfläche nahtlos integriert werden können und ein minimaler Satz aus Standardmodulen beschrieben, die eine grundlegende Bearbeitung und Integration von Spielinhalten ermöglichen. Darüber hinaus werden verschiedene Ansätze zur Verbesserung des Nutzererlebnisses anhand ihrer konkreten Umsetzung erläutert.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Was ist eine Game Engine?	1
1.2 Motivation und Ziele	1
2 Grundlagen	4
2.1 Die Programmiersprache C#	4
2.2 Game Engines	9
3 Engine	14
3.1 Architektur	14
3.2 Objektmanagement	20
3.3 Ressourcenverwaltung	28
3.4 Serialisierung	34
3.5 Rendering	44
4 Editor	54
4.1 Aufbau	54
4.2 Module	58
4.3 Usability	67
5 Zusammenfassung	79
6 Ergebnisse und Diskussion	81

1 | Einleitung

1.1 Was ist eine Game Engine?

Viele Computerspiele lassen sich bei grober Betrachtung in zwei Teile zerlegen: Den eigentlichen Spielinhalt sowie ein Softwaresystem, das eine Interaktion mit diesem ermöglicht - die sogenannte *Game Engine*.

Während es sich beim Spielinhalt in der Regel um passive Daten handelt, ist es die aktive Aufgabe der Engine, diese unter Einbeziehung von Nutzereingaben in Echtzeit zu verarbeiten. Die generierte Ausgabe wird dem Nutzer audiovisuell präsentiert und erzeugt so eine Rückkopplung, welche die Basis für Interaktivität bildet. Eine Game Engine definiert damit das Grundgerüst, mit dem ein Spiel entwickelt werden kann. Sie legt die Regeln und Gesetze fest, nach denen die Spielwelt funktioniert und stellt alle Infrastruktur bereit, die für deren Simulation, Interaktion und audiovisuelle Darstellung notwendig ist.

Nicht selten handelt es sich dabei um ein komplexes und vielschichtiges Softwaresystem, dessen Konzeption und Implementierung einen wesentlichen Teil der Entwicklungszeit eines Spiels in Anspruch nimmt. Es ist daher üblich, auf bereits existierende Engines zurückzugreifen oder Engines vergangener Projekte in modifizierter Form wiederzuverwenden, um redundante Arbeit zu vermeiden. Insbesondere innerhalb eines Genres¹ zeigt sich oft eine starke Ähnlichkeit in Spielmechanik und Darstellung, was sich in den Anforderungen an die jeweilige Game Engine widerspiegelt.

1.2 Motivation und Ziele

Die Entwicklung von Computerspielen sowie der dafür notwendigen Software-Infrastruktur stellt für mich seit vielen Jahren ein gut gepflegtes Hobby dar.

¹Beispiele: »First Person Shooter«, »Rollenspiel«, »Point-and-Click Adventure«, »Rundenstrategie«, etc.

Gleichzeitig ergab sich durch meine Arbeit in den Softwareunternehmen *Deck13* und *Limbic Entertainment* auch im professionellen Bereich der Spieleentwicklung die Gelegenheit, vielfältige Erfahrungen zu sammeln.

In meiner Zeit bei Limbic Entertainment arbeitete ich an einem Projekt, das mit der kommerziellen Game Engine *Unity*^[7] realisiert wurde, welche sich als Grundlage für eine Vielzahl verschiedener Spieltypen eignet. Einen wesentlichen Bestandteil der Unity Engine stellt das mitgelieferte Editorsystem dar, das zur Integration von Spielinhalten dient und als Testumgebung verwendet werden kann. Jede Veränderung der Inhalte wird sofort sichtbar und selbst die Arbeit am Quellcode des Spiels kann in vielen Fällen anhand ihrer Auswirkungen »live« beobachtet werden. Was mich daran faszinierte, war die Einfachheit und Eleganz des Arbeitsablaufs, der sich daraus ergab, dass jeder Entwicklungsschritt sofort ein entsprechendes Feedback erzeugte.

Als ich etwas später aus Gründen der Wiederverwendbarkeit damit begann, eine allgemeine Game Engine für zukünftige Hobbyprojekte zu entwickeln, machte ich es mir zum Ziel, mit dieser einen ähnlich effizienten Arbeitsablauf erreichen zu können wie mit Unity. In vieler Hinsicht diente mir die Unity Engine als Referenzpunkt und Orientierungshilfe - jedoch beschränkte ich mich in meiner Eigenentwicklung konsequent auf den zweidimensionalen Raum, was einerseits aus persönlichem Interesse geschah, andererseits aber auch der Verringerung des Arbeitsaufwandes diente. Es ergab sich eine Liste mit den folgenden Anforderungen:

Infrastruktur: Grafische Darstellung, Audioausgabe, Erkennung von Nutzereingaben und Ressourcen- sowie Objektmanagement sollten abstrahiert und dem Nutzer in einfacher Form zur Verfügung gestellt werden.

Flexibilität: Engine und Editor sollten sich für verschiedenste Projekte einsetzen lassen und kein bestimmtes Spielgenre bevorzugen. Es war nicht das Ziel, einen »Spiele-Baukasten« mit vorgefertigten Einzelteilen zu entwickeln, sondern eine Umgebung zur Unterstützung von Eigenentwicklungen zu schaffen.

Live Editing: Wie in Unity sollten Veränderungen an Spielinhalt oder Quellcode sofort sichtbar werden. Ferner sollte es möglich sein, das Spiel innerhalb des Editors ohne Einschränkung der Editorfunktionalität zu testen.

Usability: Das Editorsystem der Engine sollte als Bereicherung empfunden werden, nicht als notwendiger Ballast. Es war daher wichtig, eine intuitive und

verlässliche Benutzerschnittstelle bereitzustellen.

Als Anspielung auf den konzeptionellen Ursprung der Engine und den Fokus auf zweidimensionale Spiele entschied ich mich, der Engine den Namen *Duality* zu geben.



Abbildung 1.1: Das Logo der Duality Engine

In dieser Arbeit soll ein Teil des Entwicklungsprozesses exemplarisch beschrieben werden, um anhand dessen wesentliche Designentscheidungen, Verfahren und Optimierungen zu erläutern. Das Kapitel »Grundlagen« soll zunächst einen groben Überblick von Entwicklungsumgebung und Game Engine Konzepten vermitteln. Den zweiteiligen Kern der Arbeit stellen die Kapitel »Engine« und »Editor« dar, welche auf ausgewählte Unterthemen näher eingehen und diese im Detail behandeln.

2 | Grundlagen

2.1 Die Programmiersprache C#

Sowohl die Duality Game Engine als auch das zugehörige Editorsystem wurden vollständig in der Programmiersprache C# (»C Sharp«) implementiert. Gegenüber C++ zeichnet sie sich durch einen höheren Abstraktionslevel von der zugrundeliegenden Hardware aus, was in der Praxis zu einer geringeren Fehleranfälligkeit und höherer Entwicklungsgeschwindigkeit führt - beides waren aufgrund der Projektgröße wichtige Faktoren.

Anders als C++ wird C# bei der Kompilierung nicht in Maschinensprache übersetzt, sondern in einen *Common Intermediate Language* genannten Bytecode, welcher bei der Ausführung des Programms zur Laufzeit (»Just in Time«) in nativen Code überführt wird. Die sogenannte *Common Language Runtime* (CLR) fungiert dabei als Ausführungsumgebung und kümmert sich darüber hinaus um Infrastruktur Aufgaben wie Speicherverwaltung oder Sicherheitsrichtlinien.^[1] Darüber hinaus steht mit dem begleitenden *.Net Framework* eine umfangreiche Klassenbibliothek zur Verfügung, welche einen Großteil üblicherweise benötigter Grundfunktionalität bereits mitbringt. Dazu zählen neben Strings, Containerklassen, Iteratoren, Streams, Thread- und Prozessverwaltung auch komplexere Themengebiete wie grafische Benutzeroberflächen.

Die aus C++ bekannten Pointer werden in C# begrenzt unterstützt^[2], aber sind nur in wenigen Situationen sinnvoll einsetzbar. Stattdessen wurde das Konzept der *Referenzen* eingeführt, welche sich zwar im Wesentlichen wie typisierte Pointer verhalten, jedoch nicht als Verweis auf eine Speicheradresse zu verstehen sind. Es wird auf ein Objekt verwiesen - wo dieses genau liegt, ist dabei weder relevant noch ermittelbar und kann von der CLR nach belieben verändert werden. Dementsprechend ist keinerlei »Referenzarithmetik« jenseits von Zuweisung und Gleichheitsprüfung möglich. Einen weiteren Nutzen erfüllen Referenzen bei der systemseitigen Speicherverwaltung: Es ist in C# nicht möglich,

den Speicher eines Objekts explizit freizugeben - dies geschieht in einem *Garbage Collection* Mechanismus automatisch, sobald keine Referenzen mehr auf das jeweilige Objekt existieren.

Listing 2.1: Referenztypen und Werttypen

```
public class MyRefType
{
    public int Number;
}
public struct MyValueType
{
    public int Number;
}

public class Program
{
    public static void Main(string [] args)
    {
        MyRefType myRefA = new MyRefType();
        MyRefType myRefB = myRefA;
        myRefA.Number = 42;
        Console.WriteLine(myRefA.Number); // 42
        Console.WriteLine(myRefB.Number); // 42

        MyValueType myValA = new MyValueType();
        MyValueType myValB = myValA;
        myValA.Number = 42;
        Console.WriteLine(myValA.Number); // 42
        Console.WriteLine(myValB.Number); // 0
    }
}
```

Ob bestimmte Objekte als Referenz oder Wert gehandhabt werden, hängt davon ab, wie der Objekttyp definiert wurde. Wird die Instanz eines Referenztyps mehreren Variablen zugewiesen, so halten sie alle eine Referenz auf dasselbe Objekt. Im Falle eines Werttyps enthielte jede Variable stattdessen eine Kopie des Objekts (Lst. 2.1). Die .Net Umgebung unterscheidet elementar zwischen fünf verschiedenen Typen, die sich wie folgt beschreiben lassen:^[3]

Klassen sind Referenztypen, die unter anderem Datenfelder, Methoden und so genannte *Properties* enthalten können. Letztere stellen im Grunde ein Aggregat aus Getter- und Setter-Methode dar. Eine Klasse kann von maximal einer Basisklasse erben. Wird keine solche Basisklasse angegeben, wird implizit von der Systemklasse *object* geerbt. Damit ist jede Klasseninstanz ein *object*.

Strukturen können wie auch Klassen unter anderem Datenfelder, Methoden und Properties enthalten, sind jedoch Werttypen und erlauben keine nutzerseitige Vererbung. Insgesamt sind Strukturen eher darauf ausgelegt, Da-

Listing 2.2: Klassen

```

public abstract class Animal
{
    private string name;
    private int age;

    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }
    public int Age
    {
        get { return this.age; }
    }

    public void Sleep() { /* ... */ }
    public void Eat() { /* ... */ }
}

public class Cat : Animal
{
    private int numLives = 9;
    public int NumLives
    {
        get { return this.numLives; }
    }

    public void Purrr() { /* ... */ }
    public void Scratch() { /* ... */ }
}

```

ten zu gruppieren, als komplexe Objekte zu beschreiben. Primitive Datentypen¹ werden ebenfalls als Strukturen gehandhabt.

Listing 2.3: Datenstrukturen

```

public struct Vector3
{
    public float X;
    public float Y;
    public float Z;

    public static Vector3 operator +(Vector3 left, Vector3 right) { /* ... */ }
}

```

Enumerationen sind primitive numerische Werttypen, die einen abgeschlossenen Satz von benannten Konstanten definieren. Üblicherweise nimmt eine Enum Variable in der Ausprägung genau einen dieser benannten Werte an, ist jedoch nicht ausschließlich an diese gebunden. Im Sonderfall einer Bitmaske können mehrere benannte Werte gleichzeitig angenommen werden.

¹bool, byte, short, int, long, char, ...

Listing 2.4: Enumerationen

```

public enum Color
{
    Red, Green, Blue,
    Purple = 17
}
public enum Feature
{
    A = 0x1,
    B = 0x2,
    C = 0x4,
    All = A | B | C
}

public class Program
{
    public static void Main(string[] args)
    {
        Color c = Color.Green;
        Console.WriteLine(c);           // Green
        Console.WriteLine((int)c);      // 1
        Console.WriteLine((Color)17);   // Purple
        Console.WriteLine((Color)100);  // 100

        Feature f = Feature.A | Feature.B;
        Console.WriteLine(f);          // A, B
        Console.WriteLine(f | Feature.C); // All
    }
}

```

Schnittstellen bilden einen Implementierungsvertrag für Klassen und Strukturen. Sie definieren eine Reihe von abstrakten Methoden oder Properties, die vom jeweiligen Typ implementiert werden müssen. Die Implementierung einer Schnittstelle gilt nicht als Vererbung und unterliegt daher auch nicht den entsprechenden Einschränkungen: Sowohl Klassen als auch Strukturen können eine beliebige Zahl von Schnittstellen implementieren.

Listing 2.5: Schnittstellen

```

public interface ILanguageUser
{
    void Speak();
    void Listen();
}

public class Human : Animal, ILanguageUser
{
    // ...
    public void Speak() { /* ... */ }
    public void Listen() { /* ... */ }
}

```

Delegattypen sind vergleichbar mit den aus C++ bekannten Definitionen von Funktionspointern, können jedoch eine beliebige Anzahl von statischen

oder Objekt Methoden repräsentieren.

Listing 2.6: Delegattypen

```
// Define a Delegate that takes an int and returns nothing
public delegate void IntFunc(int value);

public class Program
{
    public static void Main(string[] args)
    {
        IntFunc someIntFunc = WriteInt;
        someIntFunc(42); // Writes 42 to the console
        someIntFunc += WriteInt;
        someIntFunc(42); // Writes 42 to the console two times
    }
    public void WriteInt(int value) { Console.WriteLine(value); }
}
```

Alle Typen der .Net Umgebung erben auf direktem oder indirektem Weg von der Systemklasse `object` und stellen somit eine universelle Basisfunktionalität bereit. Zu dieser zählt neben der »`ToString`« Konvertierung in eine Zeichenkette auch die »`GetType`« Methode, mit der es möglich ist, Typinformationen einer bestimmten Instanz zu erlangen. Über die zurückgelieferte Systemklasse `Type` können zur Laufzeit umfangreiche Informationen zu Definition, Art und Struktur eines Objekts abgefragt werden. Man spricht dabei auch von einem *Reflection* Mechanismus. Dieser stellt für die Entwicklung erweiterbarer Systeme ein wesentliches Feature dar und erleichtert die Handhabung von zur Compilezeit unbekannten Datentypen.

2.2 Game Engines

Im Hinblick auf den Designgrundsatz einer Game Engine gibt es verschiedene Ansätze, die sich in Flexibilität und Arbeitersparnis für den Benutzer stark unterscheiden. Meist verhalten sich diese Eigenschaften grob gegenläufig, was nicht einer gewissen Logik entbehrt: Je abgeschlossener ein System bereits in sich ist, desto weniger Ansatzpunkte für die Einführung neuer Konzepte gibt es. Die Übergänge sind dabei fließend, jedoch können die Eigenschaften verschiedener Ansätze anhand der folgenden exemplarischen Beispiele verdeutlicht werden:

Funktionsbibliotheken stellen vollständig offene Systeme dar. Sie bieten dem Benutzer eine Reihe von Klassen, Algorithmen und Datenstrukturen, die er für die Implementierung eines Spiels verwenden kann, legen jedoch nicht fest auf welche Art und Weise die einzelnen Elemente zu verwenden sind. Diese Art der Game Engine verhält sich weitgehend passiv: Sie bietet dem Benutzer lediglich Werkzeuge an, die er bei Bedarf verwenden kann, bringt jedoch keine vorgefertigte Architektur mit sich. So kann maximale Flexibilität geboten werden, die jedoch auf Kosten der Arbeitersparnis geht: Bevor tatsächlich ein Spiel entwickelt werden kann, müssen die einzelnen Komponenten der Bibliothek zusammengefügt und mittels zusätzlicher Logik zu einem funktionierenden Gesamtsystem ergänzt werden. Ein bekannter Vertreter dieser Kategorie ist die *Simple and Fast Multimedia Library*^[4].

Datengetriebene Engines bilden das logische Gegenstück zum Ansatz der Funktionsbibliothek: Sie stellen vollständig abgeschlossene Komplettsysteme dar, bei denen eine Modifikation von außen weder vorgesehen noch ohne Weiteres möglich ist. Organisationsstrukturen sowie Logik sind bereits klar definiert und deren konkrete Implementierung ist für den Nutzer nicht von Belang. Die Entwicklung eines Spiels erfolgt hier auf Datenebene: Anstatt Spielinhalte und Spielmechanik über das Schreiben von Quellcode zu integrieren, werden diese in Form von Datenstrukturen ausgedrückt, die von der Engine definiert und ihrem jeweiligen Zweck entsprechend interpretiert werden. Auf diese Weise geht keine Entwicklungszeit für die Schaffung der nötigen Infrastruktur verloren, allerdings gibt es auch keine Möglichkeit, von den vorgegebenen Pfaden der Engine abzuweichen. Die Flexibilität wird also zugunsten maximaler Arbeitersparnis geopfert. Häufig sind datengetriebene Engines an passende Editorsysteme gekoppelt, die es auch technisch weniger begabten Nutzern ermöglichen, zufriedenstellende

Ergebnisse zu erzielen. Ein gutes Beispiel hierfür ist der *Rpg Maker 2000*^[5] welcher die Erstellung einfacher 2D Rollenspiele erlaubt ohne dabei jegliche Programmierkenntnisse vorauszusetzen.

Komponentenbasierte Systeme entsprechen einem Mittelweg zwischen beiden Ansätzen und stellen den Versuch dar, einen Kompromiss zwischen Arbeitsersparnis und Flexibilität zu finden. Dabei werden systemfremden Programmierern öffentliche Schnittstellen zur Verfügung gestellt, mit denen die bisherige Funktionalität einer grundsätzlich einsatzbereiten Engine durch das Hinzufügen neuer oder den Austausch existierender Softwarekomponenten angepasst werden kann.^[6] Das Maß der Anpassungsfähigkeit kann dabei natürlich von Fall zu Fall variieren. Dieser Ansatz ist unter anderem in der *Unity Engine*^[7] zu finden.

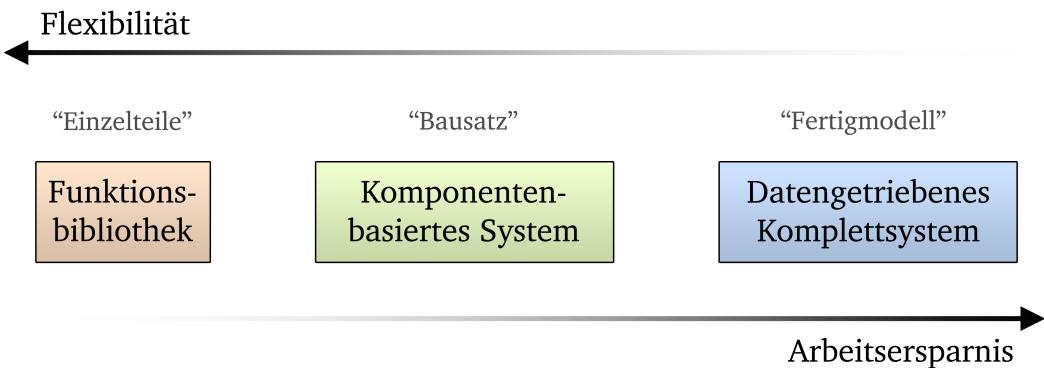


Abbildung 2.1: Verschiedene Ansätze des Game Engine Designs

Funktionsbibliotheken sowie vollständig datengetriebene Engines stellen zwei Gegenpole dar, zwischen denen sich die konkrete Umsetzung einer Game Engine frei bewegen kann (Abb. 2.1). Dennoch erfüllen sie alle grundsätzlich denselben Zweck: Die Simulation und Präsentation eines Spiels. Obwohl sich die konkreten Anforderungen dieser Aufgabe von Fall zu Fall unterscheiden, gibt es einige wiederkehrende Aufgabenfelder, die typischerweise behandelt werden:

Rendering bezeichnet den Vorgang einer grafischen Darstellung der Spielwelt und ist damit unverzichtbarer Bestandteil einer Game Engine. Die Umsetzung erfolgt meist durch Anbindung einer Schnittstelle zur Grafikkarte (z.B. OpenGL oder Direct3D), mit deren Hilfe Renderoperationen durchgeführt werden können. Durch eine zusätzliche Abstraktionsebene werden

diese vor dem Nutzer versteckt: Er teilt der Engine lediglich mit, *was* er darstellen möchte und das Rendering-Subsystem kümmert sich darum, *wie* dies zu bewerkstelligen ist. Dabei können weitreichende Optimierungen zugunsten der Performance stattfinden, ohne dass vom Nutzer ein Verständnis der internen Vorgänge vorausgesetzt wird.

Audioausgabe befasst sich analog zum Rendering mit der akustischen Darstellung der Spielwelt. Auch hier kann eine Umsetzung über die Anbindung und Abstraktion einer Treiberschnittstelle (z.B. OpenAL) erfolgen, allerdings ist es ebenfalls üblich, diese Aufgabe an eine spezialisierte Audio-Middleware² wie beispielsweise *FMOD*^[8] zu delegieren und deren wesentliche Funktionalität auf die Schnittstellen der Engine abzubilden.

Nutzereingaben können über verschiedene Eingabegeräte registriert werden und sollten auf einfache Art für ihre Behandlung innerhalb der Spiellogik zur Verfügung gestellt werden. Klassische Eingabegeräte sind beispielsweise Maus, Tastatur, Gamepad oder Joystick. Weitere spezialisierte Hardware findet sich oft in Verbindung mit modernen Spielekonsolen, so auch die *Microsoft Kinect*- oder *PlayStation Move* Bewegungssteuerungssysteme. Durch die konzeptionell starken Unterschiede verschiedener Eingabemethoden ist eine geräteübergreifende Abstraktion in diesem Bereich nur schwer möglich, weswegen hier durchaus jede Eingabemethode über eine spezialisierte API zugänglich sein kann.

Physikalische Simulation umfasst primär die Kollisionsprüfung zwischen Festkörper Objekten sowie deren nachvollziehbare Reaktion auf stattfindende Kollisionen. Andere Aufgabenfelder können Fluidsimulation, inverse Kinematik sowie die Handhabung von deformier- oder zerstörbaren Materialien sein. Aufgrund der komplexen Natur dieses Subsystems sowie seiner weitgehenden Abgeschlossenheit in sich selbst wird auch hier häufig auf eine spezialisierte Middleware zurückgegriffen, die lediglich in das Gesamtsystem integriert werden muss.

Künstliche Intelligenz ist ein Teilbereich, der sehr stark auf die Anforderungen eines konkreten Spiels zugeschnitten werden muss, daher findet sie meist nur im Fall Genre-spezialisierter Game Engines ihren Weg zur Kernfunktionalität. Eine Ausnahme stellt dabei jedoch das Themengebiet des Pathfin-

²Middleware bezeichnet in diesem Kontext ein unabhängiges Subsystem, das von einem externen Hersteller entwickelt wird.

dings dar, bei dem es darum geht, den optimalen Weg von Punkt A nach Punkt B zu finden, ohne dabei mit Hindernissen zusammenzustoßen.

Netzwerkfunktionalität bildet die Basis für die Unterstützung eines Multiplayer Modus, der es ermöglicht, einen gemeinsamen Spielzustand über mehrere Rechner verteilt zu simulieren und synchronisieren, um mehrere Spieler an einer gemeinsamen Partie teilhaben zu lassen. Weitere Anwendungsmöglichkeiten können globale Online Ranglisten, extern gespeicherte Spielerprofile oder die Integration mit sozialen Netzwerken wie Facebook oder Google+ sein.

Scripting bezeichnet die Möglichkeit, programmierbare Spiellogik auf eine Art und Weise zu implementieren, die keine Veränderungen am Quellcode der Engine voraussetzt. Dies kann beispielsweise über Skriptsprachen realisiert werden, die von der Engine zur Laufzeit interpretiert werden können. Eine andere Möglichkeit stellen datengetriebene »Logik-Baukästen« dar, mit deren Hilfe sich einfache Skripte aus einzelnen Aktions- und Bedingungs-Bausteinen zusammensetzen lassen. Zu den Vorteilen eines Skriptsystems zählt neben der Trennung von Spiel- und Enginelogik auch eine Beschleunigung des Arbeitsprozesses, da bei Veränderungen am Skriptcode sowohl Neukompilierung der Engine als auch der Neustart des Spiels entfallen. Außerdem stellt ein flexibles Skriptsystem gerade in kleineren Firmen auch eine Entlastung der Programmierer dar, weil die Implementierung von Spiellogik nun kein tiefgreifendes technisches Verständnis mehr voraussetzt und so an andere Abteilungen delegiert werden kann.

Ressourcenverwaltung befasst sich mit der Organisation und Strukturierung von Spielinhalten sowie anderen Arten von persistenten Daten. Sie sorgt dafür, dass diese geladen, genutzt, gegebenenfalls gespeichert und wieder freigegeben werden können und kann darüber hinaus die zugrundeliegenden Dateisysteme und Speichermedien abstrahieren. Zur Ressourcenverwaltung kann auch die Handhabung von Daten gezählt werden, die keine Spielinhalte im eigentlichen Sinn darstellen, beispielsweise Spielstände oder Profilinformationen des Spielers.

Objektverwaltung steht für die Organisation von spielrelevanten Laufzeitdaten und legt damit den Grundstein für die Simulation der Spielwelt. Sie ist zuständig für die Verwaltung von Szenengraphen, das Übermitteln von

Nachrichten zwischen einzelnen Objekten, sowie die ordnungsgemäße Erstellung, Zerstörung und Modifikation von Objekten.

Wie die obige Auflistung unschwer erkennen lässt, handelt es sich bei Game Engines um alles andere als simple Konstrukte (Abb. 2.2). Da Wiederverwendbarkeit eines der primären Ziele bei der Entwicklung einer Game Engine darstellt, ist ein verlässliches und nach außen hin zugängliches Softwaredesign unumgänglich.

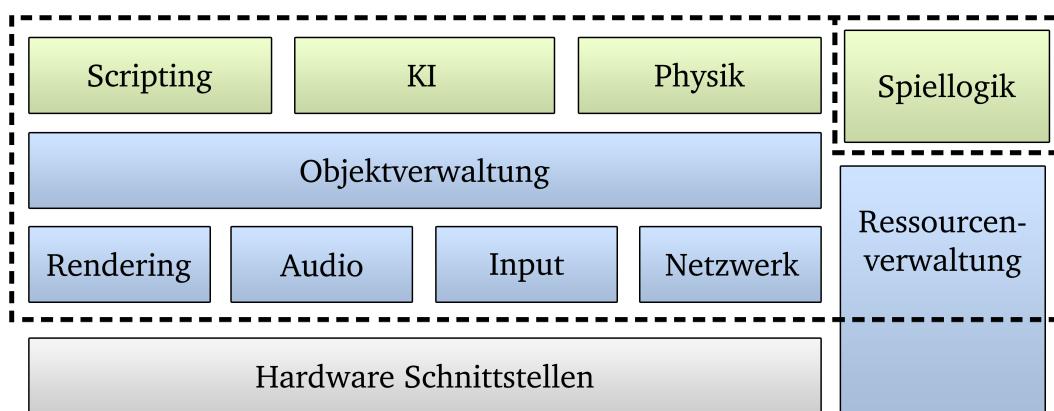


Abbildung 2.2: Eine Auswahl typischer Game Engine Infrastruktur

3 | Engine

Bei der Umsetzung der Duality Engine entschied ich mich für den Ansatz eines komponentenbasierten Systems, der einen Kompromiss zwischen Flexibilität und Arbeitersparnis für den Nutzer darstellt. Es wurde ein bereits funktionstüchtiges Gesamtsystem geschaffen, das jedoch an definierten Punkten vom Benutzer erweitert werden kann, ohne dass dieser dazu den Quellcode der Engine verändern muss. Dazu wurden wesentliche Konzepte der Engine abstrahiert und über offene Programmierschnittstellen nach Außen hin zugänglich gemacht.

3.1 Architektur

Grundsätzlich gibt es mehrere Formen, die eine Game Engine annehmen kann. Für die eher passiven Funktionsbibliotheken bietet sich beispielsweise die Auslieferung als *Dynamically Linked Library* (.dll) Datei an, die vom Benutzer in eigene Projekte eingebunden und anschließend verwendet werden kann. Datengetriebene Komplettsysteme dagegen erlauben oft keine Einmischung auf Seiten des Quellcodes und kommen als bereits ausführbares Programm daher, dem nur noch die entsprechenden Spieldaten hinzugefügt werden müssen.

Einen Mittelweg stellt die Einführung eines Plugin Konzepts dar, bei dem ein solches Komplettsystem zur Laufzeit vom Nutzer geschriebenen Code nachlädt und zur weiteren Ausführung verwendet. Der Nutzer implementiert dazu eine oder mehrere vom System angebotene Schnittstellen, die es der Engine ermöglichen, den zuvor unbekannten Code anhand einer abstrakten Definition auszuführen. Die Entscheidung, an welchen Punkten externe Erweiterungen zulässig sind und welche Rolle diese innerhalb des Gesamtsystem erfüllen, kann also über das Design der Engine gesteuert werden - der konkrete Inhalt dieser Erweiterungen bleibt jedoch frei wählbar dem Nutzer überlassen.

Praktischerweise wird das dynamische Nachladen von kompiliertem Code mit der Funktionalität des .Net Frameworks bereits abgedeckt: Der Code wird

in Paketen organisiert, die als *Assembly* bezeichnet und von der .Net Laufzeitumgebung ausgeführt werden können. Ein vollständiges Programm kann dabei aus einer oder mehreren Assemblies bestehen, die jeweils von einer .dll oder .exe Datei repräsentiert werden¹ und ihrerseits weitere Assemblies als Abhängigkeiten referenzieren können.^[9] Es ist ebenfalls möglich, zuvor unbekannte Assemblies aus einer Datei auszulesen und gleichberechtigt mit bereits geladenem Code zu verwenden.

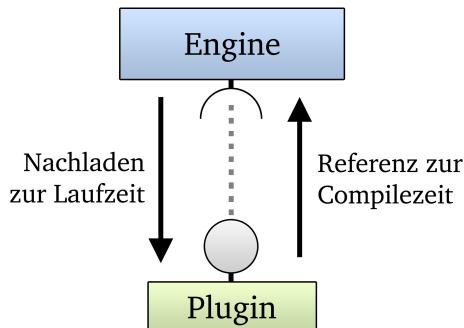


Abbildung 3.1: Das Plugin Konzept

Ein Plugin kann nun als einfache .Net Klassenbibliothek realisiert werden, die in eine Assembly übersetzt und der Engine in Dateiform zur Verfügung gestellt wird; beispielsweise durch das Ablegen in einem dafür vorgesehenen Ordner. Beim Systemstart durchsucht die Engine diesen nach .dll Dateien, lädt die entsprechenden Assemblies und bereitet sie für die spätere Verwendung vor (Abb. 3.1). Eine zentrale Rolle spielt dabei der sogenannte *Reflection Mechanismus* mit dem es möglich ist, Informationen über die innere Struktur einer Assembly zu erlangen.^[10] Dazu zählt neben umfangreichen Metadaten bezüglich einzelner Datentypen, Schnittstellen, Methoden und Membervariablen auch die Auflistung aller öffentlich zugänglichen Klassen einer Assembly. In einem ersten Schritt zur Einbindung eines Plugins kann also ein Register aller Klassen angelegt werden, die eine bestimmte, von der Engine definierte Schnittstelle implementieren. Im zweiten Schritt können diese instanziert und entsprechend der jeweils zugrunde liegenden Schnittstellendefinition verwendet werden (Lst. 3.1).

Als ein erster Ansatz erscheint diese Implementierung eines Plugin Systems bereits ausreichend, doch ein Blick in die Liste der anfänglich festgelegten Zielsetzung² offenbart ein Problem: Für die geplante Live Editing Funktionalität ist

¹Eine Ausnahme stellen hierbei die an dieser Stelle nicht näher behandelten *Multi File Assemblies* dar.

²Siehe Abschnitt 1.2 Motivation und Ziele

Listing 3.1: Plugins: Ein erster Ansatz

```
// Prepare register for usable plugin types
List<Type> usablePluginTypes = new List<Type>();

// Gather paths to all available plugin .dll files
string[] pluginDllPaths = Directory.GetFiles("Plugins", "*.dll");
foreach (string dllPath in pluginDllPaths)
{
    // Load each Assembly and iterate over its public Types
    Assembly pluginAssembly = Assembly.LoadFrom(dllPath);
    foreach (Type pluginType in pluginAssembly.GetExportedTypes())
    {
        // If one implements a certain interface...
        if (typeof(IMyInterface).IsAssignableFrom(pluginType))
        {
            // ...register it for later use.
            usablePluginTypes.Add(pluginType);
        }
    }
}

// Using the registered plugin types
foreach (Type pluginType in usablePluginTypes)
{
    object myInstance = Activator.CreateInstance(pluginType);
    IMyInterface myUsefulInstance = (IMyInterface)myInstance;
    myUsefulInstance.PerformSomeOperation();
}
```

es nicht ausreichend, Plugins einmal bei Systemstart zu laden, denn der Nutzer könnte sein selbstgeschriebenes Plugin zur Laufzeit der Engine neu kompilieren, um Veränderungen am Quellcode zu testen und direkt nutzbar zu machen. Die Behandlung dieses Falls ist für eine zufriedenstellende Umsetzung des Live Editing Features durchaus wichtig, da ein Neustart des Gesamtsystems einen Bruch im Arbeitsablauf darstellen würde. Es stellt sich also die Frage, wie auf einen solchen Fall zu reagieren ist.

Zunächst einmal muss die alte Version des betroffenen Plugins aus dem System entfernt werden, um sie im nächsten Schritt durch ihr aktualisiertes Pendant zu ersetzen. Das Entfernen eines dynamisch geladenen Plugins stellt allerdings keine triviale Aufgabe dar, denn dazu müssen nicht nur die entsprechenden Klassen aus dem zuvor angelegten Register entfernt werden - sondern auch all ihre möglicherweise systemweit verstreuten Instanzen. Darüber hinaus darf diese Entfernung keinerlei destruktive Auswirkungen haben, um ungespeicherte Arbeiten an Spielinhalten nicht durch ein Neuladen von Plugins zu gefährden. Gerade im Hinblick darauf, dass einige dieser Spielinhalte Abhängigkeiten innerhalb des zu entfernenden Plugins aufweisen können, ist auch diese Bedingung alles andere als eine Vereinfachung der Problemstellung.

Ein möglicher Lösungsansatz ist es, alle potentiell betroffenen Klasseninstan-

zen in einen Datenstrom zu serialisieren³, anschließend zu löschen und durch eine Deserialisierung des Datenstroms wieder zu rekonstruieren. Da im Zuge der Serialisierung jegliche Typisierung auf Datenebene abgebildet wird, kann auf diese Weise sichergestellt werden, dass beim Entfernen eines veralteten Plugins keine der dort definierten Typen mehr referenziert werden. Bei der darauffolgenden Rekonstruktion der betroffenen Objekte können die notwendigen Typinformationen nun aus der *aktualisierten* Version des Plugins abgeleitet werden. Auf diese Weise können auf einen Schlag alle Klasseninstanzen einer veralteten Pluginversion entfernt und durch eine äquivalente Instanz der neueren Version ersetzt werden (Abb. 3.2).

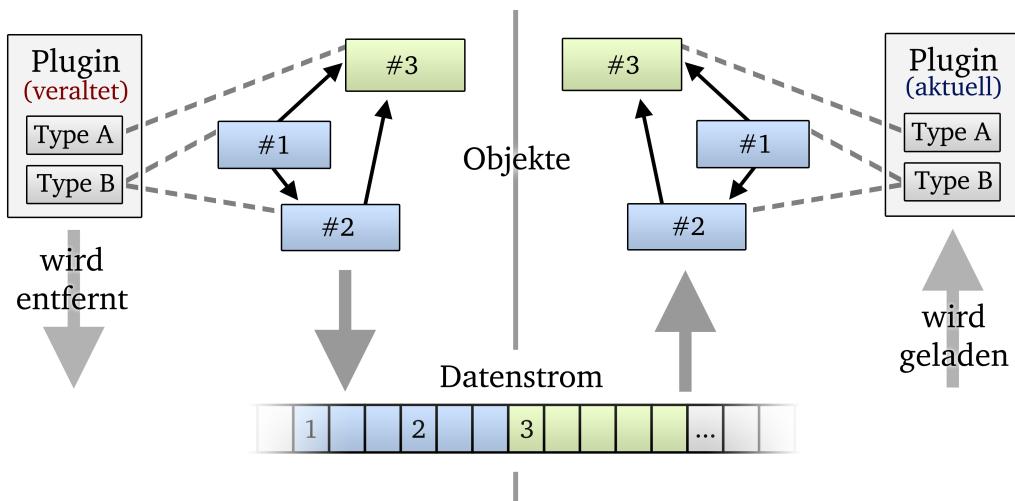


Abbildung 3.2: Neuladen eines Plugins zur Laufzeit

Unglücklicherweise ist es innerhalb der .Net Umgebung aus Gründen der Sicherheit und Fehlervermeidung nicht möglich, einmal geladene Assemblies wieder zu entladen.^[11] Ein vollständiges Entfernen veralteter Plugins ist also technisch gesehen gar nicht durchführbar: In der Praxis verbleiben alle jemals geladenen Versionen eines Plugins weiterhin verfügbar und müssen manuell verwaltet werden, um eine fälschliche Nutzung veralteter Assemblies auszuschließen.

Bei einer Plugin Aktualisierung ist der komplette Austausch »alter« gegen »neue« Klasseninstanzen daher ganz besonders im Auge zu behalten, denn augenscheinlich sind diese weder im Debugger noch durch ihr Laufzeitverhalten auf Anhieb unterscheidbar - obwohl es sich intern um zwei völlig inkompatible Klassen handelt. Die Vermischung der Objekte verschiedener Pluginversionen

³Serialisierung befasst sich mit der Abbildung von Laufzeitobjekten auf ein persistentes Datenformat. Einige Details zu diesem Thema werden in einem späteren Kapitel erläutert.

kann daher zu inkonsistentem Verhalten, Abstürzen und kaum nachvollziehbarem Fehlverhalten führen und ist um jeden Preis zu vermeiden. Langfristig stellen solche »Doppelgänger« eine schwer lokalisierbare Fehlerquelle dar.

Um eine »saubere« Versionierung von Plugins und zugleich eine abgekapselte Ausführung von externem Code zu ermöglichen, bietet das .Net Framework die sogenannten *AppDomains* an. Diese stellen eine Isolationsebene zwischen ausgeföhrtem Code und der ausführenden Umgebung dar.^[12] Ein Programm enthält implizit mindestens eine AppDomain; es ist aber auch möglich, zur Laufzeit weitere AppDomains zu erzeugen, die als separate Ausführungsumgebung verwendet werden können. Anders als einzelne Assemblies können AppDomains als Ganzes auch ohne Risiko wieder entladen werden, da es sich um vollständig abgeschlossene Systeme handelt (Abb. 3.3). Würde man nun jedes Plugin mittels eigener AppDomain laden, müsste man sich zwar dennoch um den zuvor erwähnten Austausch von Klasseninstanzen kümmern; es wäre aber im Anschluss daran möglich, das Plugin tatsächlich wieder vollständig zu entfernen und so eine zukünftige Fehlerquelle auszuschließen.

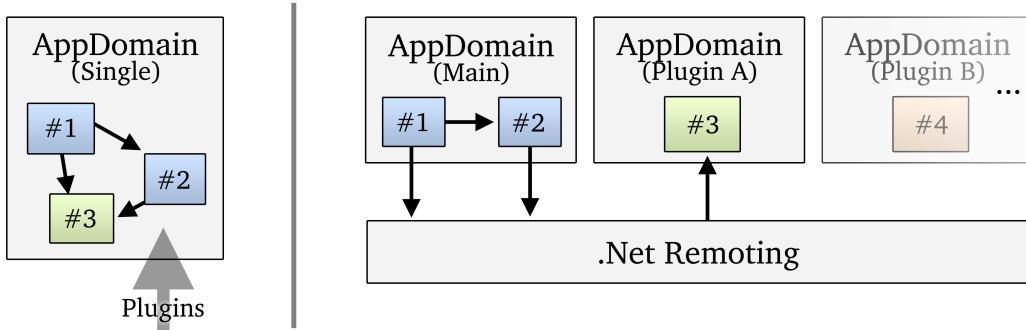


Abbildung 3.3: Single- vs. Multi-Domain Plugins

Der größte Vorteil dieses Ansatzes ist zugleich sein größter Nachteil: Um besagte Abgeschlossenheit zu ermöglichen, müssen Objekte zwischen AppDomains entweder als Kopien übergeben oder auf Proxy-Objekte abgebildet werden. Das kann zwar weitgehend automatisch erfolgen; jedoch ist die Zeitdauer eines einzigen Funktionsaufrufs mittels Proxy-Objekt um ein Vielfaches höher als die eines Direktaufrufs.^[13] Im zeitkritischen Kontext einer Game Engine stellen die damit verbundenen Performance Einbußen durchaus ein Problem dar, daher sei der AppDomain Ansatz an dieser Stelle nur der Vollständigkeit halber erwähnt. Als Grundlage für eine tatsächliche Implementierung kommt er damit nicht in Frage.

Nachdem der grundlegende Aufbau einer Plugin Architektur damit skizziert

wäre, stellt sich noch die Frage der konkreten Strukturierung des Projekts. Grundsätzlich wäre es denkbar, sowohl Engine als auch Editor vollständig in einer einzigen ausführbaren Assembly unterzubringen, um die Interaktion der einzelnen Subsysteme zu vereinfachen. Dies würde die Wartbarkeit des Gesamtsystems jedoch langfristig reduzieren, da auf diese Weise ein systemweiter Nährboden für Klassenkopplung und kaum überschaubare Querverweise entstünde. Eine klare Trennung von Engine und Editor erscheint daher sinnvoll. Es lässt sich ebenfalls ein für den Spieler gedachtes »Launcher«-Programm abspalten, mit dem ein fertiges Spiel gestartet werden kann. Die Engine wird dabei als eigenständige Assembly gestaltet, die eine öffentliche Schnittstelle bereitstellt, mit der es möglich ist, sie in Launcher und Editor einzubinden (Abb. 3.4). Prinzipiell könnte sie auf diese Weise auch in zuvor unbekannter Umgebung ausgeführt und vom Benutzer bei Bedarf in eigene Programme eingebunden werden.

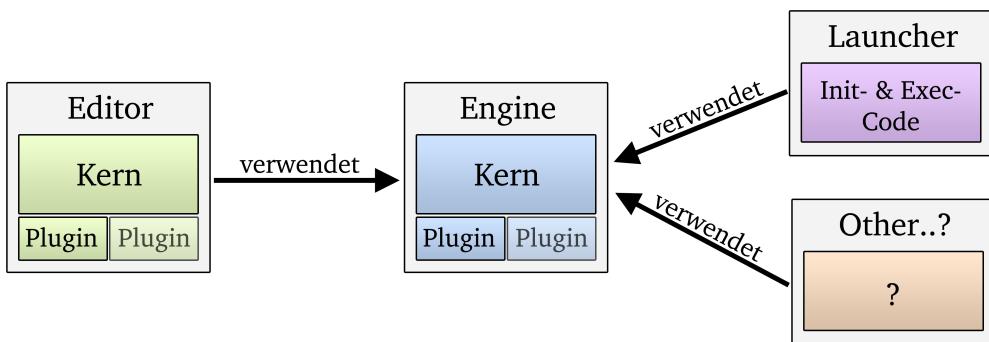


Abbildung 3.4: Struktur des Gesamtsystems

Durch die Trennung von Engine und Editor wird eine entsprechende Zweiteilung des skizzierten Pluginsystems ebenfalls sinnvoll - einerseits zur klaren Aufgabentrennung, andererseits auch um die Zahl der Abhängigkeiten zu verringern: Während die von der Engine geladenen Plugins lediglich spielrelevante Erweiterungen einbringen, ist es nur in den separaten Editor Plugins nötig, auf Editor- und GUI-Funktionalität zuzugreifen. Eine klare Aufgabentrennung vereinfacht auch die Behandlung einer Plugin Aktualisierung zur Laufzeit, da diese so gezielt im jeweils betroffenen Teilsystem stattfinden kann.

3.2 Objektmanagement

Die meisten denkbaren Spielwelten lassen sich durch einen Graph aus einzelnen *Objekten* repräsentieren. Was genau durch ein solches Objekt ausgedrückt wird und wie feingranular die Spielwelt in einzelne Objekte aufgelöst wird, unterscheidet sich jedoch von Spiel zu Spiel. Es sollte daher dem Nutzer überlassen werden, dies zu definieren. Die Aufgabe der Engine besteht nun darin, einen Rahmen für die Definition eigener Objekttypen zu schaffen sowie grundlegende Verwaltungsaufgaben zu übernehmen. Es gibt im Kontext der Game Engine Entwicklung viele Designgrundsätze, die sich für die Betrachtung eines Objekts anbieten; zwei davon möchte ich als Beispiele herausgreifen: Die klassische Vererbungshierarchie sowie Komponentenbasierte GameObjects.^[14]

Die alles entscheidende Frage einer klassischen Vererbungshierarchie lautet »Was ist das für ein Objekt?«. Ist es beweglich oder statisch? Ist es ein Fahrzeug oder ein Lebewesen? Motorrad oder Ruderboot? Setzt man alle anfallenden Klassifizierungen in Beziehung zueinander, erhält man eine Hierarchie, welche bei der Implementierung auf tatsächliche Klassenstrukturen abgebildet werden kann (Abb. 3.5). Verfolgt man diese von einem Blattknoten aus bis zu ihrer Wurzel, ergibt sich ein logischer Zusammenhang: Ein Ruderboot ist ein Fahrzeug ist ein bewegliches Objekt ist ein Objekt. Jede Stufe in dieser Hierarchie fügt der geerbten Funktionalität eine bestimmte Verhaltensweise hinzu. Der Code für die Bewegung eines Objekts muss nur ein einziges Mal in »MovableObject« implementiert werden und alle davon erbenden Objekte sind automatisch zur Simulation von Bewegung in der Lage. Redundanzen können so effektiv vermieden und kleinere Änderungen zentral durchgeführt werden.

Ein Problem mit größeren Vererbungshierarchien ist, dass zum Entwurf dieser möglichst früh alle existierenden Klassen bekannt sein sollten, da sich spätere Veränderungen nur schwer umsetzen lassen, insbesondere wenn diese an einem tragenden Ast des Vererbungsbaumes stattfinden sollen. Ebenfalls schwierig zu handhaben ist die Tatsache, dass eine bestimmte Objekteigenschaft oder -verhaltensweise um so weiter in der Hierarchie nach oben wandert, je flexibler sie einsetzbar sein soll: Das Ruderboot kann beispielsweise schwimmen, es wäre also naheliegend, diese Funktionalität in der Ruderboot-Klasse zu implementieren. Das U-Boot kann allerdings auch schwimmen und beides sind Fahrzeuge - warum also nicht die Schwimmfähigkeit in die Fahrzeugklasse verschieben und je nach Bedarf aktivieren oder deaktivieren? So weit so gut - was aber, wenn wir nun die Klasse »Delfin« hinzufügen? Delfine sind offensichtlich Lebewesen und

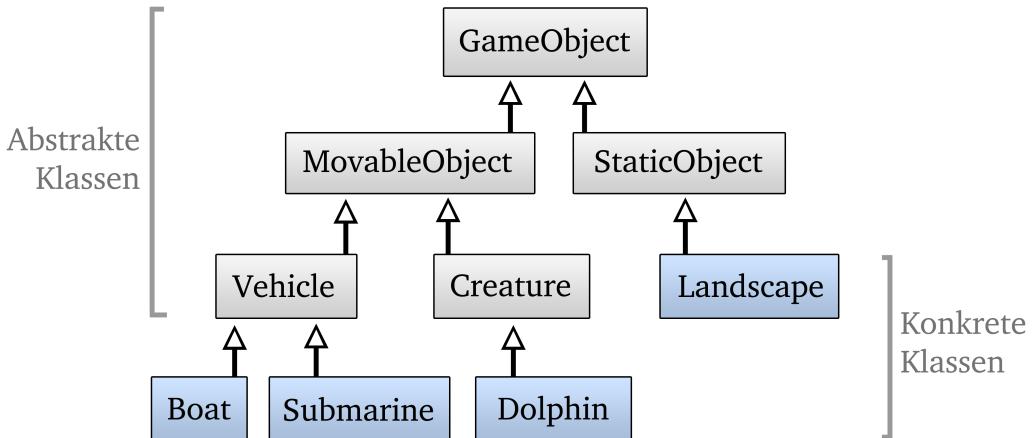


Abbildung 3.5: Beispiel einer Vererbungshierarchie

keine Fahrzeuge, können aber nichtsdestotrotz schwimmen. Um eine Mehrfachimplementierung der Schwimmfähigkeit zu verhindern bliebe nun nichts anderes übrig, als sie in der Hierarchie in die gemeinsame Basisklasse von »Fahrzeug« und »Delfin« zu verschieben - in diesem Fall »MovableObject«.

In ähnlicher Form haben die meisten Verhaltensweisen und Eigenschaften die Tendenz, in einer Vererbungshierarchie nach oben zu wandern, da sich früher oder später immer ein konkretes Objekt finden lässt, welches das bisherige Schema durcheinander bringt und nach einer allgemeineren Implementierung verlangt. In höchstmöglicher Eskalation führt dies zu einer alleskönnenden »Blob«-Klasse, die sämtliche existierenden Verhaltensweisen in sich vereint (Abb. 3.6) und per Konfiguration entscheidet, welche davon jeweils aktiv sind. Die gewonnene Flexibilität hat ihren Preis: Da sämtliche Funktionalität in einer einzigen Klasse implementiert wird, trägt jedes in der Praxis noch so einfach gezeichnete Objekt die maximal mögliche Datenlast mit sich herum und enthält die Algorithmen und Funktionen *aller* Objekttypen auf einmal - eine Katastrophe für Wartbarkeit und Nutzerfreundlichkeit.

Spätestens an diesem Punkt ist es ratsam, das Softwaredesign etwas aufzuräumen. Dabei soll jedoch die neu gewonnene Flexibilität nicht wieder verloren gehen! Prinzipiell sollte weiterhin jedes Objekt in der Lage sein, auf die gesamte verfügbare Funktionalität zurückzugreifen - was bisher fehlt ist hauptsächlich eine saubere Strukturierung. Eine erste Maßnahme wäre es, die einzelnen zu- und abschaltbaren Teilfunktionen der »Blob«-Klasse in separate Klassen auszulagern, sozusagen eine Zerlegung in einzelne Teilkomponenten - womit wir bei der Grundlage der *Komponentenbasierten GameObjects* angelangt wären.

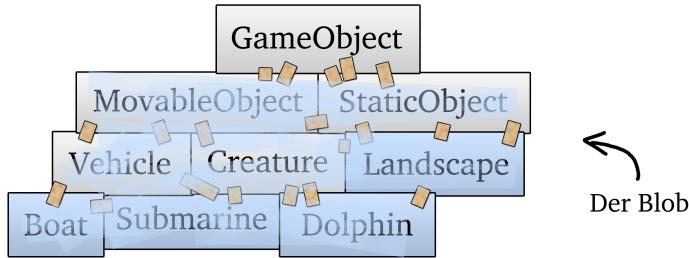


Abbildung 3.6: Resultat eskalierter Verallgemeinerung

Anders als die Vererbungshierarchie fragt der komponentenbasierte Ansatz nicht »Was ist das?« sondern »Wie verhält es sich?«. Das zentrale Designelement stellt nicht eine klassifizierende Hierarchie dar, sondern ein Pool aus modularen Verhaltensweisen und Eigenschaften, die in ihrer Aggregation ein konkretes Objekt ausmachen. Deren Einzelteile bezeichnet man als Komponenten. Jede Komponente erfüllt eine klar abgegrenzte Aufgabe und kann einem Objekt dynamisch hinzugefügt oder wieder abgenommen werden. Die Gestaltung konkreter Komponenten kann viele Formen annehmen, daher sind vielleicht einige Beispiele angebracht:

Transform stattet ein Objekt mit einer räumlichen Position, Rotation und Skalierung aus und bezieht dabei auch den Szenengraph ein: Handelt es sich beim Objekt der Transform-Komponente um ein »Child« Objekt, werden alle räumlichen Eigenschaften als relativ zum »Parent« Objekt interpretiert.

SpriteRenderer sorgen für eine grafische Darstellung des Objekts als sogenanntes »Sprite«, ein texturiertes Rechteck, das von vier Vertexpunkten aufgespannt wird. Hierfür wird eine Transform-Komponente vorausgesetzt.

SoundEmitter sorgen für eine akustische Darstellung des Objekts. Sofern eine Transform-Komponente vorhanden ist, kann der abgespielte Klang durch Anpassung von Lautstärke und Links / Rechts Verschiebung räumlich positioniert werden.

RigidBody stattet ein Objekt mit physikalischen Eigenschaften aus und sorgt für die Simulation von Kräften sowie die Erkennung von und Reaktion auf Kollisionen. Auch hier wird eine Transform-Komponente vorausgesetzt.

Das Objekt selbst erfüllt in diesem Ansatz lediglich verwaltende Aufgaben: Es stellt einen Knoten im Szenengraph dar, verwaltet die zugehörigen Komponenten

und ermöglicht die notwendige Kommunikation zwischen ihnen. Eine einzelne Komponente hat im Szenengraph keine Existenzgrundlage und ist daher stets an ein Objekt gekoppelt.

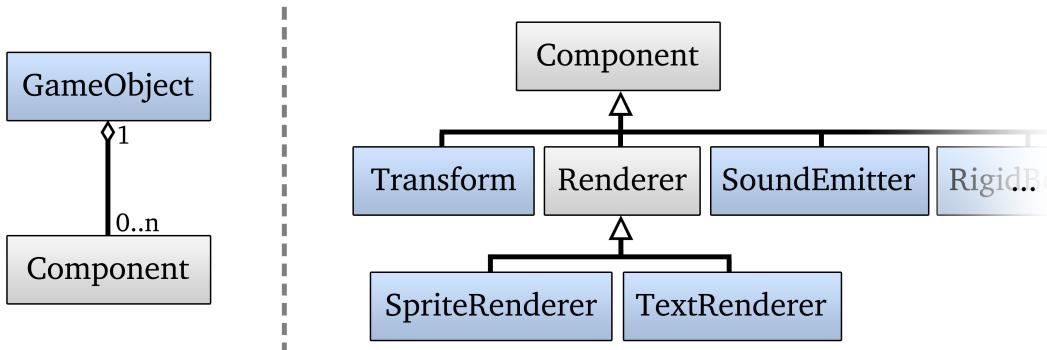


Abbildung 3.7: Komponentenbasierte GameObjects

Zwischen verschiedenen Komponentenklassen kann bei Bedarf weiterhin eine Vererbungshierarchie eingerichtet werden, um die Vorteile beider Ansätze nutzbar zu machen (Abb. 3.7). Dies kann sinnvoll sein, wenn verschiedene Komponenten auf einer gemeinsamen Funktionalität aufbauen, die jedoch im allgemeinen Fall keinen Sinn ergibt: Unterschiedliche Typen von Fahrzeugen könnten beispielsweise von der Eigenschaft vereint werden, Passagiere aufnehmen zu können und sich anschließend von diesen steuern zu lassen. Weil diese Funktionalität nur im sehr spezifischen Kontext eines Fahrzeugs sinnvoll einsetzbar ist, könnte sie in eine gemeinsame »Fahrzeug«-Basisklasse ausgelagert werden, die selbst wiederum eine Komponente ist. Die genaue Ausprägung und Verknüpfung einzelner Komponentenklassen hängt jedoch stets vom Kontext des jeweiligen Spiels ab! Während es in einem Rollenspiel sinnvoll sein kann, selbst eine einfache Kiste aufgrund ihrer vielfältigen und einzeln wiederverwendbaren Interaktionsmöglichkeiten auf verschiedene Teilkomponenten abzubilden⁴, ist sie in einem Strategiespiel als reines Dekorationsobjekt möglicherweise nicht einmal eine einzige spezialisierte Komponente wert.

In Kombination mit einer Plugin Architektur entfaltet der komponentenbasierte Ansatz weiteres Potential: Durch Exponierung der abstrakten Komponenten Basisklasse wird es möglich, einzelne Komponententypen per Plugin nachzurüsten und nahtlos in das bestehende System einzufügen. Durch die Definition einer allgemeinen Schnittstelle für die Interaktion mit Komponenten ist es nicht notwendig, deren konkrete Ausprägung zu kennen, um mit ihnen zu arbeiten -

⁴zerstörbar, benutzbar, blockiert Laufweg, visuell hervorgehoben, ...

Listing 3.2: Implementierungsskizze eines komponentenbasierten GameObjects

```

public sealed class GameObject // "sealed": Don't derive from GameObject
{
    // Properties
    public GameObject Parent { get; set; }
    public string Name { get; set; }
    public IEnumerable<GameObject> Children { get; }

    // Component Access
    public IEnumerable<T> GetComponents<T>() { /* ... */ }
    public T GetComponent<T> () { /* ... */ }

    // Object Setup
    public T AddComponent<T> () where T : Component { /* ... */ }
    public T RemoveComponent<T>() where T : Component { /* ... */ }
    public void AddComponent (Component cmp) { /* ... */ }
    public void RemoveComponent(Component cmp) { /* ... */ }
}

```

und ob diese nun aus dem Kern der Engine stammen oder aus einem Plugin spielt dabei keine Rolle. Im Entwicklungsverlauf der Duality Engine hat sich die Implementierung von Nutzerkomponenten in Plugins daher als primärer Ankerpunkt zur Erweiterung der Engine bewährt.

Für die Definition einer allgemeinen Komponentenschnittstelle gibt es mehrere mögliche Ansätze. Besonders naheliegend ist es, in der Basisklasse einfach eine Reihe von virtuellen Methoden zu definieren, beispielsweise OnInit und OnShutdown zur Aktivierung und Deaktivierung, OnUpdate zur Simulation des Verhaltens, OnRender zur visuellen Darstellung, OnCollision zur Handhabung von Kollisionen und so weiter und so fort (Lst. 3.3 / Abb. 3.8). Eine abgeleitete Komponente könnte nun diejenigen Methoden überschreiben, die für sie von Belang sind. Problematisch dabei ist jedoch die nach oben offene Anzahl von Spezialfällen, für welche die Komponentenklasse bei diesem Ansatz Methoden bereitstellen müsste. Außerdem haben nicht alle Komponenten wirklich etwas darzustellen - dennoch muss »OnRender« für alle Komponenten sicherheitshalber aufgerufen werden, da von Außen nicht ersichtlich ist, ob eine solche Methode wirklich überschrieben und benutzt wird oder nur als Gerüst in der Basisklasse vorhanden ist. Es muss daher davon ausgegangen werden, dass jede Komponente das gesamte Spektrum der Schnittstelle ausnutzt.

Zwar sind die Auswirkungen der zusätzlichen Funktionsaufrufe für die Performance vernachlässigbar gering, allerdings wäre es aus Gründen der Strukturierung und des Debuggings durchaus nützlich, bereits im Vorfeld zu wissen, welche Teile der allgemeinen Komponentenschnittstelle von einer konkreten Komponente tatsächlich benutzt werden.

Ein weiteres Problem dieses Ansatzes betrifft die Offenheit für Erweiterun-

Listing 3.3: Komponentenschnittstelle über virtuelle Methoden

```
// ----- Component base class -----
public abstract class Component
{
    // ... Base Implementation ...

    // General Component interface
    public virtual void OnInit( /*...*/ ) {}
    public virtual void OnShutdown( /*...*/ ) {}
    public virtual void OnUpdate( /*...*/ ) {}
    public virtual void OnRender( /*...*/ ) {}
    public virtual void OnCollisionBegin( /*...*/ ) {}
    public virtual void OnCollisionEnd( /*...*/ ) {}
    public virtual void OnCollisionSolve( /*...*/ ) {}
    // ... etc. ...
}

// ----- Concrete Component class -----
public class SpriteRenderer : Component
{
    public override void OnUpdate( /*...*/ ) { /* Implementation */ }
    public override void OnRender( /*...*/ ) { /* Implementation */ }
}
```

gen: Die abstrakte Komponentenklasse wird einmalig in der Engine definiert und kann von außen nicht mehr verändert werden. Wird dort als allgemeine Schnittstelle eine Reihe von virtuellen Methoden festgelegt, ist es später nicht mehr möglich, diese über ein Plugin zu erweitern - was allerdings insbesondere bei größeren Erweiterungen durchaus wünschenswert sein kann.

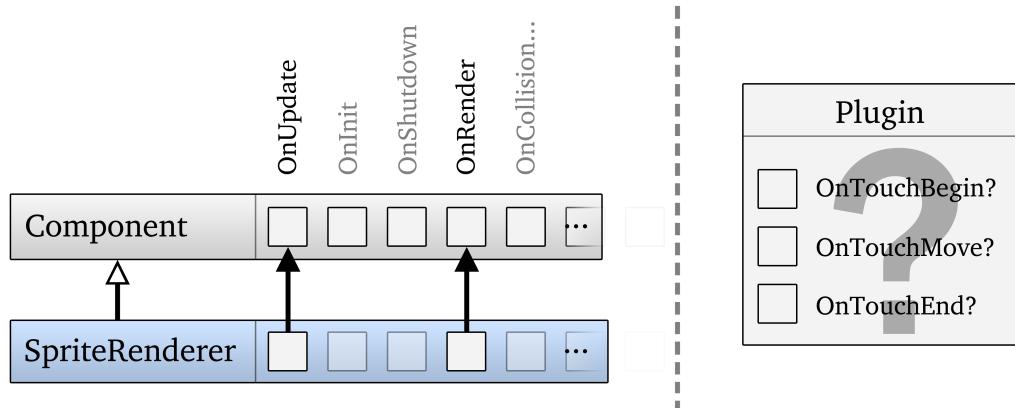


Abbildung 3.8: Komponentenschnittstelle über virtuelle Methoden

Nehmen wir beispielsweise an, die Engine würde durch ein Plugin um den Support für Multitouch⁵ Eingaben erweitert: Um es Komponenten anderer Plugins zu vereinfachen, auf diese neuartigen Eingaben zu reagieren, könnte der Au-

⁵Touchscreens, die mehrere Berührungen gleichzeitig detektieren und verarbeiten können, werden auch als »Multitouchscreens« bezeichnet.

tor selbst einige überschreibbare Methoden anbieten, beispielsweise »OnTouchBegin«, »OnTouchMove« und »OnTouchEnd«. Weil er die allgemeine Komponentenklasse aber natürlich nicht erweitern kann, bleiben ihm zwei Möglichkeiten:

1. Die Definition einer eigenen »MultitouchComponent« Klasse, die von der allgemeinen Komponentenklasse erbt und die entsprechenden virtuellen Methoden hinzufügt. Der Nutzer müsste seine eigenen Komponenten nun von »MultitouchComponent« erben lassen. Das funktioniert allerdings nur solange nur ein einziges Plugin Erweiterungen auf diesem Weg einzubringen versucht. Wird nun beispielsweise ein Multiplayer Plugin hinzugefügt, das seine Funktionalität über eine abstrakte »NetworkComponent« zur Verfügung stellt, steht der Nutzer vor einem Problem: Er kann seine eigenen Komponenten nicht von beiden Basiskomponenten gleichzeitig erben lassen und muss sich daher für nur eine von beiden Zusatzfunktionen entscheiden. Dieser Ansatz würde also begrenzt funktionieren, bleibt insgesamt aber unpraktikabel.
2. Die Definition einer separaten Schnittstelle, die von Nutzerkomponenten implementiert werden kann, um auf die zusätzliche Funktionalität zurückzugreifen. Die Probleme aus Variante 1 bestehen hier nicht - allerdings stellt dieser Ansatz einen Bruch im Gesamtkonzept dar. Bisher war der Nutzer es gewohnt, in seinen eigenen Komponenten lediglich virtuelle Methoden zu überschreiben - und für ein Plugin soll er nun plötzlich eine separate Schnittstelle implementieren? Es gibt keinen für ihn relevanten Grund, warum dieser Bruch im Softwaredesign existiert, was die Zusatzfunktionen des Plugins für ihn zu »Bürgern zweiter Klasse« machen würde.

Eine mögliche Lösung für die genannten Probleme besteht darin, bereits die im Kern der Engine definierten Methoden in separate Schnittstellen auszulagern, die jeweils eine logische Einheit bilden und bei Bedarf einzeln implementiert werden können (Lst. 3.4 / Abb. 3.9). Auf diese Weise sind Erweiterungen über Plugins möglich, die sich nahtlos einfügen ohne dabei weiter aufzufallen oder eine Inkonsistenz im Softwaredesign zu erzeugen.

Ein weiterer Vorteil ist, dass schon anhand der Klassendefinition einer konkreten Komponente ersichtlich wird, auf welche Weise sie zu behandeln und für welche Subsysteme der Engine sie interessant ist. Im Zuge der grafischen Darstellung müssen zum Beispiel nur Komponenten beachtet werden, welche die zuständige Rendering Schnittstelle implementieren - alle anderen können in einer Vorauswahl bereits aussortiert werden.

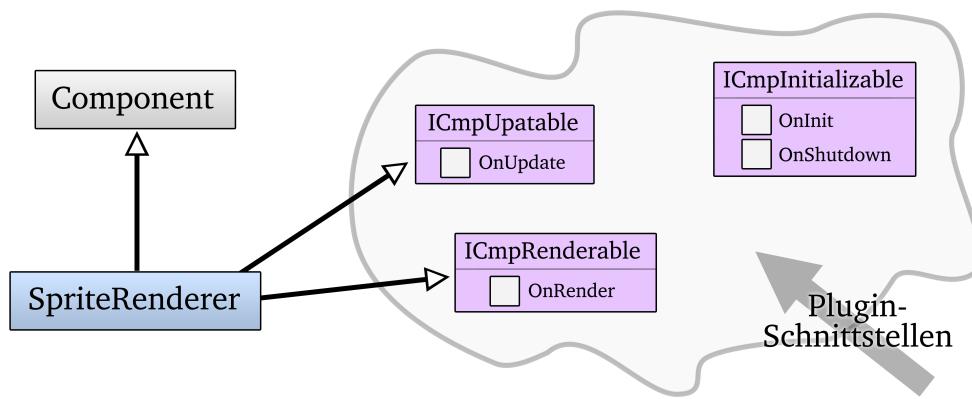


Abbildung 3.9: Modulare Komponentenschnittstelle

Listing 3.4: Modulare Komponentenschnittstelle

```

// _____ Component base class _____
public abstract class Component { /* Base Implementation */ }

// _____ General Component interface _____
public interface ICmpInitializable      // "I want to be setup and shut down!"
{
    void OnInit( /*...*/ );
    void OnShutdown( /*...*/ );
}

public interface ICmpUpdatable           // "I want to be updated each frame!"
{
    void OnUpdate( /*...*/ );
}

public interface ICmpRenderable          // "I can be visually displayed!"
{
    void OnRender( /*...*/ );
}

public interface ICmpCollisionListener   // "Inform me about collisions!"
{
    void OnCollisionBegin( /*...*/ );
    void OnCollisionEnd( /*...*/ );
    void OnCollisionSolve( /*...*/ );
}

// _____ Concrete Component class _____
public class SpriteRenderer : Component, ICmpUpdatable, ICmpRenderable
{
    public void OnUpdate( /*...*/ ) { /* Implementation */ }
    public void OnRender( /*...*/ ) { /* Implementation */ }
}

```

3.3 Ressourcenverwaltung

Für die Entwicklung eines Spiels reicht es nicht allein aus, den dafür nötigen Quellcode zu schreiben; es wird in nahezu jedem Fall einen Punkt geben, an dem darüber hinaus auf externe Daten zugegriffen werden muss. Die Notwendigkeit kann im Laden von Grafiken, Sound, Teilen der Spiellogik oder einzelnen Levels bestehen, kann sich aber ebensogut auf weitere Arten von Spielinhalten erstrecken. All diese oft als *Ressourcen* bezeichneten externen Daten müssen verwaltet werden, um Programmierer und Engine den Zugriff darauf zu vereinfachen, sowie deren Verwendung überhaupt erst zu ermöglichen.

Auch wenn jeder Ressourcentyp grundsätzlich eigene Anforderungen mitbringt, so zeigen sich dennoch Ähnlichkeiten in der Handhabung. Üblicherweise muss eine Ressource zunächst einmal initialisiert werden, wobei zum Beispiel die entsprechenden Dateien eingelesen werden können und Speicher angefordert werden kann. Anschließend kann die Ressource entsprechend ihrer Bestimmung verwendet oder modifiziert werden. Zum Abschluss müssen die während der Initialisierung angeforderten Systemressourcen wieder freigegeben werden, zuvor können gegebenenfalls durchgeführte Änderungen in den zugrundeliegenden Dateien gespeichert werden. Diese gemeinsamen Eckpunkte aller Ressourcen können in der gemeinsamen Basisklasse bereits angelegt werden: Laden bzw. Initialisieren, Speichern und Freigeben sowie eine identifizierende Pfadangabe bilden damit das öffentliche Grundgerüst einer allgemeinen Ressourcen-Basisklasse (Abb. 3.10). Von dieser können separate Klassen für Texturen, Audiodaten und andere Ressourcentypen abgeleitet werden, welche die jeweilige konkrete Implementierung enthalten und typspezifische Funktionen zu ihrer Benutzung anbieten.

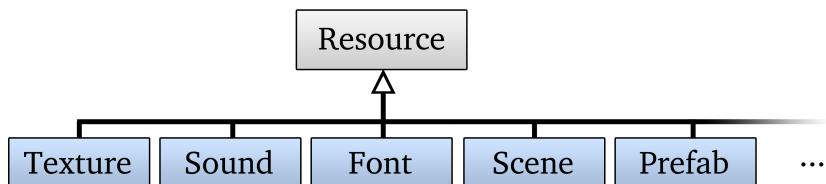
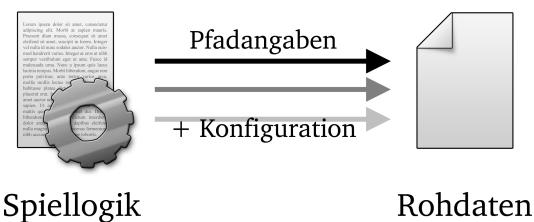


Abbildung 3.10: Die gemeinsame Basis aller Ressourcen

Eine weitere allgemeine Designentscheidung betrifft die Konfiguration von Ressourcen: Texturen beispielsweise können durch ein einfaches Laden von Bilddaten initialisiert werden. Um eine einfache Grafik darzustellen würde in vie-

len Fällen also eine Pfadangabe ausreichen, welche sich auf die zu verwendenden Bilddaten bezieht. Bei dieser Vorgehensweise wird jedoch schnell auffallen, dass Texturen nicht allein durch die zugrundeliegenden Pixeldaten definiert werden! Darüber hinaus sind Konfigurationsdaten notwendig, die über Dinge wie Mipmapping und Pixelformat entscheiden oder die Handhabung von NPOT-Seitenlängen⁶ beeinflussen. Woher werden diese Informationen genommen?

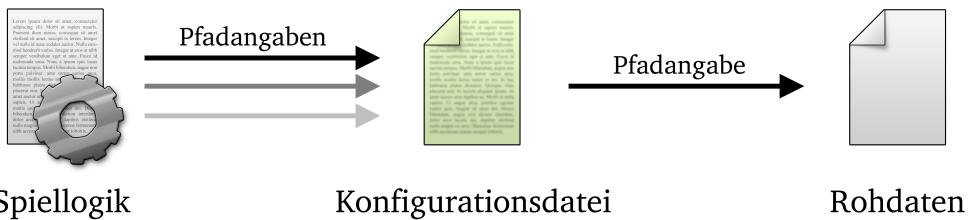
Eine naheliegende Möglichkeit ist es, diese Informationen aus dem Kontext der Verwendung abzuleiten. Damit liegt die Verantwortlichkeit für die Konfiguration einer Ressource immer dort, wo sie angefordert wird. Durch die Festlegung eines sinnvollen Standardverhaltens kann diese Aufgabe in vielen Fällen auf wenige Zeilen reduziert und somit durchaus praktikabel gemacht werden. Ein Nachteil dieser Methode ist das Fehlen einer Gesamtübersicht aller Ressourcen - zwar kann man sich die einzelnen Dateien ansehen, jedoch bleiben die so erhaltenen Informationen ohne Kenntnis des Verwendungskontexts und dessen interner Verhaltensweise unvollständig. Außerdem können einzelne Dateien als Rohdaten für mehrere Ressourcen dienen, die in jeweils unterschiedlichem Kontext verwendet werden - auch das ist so nicht ersichtlich. Das Problem der Übersicht einmal beiseite gelassen könnte man es unter Umständen auch als Designfehler betrachten, zentral verfügbare Ressourcen dezentral zu konfigurieren; in jedem Fall aber stellt die so entstehende Redundanz eine mögliche Fehlerquelle dar.



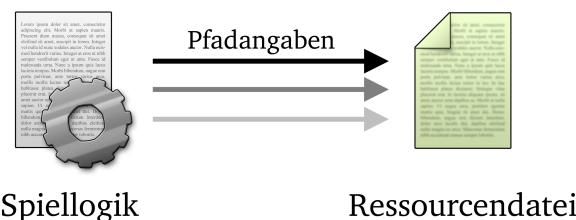
Die oben genannten Schwächen lassen sich durch die Einführung einer Metadaten Ebene ausräumen: Anstatt direkt die zu verwendenden Rohdaten per Pfadangabe zu referenzieren, gilt diese nun stattdessen einer separaten Konfigurationsdatei. Diese verweist ihrerseits auf genannte Rohdaten und stellt zusätzlich alle Informationen bereit, die zur Initialisierung, Verwendung und Verwaltung der jeweiligen Ressourcen notwendig sind. So kann einerseits das Problem de-

⁶NPOT steht für »Non Power-Of-Two« und bezieht sich auf die Tatsache, dass Grafikkarten Texturen bevorzugen, deren Seitenlängen Zweierpotenzen sind, also beispielsweise 256x256 oder 64x1024. Auf modernen Grafikkarten sind auch beliebige andere Texturmaße möglich, jedoch sollten diese aus Kompatibilitätsgründen vermieden werden.

zentraler Konfiguration gelöst werden während andererseits zu jedem Zeitpunkt eine vollständige Übersicht aller Ressourcen möglich ist. Um das Hinzufügen neuer Ressourcen in den aktiven Spielinhalt zu erleichtern, können entsprechende Konfigurationsdateien auch automatisch erstellt und mit sinnvollen Standardwerten voreingestellt werden - einen Vorgang, der auch als *Import* bezeichnet wird.



In einem nächsten Schritt können weiterhin alle von einer Ressource verwendeten Daten in einer einzigen Datei zusammengefasst werden, beispielsweise durch die Verwendung eines eigenen Datenformats. Auf diese Weise wird der zusätzliche Indirektionslevel zwischen Konfigurationsdatei und Rohdaten⁷ entfernt, was zur Verbesserung der Übersicht und Vermeidung von Fehlerquellen beitragen kann. Auch wird das System durch die Zusammenfassung nach außen hin abgeschottet - etwas, das sowohl Vor- als auch Nachteile birgt: Spielinhalte werden so in begrenztem Maß vor unerwünschten Fremdzugriffen geschützt, sind aber auch vom Entwickler selbst nur noch über die Funktionalität der Engine erreichbar, was Veränderungen an bestehenden Inhalten erschweren kann.



Im Kontext der Multiplattform Entwicklung kann es für die Ressourcenverwaltung wichtig werden, zusätzlich zu den bereits beschriebenen Strukturen ein sogenanntes *Asset Management System* zur Verfügung zu stellen. Dessen Aufgabe ist es, die verwendeten Rohdaten sowie deren Beziehung zu den generierten Ressourcen zu verwalten, um den zuvor einmaligen Importvorgang nun zu

⁷Indirektion bezeichnet die Referenzierung eines Objekts mittels Name, Speicheradresse, Pfadangabe o.Ä. anstelle des eigentlichen Wertes. In diesem Fall referenziert Spiellogik eine externe Konfigurationsdatei, welche ihrerseits externe Rohdaten referenziert: Es liegt eine doppelte Indirektion vor.

einer wiederhol- und konfigurierbaren Aufgabe zu machen, die verschiedenen Plattformen und Qualitätsstufen angepasst werden kann.^[15] So kann aus einem konstanten Satz an Rohdaten ein je nach Anforderung unterschiedlicher Satz von Ressourcen erzeugt werden - beispielsweise können auf einem Embedded System besonders niedrig aufgelöste Texturen oder ein spezielles Kompressionsverfahren verwendet werden, auf dem Pc jedoch die volle Auflösung bei verlustfreier oder keiner Kompression. Da die Duality Engine jedoch nur für die Entwicklung innerhalb der Pc Plattform vorgesehen war, ein Asset Management System jedoch eine größere Arbeitsinvestition darstellt, wurde ein solches nicht implementiert. Es sei daher nur der Vollständigkeit halber erwähnt und soll im Rahmen dieser Arbeit nicht näher thematisiert werden.

Durch die Einführung von Konfigurations- oder Ressourcendateien wird das Problem redundanter Ressourcenkonfiguration zwar gelöst, aber noch nicht verhindert, dass eine Ressource zur Laufzeit mehrfach geladen wird. Sofern keine systemweite Verwaltung von Ressourcen existiert, werden Ressourcen überall dort instantiiert, wo sie benötigt werden - ohne Kenntnis darüber, dass die jeweilige Ressource möglicherweise bereits an anderer Stelle geladen wurde! Einseitig wird durch die resultierende Mehrfachinstantiierung sowohl Rechenzeit als auch Speicher verschwendet, sobald mehrere Komponenten der Engine auf dieselbe Ressource zugreifen; andererseits stellt dies bei der Bearbeitung von Ressourcen zur Laufzeit ein prinzipielles Problem dar, das sich nicht allein durch leistungsfähigere Hardware lösen lässt: Bearbeitet der Nutzer im Editor die Konfiguration einer Ressource zur Laufzeit, verlangt es die Anforderung des Live Editings, dass die Auswirkungen sofort und systemweit sichtbar werden. Durch die dezentrale Anforderung von Ressourcen ist dies jedoch nicht möglich, da überhaupt nicht bekannt ist, welche Instanzen bereits existieren und zur Laufzeit aktualisiert werden müssen.

Es wäre also sinnvoll, eine zentrale Verwaltungsstelle zu implementieren, über die Ressourcen angefordert und wieder freigegeben werden können. So kann sichergestellt werden, dass von jeder Ressource nur eine einzige Instanz existiert, die gemeinsam referenziert wird. Die *ContentProvider* genannte Duality Implementierung einer solchen Verwaltungsstelle basiert im Wesentlichen auf einer simplen Zuordnung zwischen Pfadangaben und derzeit aktiven Ressourcen, die in einem sortierten Register hinterlegt und gegebenenfalls nachgeschlagen werden (Lst. 3.5).

Zusätzlich kann die Referenzierung an sich über einen eigenen Datentyp abstrahiert werden, um auch den kompletten Austausch von Ressourcen zur Lauf-

Listing 3.5: Implementierungsskizze des ContentProviders

```

public static class ContentProvider
{
    // Global register for all active Resources
    private static Dictionary<string, Resource> resLibrary = /* ... */;

    // Call this method to request certain content
    public static T RequestContent<T>(string path) where T : Resource
    {
        // Return cached content
        Resource res;
        if (resLibrary.TryGetValue(path, out res) && !res.Disposed)
            return res as T;

        // Load new content
        return LoadContent(path) as T;
    }

    private static Resource LoadContent(string path)
    {
        /* Actually load the Resource and return it */
    }
}

```

zeit zu ermöglichen sowie die Handhabung von nicht verfügbaren Ressourcen zu vereinfachen (Lst. 3.6). Anstatt also wie im ContentProvider Beispiel auf Anforderung eine direkte Referenz einer Ressource zu erhalten, wird als Resultat eine *ContentRef* Datenstruktur zurückgeliefert, über die bei Bedarf auf die tatsächliche Ressource zugegriffen werden kann.

Durch die *ContentRef* Abstraktion kann das System sicherstellen, dass alle Referenzen auf eine freigegebene Ressource⁸ gelöst und bei Bedarf neu angefordert werden. Um nun beispielsweise das systemweite Neuladen einer Ressource zu erzwingen, muss lediglich die bestehende Instanz freigegeben werden - die entsprechenden *ContentRefs* werden die Freigabe ihrer Ressource beim nächsten Zugriff erkennen und sie auf Basis des hinterlegten Pfads neu anfordern.

Ein weiterer Nutzen ergibt sich, sollte eine Ressourcendatei unerwarteterweise fehlen: Weil es in diesem Fall nicht möglich ist, eine entsprechende Ressource zu instantiiieren, ergäbe eine Auflösung der betroffenen Ressourcenpfade nichts weiter als eine Ansammlung von Nullreferenzen, die sich zur Weiterarbeit nicht eignen. Es ist nicht mehr feststellbar, welche Ressource ursprünglich geladen werden sollte oder ob der jeweilige Null Wert überhaupt einen Fehler darstellt - schließlich ist eine Nullreferenz im Allgemeinen Fall noch nichts ungewöhnliches und kann durchaus Absicht sein. Die *ContentRef* Datenstruktur bleibt in solchen

⁸Freigegebene Ressourcen sind nicht länger zur Verwendung geeignet und enthalten möglicherweise alte oder korrupte Daten, daher werden sie über ein *Disposed* Property entsprechend markiert.

Listing 3.6: Vereinfachte Implementierung der ContentRef Datenstruktur

```

public struct ContentRef<T> where T : Resource
{
    private T      contentInstance;
    private string contentPath;

    public T Res
    {
        get
        {
            if (this.contentInstance == null ||
                this.contentInstance.Disposed)
            {
                this.RetrieveInstance();
            }
            return this.contentInstance;
        }
    }

    public string Path
    {
        get { return this.contentPath; }
    }

    private void RetrieveInstance()
    {
        /* Assure that the contentInstance field has an up-to-date,
         * non-null reference to the actual Resource. This might
         * trigger loading it for the first time or reloading it, if
         * the last known instance has become invalid.
        */
    }
}

```

Fällen flexibler, denn sie enthält neben der konkreten Referenz immer auch den unaufgelösten Ressourcenpfad. So bleibt die Referenz selbst bei Nichtverfügbarkeit der Ressource mit anderen vergleichbar und kann in ihrer unaufgelösten Form weiterhin verwendet und weitergereicht werden. Sobald die fehlenden Ressourcen wieder verfügbar sind, werden es beim nächsten Zugriff auch die entsprechenden ContentRefs.

3.4 Serialisierung

Das Themenfeld der Serialisierung befasst sich mit der Abbildung von Laufzeitobjekten auf ein Format, das mittels Datenstrom in persistenter Form abgelegt werden kann. Zu einem späteren Zeitpunkt können aus den erzeugten Daten die ursprünglichen Objekte rekonstruiert werden.^[16]

Ein verlässliches Serialisierungssystem bildet die Grundlage der Ressourcenverwaltung und kann auf vielseitige Weise zum Speichern und Laden von Objekten eingesetzt werden: Sowohl Ressourcen im eigentlichen Sinn als auch Spielstände und Nutzereinstellungen können damit als Dateien auf der Festplatte abgelegt oder im Speicher für die spätere Verwendung hinterlegt werden. Auch für das zuvor beschriebene Aktualisieren von Plugins zur Laufzeit wird ein Serialisierungsverfahren eingesetzt, um die Typisierung aktiver Objekte neu aufzulösen.

Während die Serialisierung einer primitiven Datenstruktur mit den Mitteln des .Net Frameworks eine triviale Aufgabe darstellt, wird im allgemeinen Fall durch die Referenzierung weiterer Objekte ein beliebig komplexer Objektgraph aufgespannt. Dieser muss als Ganzes serialisiert werden, um den ursprünglichen Objektzustand wiederherstellen zu können. Darauf hinaus verlangt die Behandlung von Objektreferenzen weitere Überlegungen, um sowohl die Identität einzelner Objekte zu bewahren, als auch eventuelle Schleifen im Objektgraph korrekt zu behandeln. Für die Duality Engine wurde ein allgemeines Serialisierungssystem angestrebt, das systemweit für beliebige Zwecke einsetzbar sein sollte. Dabei galten die folgenden Anforderungen:

Symmetrie: Das im Zuge der Rekonstruktion entstandene Objekt sollte dem ursprünglich serialisierten Objekt in allen wesentlichen Eigenschaften gleichen.

Datenorientierung: Die De/Serialisierung eines Objekts sollte vollständig anhand der reinen Objektdaten stattfinden können, also vollkommen unabhängig von jeglicher im Objekt implementierten Logik.

Identitätswahrung: Die Identität eines Objekts darf nicht verloren gehen, Referenzen müssen als solche abgebildet werden können. Ein vielfach referenziertes Objekt darf nicht auf eine Vielzahl von Objekten abgebildet werden.

Automatisierung: Im allgemeinen Fall sollte eine De/Serialisierung vollautomatisch ablaufen können und keine explizite Implementierung seitens des

Nutzer verlangen. Für diese Aufgabe können Reflection Mechanismen des .Net Frameworks eingesetzt werden.

Flexibilität: Es sollte möglich sein, bestimmte Objekttypen bei Bedarf manuell zu serialisieren oder deren Serialisierung auf anderem Weg in vergleichbarem Ausmaß zu beeinflussen.

Selbstbeschreibendes Format: Alle zur Interpretation des Datenstroms notwendigen Informationen sollten im Datenstrom selbst hinterlegt werden⁹.

Fehlertoleranz: Fehler, die während der De/Serialisierung auftreten, sind so gut es geht abzufangen und einzudämmen, um den entstandenen Schaden so gering wie möglich zu halten. Ein Misserfolg bei der Rekonstruktion eines einzelnen Objekts sollte auf keinen Fall den gesamten De/Serialisierungsprozess korrumpern.

Die letzten beiden Punkte ergeben sich primär aus der Tatsache, dass im Zuge von Live Editing und iterativen Entwicklungsprozessen eine Inkonsistenz zwischen persistenten Daten und den codeseitigen Datenstrukturen zu erwarten ist: Ein in Dateiform abgespeicherter Level eines Spiels kann beispielsweise Objekte enthalten, deren Klassendefinition seitdem größeren Änderungen unterworfen war. Felder könnten hinzugefügt, entfernt oder umbenannt worden sein und die verwendeten Datentypen könnten sich verändert haben. Es wäre jedoch untragbar für den Entwicklungsprozess, wenn betroffene Objekte durch solche Änderungen nicht länger rekonstruierbar wären, daher ist es Aufgabe des Serialisierungssystems, Probleme dieser Art so gut es geht abzufedern. Sollte eine Klassendefinition vollständig entfernt werden, ist dies natürlich nicht mehr möglich, da es in diesem Fall nichts mehr gibt, das hier instantiiert werden könnte - allerdings kann der Rest des Objektgraphen möglicherweise unbeküllt weiter existieren.

Das .Net Framework selbst bietet mehrere Arten der Serialisierung an, darunter sowohl eine Form von XML- als auch binärer Serialisierung. Erstere wird über die *XmlSerializer* Klasse abgewickelt und nutzt die öffentlich zugreifbaren Daten eines Objekts¹⁰, um dieses vollautomatisch auf eine entsprechende XML Baumstruktur abzubilden (Lst. 3.8, 3.9). Die Art und Weise der Abbildung lässt sich bei Bedarf über Attribute steuern.^[17] Das ist jedoch nicht in allen Fällen

⁹Eine Ausnahme bildet natürlich der Serialisierungsalgorithmus selbst.

¹⁰Öffentliche Felder sowie Properties mit öffentlichem Lese- und Schreibzugriff

Listing 3.7: Referenzklasse zum Vergleich der Serialisierung

```
[Serializable]
public class TestObject
{
    private int      myPrivateVar = 17;
    private string   myString     = "HelloWorld";
    private List<float> myFloatList = new List<float> { 0.1f, 0.5f, 0.7f };

    public string MyStringProperty
    {
        get { return this.myString; }
        set { this.myString = value; }
    }
    public List<float> MyFloatListProperty
    {
        get { return this.myFloatList; }
        set { this.myFloatList = value; }
    }
}
```

ausreichend: Zum einen kann es für die vollständige Rekonstruktion eines Objekts durchaus notwendig sein, auch nichtöffentliche Daten zu serialisieren - und zum anderen wird beim Setzen eines Properties stets auch der Code der impliziten Setter Methode ausgeführt, was nicht vorhersehbare Auswirkungen auf den Objektzustand haben kann. Diese können das Ergebnis der Deserialisierung verfälschen oder zu kaum vorhersehbaren Fehlern führen, da das Objekt zum Ausführungszeitpunkt des Setter Codes noch gar nicht vollständig initialisiert wurde.

Listing 3.8: .Net XML Serialisierung

```
TestObject data = new TestObject();

XmlSerializer xml = new XmlSerializer(typeof(TestObject));
using (FileStream f = File.OpenWrite("test.xml"))
{
    xml.Serialize(f, data);
    // data = xml.Deserialize(f) as TestObject;
}
```

Ein weiteres Problem besteht in der Unfähigkeit des XmlSerializers, mit Objektreferenzen als solchen umzugehen: Ein zweifach referenziertes Objekt wird ohne Rücksicht auf dessen Identität auch zweifach serialisiert. Nach der Rekonstruktion hat sich das zuvor einmalige Objekt verdoppelt und zwei unterschiedliche Instanzen existieren, die unabhängig voneinander modifiziert werden können - ein meist unerwünschtes Verhalten, das in vielen Fällen Fehler im Programmablauf hervorruft. Darüber hinaus stößt die Funktionalität des XmlSerializers an eine Grenze, wenn es um die Serialisierung von Referenzen geht, deren

Typ nicht im Vorfeld bekannt oder nicht eindeutig ist: Eine Referenz vom Typ `object` kann prinzipiell jedes beliebige Objekt enthalten. Da im resultierenden XML keine Typinformationen hinterlegt werden, ist es dem `XmlSerializer` jedoch unmöglich, mit der resultierenden Mehrdeutigkeit umzugehen. Ebenso verhält es sich mit der Referenzierung von Schnittstellen oder abstrakten Basisklassen.

Listing 3.9: Ergebnis der .Net XML Serialisierung

```
<?xml version="1.0"?>
<TestObject xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <MyStringProperty>HelloWorld</MyStringProperty>
  <MyFloatListProperty>
    <float>0.1</float>
    <float>0.5</float>
    <float>0.7</float>
  </MyFloatListProperty>
</TestObject>
```

Auch wenn der von .Net bereitgestellte `XmlSerializer` für die Handhabung von reinen Datenstrukturen bestens geeignet ist, so kommt er aufgrund der genannten Einschränkungen nicht für eine allgemeine Verwendung in Frage. Eine mögliche Alternative stellt der *BinaryFormatter* des .Net Frameworks dar, welcher Objekte in binärer Form serialisiert (Lst. 3.10, Abb. 3.11).

Anders als beim `XmlSerializer` wird hier rein auf Datenebene und unabhängig von Sichtbarkeitsregeln einzelner Klassenmember gearbeitet; die zu serialisierenden Daten werden direkt aus den einzelnen Feldern eines Objekts extrahiert, unabhängig davon ob diese als öffentlich deklariert sind oder nicht. Durch den Verzicht auf jegliche Interaktion mit Properties werden im De/Serialisierungsprozeß keine Getter- oder Setter Methoden ausgeführt, was einer Verfälschung der Daten vorbeugt. Auch die Behandlung von mehrdeutigen Referenzen gestaltet sich problemlos, ebenso wie die Bewahrung der Identität von Objekten, die mehrfach referenziert werden.

Listing 3.10: Binäre .Net Serialisierung

```
TestObject data = new TestObject();

BinaryFormatter bin = new BinaryFormatter();
using (FileStream f = File.OpenWrite("test.bin"))
{
  bin.Serialize(f, data);
  // data = xml.Deserialize(f) as TestObject;
}
```

Typischerweise zeichnen sich binäre Serialisierungsverfahren durch eine hohe Performanz gegenüber »von Menschen lesbaren« Verfahren aus, da sowohl

die Konvertierung der Daten in lesbare Werte als auch die Strukturierung dieser gemäß ihrer jeweiligen Bedeutung entfällt. In vielen Fällen ist es bereits ausreichend, einzelne Variablenwerte völlig unstrukturiert aneinander zu reihen - solange sie nur in derselben Reihenfolge wieder eingelesen werden wie sie geschrieben wurden. Hier liegt allerdings auch der größte Nachteil vieler binärer Serialisierungsverfahren: Eine einzelne Änderung an den zugrundeliegenden Datenstrukturen ist ausreichend, um alle vorher serialisierten Datenpakete unlesbar zu machen!

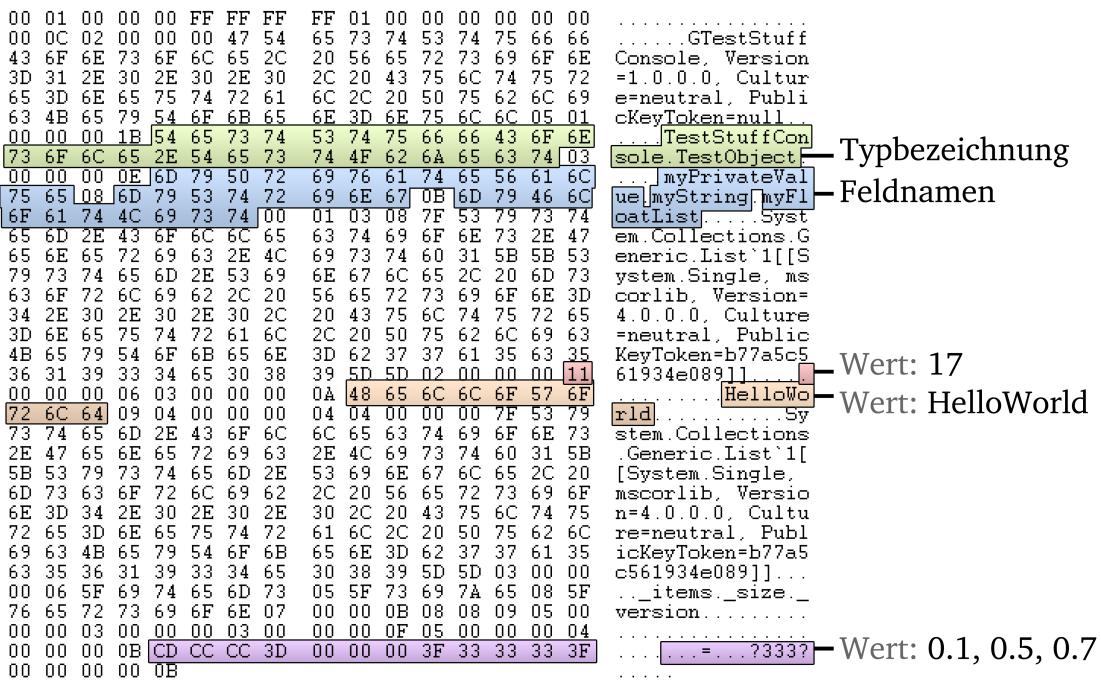


Abbildung 3.11: Ergebnis der binären .Net Serialisierung

Der BinaryFormatter umgeht dieses Problem, indem er zusätzlich zu den reinen Nutzdaten auch Metadaten im Datenstrom ablegt, die auf verwendete Datentypen verweisen und einzelne Werte ihren jeweiligen Feldern innerhalb der ursprünglichen Klassendefinition zuweisen. So können in der Vergangenheit serialisierte Daten auch dann noch erfolgreich deserialisiert werden, wenn sich die Reihenfolge der Felddefinitionen einer Klasse geändert haben, Felder entfernt wurden, oder neue hinzugefügt. Die Robustheit der binären Serialisierung des .Net Frameworks stößt allerdings dann an ihre Grenzen, wenn die Klasse eines zu deserialisierenden Objekts nicht auffindbar ist - etwa, weil sie gelöscht oder umbenannt wurde - oder der Datentyp eines serialisierten Feldes ausgetauscht wurde. In beiden Fällen wird die Leseoperation abgebrochen und eine entspre-

chende Exception geworfen. Unabhängig von der genauen Fehlerposition oder dem Ausmaß der Fehlfunktion sind damit sämtliche Daten verloren! In der Praxis würde dies bedeuten, dass die unachtsame Entfernung einer einzelnen Klasse den kompletten Verlust sämtlicher Levels und Spielstände bedeuten könnte - selbst wenn das Fehlen der entsprechenden Klasseninstanzen keine wesentlichen Auswirkungen auf das Gesamtsystem hätte. Dieses Verhalten ist für einen so dynamischen Prozess wie die Spieleanwendung untragbar - womit auch der BinaryFormatter für eine weitläufige Nutzung wegfällt.

Aufgrund der Tatsache, dass keine der beiden Lösungen des .Net Frameworks die gestellten Anforderungen vollständig erfüllen kann, erschien es sinnvoll, ein eigenes Serialisierungsverfahren zu entwickeln. Das dabei verwendete Format wird über eine allgemeine *Formatter* Basisklasse abstrahiert und findet seine konkrete Ausprägung in den *XmlFormatter* und *BinaryFormatter* Klassen¹¹. Die Information darüber, welche dieser Implementierungen in einem bestimmten Fall aktiv ist, wird konsequent versteckt. Da es für die Benutzung des Systems auch gar nicht nötig ist, Kenntnis darüber zu haben, kann so ein gewisses Maß an Flexibilität erkauft werden (Lst. 3.11). Dieses kommt später dem Nutzer zugute, der sich nach Belieben für eine der zur Verfügung stehenden Methoden entscheiden kann. Ebenfalls wäre es durch den zusätzlichen Abstraktionslevel möglich, weitere Serialisierungsverfahren nachträglich hinzuzufügen.

Listing 3.11: Duality Serialisierung

```
TestObject data = new TestObject();

using (FileStream f = File.OpenWrite("test.res"))
{
    // Auto-select an appropriate formatting method
    using (Formatter serializer = Formatter.Create(f))
    {
        serializer.WriteObject(data);
        // data = serializer.ReadObject(f) as TestObject;
    }
}
```

Beide derzeit verfügbaren Implementierungen arbeiten trotz ihrer unterschiedlichen Formatierung nach denselben Grundsätzen und erfüllen alle der obigen Anforderungen in ausreichendem Maß. In vereinfachter Darstellung kann die

¹¹Diese liegen natürlich in einem entsprechenden Duality Namespace, sodass zwischen dem eigens implementierten BinaryFormatter und der .Net Variante keine Kollision stattfindet. Die vollständigen Namen der beiden Klassen lauten *Duality.Serialization.BinaryFormatter* und *System.Runtime.Serialization.BinaryFormatter*. In den folgenden Paragraphen geht es natürlich um die Eigenimplementierung des BinaryFormatters, nicht die .Net Variante.

Vorgehensweise beim Schreiben von Objekten wie folgt beschrieben werden:

1. Per Reflection Mechanismus wird der Typ des aktuellen Objekts bestimmt, der über weitere Schritte entscheidet.
2. Primitive Datentypen werden direkt geschrieben, Klassen und Datenstrukturen werden weiter untersucht, um eine Liste aller relevanten Felder zu erstellen. Arrays entsprechen der N-fachen Behandlung ihres Elementtyps.
3. Wurde eine Feldliste erstellt, erfolgt für jeden Eintrag ein rekursiver Aufruf der allgemeinen Serialisierungsmethode.

Beim Schreiben der einzelnen Einträge werden umfangreiche Metadaten bezüglich verwendeter Typen und Zuordnungen von Datenfeldern hinterlegt (Lst. 3.12, Abb. 3.12). Diese beschreiben die Struktur der serialisierten Objekte vollständig genug, um bei Bedarf auch dann noch eine Deserialisierung durchführen zu können, wenn sich die entsprechende Klassendefinition grob verändert hat oder nicht länger auffindbar ist. Zwar können bei der Instantiierung von Objekten weiterhin Probleme auftreten - das Serialisierungsformat an sich ist damit jedoch komplett unabhängig von den ursprünglichen Klassendefinitionen und kann in jedem Fall vollständig gelesen werden.

Treten bei der Rekonstruktion einzelner Objekte Fehler auf, werden diese über zur Verfügung stehende Logging Funktionalität¹² gemeldet und nehmen im Folgenden einen Standardwert an - je nach Typ also eine einfache Nullreferenz oder eine »Null-initialisierte« Datenstruktur. Der restliche Deserialisierungsprozess wird davon nicht beeinträchtigt. Kann also beispielsweise die Nutzerkomponente eines GameObjects nicht mehr gelesen werden, wird anstelle ihrer Instanz eine Nullreferenz angenommen, das GameObject selbst sowie die umschließende Szene werden allerdings nicht betroffen. Ein unverhältnismäßiger Datenverlust wie bei der binären .Net Serialisierung tritt hierbei nicht ein, auch wenn es natürlich möglich ist, dass bestimmte Sektionen des Nutzercodes nicht adäquat auf unerwartete Nullreferenzen reagieren. Diese Angelegenheit liegt jedoch im Verantwortungsbereich des Nutzers.

Die Bewahrung einzelner Objektidentitäten sowie die Unterscheidung zwischen Objekten und Objektreferenzen kann durch einen einfachen Trick gewährleistet werden: Beim ersten Schreiben eines identitätstragenden Objekts wird diesem eine eindeutige ID zugewiesen und die Zuordnung in einem temporären

¹²Log: Ein als Text kodierter Datenstrom, in den bei Bedarf kontinuierlich Nachrichten geschrieben werden können.

Listing 3.12: Ergebnis der Duality XML Serialisierung

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <object dataType="Class" type="Duality.DualityApp+TestObject" id="1">
    <myPrivateValue dataType="Int">17</myPrivateValue>
    <myString dataType="String">HelloWorld</myString>
    <myFloatList dataType="Class" type="System.Collections.Generic.List`1[[System.←
      Single]]" id="2">
      <_items dataType="Array" type="System.Single[]" id="3" length="4">
        <object dataType="Float">0.1</object>
        <object dataType="Float">0.5</object>
        <object dataType="Float">0.7</object>
        <object dataType="Float">0</object>
      </_items>
      <_size dataType="Int">3</_size>
      <_version dataType="Int">3</_version>
    </myFloatList>
  </object>
</root>
```

Objektregister zwischengespeichert. Kommende Schreibversuche prüfen nun zunächst, ob das zu schreibende Objekt bereits in diesem Register hinterlegt wurde; ist dies der Fall, wird anstelle des Objekts nur ein Rückverweis auf die zugeordnete ID gespeichert. Im Deserialisierungsprozess wird das Register anhand bereits gelesener Objekte nach und nach wieder aufgebaut und zur Auflösung der entsprechenden Objektverweise genutzt. Diese Vorgehensweise verhindert nicht nur das unabsichtliche Klonen mehrfach referenzierter Objekte, sondern ermöglicht auch die korrekte Handhabung von Schleifen im zu serialisierenden Objektgraph.

Auch wenn eine vollautomatische Serialisierung von Objekten eine Menge Arbeit bei der Implementierung erspart, so gibt es gerade bei der Ressourcenverwaltung auch Fälle, in denen eine manuelle Handhabung notwendig ist. Die Ressourcenklasse *Texture* beinhaltet zum Beispiel neben den üblichen Konfigurationsdaten auch ein OpenGL Handle, mit dem auf die treiberseitige Repräsentation der Texturdaten verwiesen wird. Dieses kann - wie auch andere externe Systemressourcen - nicht auf dem üblichen Weg serialisiert werden, da hier auf Daten verwiesen wird, die weder im Einflussbereich der Game Engine liegen, noch auf sinnvolle Art und Weise persistent gespeichert werden können. Es muss also mindestens möglich sein, einzelne Felder einer Klasse von der Serialisierung auszuschließen, um deren Inhalte bei Bedarf manuell zu rekonstruieren. Aus diesem Grund gibt es das sogenannte *NonSerialized* Feldattribut: Im automatisierten Serialisierungsprozess werden nur diejenigen Felder einer Klasse als relevant betrachtet, die kein solches Attribut tragen.

Darüber hinaus kann es allerdings wünschenswert sein, bestimmte Objekt-

15 42 69 6E 61 72 79 46 6F 72 6D 61 74 74 65 72	.BinaryFormatter	
48 65 61 64 65 72 02 00 08 00 00 00 00 00 00 00	Header.....	
01 18 00 B4 01 00 00 00 00 00 00 00 1D 44 75 61 6C	Dual	
69 74 79 2E 44 75 61 6C 69 74 79 41 70 70 2B 54	ity.DualityApp+T	
65 73 74 4F 62 6A 65 63 74 01 00 00 00 00 00 FF	estObject.....	
FF FF FF FF FF FF FF 03 00 00 00 0E 6D 79 50 72	myFr	
69 76 61 74 65 56 61 6C 75 65 0C 53 79 73 74 65	ivateValue.Syste	
6D 2E 49 6E 74 33 32 08 6D 79 53 74 72 69 6E 67	m.Int32 myString	
0D 53 79 73 74 65 6D 2E 53 74 72 69 6E 67 0B 6D	System.String m	
79 46 6C 6F 61 74 4C 69 73 74 32 53 79 73 74 65	yFloatList2Syste	
6D 2E 43 6F 6C 6C 65 63 74 69 6F 6E 73 2E 47 65	m.Collections.Ge	
6E 65 72 69 63 2E 4C 69 73 74 60 31 5B 5B 53 79	neric.List`1[[Sy	
73 74 65 6D 2E 53 69 6E 67 6C 65 5D 5D 01 06 00	stem.Single]].....	
0C 00 00 00 00 00 00 00 11 00 00 00 01 16 00 13	Id.....	
00 00 00 00 00 00 00 00 0A 4B 65 6C 6C 6F 57 6F 72	25y	
6C 64 01 18 00 E2 00 00 00 00 00 00 00 32 53 79	stem.Collections	
73 74 65 6D 2E 43 6F 6C 6C 65 63 74 69 6F 6E 73	.Generic.List`1[
2E 47 65 6E 65 72 69 63 2E 4C 69 73 74 60 31 5B	[System.Single]]	
5B 53 79 73 74 65 6D 2E 53 69 6E 67 6C 65 5D 5D	02 00 00 00 00 00 FF FF FF FF FF FF 03 00
00 00 06 5F 69 74 65 6D 73 0F 53 79 73 74 65 6D_items.System	
2E 53 69 6E 67 6C 65 5B 5D 05 5F 73 69 7A 65 0C	.Single[]._size.	
53 79 73 74 65 6D 2E 49 6E 74 33 32 08 5F 76 65	System.Int32._ve	
72 73 69 6F 6E 0C 53 79 73 74 65 6D 2E 49 6E 74	rsion.System.Int	
33 32 01 17 00 34 00 00 00 00 00 00 00 00 0F 53 79	32...4.....Sy	
73 74 65 6D 62 6E 53 69 6E 67 6C 65 5B 5D 03 00 00	stem.Single[].....	
00 01 00 00 00 04 00 00 00 CD CC CC 3D 00 00 00 00	
3F 33 33 33 3F 00 00 00 00 01 06 00 0C 00 00 00 00	2333?.....	
00 00 00 00 03 00 00 00 01 06 00 0C 00 00 00 00 00	Wert: 0.1, 0.5, 0.7	
00 00 00 03 00 00 00 00	

Abbildung 3.12: Ergebnis der binären Duality Serialisierung

typen komplett manuell zu serialisieren, etwa um Vorverarbeitungsschritte wie Bildkompression durchzuführen. Zu diesem Zweck überprüft der jeweilige Formatter, ob ein zu serialisierendes Objekt die *ISerializable*¹³ Schnittstelle implementiert und nutzt gegebenenfalls die dort definierten Lese- und Schreibmethoden anstelle des bereits beschriebenen Reflection Automatismus (Abb. 3.13). Um die Abstraktion der Serialisierung von konkreten Datenformaten aufrecht zu erhalten, erlaubt die *ISerializable* Schnittstelle keinen direkten Zugriff auf Lese- und Schreibfunktionalität sondern bietet stattdessen *IDataReader* bzw. *IDataWriter* Schnittstellen an, die der jeweilige Formatter entsprechend implementiert. Diese ermöglichen das Hinterlegen bzw. Abfragen von beliebigen (Objekt-) Daten unter Angabe einer als Schlüssel fungierenden Zeichenkette. Diese ist frei wählbar und dient lediglich dazu, beim Serialisieren geschriebene Daten während der Deserialisierung wieder korrekt zuordnen zu können.

Eine manuelle Serialisierung mittels *ISerializable* ist in den meisten Fällen ausreichend, steht jedoch nur für Klassen zur Verfügung, die speziell im Hinblick auf eine Verwendung innerhalb der Duality Engine entworfen wurden. Um eine manuelle Handhabung von Library- und .Net Klassen zu erlauben, wurde darüber hinaus das Konzept der *Surrogates* eingeführt. Diese funktionieren analog zu *ISerializable* Implementierungen, dienen jedoch dazu, das Serialisierungsver-

¹³Wird im Duality Namespace definiert und ist nicht zu verwechseln mit der .Net Schnittstelle *System.Runtime.Serialization.ISerializable*

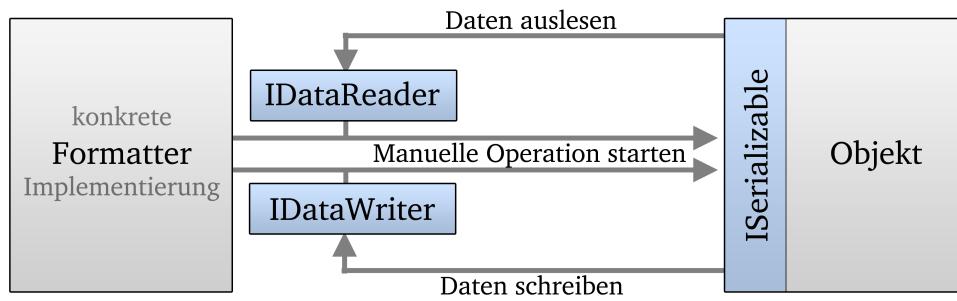


Abbildung 3.13: Manuelle Serialisierung mit ISerializable

fahren anderer Objekttypen zu überschreiben. Sie werden global registriert und ermöglichen so die manuelle Handhabung externer Datentypen.

3.5 Rendering

Rendering bezeichnet den Prozess der grafischen Darstellung eines virtuellen räumlichen Modells.^[18] Dieses wird auch als Szene bezeichnet und ist aus den vorliegenden Spielinhalten sowie den gegenwärtigen Simulationsdaten der Spielwelt ableitbar. Typischerweise lässt sich Rendering in eine Abfolge von Teilaufgaben zerlegen:

Generierung und Optimierung des darzustellenden räumlichen Modells: Es muss festgestellt werden, welche Objekte darzustellen sind, in welcher räumlichen Beziehung diese zueinander stehen und wie genau sich die einzelnen Objekte darstellen lassen. Ein *Szenengraph* enthält all diese Informationen und kann von der Game Engine verwaltet werden - jedoch ist es notwendig, die entsprechenden Daten in eine Form zu bringen, die der verwendeten Grafiksschnittstelle geläufig ist. Konkret bedeutet dies, den Szenengraph in eine Sequenz von sogenannten *Batches* aufzulösen. Ein Batch bezeichnet Vertexdaten, die mit Rendering Parametern wie beispielsweise den zur Darstellung verwendeten Texturen oder Shadern versehen wurden.

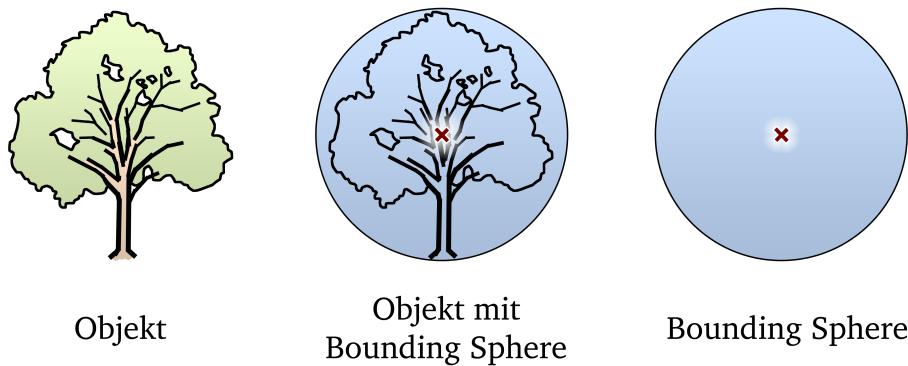


Abbildung 3.14: Bounding Sphere Repräsentation eines Objekts

Bei der Auflösung des Szenengraphs kann weiterhin eine grobe Form von *Culling* durchgeführt werden. Dabei wird versucht, die Performance des Renderingvorgangs zu verbessern, indem derzeit nicht im Sichtfeld befindliche Objekte direkt aussortiert werden. Das Feststellen der Sichtbarkeit sollte so wenig Zeit wie möglich beanspruchen und daher möglichst effizient approximiert werden. Ein einfacher Ansatz ist es, jedes Objekt durch seine umschließende *Bounding Sphere* zu repräsentieren und diese auf Sichtbarkeit zu prüfen (Abb. 3.14).

Übertragung der gesammelten Vertex- und Zustandsdaten an die Grafikkarte. Gegebenenfalls können an dieser Stelle weitere Optimierungsschritte durchgeführt werden, um die Daten- und Übertragungslast möglichst gering zu halten. Für die Interaktion zwischen Game Engine und Grafikkarte wird im Rahmen des Duality Projekts die offene Schnittstelle OpenGL^[20] verwendet. Eine gleichwertige Alternative stellt das von Microsoft entwickelte Direct3D^[21] dar.

Transformation und Projektion der Vertexdaten in den Koordinatenraum des Ausgabemediums. Üblicherweise beschreiben die in roher Form vorliegenden Vertexdaten ein Objekt relativ zu seinem Mittel- oder lokalen Referenzpunkt: Sie liegen im Objektraum vor. Im Verlauf des Renderings gilt es, diese zunächst in den Welt- und anschließend den Beobachterraum zu übertragen. Mit einer perspektivischen oder orthografischen Projektion können die Vertexdaten schließlich auf das Koordinatensystem des Sichtfeldes abgebildet werden (Abb. 3.15). Der Wechsel zwischen zwei Koordinatenräumen kann durch eine einfache Multiplikation mit der jeweiligen Transformationsmatrix erfolgen.

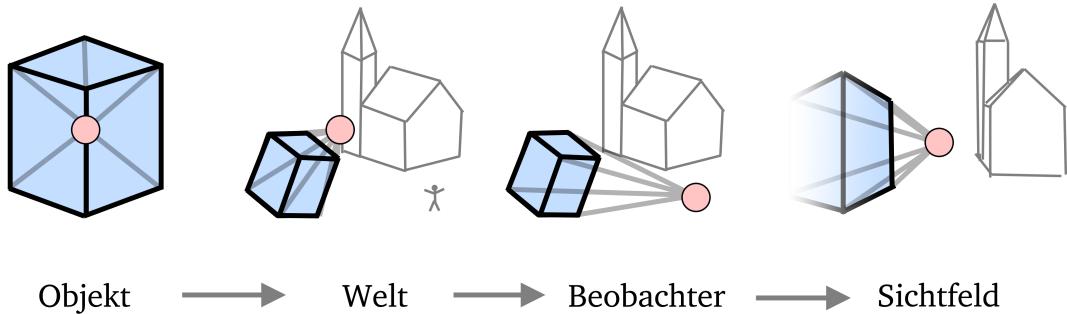


Abbildung 3.15: Eine Auswahl der Koordinatenräume

Aufgrund der großen Datenmengen komplexer 3D Modelle wird die Transformation der Vertexdaten normalerweise auf die Grafikkarte ausgelagert, um deren ausgeprägte Rechenkapazität auszunutzen und so die Performance zu steigern.

Game Engines, die sich auf zweidimensionale Spiele spezialisiert haben, stellen allerdings einen Sonderfall dar: Weil die meisten Objekte als einfache Sprites¹⁴ dargestellt werden und lediglich vier Vertexpunkte benötigen, sind die zu behandelnden Datenmengen eher klein. Zudem ist es

¹⁴Texturierte Rechtecke

in zweidimensionalen Spielen durchaus üblich, die beschriebenen Koordinatentransformationen objektweise zu modifizieren, beispielsweise um perspektivische Verschiebung vorzutäuschen oder Kachelungseffekte umzusetzen. Die für eine Verarbeitung auf der Grafikkarte ideale Konstellation »Viele Daten, wenige Ausnahmen« ist für zweidimensionale Spiele also nicht in jedem Fall gegeben. Es kann daher sinnvoll sein, einen Teil der Vertextransformation manuell durchzuführen.

Rasterisierung¹⁵ der vorliegenden Daten sowie die Berechnung konkreter Farbwerte einzelner Pixel des Ergebnisbildes unter Einbeziehung verschiedener Zustands- und Vertexdaten. Neben einfacher Texturierung können auch Lichtberechnungen und andere per-Pixel Effekte ausgeführt werden.

Nachbearbeitung des resultierenden Bildes mittels Techniken der Bildverarbeitung. Man spricht dabei auch von »Postprocessing«. Eine häufige Anwendung findet sich in Korrekturen von Farbton, Sättigung, Helligkeit und Kontrast zur dynamischen Veränderung der Stimmung einer bestimmten Szene.

Als Knotenpunkt für die oben aufgeführten Teilaufgaben des Renderings kann eine als Komponente implementierte *Camera* Klasse dienen. Diese repräsentiert den Beobachter einer Szene und bietet eine öffentliche Schnittstelle an, mit der diese in ihrem aktuellen Zustand aus Sicht der Kamera gerendert werden kann. Auf Seite der Engine werden zur Darstellung der Spielwelt also die derzeit aktiven Kamera Objekte ermittelt, für die der grob beschriebene Rendervorgang dann jeweils einmal durchlaufen wird.

Wie zuvor beschrieben gilt es zunächst, den vorliegenden Szenengraph in eine Sequenz von Daten aufzulösen, die zur Darstellung an die Grafikkarte übertragen werden kann. Diese Aufgabe könnte der Kamera selbst anvertraut werden, denn diese ist schließlich für die Darstellung der Szene zuständig und vereint das Know-How des Renderings in sich. Was die Darstellung eines konkreten Objekts nun ausmacht, bleibt dabei allerdings offen: Handelt es sich um eine einfarbige geometrische Form? Ein animiertes Sprite? Ein Partikelsystem? Die Implementierung der Kamera weiß, »Wie« etwas darzustellen ist, jedoch nicht »Was«. Ein Objekt dagegen kennt das »Was«, sollte sich jedoch nicht um das »Wie« kümmern müssen. Um diese konzeptionelle Trennung auch im Software-design aufrecht zu erhalten, bietet sich eine Abstraktionsebene zwischen Kamera

¹⁵ Abbildung von Vektordaten auf ein diskretes Pixelraster^[19]

und darzustellendem Objekt an. In Duality setzt sich diese im Wesentlichen aus zwei Schnittstellen zusammen: *IDrawDevice* und *ICmpRenderer* (Abb. 3.16).

Die *IDrawDevice* Schnittstelle wird von der Kamera selbst implementiert und stellt Methoden zur Verfügung, mit denen parametrisierte Vertexdaten im laufenden Renderingvorgang registriert werden können. Zu den verfügbaren Parametern zählen neben dem Vertexmodus¹⁶ auch die zu verwendenden Texturen, Shader und sonstige Zustandsvariablen. Zur Unterstützung einer bei Bedarf manuellen Vertextransformation befinden sich die am *IDrawDevice* registrierten Vertices bereits im Koordinatenraum des Beobachters.

Auf der anderen Seite wird die *ICmpRenderer* Schnittstelle von allen Komponentenklassen implementiert, die etwas darstellen wollen: Sie dient als Erkennungszeichen der jeweiligen Objekte im Szenengraph und bietet Methoden zur Feststellung ihrer Sichtbarkeit (*IsVisible*) sowie zur Anforderung besagter Vertexdaten (*Draw*) an. Durch einen sequentiellen Aufruf der *Draw* Methode in allen sichtbaren Renderer Komponenten können nun sämtliche für die Darstellung der Szene notwendigen Informationen gesammelt werden.

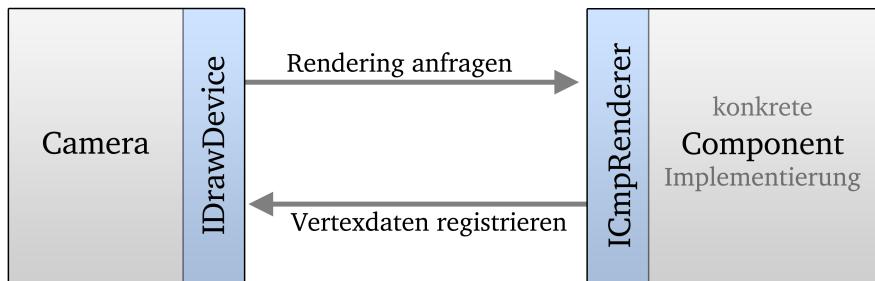


Abbildung 3.16: Die Abstraktion des Renderings

»Was« dargestellt wird, ist in den übermittelten Vertexdaten sowie deren Parametrisierung kodiert - und liegt damit im Verantwortungsbereich der konkreten Implementierung eines *ICmpRenderers*. »Wie« diese Vertexdaten im Folgenden verwendet werden, ist Sache der Kamera und für den Nutzer irrelevant, solange das Endergebnis seinen Erwartungen entspricht. Dieser Umstand schafft einen gewissen Freiraum bei der Implementierung der Kamera und bietet verschiedene Erweiterungs- und Optimierungsmöglichkeiten an.

In einem ersten Ansatz zur Handhabung eingehender Vertexdaten könnte die Implementierung der Kamera eine direkte Weitergabe der Daten an die Grafikkarte vorsehen, beispielsweise durch die Nutzung des mittlerweile ausgemu-

¹⁶Points, Lines, Triangles, Quads, ...

sterten *Immediate Mode* von OpenGL oder durch das wiederholte Befüllen und Rendern eines Vertex Buffers je Aufruf. Es ergibt sich jedoch ein Problem, sobald *Blending* ins Spiel kommt, denn zur korrekten Darstellung halbtransparenter Flächen müssen diese in sortierter Reihenfolge gezeichnet werden¹⁷ - die Registrierung der Vertexdaten erfolgt jedoch unsortiert. Eine Vorsortierung der darzustellenden Objekte würde zwar in einigen Fällen Abhilfe schaffen, könnte jedoch keine verlässliche Sortierung der generierten Vertices garantieren: Nicht alle Renderer Komponenten gehören zu Objekten die räumlich lokalisierbar sind und auch bei den anderen gibt es keine Garantie dafür, dass die Tiefenwerte registrierter Vertexdaten mit denen des zugehörigen Objekts übereinstimmen. Das direkte Rendern aller registrierten Daten ist daher unter den gegebenen Umständen nicht praktikabel.

Durch eine Trennung von Datensammlung und Rendering kann das beschriebene Problem gelöst werden: Anstatt alle eingehenden Vertexdaten direkt zu bearbeiten, können diese zunächst in jeweils einer *DrawBatch* Datenstruktur zusammengefasst und für die spätere Verwendung gespeichert werden (Abb. 3.17). In einem Zwischenschritt werden alle DrawBatches anhand der Tiefenwerte ihrer Vertexdaten sortiert, um ein korrektes Blending bei der anschließenden Verarbeitung zu ermöglichen. Damit steht ein erstes, funktionierendes Verfahren zur Verfügung, das im Folgenden weiter optimiert werden kann.

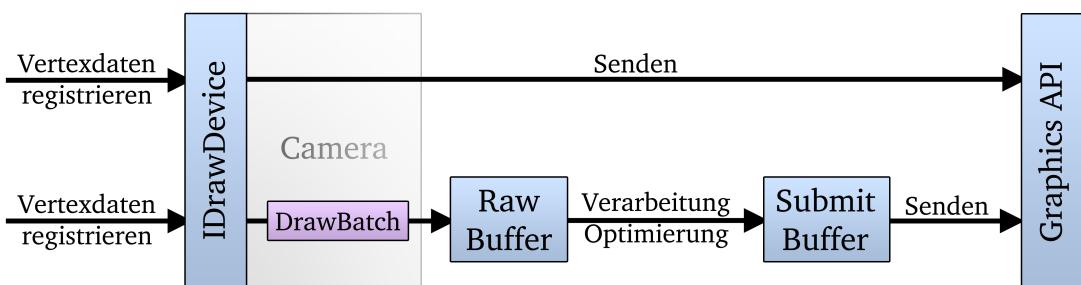


Abbildung 3.17: Direktes und Indirektes Rendering

In einer Testszene mit rund 2300 als Sprites dargestellten Objekten benötigt ein einzelner Frame¹⁸ eine Rechenzeit von etwa 16.13 ms (Abb. 3.21), womit eine erste Performancegrenze bereits erreicht ist: Bei einem Ziel von 60 FPS¹⁹ stehen

¹⁷Blending unterscheidet zwischen Vorder- und Hintergrund: Werden diese durch eine Umkehrung der Zeichenreihenfolge vertauscht, ergeben sich andere Farbwerte.

¹⁸Ein Frame bezeichnet einen vollständigen Berechnungszyklus der Anwendung inklusive Update und Rendering.

¹⁹Frames Per Second

pro Frame maximal 16.66 ms zur Verfügung. Behält man die vergleichsweise geringe Zahl von Vertices ($2300 * 4 = 9200$) und verwendeten Texturen im Auge, erscheint die gemessene Zeit unverhältnismäßig hoch. Die Vermutung liegt nahe, dass ein Großteil der Rechenzeit nicht durch die Komplexität des eigentlichen Renderings zustande kommt, sondern durch die Strukturierung der zugrundeliegenden Daten: Im bisherigen Ansatz wird für jeden einzeln registrierten Satz von Vertexdaten ein eigener DrawBatch angelegt, welcher individuell verwaltet, sortiert und dargestellt wird. Da sich jedes der 2300 Objekte der Testszene nur um seine eigene Darstellung kümmert, fallen 2300 separate DrawBatches mit je vier Vertexpunkten an - allein die Tiefensortierung dieser nimmt rund 3.71 ms in Anspruch. Zudem liegen die Daten in einer für die Grafikkarte denkbar ungünstigen Form vor, denn jeder DrawBatch bringt seine eigene Parametrisierung mit sich und muss individuell behandelt werden: Für ein Minimum an Daten wird so ein Maximum an Ausnahmen geschaffen, was einer effizienten Verarbeitung im Wege steht.



Abbildung 3.18: Darstellung eines Sprites mit verschiedenen Transparenzmodi

Glücklicherweise gibt es gleich mehrere Ansatzpunkte für Optimierungen. Zunächst einmal ist eine Tiefensortierung nur dann nötig, wenn bei der Darstellung der jeweiligen Vertices eine Blending Operation durchgeführt wird, also beispielsweise Alpha- oder Additives Blending. In den meisten Fällen ist jedoch bereits das weitaus pflegeleichtere *Alpha Testing* ausreichend, das eine binäre Unterscheidung der Pixelsichtbarkeit darstellt und keine Tiefensortierung voraussetzt. Alternativ kann der *Alpha To Coverage* Modus verwendet werden, der sich im Grunde identisch verhält, zusätzlich jedoch hardwareseitiges Multisampling ausnutzt, um weichere Kanten hervorzubringen (Abb. 3.18).^[22]

Durch die flächendeckende Nutzung der genannten Techniken bei der Darstellung von Sprites lassen sich eingehende Batches in zwei Kategorien unterteilen: Jene, die Blending benötigen und jene, die ohne auskommen. Diese können in separaten Listen zwischengespeichert und nacheinander gerendert werden - eine vorherige Tiefensortierung ist jedoch nur bei der Blending Liste notwendig (Abb. 3.19). Auf diese Weise lassen sich im Testfall 3.4 ms Berechnungszeit einsparen. Die Gesamtdauer eines Frames bleibt dennoch konstant, was darauf hindeutet, dass der Flaschenhals in diesem Fall auf Seite der Grafikkarte zu verordnen ist.

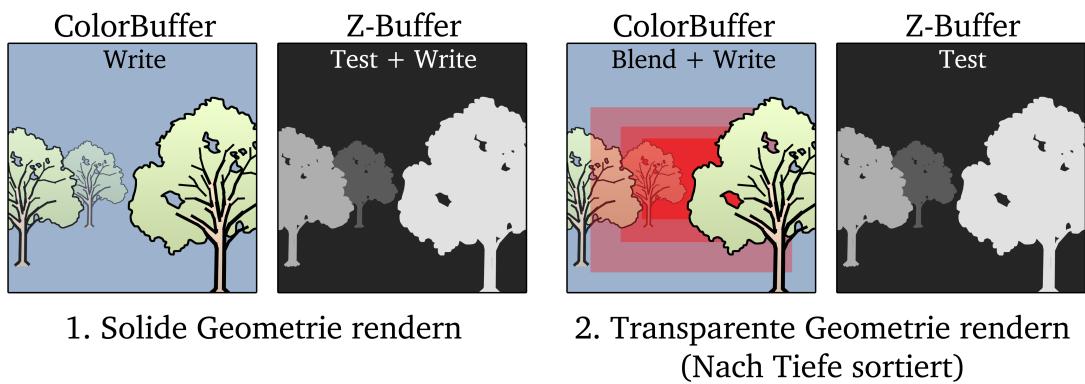


Abbildung 3.19: Rendering in zwei Schritten

Dies ist nicht weiter verwunderlich, wenn man die bisher völlig unstrukturierten Eingangsdaten betrachtet. Grafikkarten sind auf die parallele Massenverarbeitung von gleichartigen Daten ausgelegt; tendenziell sind die besten Ergebnisse also dann zu erreichen, wenn große Datenmengen auf einmal verarbeitet werden können. Momentan liegt jedoch eine große Zahl von sehr kleinen Datenmengen vor - in Form von 2300 DrawBatches à vier Vertices. Das primäre Ziel bei der weiteren Optimierung sollte also die Zusammenfassung von Batches zu möglichst wenigen, großen Einheiten sein.

Zwei DrawBatches können genau dann zusammengefasst werden, wenn sowohl ihre Parametrisierung²⁰ als auch das verwendete Vertexformat vollständig übereinstimmen. Die Zahl der Vertices sowie deren Konfiguration sind dabei unerheblich - was zählt ist allein die Art und Weise ihrer Darstellung, denn sie bestimmt den Zustand des Renderingsystems. Solange dieser nicht geändert werden muss, können beliebig viele Vertices auf einmal, also in einem einzigen Batch, verarbeitet werden. Eine einfache Vorgehensweise zur Optimierung

²⁰Verwendete Shader, Texturen und sonstige Zustandsvariablen

der zwischengespeicherten DrawBatches ist es, diese schrittweise durchzugehen und die aktuelle Instanz so lange mit der jeweils nächsten zu vereinen, wie diese kompatibel sind. Zusätzlich kann ausgenutzt werden, dass für die Liste der DrawBatches ohne Blending Features keine Tiefensortierung erforderlich ist: Indem diese stattdessen nach der Parametrisierung der enthaltenen Batches sortiert wird, kann eine maximale Zusammenfassung von Vertexdaten erreicht werden (Abb. 3.20).

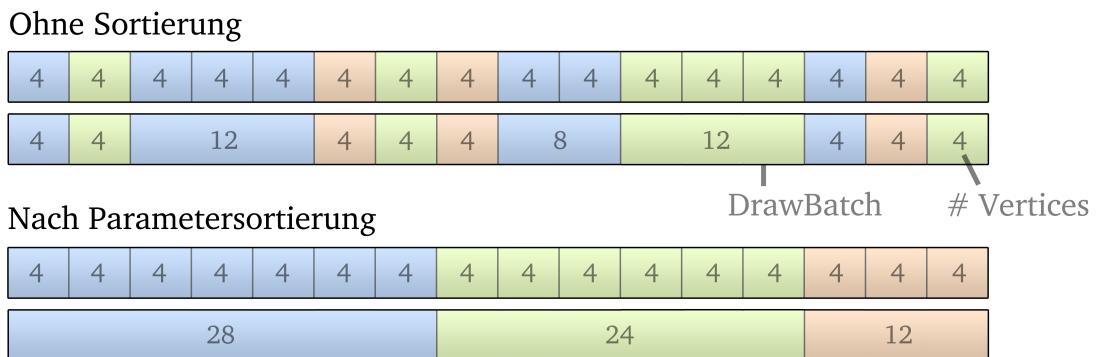


Abbildung 3.20: Optimierung von Batches mit und ohne Sortierung

Im Test zeigen sich deutliche Erfolge: Die Anzahl der DrawBatches kann von etwa 2300 auf insgesamt 23 reduziert werden und die Berechnungszeit eines Frames sinkt auf 9.81 ms. Es fällt ebenfalls auf, dass diese Zeit der Gesamtdauer aller CPU-seitigen Operationen entspricht, was darauf hindeutet, dass dort der neue Flaschenhals zu suchen ist; ferner stieg die Vorverarbeitungszeit der zwischengespeicherten Batchdaten um 4.16 ms an, was auf den neuen Optimierungsschritt zurückzuführen ist, der mit einer relativ großen Datenmenge konfrontiert wird. Um diese von vornherein gering zu halten, kann zusätzlich eine *Just In Time* Optimierung durchgeführt werden, die eingehende Vertexdaten wenn möglich direkt zusammenfasst, ohne überhaupt erst einen neuen DrawBatch zu erzeugen. Auf diese Weise können nochmals 2.39 ms pro Frame eingespart werden.

Mit den beschriebenen Verfahren lässt sich die Berechnungszeit eines Frames im Testfall auf bis zu 7.42 ms verringern. Durch algorithmische Optimierungen an verschiedenen Stellen²¹ war nach Abschluss der Testreihen eine weitere Reduktion auf 6.41 ms möglich.

²¹Parallelisierung des Sortieralgorithmus, verbesserte Gleichheitsprüfung von Batch Parametrisierungen, sonstige Änderungen

Um die gemessenen Testwerte besser einordnen zu können, wurde ein Referenzwert ermittelt, welcher die minimale Rechenzeit repräsentieren soll, in der eine vergleichbare Szene auf dem Testsystem gerendert werden kann. Dazu wurde in separater Umgebung ein einzelnes, komplexes Objekt dargestellt, das eine mit dem Testfall identische Zahl von Vertices in einem einzelnen Batch registriert. Signifikante Rechenzeiten für Objektupdates, Optimierungs- oder Verwaltungsaufgaben konnten somit ausgeschlossen werden, da trotz einer ähnlichen grafischen Komplexität nur ein einzelnes Objekt existierte. In einer Gegenprobe wurde zusätzlich die Darstellung der an die Grafikkarte übertragenen Daten blockiert, um festzustellen, welcher Anteil der gemessenen Zeit tatsächlich auf das Rendering selbst zurückzuführen ist. Als Referenzwert ergaben sich 3.71 ms pro Frame, die Gegenprobe verhielt sich mit 0.63 ms wie erwartet.

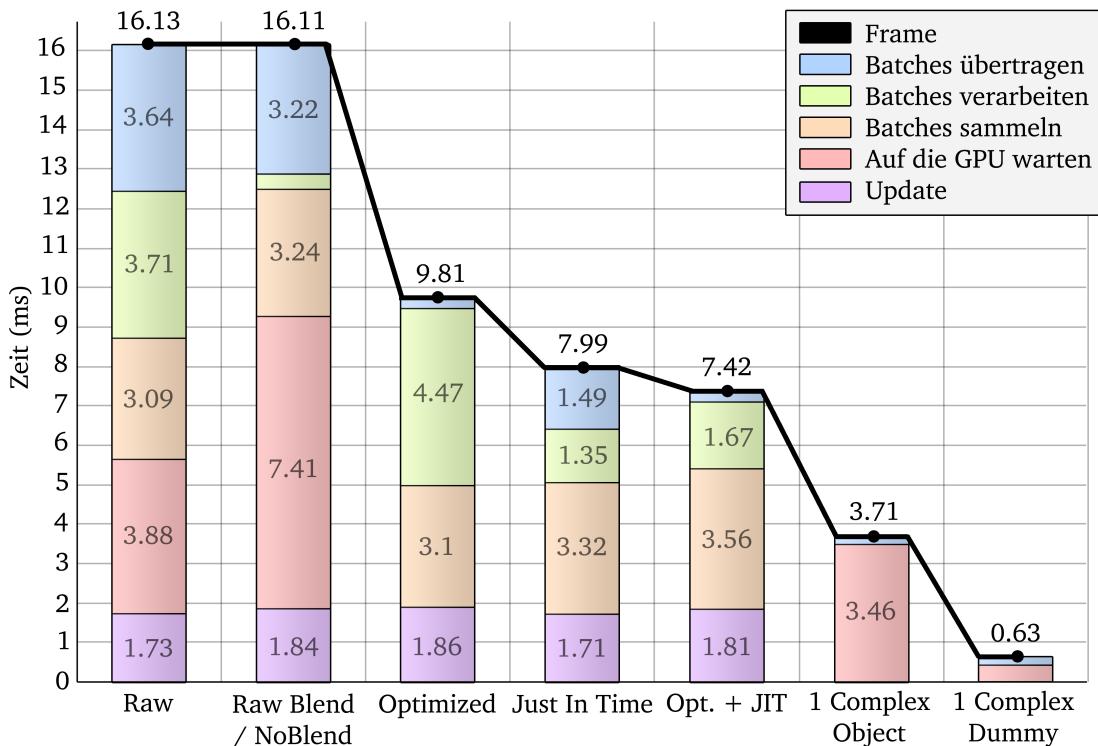


Abbildung 3.21: Performance Messwerte im Vergleich

Vergleicht man nun den Referenzwert mit der durch Optimierungen erreichten Zeit, ist dies ein durchaus zufriedenstellendes Ergebnis - wenn man bedenkt, dass die Vertexdaten im Referenzfall in vorgefertigter Idealform vorliegen, die Daten der Testszene jedoch völlig unstrukturiert eingehen. Das für die Testreihe verwendete Szenario stellt aus Sicht der Engine einen »Worst Case« dar, der

durch die implementierten Optimierungsschritte ausreichend abgedeckt werden kann. Schließlich sollte die Engine wenn möglich auch dann noch zufriedenstellend arbeiten, wenn der Nutzer sie auf nicht vorhergesehene oder suboptimale Art und Weise einsetzt.

In der Praxis wäre zu erwägen, das Rendering gleichartiger Objekte anders zu organisieren, sobald sehr große Zahlen zu erwarten sind: Für ein Partikelsystem macht es beispielsweise keinen Sinn, jeden einzelnen Partikel als vollwertiges Objekt zu modellieren. Stattdessen könnte eine »ParticleSystem« Komponente die Berechnung ihrer Partikel zentralisieren und sie ähnlich des Referenzobjekts in einem einzigen Batch auf einmal abschicken.

Eine weitere wesentliche Optimierung betrifft das Senden von Vertexdaten: Im derzeitigen Ansatz werden alle Vertexdaten dynamisch jedes Frame neu gesendet. Für besonders komplexe Objekte würde sich stattdessen das einmalige Senden der Daten in Kombination mit einer hardwareseitigen Transformation anbieten, um redundante Datenübertragung zu vermeiden. Im Kontext zweidimensionaler Spiele gibt es jedoch nur wenige Fälle, in denen sich abgeschlossene Pakete ausreichend großer Vertexzahlen ergeben, daher wurde diese Optimierung nicht implementiert.

4 | Editor

Das passende Editorsystem wurde parallel zur Duality Engine selbst entwickelt und sollte den Nutzer bei der Integration von Spielinhalten, Testing und Debugging so gut es geht unterstützen. Der Entwicklungsfokus lag dabei auf einer möglichst hohen Nutzerfreundlichkeit sowie im Speziellen der bereits genannten *Live Editing* Funktionalität: Jede Änderung an Spielinhalten oder Quellcode sollte direkt sichtbar werden und ohne ein Verlassen der Editorumgebung getestet werden können. Um die genannten Ziele erreichen zu können, wurden bereits sehr früh in der Entwicklung Überlegungen zu Aufbau und Softwaredesign notwendig.

4.1 Aufbau

Wie schon im Architektur Kapitel skizziert, wurde die Engine selbst als eine Klassenbibliothek umgesetzt, die von der *Launcher* Applikation sowie dem Editor eingebunden wird. Erstere stellt eine passive Ausführungsumgebung dar, in der ein fertiges Spiel unter Regie von Ressourcen und Engine simuliert und dargestellt werden kann. Das *Dualitor*¹ genannte Editorsystem hingegen stellt eine aktiv eingreifende Ausführungsumgebung dar: Sie bestimmt wann und in welchem Umfang die Spielwelt aktualisiert oder dargestellt wird und bietet dem Nutzer Werkzeuge an, um Objekte zur Laufzeit konstruieren, zerstören und modifizieren zu können. Ferner werden Veränderungen an geladenen Plugins und Ressourcen registriert sowie gegebenenfalls ein Neuladen dieser veranlasst. Die Engine selbst nimmt in dieser Umgebung eine eher passive Rolle ein und geht ihrer gewohnten Tätigkeit nur auf Anfrage nach.

Analog zum Pluginkonzept des Kerns basiert auch das zugehörige Editorsystem vollständig auf einer Plugin Architektur. Zur Basisfunktionalität zählen lediglich grundlegende Verwaltungsaufgaben sowie die von den Plugins genutzte

¹Die Kurzform von »Duality Editor«.

Infrastruktur. Über das zur Verfügung stehende Interface wird es diesen ermöglicht, eigene GUI² Steuerelemente im Gesamtaufbau zu registrieren. Diese sind größtenteils modular gehalten und erfüllen jeweils eine klar abgegrenzte Aufgabe, was den Austausch einzelner Editorkomponenten ermöglicht, ohne Modifikationen am Gesamtsystem vornehmen zu müssen (Abb. 4.1). Auch ist es so für den Nutzer möglich, das System nahtlos um neue Komponenten zu erweitern.

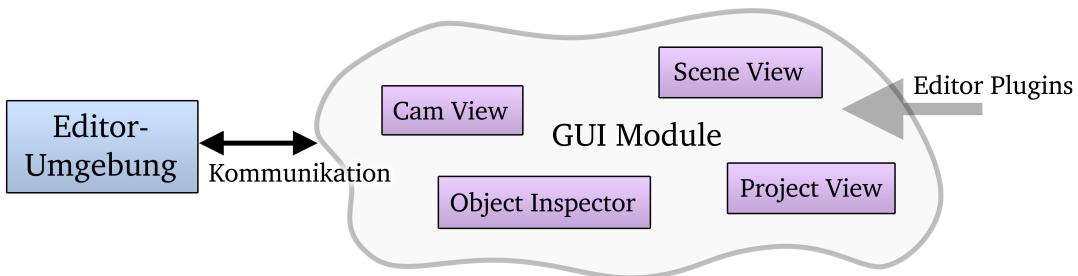


Abbildung 4.1: Basis-Editor und Zusatzmodule

Natürlich ist ein gewisses Maß an Kommunikation zwischen den einzelnen Modulen notwendig. Diese kann jedoch in vielen Fällen über die Infrastruktur des Basissystems abstrahiert werden, was eine direkte Kopplung überflüssig macht. Zwar lassen sich nicht alle Kommunikationswege auf sinnvolle Art und Weise auf eine allgemeine Editorschnittstelle abbilden, jedoch gibt es einige typische Szenarien, welche sich auf diese Weise vereinfachen lassen:

Objektselektion befasst sich mit der Handhabung von Nutzerinteraktion, bei der ein einzelnes Objekt oder eine Menge von Objekten für die weitere Bearbeitung markiert wird, beispielsweise durch eine Listenauswahl oder das Klicken auf die visuelle Repräsentation eines Objekts.

Die Informationen darüber, welche Objekte gerade ausgewählt sind, kann zunächst einmal innerhalb des jeweils betroffenen Moduls verwaltet werden: Wird ein bestimmtes Objekt in Teileditor A ausgewählt, bliebe Teileditor B davon unberührt. Jeder Teileditor des Gesamtsystems hätte eine eigene Selektion, die von allen anderen unabhängig arbeitet. Möchte nun ein Modul auf die Selektion eines anderen reagieren, muss dieses dort nachfragen - eine explizite Kopplung entsteht, welche die Wartbarkeit des Gesamtsystems verringert und zugleich dessen Erweiterbarkeit einschränkt, da zuvor unbekannte Module nicht einbezogen werden können.

²Graphical User Interface: Grafische Benutzeroberfläche

Eine alternative stellt die zentralisierte Objektselektion dar: Anstatt eine lokale Selektion innerhalb jedes Einzelmoduls zu verwalten, wird die aktuelle Auswahl von Objekten global gespeichert. Einzelne Module können diese abfragen oder per globalem Event³ auf Änderungen reagieren - unabhängig davon, woher diese Änderungen stammen. Für die Nutzerfreundlichkeit ergibt sich gleichzeitig ein positiver Nebeneffekt: Wählt er ein bestimmtes Objekt aus, reagieren simultan alle aktiven Teileditoren darauf, was zu einem konsistenten Nutzererlebnis beiträgt und den Zusammenhang verschiedener Sichten auf identische Objekte verdeutlicht.

Objektmodifikation benötigt als solche keine besondere Handhabung seitens des Editors. Es kommt jedoch nicht selten vor, dass eine bestimmte Editorkomponente wissen muss, ob sich ein Objekt oder eine bestimmte Objekteigenschaft verändert hat, beispielsweise um die GUI Ansicht des Objekts zu aktualisieren. Eine primitive Möglichkeit, dies zu erreichen, ist ein Zwischenspeichern des letzten bekannten Eigenschaftswerts in Verbindung mit einer zyklischen Überprüfung, ob dieser Wert noch aktuell ist. Diese auch als *Polling* bezeichnete Vorgehensweise schadet jedoch besonders bei großen Systemen der Performance.

Effizienter ist es, ein globales Event für Objektmodifikationen zu definieren, bei dem sich interessierte Module registrieren können, um darauf zu reagieren. Module, die selbst Veränderungen vornehmen, melden diese bei der Durchführung dem Editorsystem, welches sich darum kümmert, ein entsprechendes Event auszulösen. Die Datenübertragung vom veränderten Modul zum reagierenden Modul erfolgt auf diesem Weg ohne Kenntnis des jeweiligen Kommunikationspartners - und damit ohne direkte Koppelung und offen für Erweiterungen.

Sonstige Systemzustände, die gemeinsam von vielen Modulen genutzt werden, können ebenfalls zentral in einer allgemeinen Editorschnittstelle zur Verfügung gestellt werden. Indem diese global verfügbar gemacht werden, entfallen weitere Kommunikationswege zwischen einzelnen Modulen, da ein Austausch entsprechender Informationen indirekt über einen gemeinsamen Knotenpunkt abgewickelt werden kann.

Um die Erweiterbarkeit des Editors sicherzustellen, wurden auch standardmäßig mitgelieferte Module auf Plugin Ebene ausgelagert, fest integrierte Mo-

³Siehe auch: Observer Pattern

dule existieren nicht. Würde man tatsächlich alle Editor Plugins restlos entfernen, bliebe vom »Dualitor« System nur eine leere Hülle ohne den Großteil der Funktionalität übrig (Abb. 4.2).

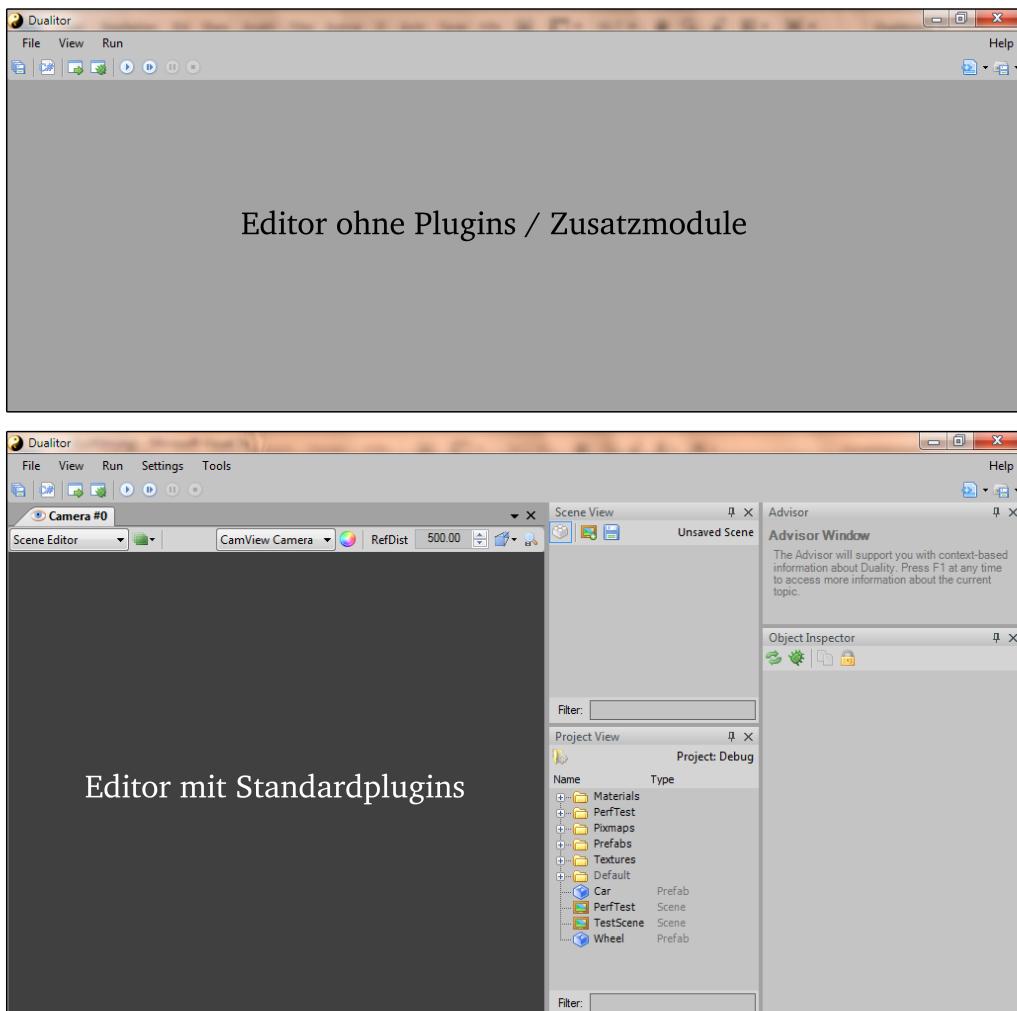


Abbildung 4.2: Das Editorsystem mit und ohne Plugins im Vergleich

4.2 Module

Auch wenn Erweiterbarkeit von Anfang an ein klares Ziel darstellte, so wäre es doch kein gutes Beispiel an Nutzerfreundlichkeit gewesen, den Nutzer all seine Editormodule selbst entwickeln zu lassen. Passend zur Grundfunktionalität der Engine sollte auch der Editor selbst in Form von Standardplugins eine gewisse Basis mitbringen, auf die der Nutzer ohne weiteres Tun zurückgreifen kann. Eine minimale Funktionalität kann dabei durch die folgenden vier GUI Module erreicht werden:

Project View

Die Projektansicht bildet eine editorseitige Darstellung der Ressourcenverwaltung. Es werden alle im Projekt verfügbaren Ressourcen in einer hierarchischen Ansicht gezeigt, welche die Ordnerstruktur der jeweiligen Dateien widerspiegelt (Abb. 4.3). Aufgrund der in Duality umgesetzten 1:1 Zuordnung zwischen Ressource und Ressourcendateien entspricht die Projektansicht damit einer gefilterten Darstellung des zugrundeliegenden Ordners im Dateisystem. Tatsächlich werden sämtliche Aktionen des Benutzers⁴ in dieser Ansicht direkt auf Dateiebene abgebildet.

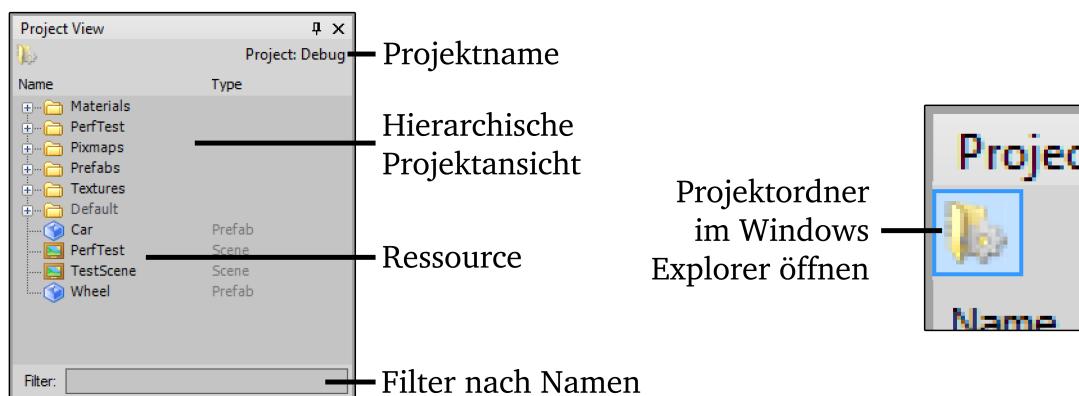


Abbildung 4.3: Die Projektansicht

Zum Erreichen einer Synchronisation von Projektansicht und Dateisystem ist es in den meisten Fällen ausreichend, die dargestellte Baumstruktur gemäß dem erwarteten Ergebnis einer Nutzeraktion zu aktualisieren: Wird beispielsweise eine Ressource gelöscht, genügt es die entsprechende Ressourcendatei zu löschen

⁴Kopieren / Einfügen, Verschieben, Umbenennen, Löschen, ...

und den Knoten der Baumansicht zu entfernen. Mit diesem Ansatz ergibt sich jedoch ein Problem, wenn die Dateisystemaktion fehlschlägt - oder der Nutzer sich entschließt, auf eigene Faust im Windows Explorer Ressourcendateien zu löschen. In beiden Fällen wäre das Ergebnis eine fehlerträchtige Inkonsistenz zwischen Dateisystem und Darstellung (Abb. 4.4).

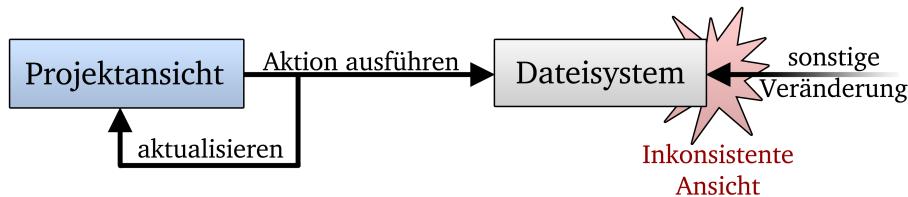


Abbildung 4.4: Inkonsistenz durch externe Veränderung

Ein verbesserter Ansatz kann Probleme dieser Art lösen, indem die Durchführung einer Aktion und die Aktualisierung der Ansicht voneinander getrennt werden: Die Entfernung einer Ressource durch den Nutzer gibt in diesem Ansatz lediglich die Löschung der zugehörigen Datei in Auftrag, nimmt jedoch keine Anpassung der Darstellung vor. Sobald besagte Löschung vom Dateisystem durchgeführt wurde, registriert ein zentraler *FileSystemWatcher*⁵ die Veränderung und löst ein globales Event aus, auf das die Projektansicht mit einer angemessenen Aktualisierung reagiert (Abb. 4.5).

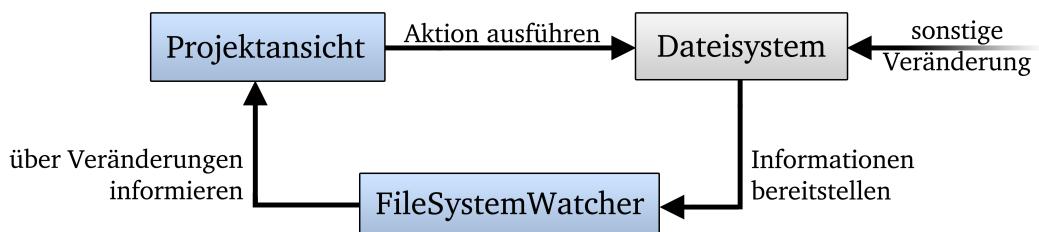


Abbildung 4.5: Indirekte Aktualisierung der Ansicht

Der große Vorteil dieses indirekten Ansatzes ist, dass es keinen Unterschied mehr macht, auf welche Art und Weise eine Aktion durchgeführt wurde - ob eine Ressource nun über die Projektansicht selbst oder den Windows Explorer gelöscht wurde, spielt keine Rolle.

⁵Eine .Net Klasse, welche bestimmte Dateien oder Ordnerstrukturen des Dateisystems auf Veränderungen überwachen kann.

Diese Indifferenz trägt für den Benutzer zur Verstärkung der Analogie zwischen Windows Explorer und Projektansicht bei - zugunsten der Nutzerfreundlichkeit sollte diese daher konsequent so weit wie möglich ausgebaut werden. Konkret kann beispielsweise der Windows Papierkorb für die Entfernung von Ressourcen einbezogen werden, anstatt die Dateien vollständig zu löschen. Weitere mögliche Anpassungen sind die Verwendung des Windows Clipboards⁶ für Kopieren / Einfügen Aktionen von Ressourcen sowie eine Integration mit DragDrop Aktionen des Windows Explorers.

Scene View

In der Szenenansicht werden alle derzeit existenten GameObjects in einer Baumansicht dargestellt (Abb. 4.6). Die verwendete Hierarchie entspricht dabei dem Szenengraph, der vom Nutzer mit verschiedenen Aktionen bearbeitet werden kann.

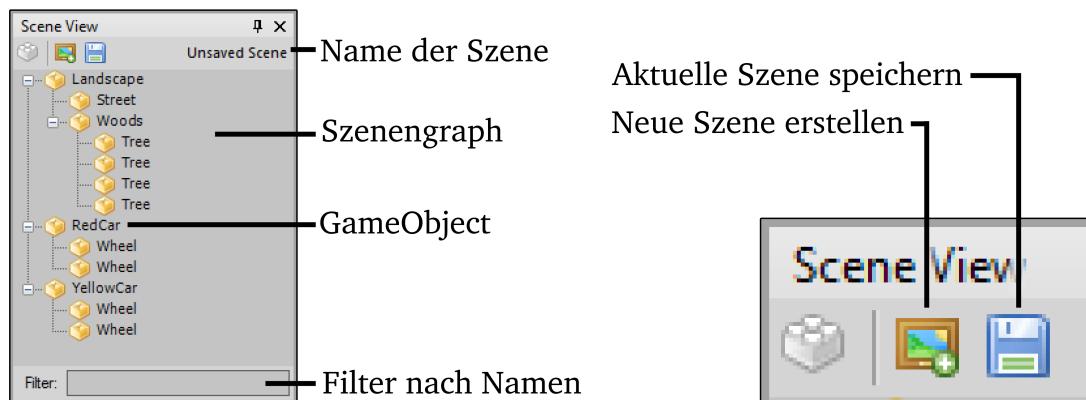


Abbildung 4.6: Die Szenenansicht

Die bereits für die Projektansicht beschriebene indirekte Aktualisierung der Darstellung wurde auch für die Szenenansicht implementiert, um auch auf Szenenveränderungen reagieren zu können, die nicht explizit vom Nutzer durchgeführt wurden. An die Stelle von Dateisystem und FileSystemWatcher tritt jedoch die derzeit aktive Szene, welche als Verwaltungsinstantz von sich aus bereits Events anbietet, um eine direkte Reaktion auf Neustrukturierungen des Szenengraphs zu ermöglichen.

⁶Ein systemglobaler Zwischenspeicher für den Austausch von Daten zwischen verschiedenen Anwendungen

Camera View

Die Aufgabe des Camera Views ist es, die aktuelle Szene aus Sicht einer frei beweglichen Kamera zu zeigen und darüber hinaus eine Interaktion zu ermöglichen, die der Bearbeitung von Objekten dient (Abb. ??). Für die Darstellung wird im Editor je Camera View ein temporäres Kameraobjekt instantiiert, das in den aktiven Bereich der Ansicht rendert; die Verfahren zur Darstellung des finalen Spiels und der Editorsicht sind damit identisch und ermöglichen ein »What You See Is What You Get« (Abk.: WYSIWYG) Erlebnis auf Nutzerseite. Der Nutzer kann eine beliebige Zahl an Kameraansichten öffnen, um die Szene gleichzeitig aus verschiedenen Blickwinkeln oder mit unterschiedlicher Darstellungsweise zu betrachten.

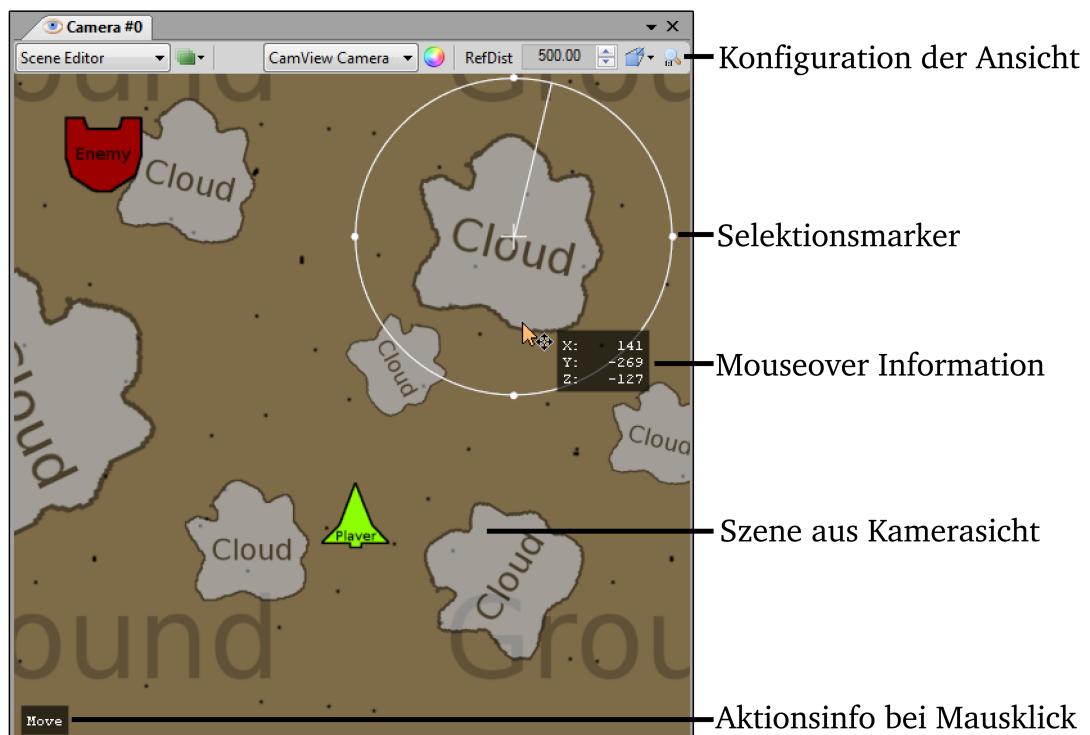


Abbildung 4.7: Die Kameraansicht

Ein wesentlicher Unterschied zwischen dem Camera View und anderen GUI Modulen ist ein erhöhter Bedarf an Erweiterungsmöglichkeiten: Die Projektansicht beispielsweise kann bereits von sich aus problemlos mit jeglicher Art von Ressourcen arbeiten und sollte in aller Regel keine dynamischen Erweiterungen benötigen, da Darstellung und Interaktion mit Ressourcen bereits klar definiert sind.

Die Kameraansicht jedoch ermöglicht die Bearbeitung der Szene oder einzelner Objekte auf sehr allgemeine Art und Weise. Neben einfachen Szenengraph Aktionen wie dem Verschieben, Rotieren oder Skalieren von Objekten ist prinzipiell auch sonst jede Form der Interaktion denkbar, für die eine WYSIWYG Umgebung einen Vorteil bedeutet - beispielsweise das Bearbeiten von Kollisionsgeometrie⁷ oder Tilemaps⁸.

Aufgrund der kaum absehbaren Zahl möglicher Anwendungsfälle erscheint es sinnvoll, einzelne Aktionen abhängig vom Kontext ihrer Benutzung zu logischen Einheiten zu gruppieren. Auf Softwareseite kann dies durch die Einführung von *CamViewStates* geschehen, die jeweils einen Bearbeitungskontext repräsentieren, also beispielsweise »Szenengraph«, »Kollisionsgeometrie« oder »Tilemaps«. Der Nutzer kann nun unter allen registrierten Zuständen den gewünschten auswählen und bekommt daraufhin kontextabhängige Bearbeitungsmöglichkeiten angeboten. Analog dazu wurde das Konzept der *CamViewLayers* eingeführt, welche anders als *CamViewStates* nicht der Bearbeitung dienen, sondern lediglich der Visualisierung zusätzlicher Informationen. Diese können in beliebiger Kombination zur primären Ausgabe des aktuellen Kameramodus zugeschaltet werden - beispielsweise kann existierende Kollisionsgeometrie jederzeit als zusätzliches Overlay⁹ angezeigt werden, um die Transformation einzelner Objekte darauf abzustimmen.

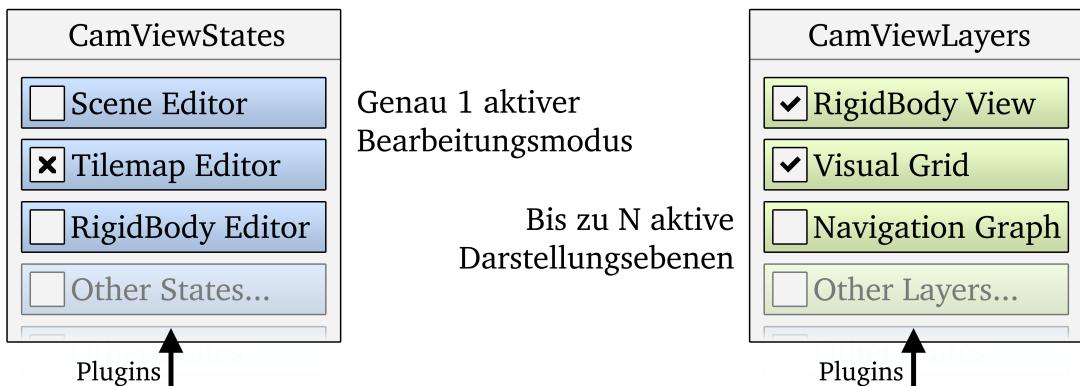


Abbildung 4.8: Layers und States im Camera View

⁷Im Rahmen physikalischer Simulation können Objekte mit einer vereinfachten mathematischen Repräsentation ihrer Form ausgestattet werden, um Kollisionsabfrage und -reaktion zu beschleunigen

⁸Eine Methode, große Landschaften durch Abbildung auf wiederkehrende *Tiles* (»Kacheln«) effizient speichern, verwalten und darstellen zu können

⁹Überlagernde Darstellung, bei der sowohl Hintergrund als auch Vordergrund sichtbar bleiben.

Ein großer Vorteil der Aggregation von Bearbeitungs- und Darstellungsfeatures zu States und Layers ist die Schaffung eines modularen Systems, das über Editor Plugins vom Nutzer beliebig erweitert werden kann (Abb. 4.8). Anders als bei einer Eigenimplementierung einer spezialisierten Kameraansicht ist er dabei gleichzeitig in der Lage, auf eine gemeinsame Basisfunktionalität zurückzugreifen, beispielsweise nutzerseitige Kamerabewegung, Selektionsverwaltung oder DragDrop Aktionen.

Object Inspector

Obwohl mithilfe der Kameraansicht bereits die Bearbeitung einiger Aspekte einer Szene oder eines Objekts ermöglicht wird, so gibt es doch Objekteigenschaften, bei denen eine visuelle oder räumlich verankerte Bearbeitung wenig sinnvoll erscheint: Der Name oder die »Lebenspunkte« eines Objekts beispielsweise, oder aber die Konfiguration einer Ressource. Für die Bearbeitung dieser Objekteigenschaften gibt es den sogenannten *Object Inspector* (Abb. 4.9).



Abbildung 4.9: Die Objektansicht

Dieser nutzt die globale Selektion aus, um die Eigenschaften der vom Nutzer gewählten Objekte zu bestimmen und ihm über eine GUI deren Bearbeitung anzubieten. Die selektierten Objekte können theoretisch einem beliebigen Typ angehören, angefangen bei primitiven Werttypen wie `int` oder `float`, über Arrays und Listen bis hin zu komplexen Objekten. Es ist aufgrund der systematischen Erweiterbarkeit ebenfalls damit zu rechnen, dass einige der möglicherweise selektierten Objekten Typen repräsentieren oder referenzieren, die zur Compilezeit

unbekannt sind. Es stellt sich also die grundsätzliche Frage, wie Darstellung und Bearbeitung verschiedenster Objekte in Form einer flexiblen Benutzeroberfläche zu realisieren sind.

Wie immer gibt es viele Wege, die zum Ziel führen. In einem ersten Ansatz könnten für jeden sinnvoll editierbaren Objekttyp spezialisierte Editoren definiert und manuell gewartet werden: Es gäbe Editoren für Textur- und Sound Ressourcen, für Transform- und SpriteRenderer Komponenten. Da mit diesem Konzept jeder Editor genau auf einen bestimmten Objekttyp zugeschnitten ist, kann effektiv auf dessen individuelle Anforderungen eingegangen und die resultierende Nutzerfreundlichkeit maximiert werden. Es ergeben sich jedoch auch eine ganze Reihe von Nachteilen:

- Durch die große Zahl benötigter Editoren, ist mit einem erheblichen Arbeitsaufwand auf Entwicklerseite zu rechnen, der sich möglicherweise negativ auf die Nutzerfreundlichkeit auswirken wird: Weil eine größere Zahl von Editoren entworfen und implementiert werden muss, bleibt für den einzelnen weniger Zeit.
- Das Konzept maximaler Spezialisierung wird auch dem Nutzer abverlangt. Die Handhabung von eigenen Komponentenklassen oder Ressourcen obliegt allein seiner Verantwortung, denn diese waren zur Compilezeit des Systems nicht bekannt und verfügen daher über keine effektive Bearbeitungsmöglichkeit. Diese müsste über zusätzliche Editoren per Plugin eigens nachgerüstet werden.
- Es ergibt sich ein hohes Maß an Redundanz: Obwohl beispielsweise die meisten komplexen Objekte über Eigenschaften verfügen, die sich als Ganz- oder Fließkommazahlen ausdrücken lassen, werden Darstellung und Bearbeitung dieser Werte in jedem Editor einzeln ausmodelliert. Zwar sind so spezialisierte Anpassungen möglich¹⁰, jedoch müssen globale Veränderungen an einer Vielzahl von Stellen durchgeführt werden.

Die genannten Probleme lassen den ersten Ansatz wenig erstrebenswert erscheinen. In einer komplementären Vorgehensweise könnte dagegen vollständig auf spezialisierte Editoren verzichtet werden: Da sich alle komplexen Objekte letztendlich auf eine Ansammlung primitiver Datentypen reduzieren lassen, kann durch eine analoge Strukturierung der Benutzeroberfläche jedes beliebige

¹⁰Layout, Grenzwerte, Sonderfälle, ...

Objekt bearbeitet werden, ohne den Entwurf eines spezialisierten Editors zu verlangen. Neben den für primitive Datentypen entworfenen Editoren wird in dieser Vorgehensweise nur ein einziger Editor benötigt: Über einen Reflection Mechanismus kann dieser die öffentlich zugreifbaren Properties und Datenfelder eines Objekttyps bestimmen und jeweils einen Sub-Editor des entsprechenden Typs anlegen. Nicht-primitive Typen werden ihrerseits in Sub-Editoren zerlegt, sodass sich eine hierarchische Struktur ergibt.

Auf diese Weise lassen sich alle Nachteile des vorherigen Ansatzes ausräumen: Der Arbeitsaufwand bleibt aufgrund der überschaubaren Zahl primitiver Datentypen gering, benutzerdefinierte Datentypen können von Haus aus gehandhabt werden und redundante Implementierungen sind nicht anzutreffen. Aus Entwicklerperspektive betrachtet mag diese Vorgehensweise daher nahezu ideal erscheinen - jedoch leidet die Nutzerfreundlichkeit stark unter der mangelnden Spezialisierung der Editoren. Anstatt Objekte auf logischer Ebene im Kontext ihrer Bearbeitung abzubilden, werden sie durch ihre systematische Zerlegung auf einer nur für involvierte Programmierer vorgesehenen Datenebene präsentiert. Dieser Umstand erschwert und verkompliziert den Arbeitsablauf des Nutzers unnötig.



Abbildung 4.10: Verschiedene Ansätze im Vergleich

Ein Mittelweg kann die Probleme beider Ansätze lösen: Begonnen mit der zuletzt skizzierten allgemeinen Variante können zusätzlich für bestimmte Typen¹¹ oder Typgruppen¹² spezialisierte Editoren festgelegt werden, die dort an die Stelle des Reflection Ansatzes treten (Abb. 4.10). Während ein Objekt im Standardfall in seine Bestandteile zerlegt und hierarchisch dargestellt wird, so kann unter bestimmten Bedingungen stattdessen ein spezialisierter Editor ver-

¹¹Vector3, Rect, Texture, ...

¹²IList<T>, ContentRef<T>, IColorData, Component

wendet werden - oder die allgemeine Zerlegung objektweise angepasst. Für den Nutzer irrelevante Daten können auf diesem Weg versteckt werden während andere Datentypen von einem spezialisierten Editor profitieren können: Anstatt die vier Komponenten eines Farbwertes einzeln numerisch zu bearbeiten, bietet sich beispielsweise ein intuitiv zu benutzender Farbeditor mit Vorschau und Auswahl-dialog an.

Die Entscheidung, welche Typen durch Spezialeditoren dargestellt und bearbeitet werden können, liegt bei einer Ansammlung aus zentral registrierten *PropertyEditorProvider* Instanzen, welche auf Anfrage den passenden Editor für einen bestimmten Datentyp zurückliefern. Ein Plugin kann nun eine eigene *PropertyEditorProvider* Instanz registrieren, welche bei Bedarf nutzerdefinierte Editoren zurückliefert. Auf diese Weise kann das System auch ohne weitreichende Gesamtkenntnisse um eigene Editoren erweitert werden - auch wenn dies dank des reflectiongetriebenen Unterbaus nur selten notwendig wird.

4.3 Usability

Eines der anfänglich gesetzten Ziele bei der Entwicklung des Duality Projekts war eine möglichst ausgeprägte Nutzerfreundlichkeit. Während eine intuitive und konsistente Bedienung der einzelnen Steuerelemente im Rahmen dieser Arbeit als selbstverständliches Rahmenkriterium angesehen wird, ergaben sich darüber hinaus viele Möglichkeiten, das Nutzererlebnis weiter zu verbessern. Einige davon sollen im Folgenden erläutert werden.

Farbneutrales Design

Der Duality Editor wurde mit dem GUI Framework *Windows.Forms* entwickelt, welches im Wesentlichen einen Wrapper um die Windows API darstellt und grundlegende Funktionalität für Windows Benutzeroberflächen innerhalb der .Net Umgebung verfügbar macht.^[23] Darunter fallen auch eine Reihe besonders verbreiteter Steuerelemente¹³, die ohne Weiteres verwendet werden können und sich dem aktuellen Windows Design anpassen (Abb. 4.11). In den meisten Anwendungsfällen macht es Sinn, dieses unverändert zu verwenden, um keine visuelle Inkonsistenz zu anderen Anwendungen oder dem Betriebssystem selbst herbeizuführen. Ein individuelles Design kann sich zwar letztendlich positiv auf die Nutzerwahrnehmung des Programms auswirken, verlangt jedoch stets eine Umgewöhnung und sollte daher nur mit Bedacht erwogen werden.

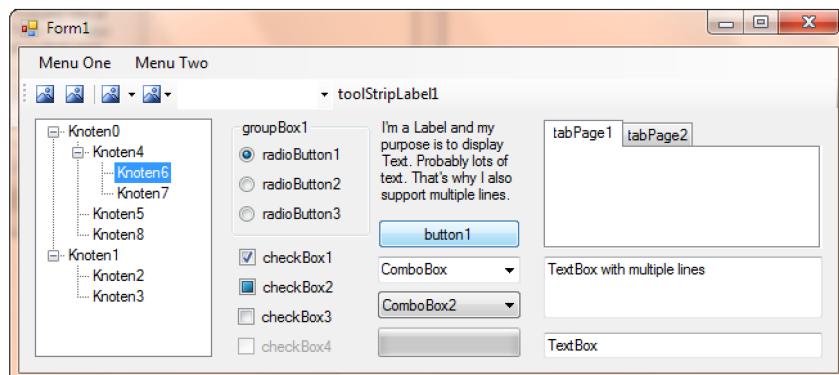


Abbildung 4.11: Typische Steuerelemente in Windows.Forms

Im Fall von Duality zeigte sich jedoch schnell ein störender Einfluss des Windows Designs auf den Arbeitsablauf des Nutzers: Durch starke Kontraste und

¹³Fenster, Buttons, Textboxen, Baumansichten, ...

den großflächigen Einsatz heller Farben erlangte die Nutzeroberfläche ein unnötig großes optisches Gewicht gegenüber den in der Regel deutlich dunkleren und weniger kontrastreichen Renderings der Spielwelt in den einzelnen Kameraansichten (Abb. 4.12). Diese Gewichtung stand im Gegensatz zur tatsächlichen Wichtigkeit der jeweiligen Bereiche für den Nutzer: Durch Live Editing Features sowie eine direkte WYSIWYG Bearbeitung von Objekten spielt sich ein großer Teil des Arbeitsprozesses mit Blick auf die Kameraansicht ab.

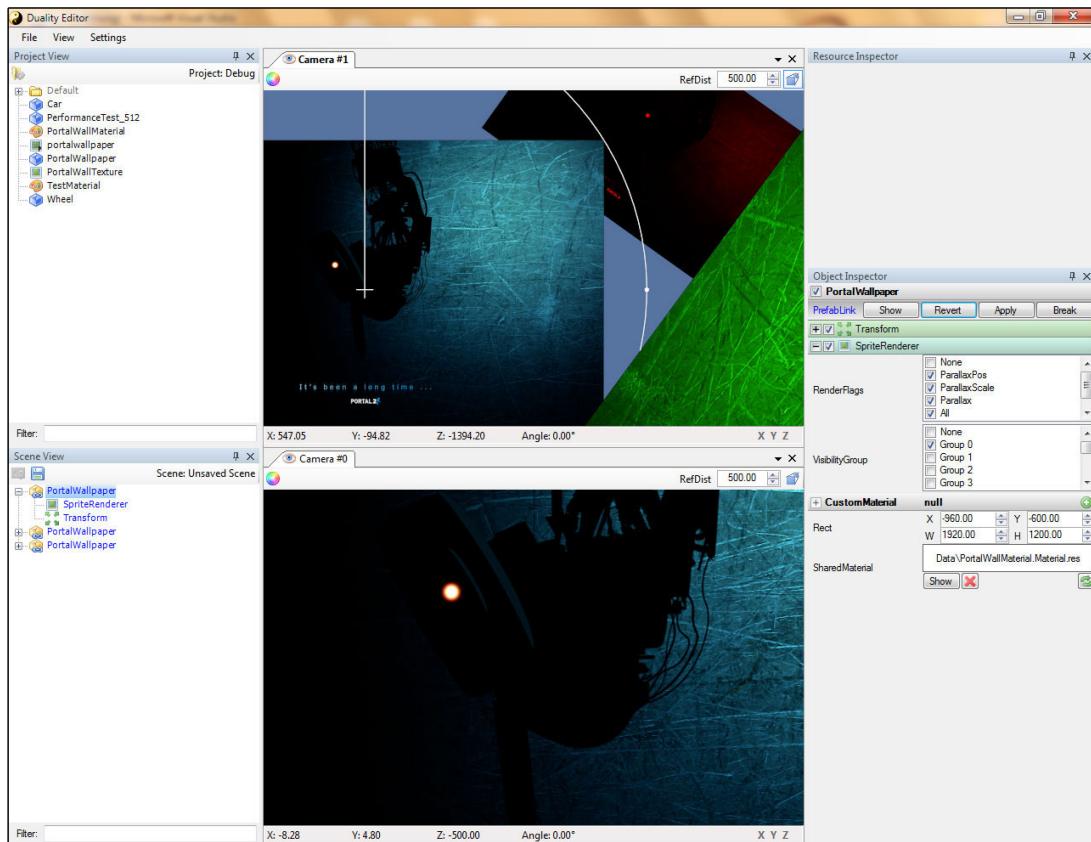


Abbildung 4.12: Der Duality Editor im Windows.Forms Stil

Das einheitliche Design der Windows Benutzeroberfläche zog damit mehr Aufmerksamkeit auf sich als die für den Nutzer tatsächlich wichtigen Bereiche. Diesen Widerspruch galt es aufzulösen, ohne dabei die Sichtbarkeit der einzelnen Bereiche als solche zu verändern, was wiederum die Gesamtübersicht gefährdet hätte - ein individuelles Design der Nutzeroberfläche erschien daher sinnvoll. Dieses sollte weniger starke Kontraste bevorzugen und sich durch eine verringerte Gesamthelligkeit verschiedenen Kameraansichten besser anpassen können (Abb. 4.13).

Darüber hinaus sollten als Primärfarben lediglich Graustufen verwendet wer-

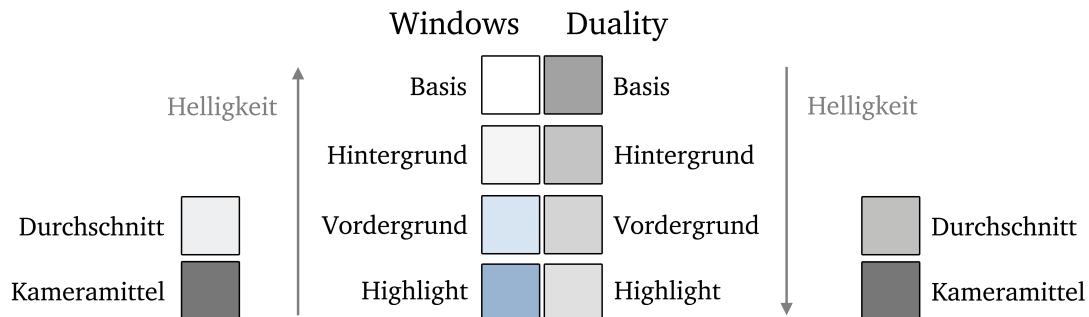


Abbildung 4.13: Die Farbschemata von Windows und Duality im Vergleich

den, um die visuelle Wechselwirkung mit der dargestellten Spielwelt so gering wie möglich zu halten. Würde man beispielsweise die gesamte Benutzeroberfläche mit einem leichten Blaustich versehen, so würden in der Wahrnehmung des Benutzers blaue Farbtöne der Spielwelt in den Hintergrund gerückt und gelbe¹⁴ verstärkt. Eine solche Wahrnehmungsverschiebung führt durch Designentscheidungen des Nutzers letztendlich zu einer entgegengesetzten Verfälschung des Grafikstils im entwickelten Spiel - daher ist es wichtig, diese durch ein farbneutrales Design zu vermeiden.

In der Praxis fällt auf, dass ein mit dem abgebildeten Farbschema konstruiertes Editor Design bei längerer Benutzung deutlich angenehmer zu betrachten ist als das unveränderte Windows Design (Abb. 4.14). Möglicherweise ist dies auf die Verminderung großflächiger Kontraste sowie die Reduktion der Gesamthelligkeit zurückzuführen. Um eine verlässliche Schlussfolgerung formulieren zu können, wären jedoch umfangreichere Tests und Recherchen notwendig, was nicht Thema dieser Arbeit sein soll. Es sei daher lediglich eine mögliche Gesamtverbesserung des Nutzererlebnisses im Zuge der Designänderung angemerkt - auch wenn diese zunächst verifiziert werden müsste.

¹⁴Gelb ist die Komplementärfarbe von Blau. In der Farbenlehre bezeichnet »Komplementär« zwei Farben, die zu gleichen Teilen vermischt einen neutralen Grauton ergeben. Die Kombination von Komplementärfarben erzeugt in der menschlichen Wahrnehmung besonders starke farbliche Kontraste.

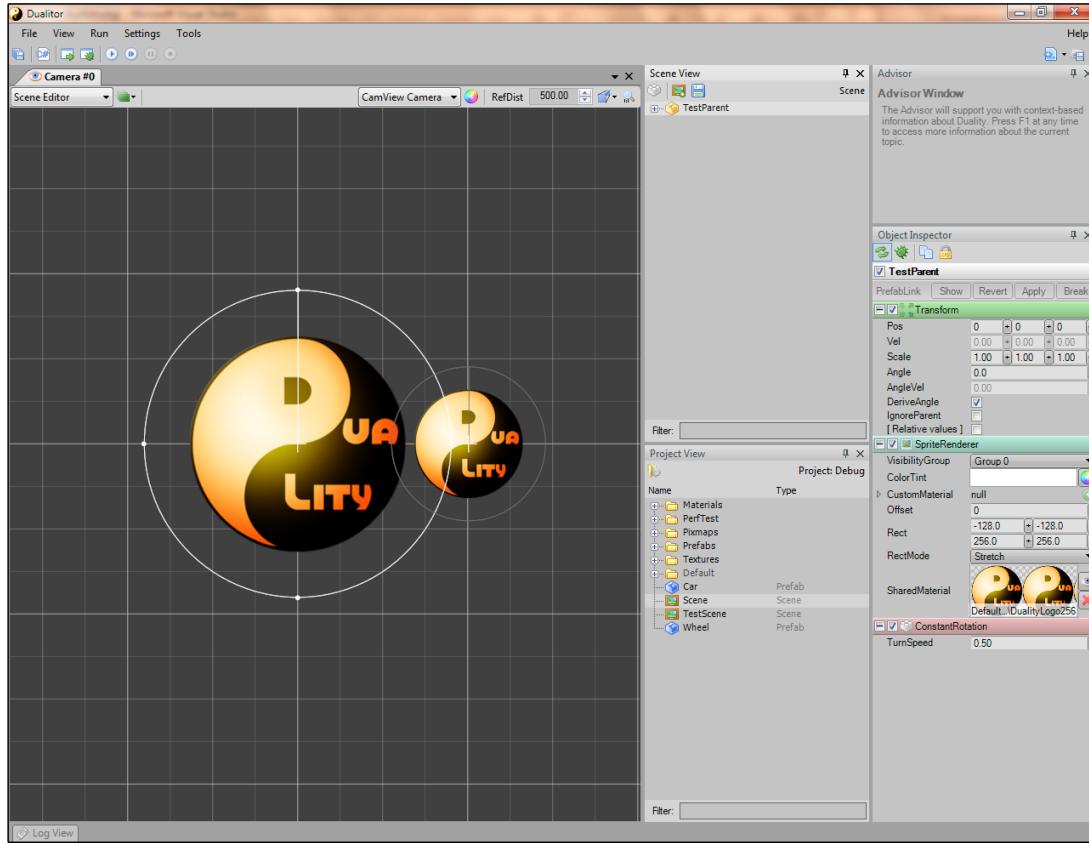


Abbildung 4.14: Der Duality Editor mit eigenem Design

Integriertes Hilfesystem

Sowohl Duality als auch der zugehörige Editor bilden ein vergleichsweise komplexes System, das von neuen Benutzern Schritt für Schritt verstanden werden muss, um effektiv benutzt werden zu können. Dieser anfängliche Lernaufwand stellt eine Einstiegshürde dar, deren Höhe mitbestimmt, wie viel Zeit für den Nutzer notwendig ist, um ein gesetztes Ziel erstmals zu erreichen. Ist sie zu hoch, wirkt sich dies negativ auf das Gesamterlebnis aus und könnte Anfänger mit wenig ausgeprägter Frustrationstoleranz unnötigerweise verschrecken.

Das größte Problem für Neueinsteiger ist in aller Regel nicht die Komplexität eines Systems, sondern mangelhafte Erklärungen, unzugängliche Metaphern und die Tatsache, dass Unklarheiten oft so lange unklar bleiben, bis der Nutzer sich durch eigene Recherchen weitergebildet hat. Eine umfangreiche Dokumentation des Systems sowie Fachartikel und Handbuch können einen Großteil dieser Problematik ähnlich gut abfedern, wie eine aktive und hilfsbereite Community.

Listing 4.1: Dokumentation von Source Code in C#

```
public class Font : Resource
{
    /// <summary>
    /// [GET / SET] The name of the font family that is used.
    /// </summary>
    public string Family { /* ... */ }
    /// <summary>
    /// [GET / SET] The size of the Font.
    /// </summary>
    public float Size { /* ... */ }
}
```

Duality hat als sehr junges Projekt jedoch bisher keinerlei Community - und selbst wenn, wäre es wenig ratsam, die Einführung von Neulingen allein auf eine bereits existierende Nutzerbasis abzuwälzen. Das Schreiben von Handbuch und Fachartikeln ist jedoch eine nicht zu unterschätzende Aufgabe, die ein großes Maß an Arbeitsaufwand bedeutet - das Dokumentieren von Klassen, Methoden und Properties im Quellcode kann jedoch bequem neben dem eigentlichen Entwicklungsprozess verlaufen. Zunächst sollte die Dokumentation also möglichst effizient zur Schulung des Anwenders genutzt werden.

Listing 4.2: Automatisch generierte XML Dokumentationsdatei

```
<doc>
<members>
    <member name="P:Duality.Resources.Font.Family">
        <summary>
            [GET / SET] The name of the font family that is used.
        </summary>
    </member>
    <member name="P:Duality.Resources.Font.Size">
        <summary>
            [GET / SET] The size of the Font.
        </summary>
    </member>
</members>
</doc>
```

Die Dokumentation von Quellcode wird in C# durch spezielle *XML Kommentare* unterstützt, die neben einer lokalen Beschreibung auch zur automatischen Generierung einer XML Dokumentationsdatei dienen (Lst. 4.1, 4.2). Diese wird im Rahmen des Kompiliervorgangs erstellt und von Visual Studio beispielsweise für Intellisense Tooltips verwendet (Abb. 4.15). Durch eine Auslieferung der Dokumentationsdatei können zukünftige Nutzer bei der Implementierung eigener Komponenten und Ressourcen unterstützt werden. Mithilfe von externen Tools kann aus der XML Dokumentation zusätzlich eine Website oder Hilfdatei als Nachschlagewerk generiert werden, um einen groben Gesamtüberblick zu bie-

ten.

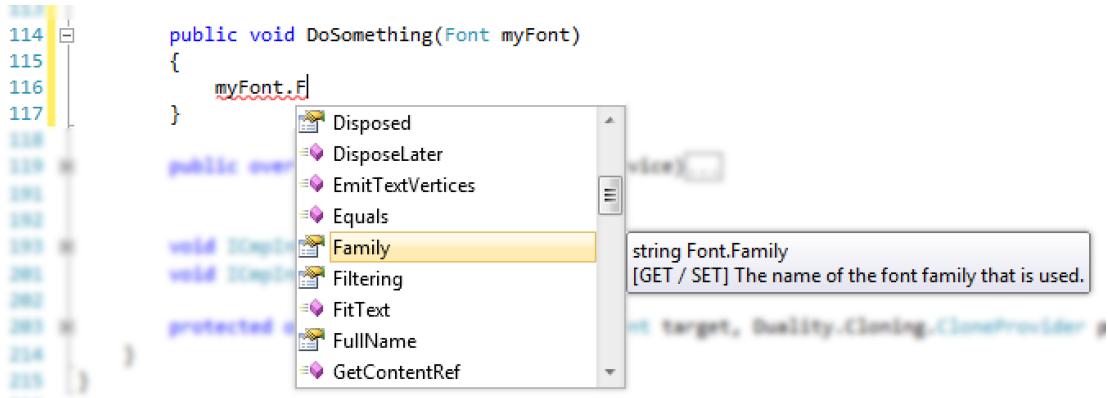


Abbildung 4.15: Dokumentationsbasiertes Intellisense

All diese Maßnahmen helfen dem Nutzer jedoch nicht bei der Verwendung des Duality Editors, womit er für den Großteil der Entwicklungszeit auf sich allein gestellt ist. Zwar kann er die Bedeutung einzelner Objekteigenschaften oder Typen von Hand in der beigelegten Hilfdatei nachschlagen, jedoch stellt dies einen Bruch im Arbeitsablauf dar und ist unnötig mühsam, daher wäre es wünschenswert, direkt im Editor auf die benötigten Informationen zugreifen zu können. Gleichzeitig dürfen insbesondere Anfänger nicht mit Informationen überflutet werden: Auf Fragen zu antworten, die noch gar nicht gestellt wurden, führt nicht selten zu Verwirrung oder Überforderung.

Es gilt also, zu jedem Zeitpunkt genau die Informationen darzustellen, die gerade benötigt werden. Um herauszufinden, welche Informationen dies sind, kann die Analyse der Mauscursor Position hilfreich sein: Der Cursor ist als Verlängerung des Zeigefingers nicht selten ein Indikator für den derzeitigen Fokus des Nutzers und stellt darüber hinaus einen intuitiven Weg dar, eine Anwendung um mehr Informationen zu bitten. Indem man auf etwas zeigt, kann man auf einfache Weise den Wunsch nach einer Erklärung äußern. Die seit langem verbreiteten Tooltips basieren auf demselben Konzept (Abb. 4.16).

Für umfangreiche Erklärungen und größere Texte sind Tooltips jedoch kaum einsetzbar: Weil sie aufgrund ihrer Nähe zum Cursor so gut wie immer einen funktionalen Bereich der Anwendung verdecken, erscheinen sie sinnvollerweise erst mit einer gewissen Verzögerung und nur im Ruhezustand des Cursors. Selbst kleinere Mausbewegungen führen zum sofortigen Verschwinden der Tooltips, damit diese im produktiven Arbeitsablauf nicht stören. Für Beschriftungen und kurze Texte ist dieses Verhalten ideal, jedoch erschwert es bei längeren Erklärungen das Lesen deutlich. Für das im Duality Editor integrierte Hilfesystem

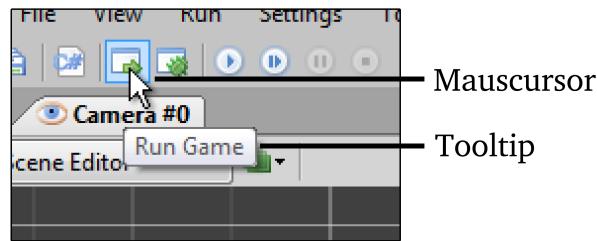


Abbildung 4.16: Tooltip eines Werkzeugeisten Buttons

wurde daher eine andere Vorgehensweise gewählt:

1. Es wird zunächst das Steuerelement bestimmt, welches derzeit unterhalb des Mausursors liegt. Dies geschieht global und ohne Zutun der involvierten Steuerelemente.
2. Implementiert dieses das *IHelpProvider* Interface, wird darüber ein parametrisierter Hilfetext angefordert, der an einer zentralen Stelle abgelegt wird. Andernfalls wird rekursiv das jeweilige Parent-Objekt der Steuerelement Hierarchie geprüft.
3. Einzelne Editormodule können auf Zustand und Veränderung des aktuellen globalen Hilfetexts reagieren.
4. Ein standardmäßig mitgeliefertes Plugin implementiert das *Advisor*¹⁵ Modul, welches den aktiven Hilfetext in einem dafür vorgesehenen Bereich der Benutzeroberfläche anzeigt (Abb. 4.17). Damit ist die Grundlage für ein editorweites Hilfesystem gelegt.
5. Beim Start des Editors werden nun alle verfügbaren XML Dokumentationsdateien geladen und über eine globale Schnittstelle verfügbar gemacht. Das *Object Inspector* Modul implementiert die *IHelpProvider* Schnittstelle und stellt für Objekte und deren dargestellte Eigenschaften die in der Dokumentation hinterlegte Beschreibung bereit.
6. Andere Editormodule können bei Bedarf ebenfalls kontextabhängige Informationen bereitstellen, indem sie *IHelpProvider* implementieren. Art und Umfang der Informationen sind frei wählbar.

Das in Duality implementierte Hilfesystem stellt insgesamt also eine Erweiterung des Tooltip Konzepts dar, das auch umfangreichere Erklärungen erlaubt. Es

¹⁵Engl.: »Berater«

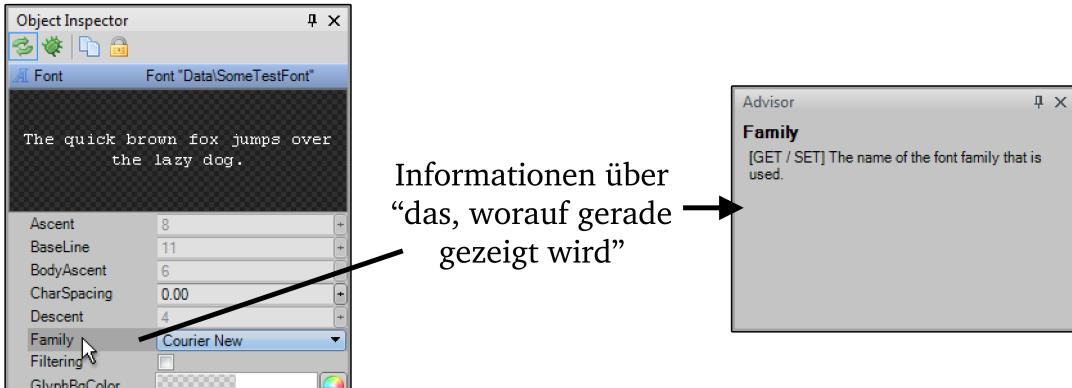


Abbildung 4.17: Das integrierte Hilfesystem in Aktion

kann den Bedürfnissen des Nutzers auf einfache Weise angepasst werden, denn dieser kann Anordnung und Existenz einzelner Editormodule selbst bestimmen - und damit auch, wie viel Platz der Advisor Bereich einnimmt, oder ob er überhaupt sichtbar ist. Zusätzlich wurde F1 als globale Hilfetaste definiert, die bei ihrer Betätigung eine themenabhängige Aktion auslöst. Im Fall der Darstellung von Dokumentationstexten wird beispielsweise die beiliegende Hilfedatei geöffnet und das entsprechende Thema aufgeschlagen.

Intelligentes DragDrop

DragDrop bezeichnet eine Aktion, bei der eine vom Nutzer vorgegebene direktionale Verbindung zwischen verschiedenen GUI Elementen hergestellt wird. Das Drücken der Maustaste markiert den Beginn der Verbindung, ein Loslassen das Ende. Auf einer metaphorischen Ebene ähnelt eine DragDrop Aktion dem »Ziehen« oder »Verschieben« eines Elements, wird jedoch auch als ein parametrisiertes »Zeigen« verstanden. Durch den gezielten Einsatz von DragDrop Funktionalität kann die Nutzerfreundlichkeit einer Anwendung deutlich gesteigert werden, also sollte auch der Duality Editor davon profitieren.

Im einfachsten Fall kann DragDrop-Funktionalität direkt an der Stelle implementiert werden, an der die Aktion ausgeführt wird: Das Project View Modul beispielsweise kann ähnlich dem Windows Explorer auf Dateien und Ordner reagieren, die darin abgelegt werden sowie bei gedrückter Maustaste selbst eine DragDrop-Aktion mit den selektierten Ressourcendateien beginnen. Dafür ist dank der umfangreichen .Net Bibliothek kein besonderes Softwaredesign notwendig und bereits der erstbeste Ansatz führt zu akzeptablen Ergebnissen. Bei komplexeren Interaktionen weist dieser jedoch einige Unzulänglichkeiten auf -

insbesondere wenn es um die Handhabung von Objekttypen geht, die zur Compilezeit des Editors noch gar nicht bekannt waren.

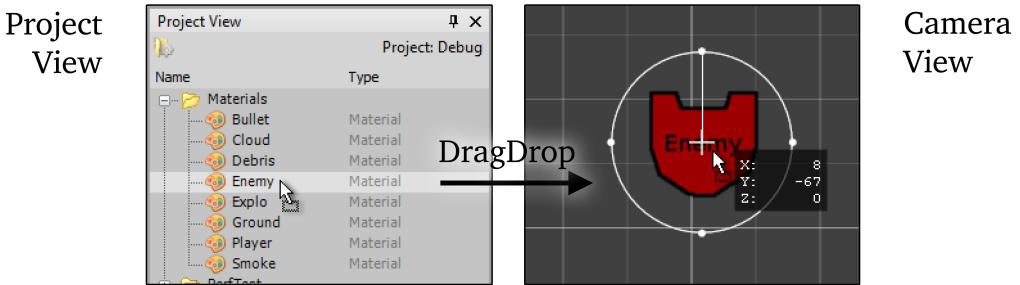


Abbildung 4.18: DragDrop Aktionen zwischen Projekt- und Kameraansicht

Wird jede mögliche Interaktion direkt an Ort und Stelle implementiert, ist es einerseits nur schwer möglich auf nutzerdefinierte Objekttypen zu reagieren während gleichzeitig die Zahl der umzusetzenden Interaktionswege in die Höhe schießt. Es soll beispielsweise ermöglicht werden, durch eine DragDrop-Aktion aus der Projektansicht in die Kameraansicht ein Objekt zu instantiiieren, das die selektierte Ressource verwendet: Werden Audiodaten in der Kameraansicht abgelegt, soll ein Objekt konstruiert werden, das über eine SoundEmitter Komponente verfügt, welche mit den jeweiligen Audiodaten arbeitet. Werden Bilddaten verwendet, soll ein Objekt mit entsprechender SpriteRenderer Komponente erstellt werden. Es gibt jedoch verschiedene Ressourcentypen, die jeweils beachtet werden müssen:

AudioData repräsentiert einen komprimierten Datenblock, der Audiodaten enthält. Dieser ist nur minimal parametrisiert und nicht für die direkte Verwendung gedacht.

Sound referenziert eine AudioData Ressource, stellt relative Parameter wie Lautstärke oder Abspielgeschwindigkeit zur Verfügung und bereitet die konfigurierten Daten für ihre Verwendung vor - beispielsweise in einer SoundEmitter Komponente.

Pixmap ist kurz für »Pixel Map« und repräsentiert einen komprimierten Datenblock, der Bilddaten enthält. Analog zu AudioData ist auch dieser nur minimal parametrisiert.

Texture referenziert eine Pixmap Ressource, ergänzt sie um einige Parameter und lädt die so konfigurierten Daten in den Speicher der Grafikkarte, um sie zur Verwendung vorbereiten.

Material geht noch einen Schritt weiter und verbindet verschiedene Texturen, Darstellungstechniken und sonstige Variablen zu einer Einheit, die im Renderingprozess zur Beschreibung eines Batches verwendet werden kann.

Wird nun ein Material oder ein Sound in der Kameraansicht abgelegt, können direkt die jeweiligen Objekte erstellt und in der Szene registriert werden. Aus einer AudioData Ressource muss jedoch zunächst eine Sound Ressource erstellt werden, um verwendet werden zu können - aber natürlich nur dann, wenn eine solche nicht bereits existiert. Analog verhält es sich mitPixmap- und Texture Ressourcen, die ihrerseits speziell behandelt werden müssen. Darüber hinaus ist es auch möglich, dass der Nutzer gleichzeitig mehrere Ressourcen in einer einzelnen DragDrop Aktion verwendet und ein einzelnes Objekt erwartet, das diese auf sinnvolle Art verknüpft (Abb. 4.19).

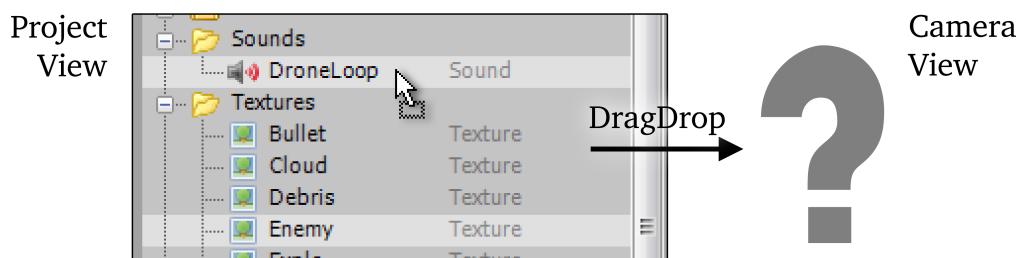


Abbildung 4.19: Eine komplexere DragDrop-Aktion

In der direkten Vorgehensweise müssten all diese Spezialfälle sowie ihre jeweiligen Kombinationen beachtet werden. Dies stellt einen Nährboden für redundanten Code dar und bildet gleichzeitig ein abgeschlossenes System, das nicht vom Nutzer erweitert werden kann. Ein Lösungsansatz, der sich bei bereits behandelten Problemstellungen bewährt hat, kann auch an dieser Stelle weiterhelfen: Anstatt ein monolithisches System zur Behandlung komplexer Probleme einzusetzen, sollten zunächst die Probleme in ihre Einzelteile zerlegt werden - die dann in einem modularen System einzeln behandelt werden können¹⁶. Die zu lösende Problemstellung lässt sich wie folgt beschreiben:

- Ein Editormodul, das DragDrop-Aktionen akzeptiert, verlangt einen bestimmten Datentyp, mit dem es arbeiten kann. Die Projektansicht beispielsweise kann wahlweise mit Windows Explorer Dateipfaden oder Referenzen

¹⁶Dieser Lösungsalgorithmus wird auch als »Divide and Conquer« / »Teile und herrsche« bezeichnet [24]

auf Ressourcen arbeiten während die Kameraansicht im Szenengraphmodus mit GameObjects arbeitet.

- DragDrop Aktionen werden bei ihrem Beginn stets mit Objekten eines oder mehrerer Datentypen ausgestattet, auf die bei Abschluss der Aktion zugegriffen werden kann.
- Die zentrale Fragestellung ist: Wie wird damit umgegangen, wenn sich Eingabe- und Ausgabetypen unterscheiden? Kann eine Konvertierung stattfinden? Auf welche Weise wird diese implementiert? Einige Konvertierungen können direkt stattfinden (AudioData → Sound), andere benötigen mehrere Schritte (Pixmap → Texture → Material).

Betrachtet man alle denkbar sinnvollen Umwandlungen, lässt sich ein Graph aufspannen, dessen Knoten Datentypen darstellen und dessen Verbindungen jeweils eine atomare Konvertierung (Abb. 4.20). Eine vollständige Überführung von Eingabe- in Ausgabedaten kann nun durch einen Weg innerhalb dieses Graphen definiert werden, der mithilfe eines Pathfinding Algorithmus bestimmt werden kann. Indem atomare Umwandlungen über eine allgemeine *DataConverter* Schnittstelle abstrahiert und als Objekte global registriert werden, bleibt das Gesamtsystem offen für benutzerseitige Erweiterungen.

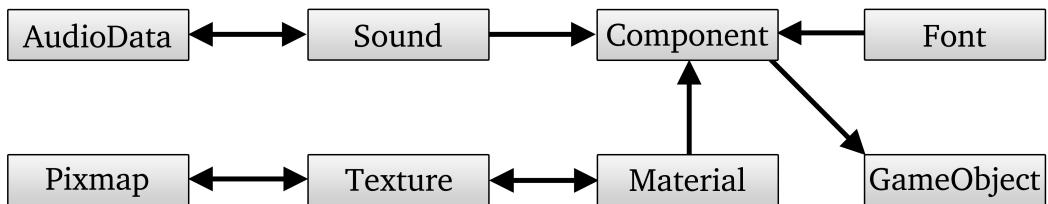


Abbildung 4.20: Ein Typumwandlungsgraph

Bestimmte DataConverter lassen dabei auch eine Komposition verschiedener verfügbarer Konvertierungspfade zu. Ein GameObject etwa kann aus beliebig vielen verschiedenen Komponenten konstruiert werden, ohne dass diese sich gegenseitig ausschließen. Es wird daher nicht in jedem Fall nur der kürzeste Pfad gewählt, sondern prinzipiell jeder verfügbare Pfad in aufsteigender Länge - bis die Operation als abgeschlossen markiert wird (Abb. 4.21).

Die beschriebene Vorgehensweise zeichnet sich dadurch aus, dass lediglich atomare Umwandlungen beschrieben werden, nicht jedoch der gesamte Pfad

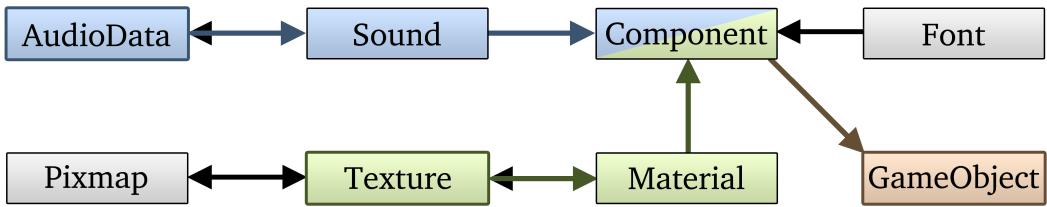


Abbildung 4.21: Generierung eines GameObjects aus AudioData und Texture

von Eingabe zu Ausgabe. Dieser wird stattdessen dynamisch aus den verfügbaren DataConverter Objekten abgeleitet und ermöglicht so nahtlose Erweiterungen durch die Definition eigener DataConverter. Als Nachteil sollte jedoch eine Verringerung des Gesamtüberblicks genannt werden: Da jeder DataConverter sich nur mit »seinen eigenen« Objekttypen befasst und Konvertierungspfade dynamisch abgeleitet werden, können sich vom Programmierer nicht vorgesehene Umwandlungsmöglichkeiten ergeben. Zwar können diese durchaus sinnvoll sein; aber da sie im Zuge der Entwicklung möglicherweise nie getestet wurden, liegt dort ein gewisses Fehlerpotential.

Natürlich kann die für Typumwandlungen geschaffene Infrastruktur auch an anderen Stellen sinnvoll eingesetzt werden, daher wurde sie über eine allgemeine Schnittstelle editorweit verfügbar gemacht und steht nur indirekt im Zusammenhang mit der beschriebenen DragDrop Funktionalität. Dort stellt sie allerdings ein wesentliches Element dar, das in vielen Fällen bereits zur Optimierung des Arbeitsablaufs beitragen konnte.

5 | Zusammenfassung

Im Verlauf dieser Arbeit wurden wesentliche Aspekte einer auf Erweiterbarkeit und Flexibilität ausgelegten 2D Game Engine thematisiert, sowie verschiedene Ansätze zur Umsetzung eines passenden Editorsystems diskutiert. Besonderes Augenmerk lag dabei auf einer Maximierung der Nutzerfreundlichkeit sowohl auf Seite der Benutzeroberfläche als auch der des Engine Frameworks.

Es wurde eine pluginbasierte Software Architektur vorgestellt, die eine Bearbeitung von Spiellogik und sonstigem Quellcode zur Laufzeit ermöglicht und so die Effizienz des Entwicklungsprozesses deutlich verbessern kann. Der Einsatz von Komponentenbasierten GameObjects zur Strukturierung der Spieltwelt stellt eine flexible Gestaltungsmöglichkeit dar und erlaubt darüber hinaus die nahtlose Erweiterung der Engine auf Nutzerseite. Zur Verwaltung von externen und internen Ressourcen wurde eine zentralisierte Lösung vorgeschlagen, welche eine systemweite Aktualisierung zur Laufzeit unterstützt, um Live Editing Features zu ermöglichen. Zur persistenten Speicherung von Daten wurden verschiedene automatisierte Verfahren des .Net Frameworks verglichen und eine Eigenimplementierung skizziert, welche eine deutlich verbesserte Fehlertoleranz aufweist. Abschließend wurde eine abstrakte Rendering Schnittstelle präsentiert, anhand deren Beispiel eine Reihe von Optimierungsverfahren erläutert werden konnte.

Im darauffolgende Kapitel dieser Arbeit wurde die Entwicklung eines modularen Editorsystems behandelt, das durch ein Pluginsystem analog zu dem des Engine Frameworks ein ähnlich hohes Maß an Erweiterbarkeit an den Tag legt. Weiterhin wurde an verschiedenen Beispielen verdeutlicht, wie durch ein flexibles Software Design die Handhabung von zur Compilezeit unbekannten Klassen und Objekten vereinfacht werden kann. Anhand einiger Standardmodule der Benutzeroberfläche konnte gezeigt werden, wie sich verschiedene Software Designentwürfe auf die Nutzerfreundlichkeit des Gesamtsystems auswirken und welche Maßnahmen im konkreten Fall ergriffen werden können, diese zu verbessern. In einer abschließenden Betrachtung wurden einige Möglichkeiten

erläutert, das Nutzererlebnis durch ein eigenes Oberflächendesign, ein integriertes Hilfesystem und intelligente DragDrop Funktionalität weiter zu verbessern.

Nicht näher behandelt wurde die bereits abgeschlossene Integration einer Library zur physikalischen Simulation zweidimensionaler Festkörper sowie die Entwicklung eines Audio Subsystems. Auch das *Prefab*¹ System zur Verwaltung von Objekt Prototypen verschlang einen nennenswerten Anteil der Entwicklungszeit, wurde jedoch aus Zeit- und Platzgründen nicht näher erläutert. Diese Themen könnten in weiterführenden Arbeiten einer eigenständigen Betrachtung unterzogen werden. Ebenso könnte auch eine detailliertere Analyse anderer Engine Komponenten weitere Erkenntnisse zutage fördern.

In ihrem derzeitigen Zustand können sowohl die Duality Engine als auch das zugehörige Editorsystem trotz einer Vielzahl von möglichen Verbesserungen als einsatzbereit gelten. Sie wurden bereits prototypisch für die Entwicklung kleinerer Spiele und Techdemos² eingesetzt.

¹ »Prefabricated Object« / »Vorgefertigtes Objekt«

² Technologie Demonstrationen

6 | Ergebnisse und Diskussion

Egal wie durchdacht und fein säuberlich geplant eine Game Engine auch sein mag - letztendlich hat sie ihren Zweck verfehlt, wenn sie nicht eines Tages auch tatsächlich produktiv eingesetzt wird. Eine Game Engine völlig isoliert zu entwickeln, birgt stets eine gewisse Gefahr: Ohne ein konkretes Spiel mit konkreten Anforderungen gibt es keinen festen Maßstab, der zur Überprüfung von Fortschritt und Möglichkeiten eingesetzt werden kann. Dieser Umstand wird in diversen Artikeln ausführlicher erläutert und wird im Allgemeinen mit dem Auspruch »Make Games, Not Engines«^[25] in Verbindung gebracht.

Um ihre Praxistauglichkeit abzusichern, sollte die Duality Engine daher nicht isoliert entwickelt, sondern stets mit konkreten Anwendungen verifiziert werden. Dies geschah mit einer Vielzahl von kleineren TechDemos und Spielen, die nach und nach entwickelt wurden - teilweise, um konkrete Features der Engine im Gebrauch zu testen, teilweise aber auch aus einem übergeordneten Kontext heraus.

Die meisten dieser Anwendungen halfen durch ihre Entwicklung dabei, Unzulänglichkeiten oder Fehler innerhalb von Engine oder Editor aufzuzeigen und zu verbessern. Nicht alle dieser Verbesserungen werden in der folgenden Auflistung von Testprojekten thematisiert: In vielen Fällen handelt es sich um triviale oder aus anderen Gründen wenig erwähnenswerte Bugfixes¹. Größere oder grundlegende Probleme wurden bereits verbessert und im Rahmen dieser Arbeit in ihrer aktualisierten Form beschrieben.

Asteroids war eines der ersten Testprojekte, die mithilfe der Engine umgesetzt wurden (Abb. 6.1). Bei der umgesetzten Variante des Spielesklassikers geht es wie auch im Original darum, mit einem in der Draufsicht gesteuerten Raumschiff umherfliegende Asteroiden durch Beschuss zu zerstören, ohne dabei mit einem davon oder den entstehenden Trümmerteilen zu kollidie-

¹Als »Bug« bezeichnet werden (kleinere) Systemfehler, ein »Bugfix« bezieht sich auf die Behebung eines solchen Fehlers.

ren. Die Umsetzung des Projekts verlief problemlos und fiel durch eine vergleichsweise kompakte Codebasis auf: Da ein Großteil der notwendigen Funktionalität bereits von Duality geliefert wird, waren für Spieler, Asteroiden, Projektilen, Highscore, Hauptmenü, Statusanzeigen und Levelgenerierung nicht mehr als 500 Zeilen Code notwendig. Kollisionsabfragen und physikalische Simulation wurden komplett von der über Rigidbody² Komponenten integrierten Physik Bibliothek übernommen und bedurften daher keiner eigenen Implementierung. Das Editorsystem konnte insgesamt als Test- und Entwicklungsumgebung effizient eingesetzt werden.



Abbildung 6.1: Screenshot der Asteroids Umsetzung in Duality

Buttermilch2 war der Arbeitstitel eines Projekts, das im Rahmen der Multitouch Lehrveranstaltung im Wintersemester 2011 in Zusammenarbeit mit Daniel Herb erstellt wurde (Abb. 6.2). Das Ziel war es, eine Neuinterpretation des Schere-Stein-Papier Prinzips als ein Spiel für zwei Teilnehmer umzusetzen, welche sich an der laboreigenen Multitouch Oberfläche gegenüberstehen und über verschiedene Gesten virtuelle »Kämpfer« erstellen und befehligen

²»Festkörper«

können. Ziel ist es, mit diesen das gegnerische Hauptquartier zu zerstören. Das Projekt verlief erfolgreich und konnte mithilfe von Duality zügig fertiggestellt werden, da ein Großteil der notwendigen Infrastruktur bereits von der Engine abgedeckt wurde. Für die Verarbeitung von Multitouch Nutzereingaben wurde eine externe Funktionsbibliothek eingebunden, die über das Buttermilch2 Plugin ebenso wie Spiellogik nahtlos integriert werden konnte.

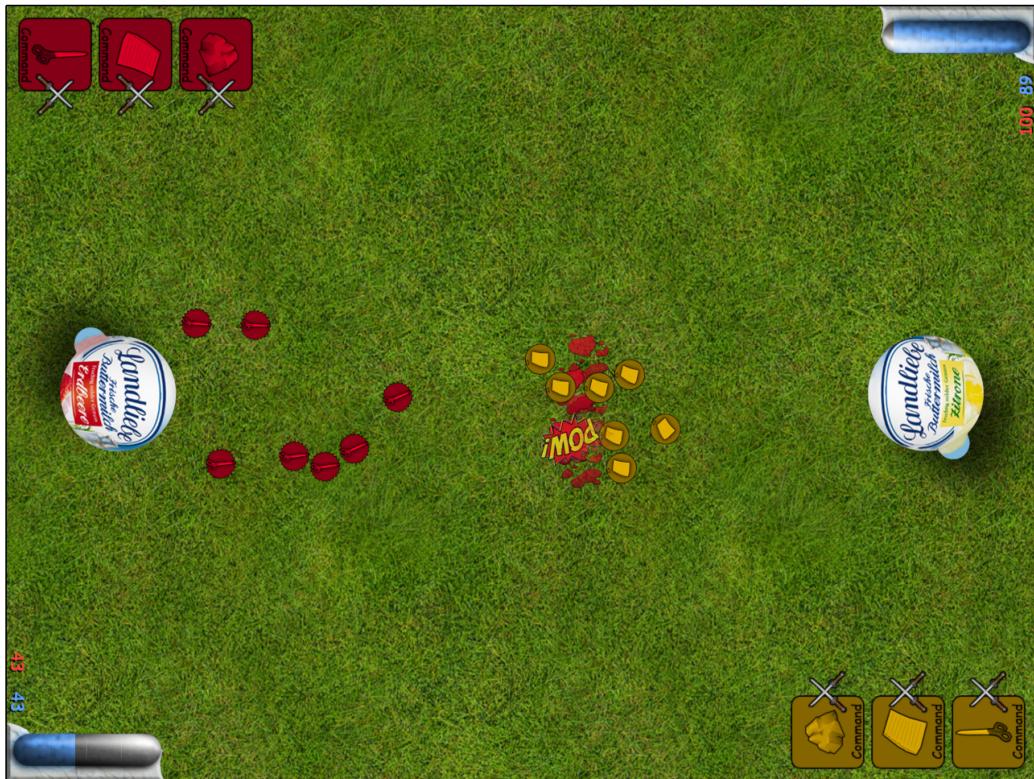


Abbildung 6.2: Screenshot des Buttermilch2 Projekts

DynamicLighting stellte den Versuch dar, das Renderingsystem von Duality durch ein Plugin um dynamische per-Pixel Beleuchtung zweidimensionaler Sprites zu erweitern (Abb. 6.3). Dies konnte durch den Einsatz des bereits verfügbaren Shadersystems, »Multitexture« Materialien³ und eine neu implementierte Variante der SpriteRenderer Komponente erreicht werden. Lichtquellen konnten dank ihrer Umsetzung als eigene Komponente komfortabel im Editor platziert und konfiguriert werden. Eine besondere Heraus-

³Die drei verwendeten Texturen Diffusemap, Normalmap und Specularmap beschreiben Farbe, Struktur und Reflektionseigenschaften eines Sprites pixelgenau.

forderung stellte die Tatsache dar, dass räumliche Informationen in Duality nur sehr begrenzt zur Verfügung stehen und dreidimensionale Perspektive lediglich emuliert wird. Durch die Definition eines eigenen Vertexformats, das zusätzliche Beleuchtungsdaten bereitstellt konnte diese Einschränkung umgangen werden.

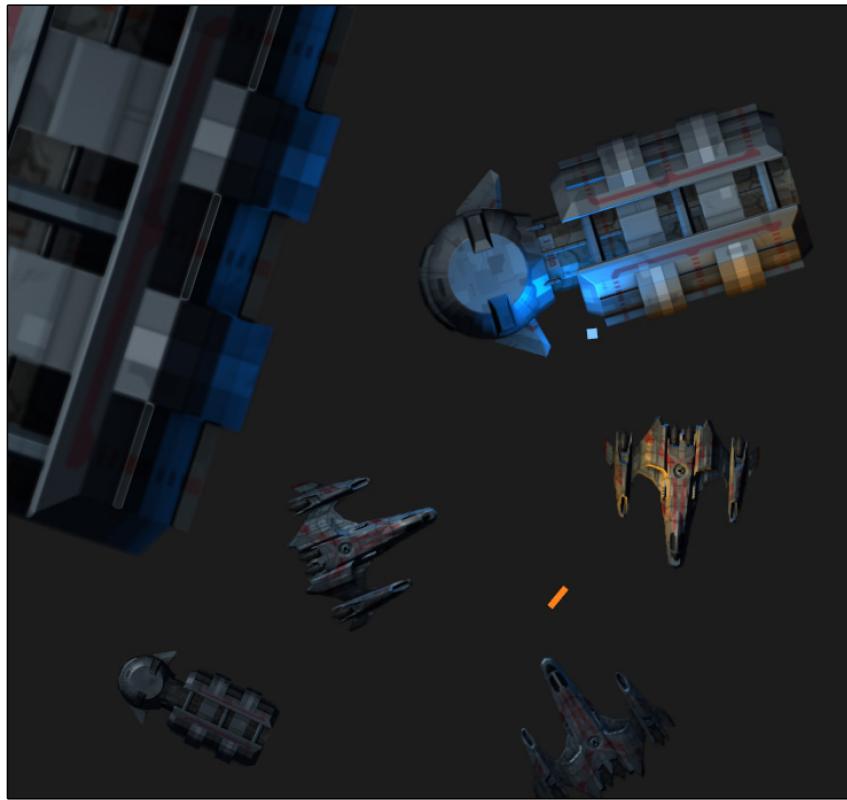


Abbildung 6.3: Screenshot des DynamicLighting Experiments

Physics TechDemo wurde im Rahmen der Duality Integration physikalischer Simulation zweidimensionaler Festkörper umgesetzt und diente dabei einerseits als Testumgebung und andererseits zur Demonstration der neu gewonnenen Features (Abb. 6.4). Es wurde eine Reihe von grafischen Debugausgaben zur Überprüfungen von mechanischen Verknüpfungen⁴ umgesetzt, die aufgrund ihrer Nützlichkeit in einer eigenen CamViewLayer Umsetzung auch dem Endnutzer zur Verfügung stehen. Obwohl besagte Verknüpfungen stets räumlich lokalisier- und darstellbar sind, existiert jedoch noch keine bequeme Möglichkeit, diese im Editor per Mausinterak-

⁴Diese auch »Joints« genannten Verknüpfungen können verschiedene Gelenke, Reibung oder vergleichbare Konzepte simulieren.

tion zu bewegen oder konfigurieren. Dies wäre für zukünftige Versionen insbesondere für größere physikalische Konstrukte wünschenswert.



Abbildung 6.4: Screenshot der Physics TechDemo

Tetris ist ebenso wie **Asteroids** ein allseits bekannter Spieleklassiker, der in einer eigenen Variation in Duality umgesetzt wurde (Abb. 6.5). Wie auch im Originalspiel geht es darum, von oben herab fallende Blöcke verschiedener Form so abzusetzen, dass sich geschlossene horizontale Reihen bilden. Diese werden entfernt und darüber liegende Blöcke werden eine Reihe herab gesetzt. Einmal abgesetzte Blöcke können vom Spieler jedoch nicht mehr bewegt werden: Das Spiel endet, wenn die durch unvollständige Reihen gewachsene Konstruktion den oberen Spielfeldrand erreicht. Die in Duality umgesetzte Variante unterscheidet sich primär durch die physikalische Simulation der einzelnen Blöcke, welche im Original in dieser Form nicht vorhanden ist: Blöcke können stufenlos bewegt werden, fallen bei ungünstiger Positionierung jedoch einfach um. Bei den Tests des Spiels auf besonders Leistungsschwachen Systemen zeigten sich erstmals Performanceprobleme durch die bisher stets in maximaler Qualität ausgeführte Voll-

bild Kantenglättung. Durch die Einführung verschiedener Qualitätsstufen konnten die Probleme auf den entsprechenden Systemen jedoch umgangen werden.

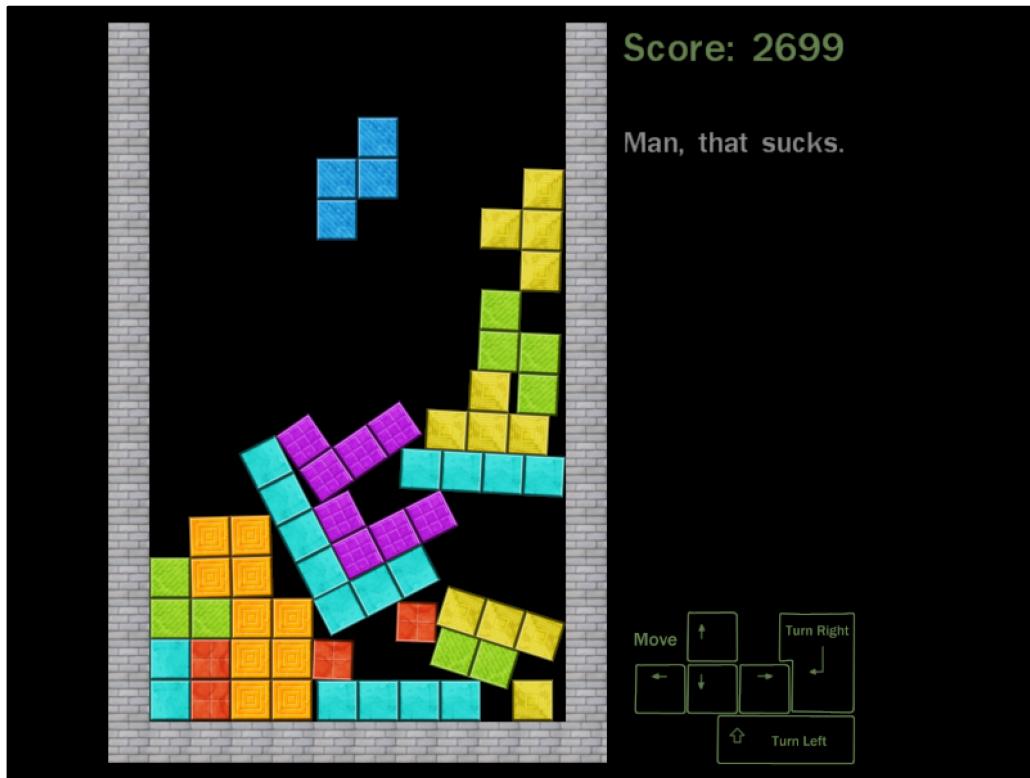


Abbildung 6.5: Screenshot der umgesetzten Tetris Variante

FluSi war der Arbeitstitel eines im Rahmen der Lehrveranstaltung »Projekt Systementwicklung« im Sommersemester 2012 eigens entwickelten Flugsimulators (Abb. 6.6). Ziel des zehnköpfigen Teams war es, nicht nur die für ein angemessenes Nutzererlebnis notwendige Hardware⁵ zu beschaffen, testen und aufzubauen, sondern auch die zugehörige Software von Grund auf selbst zu entwickeln. Insbesondere letzteres wäre rückblickend betrachtet innerhalb des gegebenen Zeitrahmens kaum ohne Zuhilfenahme der Duality Engine möglich gewesen: Natürlich ist FluSi kein in Duality entwickeltes Projekt - jedoch kann die im Rahmen des Projekts entwickelte Engine als ein Derivat der Duality Engine bezeichnet werden, da abgesehen vom Renderingsystem ein Großteil der wesentlichen Strukturen ohne größere Änderungen übernommen werden konnten. Im Projektverlauf zeigte

⁵Holzgerüst, Beamer, Joystick, diverse Ein- und Ausgabepanele, »Pilotensitz«, usw.

sich ein weiterer Vorteil des Komponentenbasierten Ansatzes zur Organisation von Objekten: Da große Teile der Engine Funktionalität auf diese Weise in klar getrennten Paketen existieren, war es ohne Probleme möglich mit einer vergleichsweise hohen Zahl von Entwicklern an einer gemeinsamen Codebasis zu arbeiten, ohne dass es zu Konflikten in der Versionierung kam.

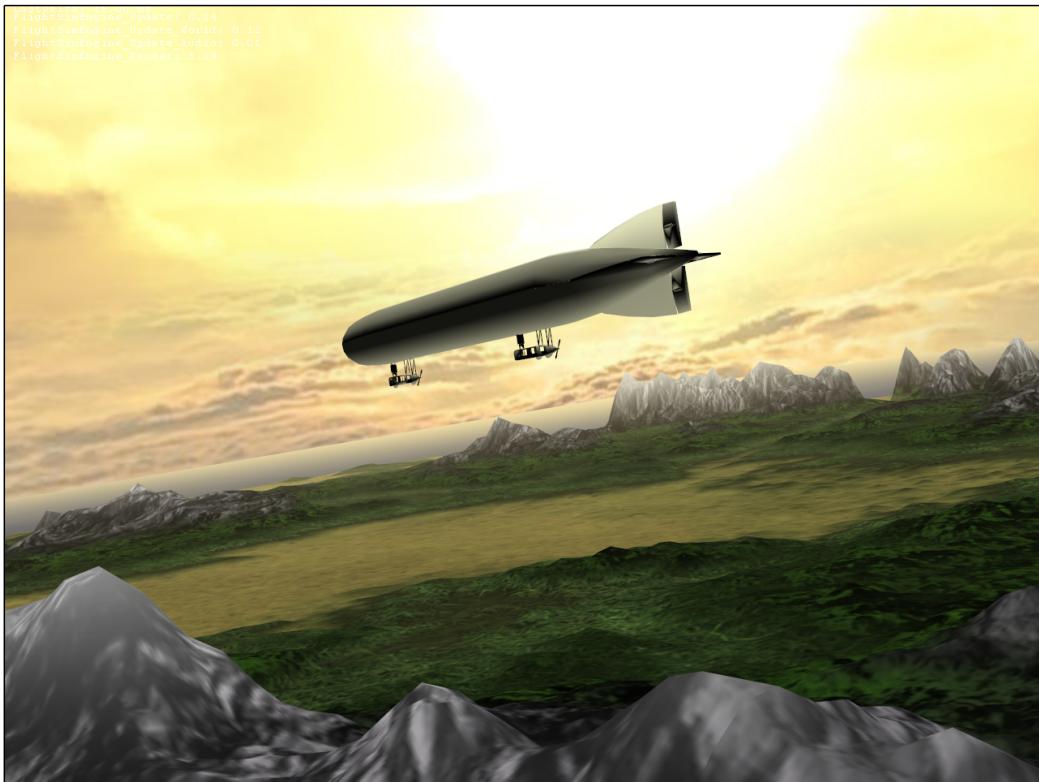


Abbildung 6.6: Screenshot des FluSi Projekts

Insgesamt konnte die Duality Engine durch stetige Verbesserungen alle der anfänglich gesetzten Ziele (Abschnitt 1.2) in zufriedenstellendem Maß erfüllen - das volle Potential von Engine und Editor ist jedoch keinesfalls ausgeschöpft. Durch die bisher vergleichsweise geringe Zahl von Testern und Testprojekten ist in Anbetracht der Größe des Gesamtsystems davon auszugehen, dass eine moderate Zahl von Bugs noch immer unentdeckt ist - diese gilt es zu finden und zu beheben. Darüber hinaus gibt es neben den bereits erwähnten Punkten eine Vielzahl von Erweiterungsmöglichkeiten für die derzeitige Standardfunktionalität:

Für die komfortable Erstellung von komplexen visuellen Effekten wären Par-

tikelsysteme sowie ein entsprechender Partikeleditor wünschenswert. Bisher können Partikel mithilfe der Standardfunktionalität lediglich durch einzelne Sprite Objekte repräsentiert werden, was einerseits ein hinsichtlich der Performanz ineffizientes Verfahren darstellt und andererseits eine Programmierung von individuellem Partikelverhalten anstelle eines datengetriebenen Ansatzes notwendig macht. Für den produktiven Einsatz ist diese Vorgehensweise nicht geeignet.

Auch gibt es bisher keine mitgelieferte Implementierung der bereits erwähnten *Tilemaps*, welche für viele zweidimensionale Spiele jedoch einen wesentlichen Aspekt grafischer Darstellung und spielerischer Gestaltung bilden. Wie auch im Falle der Partikelsysteme ist es natürlich für den Nutzer möglich, diese Funktionalität selbst nachzurüsten - allerdings würde sich eine standardisierte Umsetzung aufgrund der hohen Verbreitung von Tilemap Konzepten durchaus anbieten.

Literaturverzeichnis

- [1] MICROSOFT: *Overview of the .NET Framework*
2012,
<http://msdn.microsoft.com/en-us/library/zw4w595w.aspx>, Abruf am 02.12.2012
- [2] MICROSOFT: *Pointer types (C# Programming Guide)*
2012,
[http://msdn.microsoft.com/en-us/library/y3iyhkeb\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/y3iyhkeb(v=vs.100).aspx), Abruf am 02.12.2012
- [3] MICROSOFT: *Common Type System*
2012,
[http://msdn.microsoft.com/en-us/library/zcx1eb1e\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/zcx1eb1e(v=vs.100).aspx), Abruf am 03.12.2012
- [4] GOMILLA, LAURENT: *Simple and Fast Multimedia Library*
<http://www.sfml-dev.org/>, Abruf am 09.11.2012
- [5] WIKIPEDIA: *Rpg Maker 2000*
http://de.wikipedia.org/wiki/RPG_Maker#RPG_Maker_2000, Abruf am 09.11.2012
- [6] WIKIPEDIA: *Component-based software engineering*
http://en.wikipedia.org/wiki/Software_componentry, Abruf am 22.11.2012
- [7] UNITY TECHNOLOGIES: *Unity Game Engine*
<http://unity3d.com/>, Abruf am 09.11.2012
- [8] FIRELIGHT TECHNOLOGIES: *FMOD Audio Middleware*
<http://www.fmod.org/>, Abruf am 10.11.2012
- [9] NARAYANASWAMY, ANAND: *What is an Assembly?*
2. Februar 2004,
http://www.codeguru.com/columns/csharp_learning/article.php/c5845/C-FAQ-15--What-is-an-Assembly.htm, Abruf am 11.11.2012
- [10] MICROSOFT: *Reflection in the .NET Framework*
2012,
[http://msdn.microsoft.com/en-us/library/f7ykdhsy\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.100).aspx), Abruf am 14.11.2012
- [11] ZANDER, JASON: *Why isn't there an Assembly.Unload method?*
31. Mai 2004,
<http://blogs.msdn.com/b/jasonz/archive/2004/05/31/145105.aspx>, Abruf am 14.11.2012
- [12] MICROSOFT: *Application Domains*
2012,
<http://msdn.microsoft.com/en-us/library/2bh4z9hs.aspx>, Abruf am 14.11.2012

- [13] PAPAGIANNIS, IOANNIS: *What is the minimun Cross AppDomain communication performance penalty?*
17. Juli 2009,
<http://stackoverflow.com/questions/1144459/what-is-the-minimun-cross-appdomain-communication-performance-penalty>, Abruf am 14.11.2012
- [14] WEST, MICK: *Evolve Your Hierarchy*
05. Januar 2007,
<http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/>, Abruf am 05.01.2013
- [15] ADAM, FEDJA: *Entwicklung eines Asset Management Systems*
2. November 2012 in "Praxisbericht des Bachelorprojekts"
- [16] WIKIPEDIA: *Serialization*
<http://en.wikipedia.org/wiki/Serialization>, Abruf am 19.11.2012
- [17] T-C: *Load and save objects to XML using serialization*
1. September 2006,
<http://www.codeproject.com/Articles/4491/Load-and-save-objects-to-XML-using-serialization>,
Abruf am 19.11.2012
- [18] WIKIPEDIA: *Bildsynthese*
<http://de.wikipedia.org/wiki/Bildsynthese>, Abruf am 26.11.2012
- [19] WIKIPEDIA: *Rasterisation*
<http://en.wikipedia.org/wiki/Rasterisation>, Abruf am 26.11.2012
- [20] WIKIPEDIA: *OpenGL*
<http://en.wikipedia.org/wiki/OpenGL>, Abruf am 26.11.2012
- [21] WIKIPEDIA: *DirectX*
<http://en.wikipedia.org/wiki/DirectX>, Abruf am 26.11.2012
- [22] PERSSON, EMIL: *Alpha to coverage*
23. Juni 2005,
<http://www.humus.name/index.php?page=3D&ID=61>, Abruf am 19.11.2012
- [23] WIKIPEDIA: *Windows Forms*
http://en.wikipedia.org/wiki/Windows_Forms, Abruf am 13.12.2012
- [24] WIKIPEDIA: *Teile und herrsche (Informatik)*
[http://de.wikipedia.org/wiki/Teile_und_herrsche_\(Informatik\)](http://de.wikipedia.org/wiki/Teile_und_herrsche_(Informatik)), Abruf am 16.12.2012
- [25] PETRIE, JOSH: *Make Games, Not Engines*
30. August 2007,
<http://scientificninja.com/blog/write-games-not-engines>, Abruf am 07.01.2012