HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Bachelor Thesis

in

Allgemeine Informatik

Entwicklung eines zustandslosen Partikelsystems für den Furtwangen University Didactic Game Editor

Referent:	Prof. Dr. Lothar Piepmeyer
Korreferent:	Prof. Jirka Dell´Oro-Friedl
Vorgelegt am:	21.08.2020
Vorgelegt von:	Jonas Plotzky
	256539
	Friedrichring 7
	79098 Freiburg im Breisgau
	jonas.plotzky@hs-furtwangen.de

Abstract

There are many game engines available today to develop games with. Most of them are only partially suited to teach the development of games. To solve this problem, Furtwangen University Didactic Game Editor (FUDGE) is being developed. FUDGE is a didactic game editor and engine that is based solely on web technologies.

In the context of this bachelor thesis a stateless particle system was developed for FUDGE. Particle systems are used to display various graphical effects such as smoke or fire. A stateless particle system is based on the concept that the state of the system can be fully calculated at any given time using various parameters. This way a lot of performance can be saved in certain situations compared to state-preserving solutions, which is highly important when working with web technologies.

Heute stehen viele Spiel-Engines zur Entwicklung von Videospielen zur Verfügung. Davon sind die meisten aber nur bedingt geeignet, um die Entwicklung von Spielen zu lehren. Aus diesem Grund wird der Furtwangen University Didactic Game Editor (FUDGE) entwickelt. FUDGE ist ein auf die Lehre zugeschnittener Spiele-Editor (und Engine) und basiert komplett auf Webtechnologien.

Im Rahmen dieser Bachelorarbeit wurde ein zustandsloses Partikelsystem für FUDGE entwickelt. Partikelsysteme werden genutzt, um verschiedene grafische Effekte wie Rauch oder Feuer darzustellen. Ein zustandsloses Partikelsystem basiert darauf, dass der Zustand des Systems sich über verschiedene Parameter zu jedem Zeitpunkt vollständig berechnen lässt. Gegenüber zustandsbehafteten Lösungen kann so in bestimmten Situationen viel Performance eingespart werden, was im Bereich der Webtechnologien besonders wichtig ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CPU	Central Processing Unit
CSS	Cascading Style Sheet
DOM	Document Object Model
FUDGE	Furtwangen University Didactic Game Editor
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation
UML	Unified Modeling Language
WebGL	Web Graphics Library

1 Einleitung

Computer- und Videospiele sind das Unterhaltungsmedium der Zukunft. Der Spielemarkt entwickelt sich weltweit mit einer rasanten Geschwindigkeit. Andere Unterhaltungsbranchen wie Kino oder die Musikindustrie hat der Spielemarkt längst überholt. So hat die deutsche Spiele-Branche im Jahr 2018 mit 4,4 Milliarden Euro etwa 3-mal so viel wie die deutsche Musikindustrie und 5-mal so viel wie die deutsche Kinobranche erwirtschaftet [1].

Videospiele werden zum Großteil mithilfe von Spiel-Engines entwickelt. Spiel-Engines dienen dabei meist als Entwicklungsumgebung und stellen gleichzeitig ein weites Spektrum an Grundfunktionalitäten für die Spieleentwicklung zur Verfügung, wodurch der Spieleentwicklungsprozess beschleunigt und vereinfacht wird.

Aufgrund der großen Popularität des Mediums Spiel hat die Entwicklung von Spielen auch schon Einzug in die Hochschulen gehalten. Dies ist auch an der Hochschule Furtwangen University der Fall. Hier bietet Professor Jirka Dell’Oro-Friedl seit 2010 verschiedene Kurse im Bereich Spiel-Design und Entwicklung an. Dabei waren die meistverwendeten Werkzeuge bisher die Unity Spiel-Engine und Adobe Animate, welche beide aufgrund ihrer visuellen Editoren als besonders einsteigerfreundlich angesehen werden [2].

Diese Programme sind aber dennoch nicht für die Lehre konzipiert. Gerade der Gebrauch zusammen mit Versionskontrollsystemen wie GitHub erweist sich als schwierig. Da die Kurse von Professor Jirka Dell’Oro-Friedl ihren Fokus aber auf praktischer Übung außerhalb der Vorlesungen und Seminare haben, ist in ihnen Fernkommunikation und der Austausch sowie die Prüfung von Ergebnissen über das Internet für eine erfolgreiche Didaktik von entscheidender Bedeutung. Aus diesem Grund wird die Entwicklung des Furtwangen University Didactic Game Editor (FUDGE) vorangetrieben [2].

FUDGE ist sowohl Spiel-Engine als auch Editor und wird auf Basis der modernen Webtechnologien entwickelt. Der Fokus von FUDGE liegt auf der Lehre der Spieleentwicklung. Kriterien wie Performance, visuelle Effekte und

Sicherheit spielen im Gegensatz zu den etablierten Spiele-Engines eine untergeordnete Rolle. Mehr zu FUDGE ist in Kapitel 2.2 zu lesen [2].

Eine der in Spiel-Engines für gewöhnlich bereitgestellten Funktionalitäten ist ein Partikelsystem. Ein Partikelsystem kann dafür genutzt werden, visuelle Effekte darzustellen, die sich aus vielen kleinen Partikeln zusammensetzen. Solche Effekte können z.B. Rauch, Feuer, Explosionen oder Regen sein.

Auch FUDGE benötigt ein Partikelsystem. Für die Implementierung eines Partikelsystems gibt es zwei grundsätzlich unterschiedliche Ansätze: Partikelsysteme sind entweder zustandslos oder zustandsbehaftet. Diese zwei Arten von Systemen unterscheiden sich in vielen ihrer Eigenschaften. Mehr zu Partikelsystemen gibt es in Kapitel 2.4 zu lesen.

Ziel dieser Arbeit war es, ein zustandsloses Partikelsystem zu entwickeln und in die FUDGE-Architektur einzubetten. Die genauen Anforderungen an das Partikelsystem können in Kapitel 3.1 nachgelesen werden. Die Anforderungen ergeben dabei erst Sinn, wenn die Grundlagen zu FUDGE und Partikelsystemen verstanden wurden.

2 Grundlagen

Die Grundlagen dienen dem Verstehen der weiteren Arbeit. FUDGE und damit auch das in dessen Strukturen entwickelte Partikelsystem basiert auf den hier vorgestellten Technologien, Konzepten und Strukturen.

2.1 Webtechnologien

Die modernen Webtechnologien sind weitverbreitet, hochstandardisiert und stellen bereits ein weites Spektrum an Funktionalitäten zur Verfügung. Mithilfe dieser Technologien werden heute komplette interaktive Anwendungen in Webbrowsern implementiert.

FUDGE basiert auf den modernen Webtechnologien. Die vielen bereits vorhandenen Technologien und damit einhergehenden Möglichkeiten müssen somit nicht von Grund auf neu entwickelt werden, sondern werden vielmehr unter FUDGE vereint und zentralisiert. Die für diese Arbeit wichtigen Webtechnologien werden in den folgenden Kapiteln erläutert.

2.1.1 HTML und DOM

Die Hypertext Markup Language (HTML) bildet zusammen mit dem in den Browsern erzeugten Document Object Model (DOM), die Grundstruktur für Webseiten. Über HTML wird der Inhalt und Aufbau einer Webseite definiert. Über das DOM werden die Inhalte der Webseite dann für die Scripting API (siehe unten) zur Verfügung gestellt.

Der HTML Standard ist sehr umfangreich und in ihm sind viele Funktionalitäten bereits vorgesehen. Für die vorliegende Arbeit sind aber nur zwei dieser Funktionalitäten von besonderer Bedeutung.

Das Scripting Application Programming Interface (API) bietet die Möglichkeit mithilfe von JavaScript (siehe 2.1.2) die Objekte und Inhalte eines HTML-Dokuments zu manipulieren.

Das Canvas-Element, stellt eine Fläche dar, auf der über die Scripting API gezeichnet werden kann. Zum Zeichnen stehen hierfür verschiedene Formen wie Linien, Kreise oder auch Text zur Verfügung. Über das Canvas-Element

steht auch die Web Graphics Library (WebGL, siehe 2.1.5) Schnittstelle zur Verfügung.

2.1.2 JavaScript

JavaScript ist eine Skriptsprache, mit der sich die Inhalte von Webseiten dynamisch manipulieren lassen. Webseiten können mit Hilfe von JavaScript interaktiv gestaltet werden.

Der Sprachkern von JavaScript ist unter dem Namen ECMAScript standardisiert. ECMAScript wird hier als objektorientierte Programmiersprache mit dynamischer Typisierung beschrieben. ECMAScript Objekte sind nicht fundamental klassenbasiert wie etwa in C++ und Java [3].

2.1.3 JSON

Bei der JavaScript Object Notation (JSON) handelt es sich um eine textbasierte, sprachunabhängige Syntax zum Definieren von Datenaustauschformaten. Die Spezifikation dient nur dem Definieren einer validen Syntax. Die Semantik bzw. Interpretation der Syntax ist nicht in JSON mit enthalten [4]. Die Semantik zum Interpretieren des definierten Formats wird also erst in der Programmiersprache festgelegt. JSON-Dateien können in JavaScript standardmäßig eingelesen werden.

2.1.4 TypeScript

Bei TypeScript handelt es sich um eine auf JavaScript aufgebauter Programmiersprache. TypeScript kann dabei als Obermenge von JavaScript angesehen werden und erweitert dieses um sein Typsystem. Hierbei handelt es sich um ein strukturelles Typsystem, das sowohl die dynamische als auch die statische Typisierung erlaubt [5]. Die Kombination aus dynamischer und statischer Typisierung wird im Englischen auch als gradual typing bezeichnet. In einem strukturellen Typsystem werden zwei Objekte als gleich betrachtet, wenn sie dieselbe Form aufweisen. TypeScript ergänzt JavaScript um Konzepte wie Klassen, Interfaces und Vererbung. Auch das Paradigma der funktionalen Programmierung wird in TypeScript unterstützt. Da es sich um eine Obermenge handelt, ist jeglicher JavaScript-Code auch gültiger TypeScript-Code. Für die Verwendung wird TypeScript-Code zu JavaScript-Code kompiliert. Dieser kann dann vom Browser interpretiert werden.

In dieser Arbeit werden einige Strukturen verwendet, in denen sich TypeScript von anderen objekt-orientierten Programmiersprachen wie Java oder C++ unterscheidet. Diese Strukturen werden dann erst im Kontext ihrer Anwendung erklärt.

2.1.5 WebGL

Die Web Graphics Library ist eine plattformübergreifende API zum Erstellen von hardwarebeschleunigter 3D-Grafik in Webbrowsern. WebGL basiert auf der Open Graphics Library (OpenGL). Die WebGL API steht über das HTML Canvas-Element zum Programmieren zur Verfügung [6]. Über die WebGL kann die Graphics Processing Unit (GPU) zum Berechnen von grafischen Inhalten genutzt werden.

2.2 FUDGE

Der Furtwangen University Didactic Game Editor[1] ist ein Open Source Projekt, das von Professor Jirka Dell'Oro-Friedl von der Hochschule Furtwangen ins Leben gerufen wurde und unter dessen Leitung entsteht.

Das Hauptziel von FUDGE ist es, die Strukturen und Prozesse von Spielen und deren Entwicklung klar darzustellen und verständlich zu machen. Dabei sollte die Bedienbarkeit möglichst einfach und die Flexibilität und Anwendungsmöglichkeiten möglichst hochgehalten werden. Alle Designentscheidungen müssen sich diesem Paradigma unterwerfen [2].

Zum Erreichen seiner Ziele basiert FUDGE vollständig auf Webtechnologien. Dadurch, dass FUDGE Anwendungen komplett im Browser laufen, sind sie weitestgehend plattformunabhängig. Hierbei wird auf die bereits etablierten Standards zurückgegriffen. Graphical User Interfaces (GUI) können etwa mithilfe der im HTML Standard vorgesehenen Bedienelemente (Buttons, Eingabefelder, etc.) und über Cascading Style Sheets (CSS) implementiert werden. Für die grafische Darstellung der Spielwelt greift FUDGE auf das Canvas-Element zu und nutzt die Schnittstellen von WebGL. Programmiert wird im Web meist mit JavaScript. FUDGE Anwendungen werden in

[1] https://github.com/JirkaDellOro/FUDGE

TypeScript geschrieben. Das Kompilieren von TypeScript zu JavaScript kostet nur wenig Zeit. Dadurch können Spiele sehr schnell lokal getestet werden. Zum Abspeichern von Daten bietet sich das JSON Format an, das extra zusammen mit JavaScript entwickelt wurde. Des Weiteren bieten moderne Browser auch meist umfangreiche Werkzeuge an, um die in ihnen laufenden Anwendungen zu untersuchen. So kann mithilfe vom Debugger einfach die eigene lauffähige Anwendung direkt nach Fehlern durchsucht werden. Dieser Prozess gestaltet sich in Anwendungen, welche in anderen Spiel-Engines erstellt wurden, meist schwieriger. Ein kompiliertes Spiel kann hier nicht mehr mit dem Debugger untersucht werden. Stattdessen muss extra der Quellcode heruntergeladen und in der Engine geöffnet werden, was sich aus der Ferne als umständlich erweist. Ein Performance Monitor (z.B. im Webbrowser Chrome) kann einen tiefen Einblick in die Performance der geschaffenen Anwendung geben. Mit seiner Hilfe lassen sich Optimierungen am Code direkt überprüfen.

Die angestrebten Hauptfeatures von FUDGE bestehen darin, dass sich ein Spiele-Prototyp fast vollständig in FUDGE bauen lassen soll. Dazu sollen alle notwendigen Aufgaben für die Erstellung innerhalb des Editors erledigt werden können. Hierzu gehört das Erstellen der Spielszenen und Skripte wie auch die Erstellung von Grafiken und Animationen und vielleicht in Zukunft auch noch von Soundeffekten. Alle Datenstrukturen sollten hierbei in für Menschen lesbaren Formaten hinterlegt werden so dass sie einfach eingesehen und manipuliert werden können. Auch eine einfache Versionskontrolle (z.B über GitHub) sollte unterstützt werden, damit Arbeiten im Team und die Nachvollziehbarkeit für die Lehrenden verbessert werden können [2].

2.3 Aufbau von FUDGE

Im Folgenden werden Teile des grundlegenden Aufbaus und der Funktionsweise von FUDGE betrachtet. Dabei werden nur Teile betrachtet, die für diese Arbeit relevant sind.

2.3.1 Szenengraph

FUDGE basiert auf dem als Szenengraph bezeichneten Grundmodell, auf dem die meisten modernen Spiel-Engines basieren. Bei einem Szenengraph

handelt es sich um einen aus Knoten (nodes) und Kanten (edges) bestehenden gerichteten Graphen. Ein Knoten hält Informationen über Teile der darzustellenden Szene. Über die Kanten sind Knoten in einer Eltern-Kind-Beziehung verbunden:

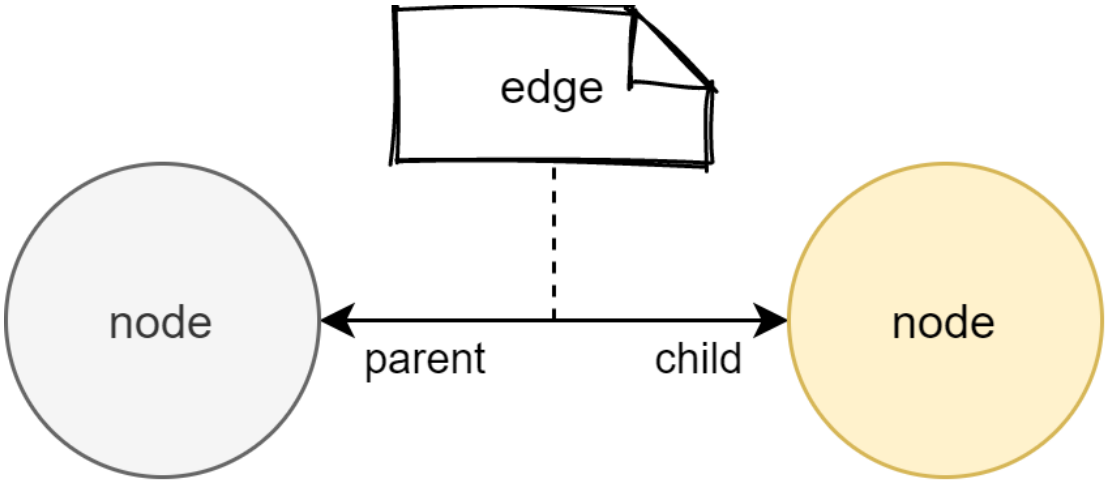

Abbildung 1: Eltern-Kind-Beziehung

Ein Knoten kann mehrere Kind-Knoten haben und hat jeweils einen Eltern-Knoten. Der Szenengraph beschreibt also eine hierarchische Baumstruktur (siehe Abbildung 2).

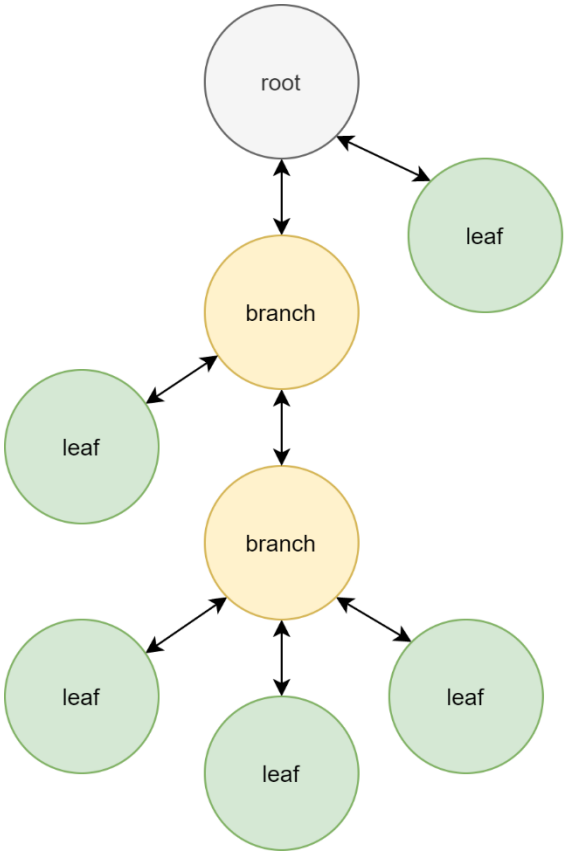

Abbildung 2: Baumstruktur

Eine Szene wird in ihrer Gesamtheit über einen solchen Graphen beschrieben. Ein Knoten enthält eine Liste von Komponenten (siehe 2.3.2), die an diesen angeheftet sind. Diese Komponenten enthalten die Informationen darüber, was der Knoten visuell darstellen soll, ob er Geräusche erzeugen soll und ob er sich besonders verhalten soll. Da die Knoten eine 3-dimensionale Szene beschreiben, brauchen sie eine Position und Orientierung im Spielraum. Diese Informationen kann dem Knoten über die sogenannte Transformations-Komponente angehängt werden. Der Szenengraph wird dann beim Rendern (siehe 2.3.3) einer Szene verwendet. Die Baumstruktur hat dabei zahlreiche Vorteile beim Berechnen von Transformationen.

2.3.2 Komponenten

In FUDGE wird das Paradigma Komposition über Vererbung verfolgt. Anstelle von einer tiefen Klassenhierarchie wird eine flache Hierarchie bevorzugt. Objekte werden in ihrer Funktionalität dadurch erweitert, dass ihnen kleine Komponenten hinzugefügt werden. Aus diesem Grund existiert in FUDGE auch nur eine einfache Klasse Node (englisch für Knoten) und keine Spezialisierungen wie MeshNode. Ein Mesh, ein Gitternetz, welches die Form eines 3D-Körpers bildet und andere Funktionalitäten werden einem Knoten einfach mithilfe von Komponenten angehängt [2].

Um dieses Schema umzusetzen, gibt es in FUDGE die abstrakte Oberklasse, Component (Komponente), von der die verschiedenen konkreten Komponentenklassen erben. Jede Komponenteninstanz, die einem Knoten angehängt wird, hält eine Referenz auf den Knoten, an dem sie hängt. Es kann unterschieden werden zwischen in sich geschlossenen Komponenten (self contained) und Komponenten, die auf eine Ressource verweisen. Eine in sich geschlossene Komponente ist die vorher erwähnte Transformations-Komponente (ComponentTransform). Diese fügt dem Knoten, wie bereits erwähnt, eine räumliche Position und Orientierung hinzu, welche in Form einer Matrix in der Komponente gespeichert wird. Eine auf eine Ressource verweisende Komponente ist die Material-Komponente (ComponentMaterial). Diese fügt dem Knoten ein zuvor definiertes Material hinzu, welches dann zusammen mit dem angebrachten Mesh (in der Mesh-Komponente) zum Rendern des Knotens genutzt wird. Manche Komponenten enthalten sowohl

eine Ressource als auch noch zusätzliche eigene Informationen. Ein Beispiel hierfür ist die Mesh-Komponente (ComponentMesh). Neben der Referenz auf ein existierendes Mesh enthält sie zusätzlich noch eine sogenannte Pivot-Matrix, in der die räumliche Position und Orientierung des Meshs relativ zur Transformation (enthalten in der Transformations-Komponente) des Knotens bestimmt werden kann [2].

In Abbildung 3 ist ein Diagramm zu sehen, das die zuvor beschriebenen Beziehung zwischen Knoten, Komponenten und Ressourcen darstellt.

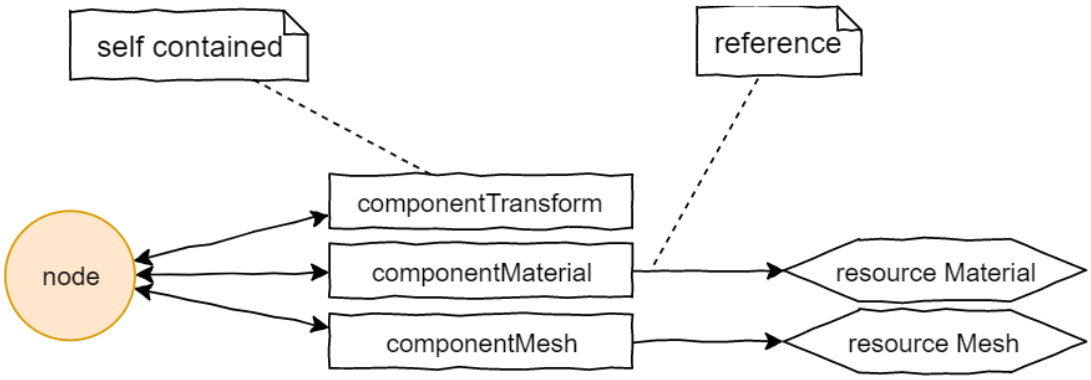

Abbildung 3: Beziehung zwischen Knoten, Komponenten und Ressourcen [2]

Zum besseren Verständnis, wie Knoten, Komponenten und Ressourcen zusammenspielen, soll folgendes Beispiel dienen. In Abbildung 4 ist ein Canvas nach dem Rendern zu sehen. In der dargestellten Szene wurden an den Wurzelknoten zwei Knoten angehängt. Beiden Knoten wurde eine Transformations-, Material- und Mesh-Komponente angehängt. Die Mesh-Komponenten der Knoten verweisen dabei beide auf dasselbe 2-dimensionale Rechteck-Mesh. Durch dieses erhalten die Knoten ihre Form. Die Material-Komponenten verweisen auf zwei unterschiedliche Materialien: ein rotfarbiges und ein blaufarbiges. Über das Material wird die Oberflächenbeschaffenheit der Knoten angegeben. Über die jeweilige Transformations-Komponente ist die Transformation der Knoten beschrieben. In diesem Fall jeweils eine Verschiebung auf der X-Achse in entgegengesetzte Richtung. Das Ergebnis ist also ein rotes und ein blaues Rechteck. Die Rechtecke haben die gleiche Form und befinden sich an verschiedenen Positionen im Raum.

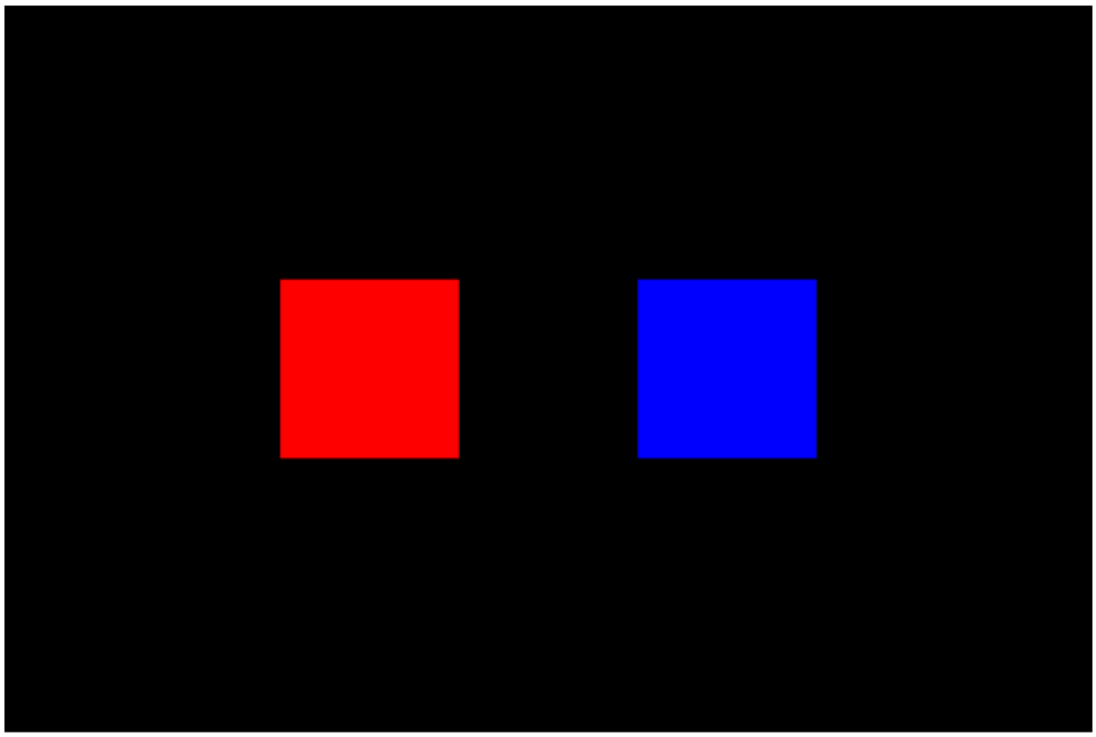

Abbildung 4: Zwei Knoten mit demselben Mesh und unterschiedlichem Material

2.3.3 Rendering

Das Rendering beschreibt den Prozess des Malens der Spielwelt bzw. einer Szene. Soll ein Spiel mit 30 Bildern pro Sekunde laufen, muss auch 30-mal pro Sekunde gerendert werden. Wie oben beschrieben, wird zum Rendern der Szenengraph verwendet. In FUDGE ist der sogenannte RenderManager für das Rendern verantwortlich.

Zusätzlich zum Szenengraph wird zum Rendern noch eine Kamera-Komponente, die angibt, wie die Szene betrachtet wird und ein Canvas-Element, auf dem gerendert wird, benötigt. Kamera, Canvas und ein Szenengraph werden jeweils in einem Viewport gebündelt. In einer FUDGE Anwendung kann es mehrere Viewports geben. Die Viewports enthalten also die zum Rendern benötigten Komponenten.

Beim Rendern wird ein Szenenbaum vom Wurzelknoten aus durchlaufen und für jeden Knoten eine Welt-Transformations-Matrix berechnet. Die Welt-Transformations-Matrix ergibt sich für jeden Knoten aus dem Produkt der Welt-Transformations-Matrix des Elternknotens mit seiner Lokal-Transformations-Matrix. Die Lokal-Transformations-Matrix eines Knotens befindet sich in seiner

Transformations-Komponente und gibt die Transformation relativ zur Transformation des Elternknotens an. Die so berechnete Welt-Transformations-Matrix beschreibt dann also jeweils die absolute Position und Orientierung eines jeden Knotens im Spielraum. Beim Rendern eines Knotens wird jetzt noch das Produkt der Welt-Transformations-Matrix und der Pivot-Matrix (aus der Mesh-Komponente des Knoten) gebildet. Daraus ergibt sich dann die finale Transformation des Knotens.

Die finale Transformation wird dann zusammen mit der Kamera, dem Mesh und dem Material dafür benutzt, um den entsprechenden Knoten auf dem Canvas zu rendern. Hierbei wird erst im letzten Schritt die WebGL-Schnittstelle des Canvas genutzt und ihr die benötigten Informationen übergeben. Das eigentliche Rendern des Bildes findet dann also mithilfe der GPU statt. Die vorangegangene Berechnung der finalen Transformation findet auf der Central Processing Unit (CPU) statt.

2.3.4 Ressourcen

Eine Ressource in FUDGE beschreibt geteilte Daten, die mehrfach referenziert werden. Daten sollten nur an einer Stelle existieren und von dort aus, wo sie benötigt werden, referenziert werden. Hingegen erhöhen redundante Daten den Speicherbedarf und die Fehleranfälligkeit des Codes. In einer Szene kann es z.B. viele Knoten geben, die ein würfelförmiges Mesh besitzen. Für den Renderprozess ist es dann genug, wenn dieses Mesh einmal vorliegt und es an vielen verschiedenen Orten gerendert wird. Wenn die Würfel nicht alle gleichförmig sein sollen, kann dies dann über die zugehörigen Transformationen manipuliert werden [2].

2.3.5 Mutatoren

In FUDGE existiert ein generischer Prozess zum Übertragen von Informationen von und auf Objekte. Die Objekte, die diesen Prozess unterstützen, erben von der abstrakten Oberklasse Mutable. Die Struktur, in der die Informationen übertragen werden, wird als Mutator bezeichnet [2].

Dieser Prozess wird hier zunächst erläutert und dann anhand eines Beispiels erklärt.

Klasseninstanzen, die von Mutable erben, verfügen über die Funktionalität Mutatoren von sich zu erstellen und sich selbst nach einem gegeben Mutator zu mutieren. Ein Mutator besteht aus einem assoziativen Array, in welchem entweder alle oder Teile der Eigenschaften der Klasseninstanz gespeichert werden. In einem Mutator befinden sich alle Eigenschaften und ihre Werte mit Ausnahme von Methoden und Referenzen auf Objekte, welche selbst nicht von Mutable erben. Je nach Unterklasse können dann noch unnötige Werte entfernt werden. Referenzen auf Objekte, die von Mutable erben, werden rekursiv durch Mutatoren ersetzt. Ein erzeugtes Mutator Objekt besteht dann also nur aus Mutatoren und einfachen Typen, die das originale Mutable Objekt repräsentieren. Beim Mutieren eines Mutable können noch zusätzliche Prüfungen der zu überschreibenden Werte stattfinden [2].

Zum besseren Verständnis, wie Mutable und Mutatoren funktionieren, soll das in Listing 1 zu sehende Beispiel dienen. Es handelt sich hierbei nur um einen Beispielcode. Die entsprechenden Klassen in FUDGE weichen in ihrem Aufbau von diesem leicht ab. Die Klasse Rectangle erbt von Mutable und verfügt über zwei Attribute der Klasse Vector2. Vector2 erbt ebenfalls von Mutable und besteht aus zwei Attributen vom Typ number. Nun werden zwei Instanzen von Rectangle erzeugt: rectangle1 und rectangle2. Im Konstruktor wird erst die Position und dann die Größe angegeben. Daraus werden dann die entsprechenden Vektoren erstellt.

```
class Rectangle extends Mutable {
  public position: Vector2;
  public size: Vector2;
  …
}

class Vector2 extends Mutable {
  private x: number;
  private y: number;
  …
}

let rectangle1: Rectangle = new Rectangle(2, 2, 1, 1);
let rectangle2: Rectangle = new Rectangle(0, 0, 0, 0);

let mutator: Mutator = rectangle1.getMutator();
rectangle2.mutate(mutator);
```

Listing 1: Beispiel Mutable

Über die Methode getMutator wird jetzt ein Mutator von rectangle1 erstellt. Das erzeugte Mutator-Objekt wird durch ein assoziatives Array beschrieben. Ein assoziatives Array besteht aus Schlüssel-Werte-Paaren. Der Mutator enthält dann genau zwei Schlüssel: „position“ und „size“. Die Werte hinter diesen Schlüsseln sind auch wieder Mutatoren und haben jeweils die Schlüssel „x“ und „y“ mit den korrespondierenden Werten eingetragen. Das erzeugte Mutator-Objekt hat also folgende Form:

```
{
  "position": {
    "x": 2,
    "y": 2
  },
  "size": {
    "x": 1,
    "y": 1
  }
}
```

Listing 2: Beipiel Mutator

Über die Methode mutate können die Werte aus dem Mutator in rectangle2 übertragen werden. Beide Rectangle sind jetzt gleich groß und an derselben Position. Mutatoren sind von ihrer Struktur also in etwa mit der Struktur von

JSON-Objekten vergleichbar, wobei sie mit Array-Syntax initialisiert und verändert werden können.

2.4 Partikelsysteme

Partikelsysteme sind ein grundlegendes Element der Computergrafik und Animation. Mithilfe von Partikelsystemen können verschiedene Effekte für Animationsfilme und Videospiele simuliert werden. Zu diesen Effekten zählen unter anderem Feuer, Rauch, Explosionen oder auch Funken, etwa beim Schrammen von zwei metallischen Objekten. In Abbildung 5 ist auf der linken Seite zu sehen, wie ein solcher Raucheffekt aussehen kann. Auf der rechten Seite ist ein ähnlicher Effekt aus kleinen Würfeln als Partikeln zu sehen.

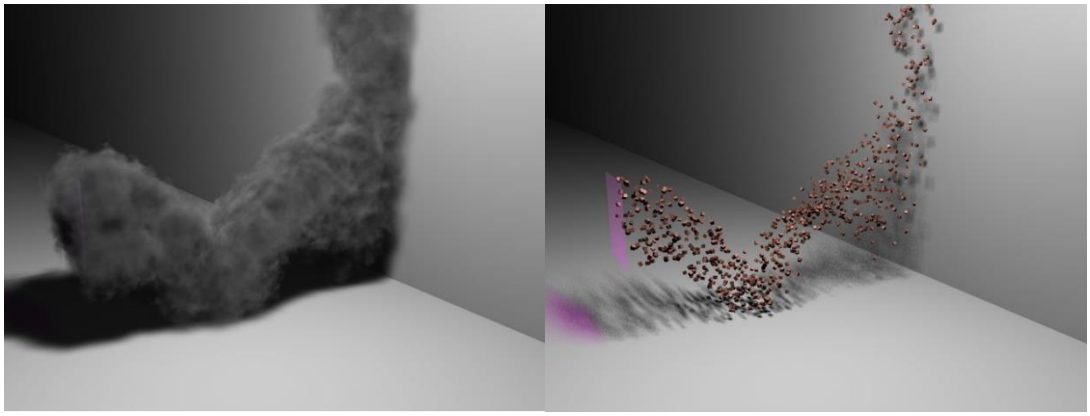

Abbildung 5: Verschiedene Anwendungen von Partikelsystemen [7]

Für die Implementierung von Partikelsystemen gibt es zwei grundsätzlich unterschiedliche Ansätze. Beide Ansätze haben ihre Vor- und Nachteile. Im Folgenden werden die zwei Ansätze vorgestellt.

2.4.1 Zustandsbehaftetes Partikelsystem

Ein zustandsbehaftetes Partikelsystem speichert seinen Zustand kontinuierlich. Bei diesem Ansatz werden alle Partikel des Systems einzeln simuliert. Dazu wird jeder Partikel mit seinen Eigenschaften (seinem Zustand) im Partikelsystem abgespeichert. Das Partikelsystem verwaltet dann alle in ihm gespeicherten Partikel. Eigenschaften eines Partikels können z.B. Position, Geschwindigkeit, Ausrichtung und Alter sein. Je nachdem, welches Verhalten von dem System erwünscht ist, können Partikel noch viele weitere Eigenschaften besitzen. Zusätzlich zu diesen Eigenschaften können dann meist noch, in Anlehnung an die Physik, sogenannte Kräfte definiert und dem

Partikelsystem übergeben werden. Das Partikelsystem sorgt dann dafür, dass die Kräfte auf die einzelnen Partikel einwirken. Eine solche Kraft kann zum Beispiel die Schwerkraft sein, also eine konstante Beschleunigung in eine festgelegte Richtung. Pro Update-Zyklus eines Spiels, also immer, wenn ein neues Bild gerendert wird, werden vom Partikelsystem alle Partikel durchlaufen und anhand ihrer Eigenschaften und der auf sie wirkenden Kräfte ihr neuer Zustand berechnet. Der neue Zustand wird dann im jeweiligen Partikel in Form einer Manipulation seiner Eigenschaften abgespeichert. So kann z.B. anhand der Geschwindigkeit und der Schwerkraft pro Bild die Position eines Partikels berechnet werden. Die einzelnen Partikel werden in der Regel von einem sogenannten Emitter erzeugt. Dieser versieht sie mit ihren Startwerten und übergibt sie zur Verwaltung an das Partikelsystem.

In einem solchen Partikelsystem werden die Zustände von jedem einzelnen Partikel gespeichert und dann ständig aktualisiert. Darum spricht man hier von einem zustandsbehafteten Partikelsystem.

Solange die Bewegung der Partikel nur von ihrer eigenen Geschwindigkeit abhängt und die auf sie wirkenden Kräfte konstant sind, verhält sich ein zustandsbehaftetes Partikelsystem deterministisch, da sich die klassische Mechanik also die zugrundeliegende Physik auch deterministisch verhält.

Die Simulation von bestimmten Eigenschaften kann in dieser Art von Partikelsystem dazu führen, dass es sich nicht mehr deterministisch verhält. Ein solcher Fall ist gegeben, wenn die Partikel sich in Weltkoordinaten bewegen sollen, d.h., dass ihre Bewegung nach dem Emittieren unabhängig von der Bewegung des Partikelsystems ist. Dies bedeutet, dass jedem Partikel zum Zeitpunkt seines Erschaffens die absolute Position des Systems übergeben werden muss. Die Bewegung des Partikelsystems muss als nicht vorhersehbar angesehen werden, da sie von externen Faktoren beeinflusst werden kann, welche nicht deterministisch sind (Spieler eingaben, Zufällen). Damit ist auch die Position des Partikelsystems nicht vorhersehbar. Seine Vergangenheit und Zukunft sind unbekannt. Daraus folgt, dass auch die Startposition der einzelnen Partikel nicht mehr vorhersehbar ist und damit das ganze Partikelsystem nicht mehr deterministisch [8].

2.4.2 Zustandsloses Partikelsystem

Ein zustandsloses Partikelsystem speichert seinen Zustand nicht. Bei diesem Ansatz werden im Gegensatz zu dem des zustandsbehafteten Partikelsystems nicht alle Partikel einzeln simuliert. Die Zustände der einzelnen Partikel werden auch nirgends abgespeichert. Stattdessen werden fest definierte mathematische Funktionen erstellt, anhand derer sich der Zustand von jedem einzelnen Partikel zu jedem gegeben Zeitpunkt berechnen lässt. Die Funktionen erhalten dafür mehrere Eingangsparameter. Der wohl wichtigste Parameter ist hier die Zeit, eine Größe, die sich kontinuierlich verändert. Ein weiterer Parameter kann der Partikelindex sein. Da in einem solchen Partikelsystem die Anzahl der Partikel meist fest angegeben ist, lassen sich die Partikel durchnummerieren und erhalten dadurch ihren Index. Diese Größe ist also für jeden Partikel unterschiedlich, aber trotzdem fest definiert. Deshalb kann sie genutzt werden, um die Bahnen der Partikel zu individualisieren. Ebenfalls sehr wichtig für die Erzeugung von realistisch wirkenden Effekten sind Zufallszahlen. Zufallszahlen als Eingangsparameter können in Abhängigkeit von der Zeit, vom Partikelindex oder von anderen Größen errechnet werden. Zeitabhängige Zufallszahlen ändern sich beim Rendern von Bild zu Bild. Indexabhängige Zufallszahlen bleiben pro Index beim Rendern von jedem neuen Bild gleich. Beim Rendern eines Bildes werden die Eigenschaften, also der Zustand, von jedem Partikel, anhand der Funktionen und Eingangsparameter berechnet und der jeweilige Partikel wird gerendert.

In einem solchen Partikelsystem werden die Zustände von jedem einzelnen Partikel stets vollständig neu evaluiert und nicht zwischengespeichert. Darum spricht man hier von einem zustandslosen Partikelsystem.

Da die Zustände der einzelnen Partikel stets berechenbar sind, ist auch der gesamte Zustand des Partikelsystems zu jedem Zeitpunkt definiert. Vergangenheit und Zukunft des Systems sind also vollständig bekannt, d.h. es verhält sich immer deterministisch.

2.4.3 Gegenüberstellung

Bevor die zwei Systeme gegenübergestellt werden können, gibt es noch eine wichtige Funktion von Spiel-Engines die bisher noch nicht erläutert wurde. Das

sogenannte Culling (englisch für auslesen). Culling beschreibt den Prozess, Objekte, die im Spiel gerade nicht sichtbar sind, zu verwerfen. Für diese Objekte müssen dann auch keine Berechnungen ausgeführt werden, was mit einem Performance-Gewinn einhergeht.

Da Partikelsysteme besonders rechenintensiv sind, kann über das Culling dieser sehr viel Rechenzeit eingespart werden. Das Culling von Partikelsystemen ist aber nur möglich, wenn diese sich vorhersehbar Verhalten. Ein zustandsloses Partikelsystem ist immer deterministisch. In diesem kann einfach zu jedem möglichen Zeitpunkt gesprungen werden. Wenn es nun also nicht mehr sichtbar ist, kann es vollständig angehalten werden. Wenn es wieder sichtbar wird, befindet es sich direkt in dem Zustand, in dem es sich nach der vergangenen Zeit befinden sollte. Ein zustandsbehaftetes Partikelsystem verhält sich meist nicht deterministisch. Es ist also nicht vorhersehbar. Deshalb müssen die Zustände der Partikel auch weiter aktualisiert werden, während es nicht sichtbar ist [8]. Würde es angehalten werden und dann wieder weiterlaufen, wäre dies für den Betrachter klar erkennbar. Er würde sehen, dass es angehalten wurde. Diese Beobachtung entspricht aber nicht der Erwartung dessen, dass die Zeit auch für Dinge weiterläuft, die man gerade nicht sieht. Aufgrund ihrer deterministischen Natur, haben zustandslose Partikelsysteme den Vorteil, dass sie immer vom Culling betroffen werden können.

Die Vorhersehbarkeit von zustandslosen Partikelsystemen birgt aber auch Probleme. Sollen die Partikel eines solchen Systems sich etwa in Weltkoordinaten bewegen, ist dies nicht ohne weiteres möglich. Die Position der Partikel wird immer in Abhängigkeit von der momentanen Position des Partikelsystems berechnet. Eine mögliche Lösung dieses Problems ist es, sich eine Positionshistorie im Partikelsystem zu speichern. Diese Historie kann dann an Stelle der momentanen Position genutzt werden, um die vergangene Position von einem Teil der Partikel zu berechnen. Da es aber nicht sinnvoll ist, die gesamte Positionshistorie des Systems aufzuzeichnen, sollten hier nur die letzten paar eingenommen Positionen gespeichert werden. Die vergangenen Positionen des Systems werden dann genutzt, um ein Nachziehen der Partikel hinter diesem darzustellen. In Wirklichkeit bewegen

sich die Partikel dann aber immer noch nicht in Weltkoordinaten. Ein weiteres Problem kommt auf, wenn die Partikel mit anderen Objekten in der Welt kollidieren sollen. Wenn die anderen Objekte sich bewegen können, ist ihre Position nicht mehr vorhersagbar. Somit ist auch die Kollision mit ihnen nicht mehr vorhersagbar. Deswegen können Kollisionen mit solchen Objekten nicht von einem deterministischen Partikelsystem berücksichtigt werden. Mögliche Kollisionen können nur umständlich in die Funktionen mit eingebaut werden und verhalten sich dann auch deterministisch. Alle Eigenschaften, die selbst nicht von deterministischer Natur sind, lassen sich in einem deterministischen Partikelsystem nicht korrekt simulieren. In einem zustandsbehafteten System lassen sich diese Probleme lösen.

Bei beiden Partikelsystemimplementierungen steigt der Rechenaufwand mit der Anzahl der Partikel linear an. Die neue Position und auch die anderen Veränderungen der Eigenschaften jedes Partikels müssen beim Rendern von jedem Bild berechnet werden.

Beim benötigten Speicherplatz unterscheiden sich die Systeme stark. Da im zustandsbehafteten System die Zustände aller Partikel gespeichert werden, steigt hier auch der benötigte Speicherplatz mit der Anzahl der Partikel linear an. Im zustandslosen System werden keine Partikelzustände gespeichert. Hier ist der Speicherbedarf also nicht von der Anzahl der Partikel abhängig und bleibt konstant.

Zusammenfassend lässt sich also feststellen, dass mit einem zustandslosen Partikelsystem Performance bzw. Ressourcen eingespart werden können. Diese Einsparung geht aber mit einer Einschränkung der Funktionalität des Partikelsystems einher. Mit einem zustandsbehafteten System lassen sich mehr Anwendungsfälle simulieren. Dafür wird aber auch dementsprechend mehr Performance benötigt.

3 Implementierung

Die Implementierung des Partikelsystems erfolgte inkrementell und iterativ. Dabei wurden pro Iteration neue Konzepte erstellt und Code restrukturiert. Der in diesem Kapitel beschriebene Aufbau des implementierten Partikelsystems ist der Stand zum Zeitpunkt des Schreibens. Die Strukturen und Namensgebungen können und werden sich voraussichtlich in der Zukunft noch ändern.

3.1 Anforderungen

Die Anforderungen an das zu entwickelnde Partikelsystem für FUDGE ergeben sich zum großen Teil aus der Zielsetzung von FUDGE und daraus wie ein zustandsloses Partikelsystem definiert ist.

Folgende funktionale Anforderungen wurden definiert:

- Ziel ist es ein zustandsloses Partikelsystem zu implementieren.
- Die Veränderung der Zustände der Partikel und damit des gesamten Systems sollen über eine Verschachtelung von Funktionen definiert werden können.
- Die Funktionen, die die Veränderung der Partikel beschreiben, sollen in einer Datei definiert und gespeichert werden können. Diese Datei kann dann zur Laufzeit importiert werden.
- Insbesondere soll die Transformation, also Position, Größe und Ausrichtung jedes Partikels über die Funktionen manipuliert werden können.
- Auch andere Eigenschaften der Partikel sollen manipuliert werden können. Dafür soll die Komponentenarchitektur von FUDGE genutzt werden. Alle Eigenschaften, die Knoten über Komponenten angeheftet werden können, sollen so auch von dem Partikelsystem manipuliert werden können.
- Es sollte möglich sein, die Transformationen der Partikel zumindest scheinbar von der Transformation des Partikelsystems zu entkoppeln. Es können dann also Partikeleffekte definiert werden, in denen bei der Bewegung des Partikelsystems die Partikel erst verzögert nachziehen.

Damit soll die Bewegung in Weltkoordinaten, die in einem zustandsbehafteten System möglich ist, nachgeahmt werden können.

- Die Transformationen der Partikel sollten von anderen Objekten abhängig gemacht werden können. Partikel sollten also z.B.: mit Flächen kollidieren können.
- Das Mesh und die Textur der Partikel sollte beliebig auswählbar sein.
- Die Anzahl der Partikel in einem Partikelsystem sollte fest definiert sein.
- Die Datei, in der sich die Definitionen der Funktion befinden, sollte im JSON-Format geschrieben sein.

Folgende nichtfunktionale Anforderungen sind implizit durch die Richtlinien von FUDGE gegeben oder wurden zusätzlich definiert:

- Die Datei, in der sich die Definitionen der Funktion befinden, soll menschenlesbar und von Hand editierbar sein. Dabei sollte ihre Struktur leicht verständlich sein.
- Alle unterliegenden Strukturen des Partikelsystems sollten möglichst übersichtlich gestaltet werden. Dadurch wird eine einfache Wartbarkeit gewährleistet. Ebenfalls soll es so gemäß den FUDGE Richtlinien auch für Anwender gut nachvollziehbar sein, wie das genutzte System funktioniert.
- Die Strukturen sollten leicht erweiterbar sein. Es sollte also möglich sein, mit minimalen Codeaufwand neue Funktionen zum Berechnen der Eigenschaften der Partikel zu ergänzen.

Bei einem Partikelsystem handelt es sich um eine sehr performancekritische Anwendung. Deshalb musste an bestimmten Stellen, im Gegensatz zu den FUDGE Richtlinien, mehr Wert auf die Effizienz des Codes gelegt werden als auf dessen Verständlichkeit.

3.2 Komponente und Ressource

Das Partikelsystem wurde entsprechend der Komponentenarchitektur von FUDGE (siehe 2.3.2) als Komponente mit dazugehöriger Ressource entworfen. Eine Partikelsystem-Komponente verwendet also einen Partikeleffekt als Ressource und wird an einen Knoten angehängt. Ein

Partikeleffekt kann mehren Komponenten gleichzeitig als Ressource dienen und über das Bestehen einer Komponente hinaus weiterverwertet werden.

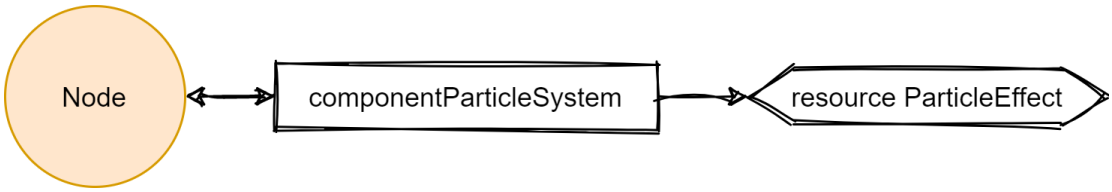

Abbildung 6: Partikelsystem als Komponente und Ressource

Der Partikeleffekt beinhaltet nur eine Information darüber, wie sich die Eigenschaften der Partikel verändern sollen. Die Information darüber, wie die Partikel zu Beginn aussehen sollen, erhält das Partikelsystem über die anderen am selben Knoten angehängten Komponenten (Mesh- und Material-Komponente). Die Transformation des Partikelsystems ist über die Transformation des Knotens (Transform-Komponente) gegeben. Abbildung 7 zeigt die anderen Komponenten, die das System zum Zeitpunkt des Schreibens der Arbeit nutzt.

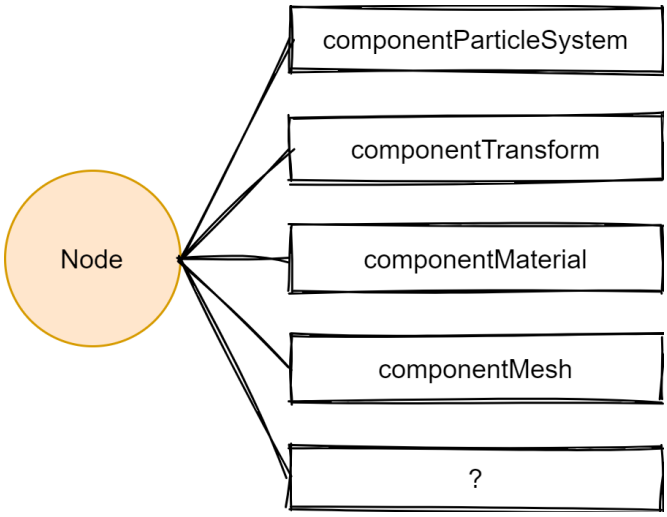

Abbildung 7: Komponenten und Partikelsystem

Das Partikelsystem ist dabei so aufgebaut, dass zukünftig hinzukommende Komponenten auch über dieses manipuliert werden können. Hierzu wird das Interface Mutable, das von allen Komponenten implementiert wird, genutzt.

Beim Rendern des Knotens wird die angehängte Partikelsystem-Komponente erkannt und eine speziell zum Rendern von Partikelsystemen entwickelte Rendermethode (siehe 3.6) aufgerufen.

3.3 Klassenstruktur

Für die Implementierung des Partikelsystem in FUDGE wurde die in Abbildung 8 zu sehende Klassenstruktur erarbeitet. Das Klassendiagramm ist nicht UML-konform, da sich einige Möglichkeiten zur Definition von Datentypen aus TypeScript nicht standardmäßig in einem UML-Klassendiagramm darstellen lassen. Interfaces verhalten sich in TypeScript anders als z.B. in Java oder C++. Mit TypeScript Interfaces lassen sich auch assoziative Arrays und Funktionen typisieren.

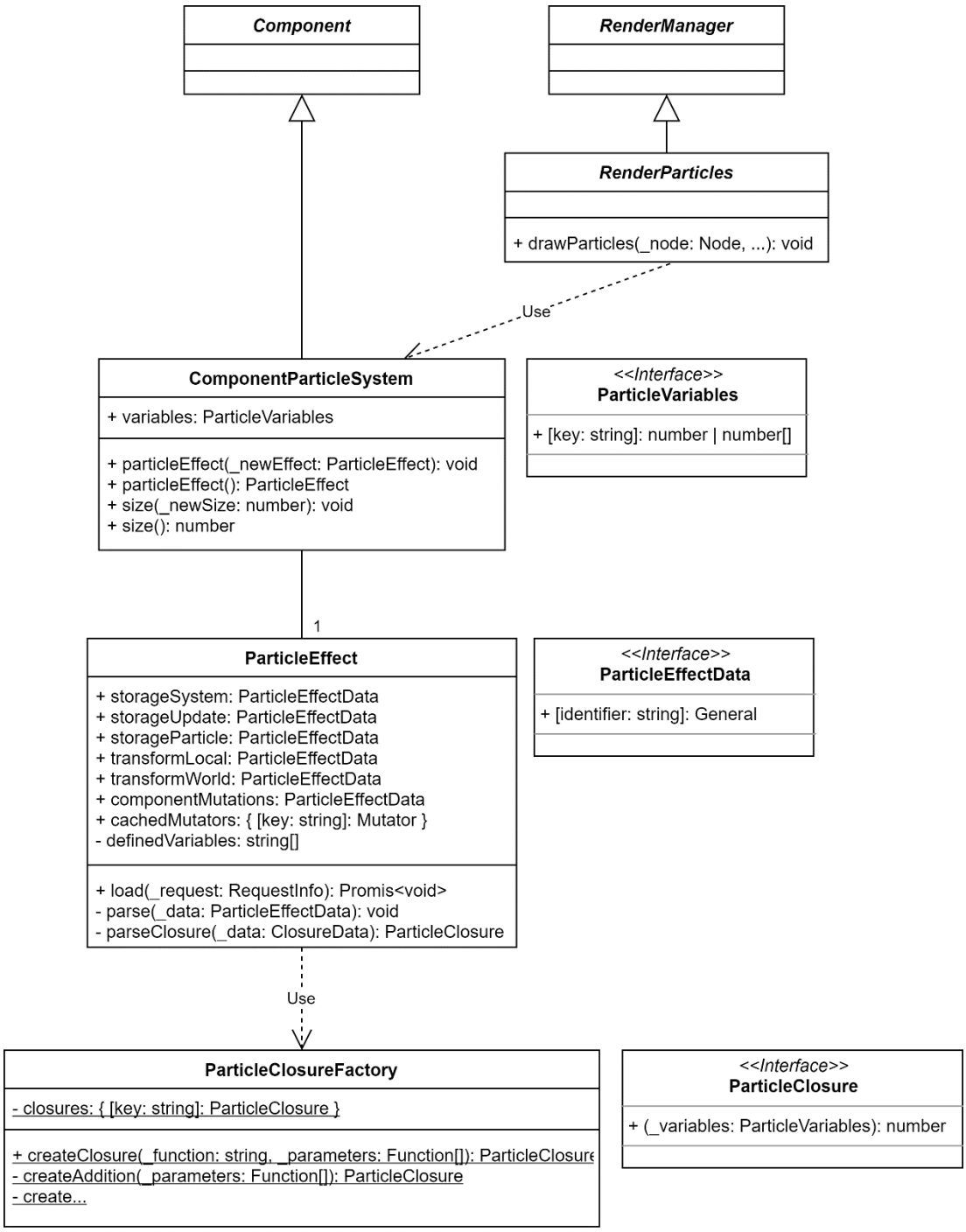

Abbildung 8: Klassendiagramm des Partikelsystems

Im Folgenden werden die Aufgaben der einzelnen Klassen und Interfaces, die im Diagramm zu sehen sind, näher erläutert. Hierbei werden auch die TypeScript spezifischen Möglichkeiten zur Definition von Interfaces erläutert.

3.3.1 ParticleEffect

In der Klasse ParticleEffect werden Funktionen zum Berechnen von Partikelzuständen eines Partikelsystems gespeichert. Diese Funktionen

müssen zunächst in einer JSON-Datei definiert werden (siehe 3.4). Die Beschreibung der Funktionen erfolgt also außerhalb vom eigentlichen Programmcode. Beim Erstellen einer ParticleEffect-Instanz muss eine solche JSON-Datei angegeben werden. Diese wird dann über die Methoden des ParticleEffect gelesen und ausgewertet. Dabei wird die ParticleEffect-Instanz mit den entsprechenden Daten initialisiert.

Der Inhalt der JSON-Datei wird als Objekt vom Typ ParticleEffectData interpretiert. Dieses Interface beschreibt ein assoziatives Array. Ein assoziatives Array besteht aus Schlüssel-Werte-Paaren. Jedem Schlüssel ist ein Wert zugeordnet. Schlüssel können dann wie Indizes verwendet werden, um die zugeordneten Werte abzurufen. Das im Klassendiagramm beschriebene Interface ParticleEffectData ist in Listing 3 zu sehen. Die Schlüssel, hier bezeichnet als „identifier“, sind vom Typ string. Die Werte sind vom Typ General. Der Typ General dient in FUDGE als Platzhalter für den TypeScript-Typ any. Das JSON-Objekt besteht dann also aus Schlüsseln vom Typ string denen Werte von einem beliebigen Typ zugeordnet sind.

```
export interface ParticleEffectData {
  [identifier: string]: General;
}
```

Listing 3: Interface ParticleEffectData

Ein Partikeleffekt dient, wie vorher beschrieben, als Ressource und kann von einem oder mehreren Partikelsystem-Komponenten (siehe 3.3.3) aus referenziert werden.

3.3.2 ParticleClosureFactory

Die Klasse ParticleClosureFactory dient dem Erstellen von anonymen Funktionen (Closures) für einen ParticleEffect. Sie wurde dem Factory-Pattern nachempfunden. Der ParticleEffekt nutzt die statischen Methoden der Factory während dem Parsen der JSON-Datei, um die in ihr definierten Funktionen zu erstellen. Die zurückgegeben Funktionen entsprechen dem Interface ParticleClosure (siehe Listing 4). Eine solche Funktionen erhält als Argument ein Objekt vom Typ ParticleVariables (siehe 3.3.3) und gibt einen Wert vom Typ number zurück.

```
export interface ParticleClosure {
  (_variables: ParticleVariables): number;
}
```

Listing 4: Interface ParticleClosure

Die erstellten Funktionen beinhalten in ihrem Rumpf verschiedene mathematische Operationen (Addition, Subtraktion, etc.) oder Abbildungen (Linear, Polynom, Wurzel), die als Ergebnis einen Wert liefern. Da das Factory-Pattern verwendet wurde, können neue Funktionen leicht im Code ergänzt werden. Auf die Funktionserstellung wird in Kapitel 3.5 näher eingegangen.

3.3.3 ComponentParticleSystem

Die Klasse ComponentParticleSystem bildet zusammen mit einem zugewiesenen ParticleEffect ein vollständiges Partikelsystem. Sie erbt ihre allgemeinen Komponenten-Eigenschaften von der abstrakten Klasse Component. In Instanzen von ComponentParticleSystem werden die verschiedenen veränderlichen Größen verwaltet, welche den Funktionen aus einem ParticleEffect als Eingangsparameter dienen. Die Eingangsparameter werden in Anlehnung an die Mathematik in einem assoziativen Array namens variables gespeichert. Das assoziative Array entspricht dem Interface ParticleVariables. Die Werte in diesem assoziativen Array sind von einem sogenannten Union-Type. Über Union-Types ist es in TypeScript möglich, dass ein Wert einen von mehreren möglichen Typen einnehmen kann. Das Besondere an diesem Interface ist also, dass hier einem Schlüssel vom Typ string nicht nur ein Wert vom Typ number zugeordnet sein kann, sondern auch ein Array von numbers. Es existieren 4 vorgefertigte Eingangsparameter in variables:

- „time“: Die Spiel-Zeit in Sekunden, die seit dem Start des Gameloops vergangen ist.
- „index“: Der Index eines Partikels also eine Nummer zwischen 0 und der Gesamtanzahl der Partikel.
- „size“: Die Gesamtanzahl der zu rendernden Partikel.

- „randomNumbers“: Ein Array gefüllt mit Zufallszahlen zwischen 0 und 1. Die Größe des Arrays ist etwas größer als die Gesamtanzahl der Partikel gewählt.

Innerhalb der JSON-Datei lassen sich außerdem eigene Variablen definieren, die in variables zwischengespeichert werden (siehe 3.4.3). Über die Funktionen aus dem ParticleEffect lässt sich dann zusammen mit den Variablen der gesamte Zustand des Partikelsystems berechnen.

3.3.4 RenderParticles

Die abstrakte Klasse RenderParticles erbt von der abstrakten Klasse RenderManger. Der RenderManger ist, wie der Name vermuten lässt, für das Rendern verantwortlich. Im RenderManager befindet sich schon sehr viel Funktionalität. Um die Funktionalität, die zum Rendern eines Partikelsystems benötigt wird, auszulagern, wurde die Klasse RenderParticles erstellt. Während des Renderns wird dann vom RenderManager aus die statische Methode drawParticles der Klasse RenderParticles aufgerufen und der momentan zu rendernde Knoten übergeben. Wie der Renderprozess genau abläuft wird in Kapitel 3.6 erläutert.

3.4 JSON-Datei

Die JSON-Datei enthält die Informationen darüber, wie sich die Funktionen zur Veränderung der Partikelzustände zusammensetzen und welche Eigenschaften der Partikel von den jeweiligen Funktionen manipuliert werden sollen.

Die Funktionsdefinitionen befinden sich in der Hierarchie des JSON eigentlich erst an tieferer Stelle. Zum besseren Verständnis wird aber zunächst auf den Aufbau dieser eingegangen. Danach wird dann die generelle Struktur der JSON-Datei erläutert.

3.4.1 Funktionen

Die Definition einer Funktion besteht aus zwei Teilen: dem Funktionstyp („function“) und der Parameterliste („parameters“). Mit dem Begriff Funktion ist eine Funktion im Sinne der Informatik als der Mathematik gemeint, weshalb

hier der Begriff Parameter gewählt wurde. Ein Bespiel für eine Funktionsdefinition im JSON-Format ist in Listing 5 zu sehen.

```
{
    "function": "addition",
    "parameters": [
        "time",
        1
    ]
}
```

Listing 5: JSON Funktionsdefinition

Über den Funktionstyp wird die eigentliche Rechenvorschrift der Funktion angegeben. Hierbei stehen verschiedene Rechenoperationen und mathematische Funktionen, also Funktionen im Sinne einer Abbildung, zur Verfügung. Parameter können sein:

- Konstanten: Eine festgelegte positive oder negative Gleitkommazahl.
- Variablen: Die veränderlichen Eingangswerte für die zu erstellende Funktion. Neben den vor definierten Größen (Zeit, Index, Größe) können auch eigene Variablen definiert werden. Die Definition von eigenen Variablen wird in Kapitel 3.4.3 näher erläutert.
- Funktionen: Als Eingangswert für eine Funktion kann das Ergebnis einer anderen Funktion genutzt werden.

Die in Listing 5 zu sehende Funktion würde bei ihrem Aufruf also immer die momentane Zeit addiert mit der Zahl 1 zurückgeben. Anstelle von Konstanten und Variablen können Parameter aber auch selbst wieder Funktionen sein. Die Konstante 1 aus dem Beispiel könnte also auch durch eine weitere Funktion ersetzt werden. Das Resultat könnte dann aussehen wie in Listing 6. Beim Aufruf der Funktion aus Listing 6 würde zunächst die innere Funktion aufgerufen werden. In der inneren Funktion wird der momentane Partikelindex mit 2 multipliziert. Das Ergebnis dieser Multiplikation würde dann auf die momentane Zeit addiert werden. Der Rückgabewert der gesamten Funktion entspricht also genau: Zeit + (Index * 2).

```
{
    "function": "addition",
    "parameters": [
        "time",
        {
            "function": "multiplication",
            "parameters": [
                "index",
                2
            ]
        }
    ]
}
```

Listing 6: JSON Funktionsverschachtelung

Die definierten Funktionen lassen sich beliebig verschachteln und miteinander verknüpfen. Eine fertige Funktion kann in sich eine Vielzahl von Funktionen enthalten. Über solche Funktionen lassen sich dann die genauen Zustandsveränderungen von jedem Partikel beschreiben.

3.4.2 Bereiche

Zum Bestimmen darüber, welche Eigenschaften eines Partikels manipuliert werden sollen und zur Verwaltung von eigens angelegten Variablen, ist die JSON-Datei in drei Oberbereiche unterteilt. Diese Bereiche sind der Speicher („storage“), die Tansformationen („transformations“) und die Veränderung der Komponenten („components“). Die Oberbereiche sind in Listing 7 zu sehen. Jeder der Bereiche dient einem sehr spezifischen Zweck, weshalb sich der weitere Aufbau innerhalb dieser mitunter stark unterscheidet. Die Bereiche werden in den folgenden Kapiteln jeweils einzeln erläutert.

```
{
    "storage": {},
    "transformations": {},
    "components": {}
}
```

Listing 7: JSON Oberbereiche

3.4.3 Storage

Der Storage-Bereich ist zum Deklarieren und Definieren von eigenen Variablen vorgesehen. In diesen Variablen können Zwischenergebnisse

gespeichert werden. Der Name der eigenen Variablen kann dabei frei gewählt werden. Einzige Ausnahme ist hierbei, dass die vorgegeben Variablen („time“, etc.) nicht überschrieben werden können. Die eigenen Variablen können dann im Rest der JSON-Datei, wie die vorgegebenen Variablen, einfach abgerufen werden.

In Listing 8 wird eine Variable mit dem selbst gewählten Namen „myVariable“ definiert. Das Schlüsselwort „myVariable“ kann dann im nachfolgenden JSON-Text genauso verwendet werden, wie z.B. das Schlüsselwort „time“.

```
{
    "myVariable": {
        "function": "addition",
        "parameters": [
            "time",
            1
        ]
    },
},
```

Listing 8: Definition einer Variable im Storage-Bereich

Zur Bestimmung davon, zu welchem Zeitpunkt in der Existenz des Partikelsystems die eigenen Variablen evaluiert werden sollen, ist der Storage-Bereich selbst nochmal in drei Unterbereiche aufgeteilt (siehe Listing 9).

```
"storage": {
    "system": {},
    "update": {},
    "particle": {}
},
```

Listing 9: Storage-Unterbereiche

Die drei Unterbereiche werden zu folgenden Zeitpunkten evaluiert:

- „system“: Variablen im Bereich „system“ werden bei der Erstellung der Partikelsystem-Komponente einmal evaluiert, wenn die Größe also die Anzahl der Partikel an der Komponente verändert wird oder der Partikeleffekt der Komponente ausgetauscht wird.
- „update“: Variablen im Bereich „update“ werden pro Update-Zyklus also immer, wenn ein neues Bild gemalt wird, genau einmal evaluiert.

- „particle“: Variablen im Bereich „particle“ werden für jeden gemalten Partikel einmal neu evaluiert.

Listing 10 zeigt die Definition von mehreren Variablen verteilt über die Storage-Unterbereiche. Wie hier zu sehen ist wird „myVariable1“ in der Definition von „myVariable2“ einfach als Parameter einer Funktion verwendet. Nicht verwendete Bereiche, wie im Beispiel der Bereich „update“, können einfach weggelassen werden.

```
"storage": {
    "system": {
        "myVariable1": {…}
    },
    "particle": {
        "myVariable2": {
            "function": "addition",
            "parameters": [
                "myVariable1",
                1
            ]
        }
    }
},
```

Listing 10: Storage mit mehreren Variablen

Der eigentliche Zweck des Storage-Bereiches besteht darin, die eigenen Partikeleffekte zu optimieren. Durch das Zwischenspeichern von Ergebnissen kann vermieden werden, dass eine Berechnung mit gleichbleibendem Ergebnis mehrmals ausgeführt werden muss.

3.4.4 Transformations

Im Transformations-Bereich können die Transformationen der Partikel beschrieben werden. Zur Berechnung der Transformation eines Partikels wird, genauso wie für die Berechnung der Transformation eines Knotens, eine 4x4-Matrix verwendet. Die JSON-Datei ist dabei so konzipiert, dass in ihr die Methodennamen von Methoden der Klasse Matrix4x4 angegeben werden können. Die wichtigsten Methoden sind hierbei das Verschieben („translate“), das Rotieren („rotate“) und das Skalieren („scale“). Auf den Methodennamen folgend sollte ein 3-dimensionaler Vektor angegeben werden. Der Vektor wird über seinen X-, Y- und Z-Wert definiert. Dabei können einzelne Werte

ausgelassen werden. Dieser Vektor wird dann für die jeweilige Methode als Eingangswert verwendet.

In Listing 11 ist eine Transformation, in der die Partikel mit der Methode „translate“ verschoben werden, zu sehen. Die Verschiebung in X-Richtung ergibt sich aus dem Ergebnis der angegebenen Funktion. Die Verschiebung in Y-Richtung wurde weggelassen, Y bleibt also unverändert. Für die Verschiebung in Z-Richtung wurde direkt die Variable „index“ eingesetzt, jeder Partikel wird also um seinen Index auf der Z-Achse verschoben werden.

```
"translate": {
    "x": {
        "function": "multiplication",
        "parameters": [
            "time",
            2
        ]
    },
    "z": "index"
}
```

Listing 11: Transformation eines Partikels

Der Transformation-Bereich ist nochmals in zwei Unterbereiche unterteilt, in welchen Transformationen angegeben werden können (siehe Listing 12). Je nach Bereich ändert sich das Bezugsystem für die angegeben Transformationen. In Anlehnung an die Bezeichnungen, die in FUDGE für diese Bezugsysteme verwendet werden, heißen die beiden Bereiche „local“ und „world“.

```
"transformations": {
    "local": {},
    "world": {}
},
```

Listing 12: Transformation Unterbereiche

Transformationen im Bereich „local“ werden auf das lokale Koordinatensystem der Partikel angewendet. Das bedeutet, dass diese Transformation in Abhängigkeit von der Welt-Transformation des Knotens geschieht, an dem die Partikelsystem-Komponente angehängt ist. Wenn also z.B. der Knoten gedreht wird, dreht sich das lokale Koordinatensystem der Partikel mit.

Transformationen im Bereich „world“ werden auf das Weltkoordinatensystem bezogen. Alle voranlaufenden Transformationen sind also egal. Dieser Bereich ist besonders nützlich, um die auf die Partikel wirkenden Kräfte zu definieren. Die Schwerkraft z.B. zieht immer in dieselbe Richtung, unabhängig davon, wie ein Objekt ausgerichtet ist.

3.4.5 Components

Der Komponenten-Bereich dient der Manipulation von allen Eigenschaften, die einem Knoten über Komponenten angehängt werden können. Dieser Bereich hat eine besondere Struktur, in welcher sich das Prinzip der Mutatoren aus FUDGE zu Nutze gemacht wird. Die Komponenten, die manipuliert werden sollen, können hier über ihren FUDGE Klassennamen angegeben werden (siehe Listing 13).

```
"components": {
    "ComponentMaterial": {…},
    "ComponentMesh": {…}
}
```

Listing 13: Komponentenangabe in JSON

Auf die Angabe der Komponentenklasse folgt eine Struktur, die einer Mutator-Struktur der jeweiligen Klasse entspricht. In der hier angegeben Mutator-Struktur können Werte ausgelassen werden, die nicht beeinflusst werden sollen.

In Listing 14 ist eine solche Mutator-Beschreibung für die Material-Komponente zu sehen. Das Attribut „clrPrimary“ (die primäre Farbe) der Material-Komponente soll hier dem Mutator entsprechend verändert werden. Hinter diesem Attribut verbirgt sich die FUDGE-Klasse Color, welche selbst, wie die Komponenten auch, von Mutable erbt. Color ist zusammengesetzt aus Attributen vom Typ number. Diese Attribute repräsentieren die einzelnen Farbwerte (r, g, b) und den Alphakanal (a), welcher die Deckkraft der Farbe angibt. Die Werte der Attribute werden jeweils mit einer Zahl zwischen 0 und 1 angegeben. Innerhalb der zweiten geschweiften Klammer (in Listing 14) befindet sich ein Mutator für die Klasse Color. Dieser bewirkt, dass der Alphakanal dann immer auf die momentane Zeit gesetzt wird. Die Folge für die

Partikel ist, dass sie zum Zeitpunkt 0 komplett durchsichtig sind und dann mit dem Verlauf der Zeit bis zum Erreichen der maximalen Deckkraft am Zeitpunkt 1, langsam deckkräftiger werden.

```
"ComponentMaterial": {
    "clrPrimary": {
        "a": "time"
    }
}
```

Listing 14: Komponentenmutator in JSON

Hier stellt sich die Frage, warum nicht auch die Transformation der Partikel über die entsprechende Transformations-Komponente mithilfe eines Mutators manipuliert werden kann. Dies ist zwar grundsätzlich möglich, erzielt aber nicht den gleichen Effekt wie die Beschreibung der Transformation über den Transformations-Bereich. Beim Mutieren einer 4x4-Matrix werden die jeweiligen Größen direkt gesetzt. Beim Verändern der Matrix über ihre Methoden werden die vorangehenden Änderungen dieser beachtet. Wird also erst skaliert und dann verschoben, sollte sich die Verschiebung entsprechend der Skalierung ändern. Dies ist das erwünschte Verhalten und es kann nur erreicht werden, wenn die Matrix über ihre Methoden verändert wird.

3.5 Parsing

Um den in der JSON-Datei beschrieben Aufbau eines Effekts in ausführbaren Code zu überführen, wurden die Möglichkeiten der funktionalen Programmierung genutzt. Beim Parsen werden die im JSON angegebenen Funktionen in anonyme Programmfunktionen, also in Closures, übersetzt. Die Closures werden dann zusammen mit der Information, welche Eigenschaften eines Partikels sie verändern sollen, in der Klasse ParticleEffect gespeichert.

Der Aufbau der Funktionen im JSON entspricht einer formalen Grammatik. Auf diese formale Grammatik soll in dieser Arbeit nicht im genauen eingegangen werden. Darum wird die Struktur in den folgenden Kapiteln anhand der zugrundeliegenden Datenstruktur und mit Hilfe von grafischen Darstellungen erläutert.

3.5.1 Datenstruktur

Wie in Kapitel 3.3.1 beschrieben, wird der Inhalt der JSON-Datei als Objekt vom Typ ParticleEffectData (siehe Listing 3) interpretiert. Auf Bezeichner vom Typ String kann also ein beliebiges Objekt folgen. An bestimmten Punkten in dieser Struktur werden dann Funktionen, Variablen oder Konstanten erwartet. Dafür dient der Datentyp ClosureData (siehe Listing 15). Bei ClosureData handelt es sich um einen Typ-Alias. Typ-Aliasse können in TypeScript dazu genutzt werden, um bestehenden Typen einen anderen Namen zu geben. Sie können aber auch in Kombination mit dem Union-Typ genutzt werden. Hinter ClosureData verbirgt sich entweder ein Objekt vom Typ ClosureDataFunction oder ein primitiver Wert vom Typ string oder number. Das Interface ClosureDataFunction (siehe Listing 15) beinhaltet einen String, der die Art der Funktion (function), die erstellt werden soll, angibt und ein Array, das ihre Parameter (parameters) enthält. Die Parameter sind wiederum vom Typ ClosureData. Diese rekursive Typdefinition beschreibt den Aufbau der Funktionen im JSON (und damit auch die zu Grunde liegende Grammatik).

```
type ClosureData = ClosureDataFunction | string | number;

interface ClosureDataFunction {
  function: string;
  parameters: ClosureData[];
}
```

Listing 15: Funktions-Parsing-Typen

Auf die Auflistung dieser zwei Typen im Klassendiagramm (Abbildung 8) wurde absichtlich verzichtet, da sie nur innerhalb der Klasse ParticleEffect und auch nur während des Parsens eine Rolle spielen.

3.5.2 Syntaxbaum

Zur Visualisierung der Datenstruktur der Funktionen eignet sich die Darstellung als Syntaxbaum. Ein Syntaxbaum dient der hierarchischen Darstellung solcher Datenstrukturen. Im Folgenden wird nur ein sogenannter abstrakter Syntaxbaum dargestellt. Dieser dient nur der groben Veranschaulichung.

In Abbildung 9 ist ein Beispiel eines abstrakten Syntaxbaums zu sehen welcher die Daten aus Listing 6 darstellt. Der Baum besteht aus drei möglichen Ausdrücken, welche die Datenstruktur widerspiegeln. Diese Ausdrücke können sein:

- Funktion: Eine Funktion besteht aus einer Operation und dazugehörigen Operanden. Die Operation beschreibt, wie die Operanden zu verrechnen sind. Die Operanden sind selbst auch wieder Ausdrücke. Operationen sind in Bezug auf die Datenstruktur die Arten von möglichen Funktionen und Operanden die Parameter für diese Funktionen. Auf den Ausdruck Funktion folgt immer mindestens ein weiterer Ausdruck; es handelt sich um einen nicht terminalen Ausdruck.
- Variable: Eine Variable ist eine veränderliche Zahl, auf die mit einem Namen verwiesen wird. Dieser Verweis erfolgt in der Datenstruktur in Form eines Strings. Auf den Ausdruck Variable folgen keine weiteren Ausdrücke mehr, es handelt sich um einen terminalen Ausdruck.
- Konstante: Eine Konstante ist ein fester Zahlenwert. In der Datenstruktur handelt es sich um eine einfache Number. Konstanten sind auch terminale Ausdrücke.

Der Beispielbaum ist also wie folgt aufgebaut: Der Start-Ausdruck ist eine Funktion mit zwei Operanden. Der erste Operand ist eine Variable. Variable ist ein terminaler Ausdruck; es folgen also keine weiteren Ausdrücke mehr im Baum. Der zweite Operand ist eine Funktion; auf ihn folgen weitere Ausdrücke. Die zwei folgenden Ausdrücke sind eine Variable und eine Konstante. Diese sind beide terminal. Damit wurde das Ende des Baums erreicht.

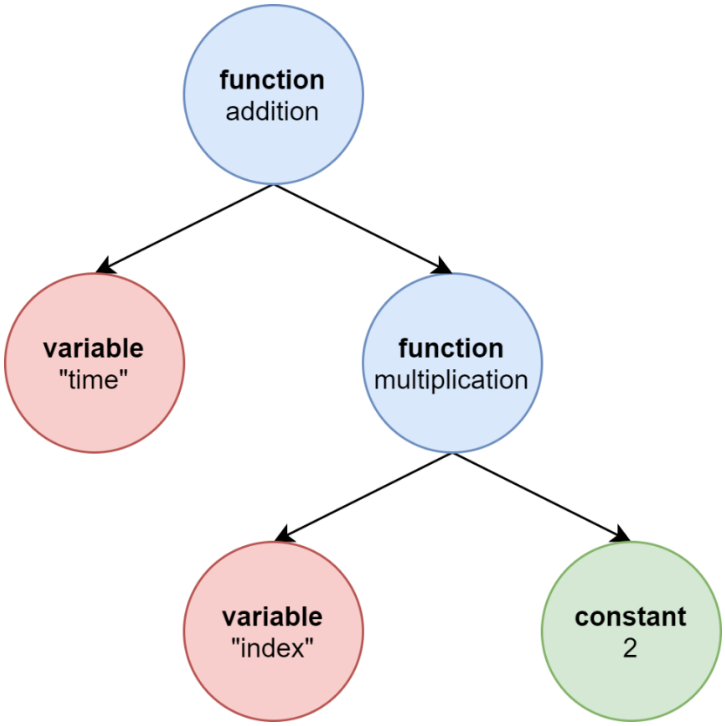

Abbildung 9: Abstrakter Syntaxbaum von Funktion

Dieser Baum dient als Grundlage für den Algorithmus zum Erstellen der Closures.

3.5.3 Funktionserstellung

Für die Funktionserstellung wird der Syntaxbaum von der Wurzel aus durchlaufen. Dabei wird für jeden Ausdruck eine Closure erstellt.

Handelt es sich beim Ausdruck um eine Funktion, werden zunächst Closures für die Folgeausdrücke erstellt. Anschließend wird die Closure erstellt, welche die eigentliche Rechenoperation enthält. Als Operanden werden dann die vorher gebildeten Closures der Folgeausdrücke eingesetzt, d.h., diese Closures werden erst ausgewertet, wenn die eigentliche Operation ausgewertet wird.

Handelt es sich beim Ausdruck um eine Konstante, dann wird diese in eine Closure verpackt, welche den entsprechenden Wert zurückgibt.

Handelt es sich beim Ausdruck um eine Variable, wird eine Closure erzeugt, die eine Referenz auf den Variablenwert hält und diesen bei einem Aufruf zurückgibt.

Aus der Funktion aus Abbildung 9 wird folgende verschachtelte Closure erstellt:

```
Closure_Addition(
    Closure_Variable(),
    Closure_Multiplication(
        Closure_Variable(),
        Closure_Constant()
    )
)
```

Abbildung 10: Verschachtelte Closure

Die Closures der Variablen und Konstanten wissen dabei selbst, welche Werte sie zurückgeben sollen. Konkret ergibt sich folgende Rechenvorschrift:

```
(time + (index * 2))
```

Die Closures entsprechen hierbei alle dem Interface ParticleClosure (siehe 3.3.2, Listing 4). In diesem ist zu sehen, dass die erstellten Closures selbst auch noch über einen Eingangsparameter, in Form eines assoziativen Arrays, verfügen. Über dieses Array werden die Variablenwerte durch die Closures hindurchgereicht. Jede Closure gibt das Array an die von ihr aufgerufenen Closures weiter. Die Variablen-Closures entnehmen ihren entsprechenden Wert dann aus diesem Array und geben ihn zurück.

Wie die verschachtelte Closure aus Abbildung 10 dann von innen aussieht, ist in Abbildung 11 dargestellt.

```
Closure_Addition(_variables) {
    Closure_Variable(_variables) {_variables[„time“]}
    +
    Closure_Multiplication(_variables) {
        Closure_Variable(_variables) {_variables[„index“]}
        *
        Closure_Constant(_variables) {2}
    }
}
```

Abbildung 11: Verschachtelte Closure mit Variablenübergabe

Der gesamte Funktionserstellungsprozess ist in Abbildung 12 zu sehen. Diese zeigt ein Aktivitätsdiagramm für die Methode parseClosure. Mit Factory ist in diesem die ParticleClosureFactory gemeint.

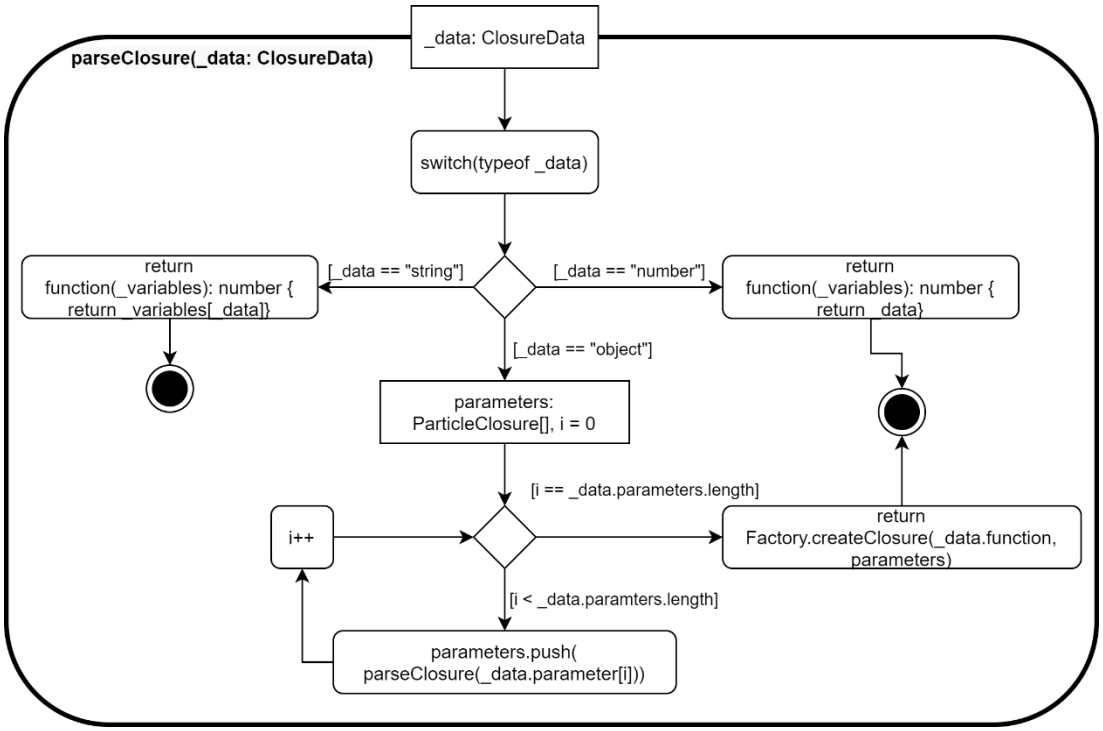

Abbildung 12: Aktivitätsdiagramm parseClosure

3.5.4 Effekterstellung

Bei der Erstellung einer ParticleEffect-Instanz wird zunächst die angegebene JSON-Datei rekursiv durchlaufen. Immer wenn dabei eine Variable, Konstante oder eine Funktionsbeschreibung gefunden wird, wird diese durch die

entsprechende Closure ersetzt. Anschließend wird die JSON-Datei in ihre Unterbereiche zerlegt und diese werden zum schnelleren Zugriff separat im ParticleEffect gespeichert.

3.6 Renderprozess

Für das Rendern des Partikelsystems wird eine eigens entwickelte Rendermethode verwendet. Während des Renderprozesses wird geprüft, ob an dem aktuell zu rendernden Knoten eine Partikelsystem-Komponente hängt. Ist dies der Fall, wird die Methode renderParticles aufgerufen und ihr wird der Knoten übergeben. Neben dem Knoten wird ihr auch die vorherberechnete Welt-Transformationsmatrix dessen übergeben. Es werden noch weitere Argumente übergeben, die aber für das Verständnis der Methode nicht von Bedeutung sind. Der ungefähre Ablauf der Methode drawParticles ist im Aktivitätsdiagramm in Abbildung 13 zu sehen.

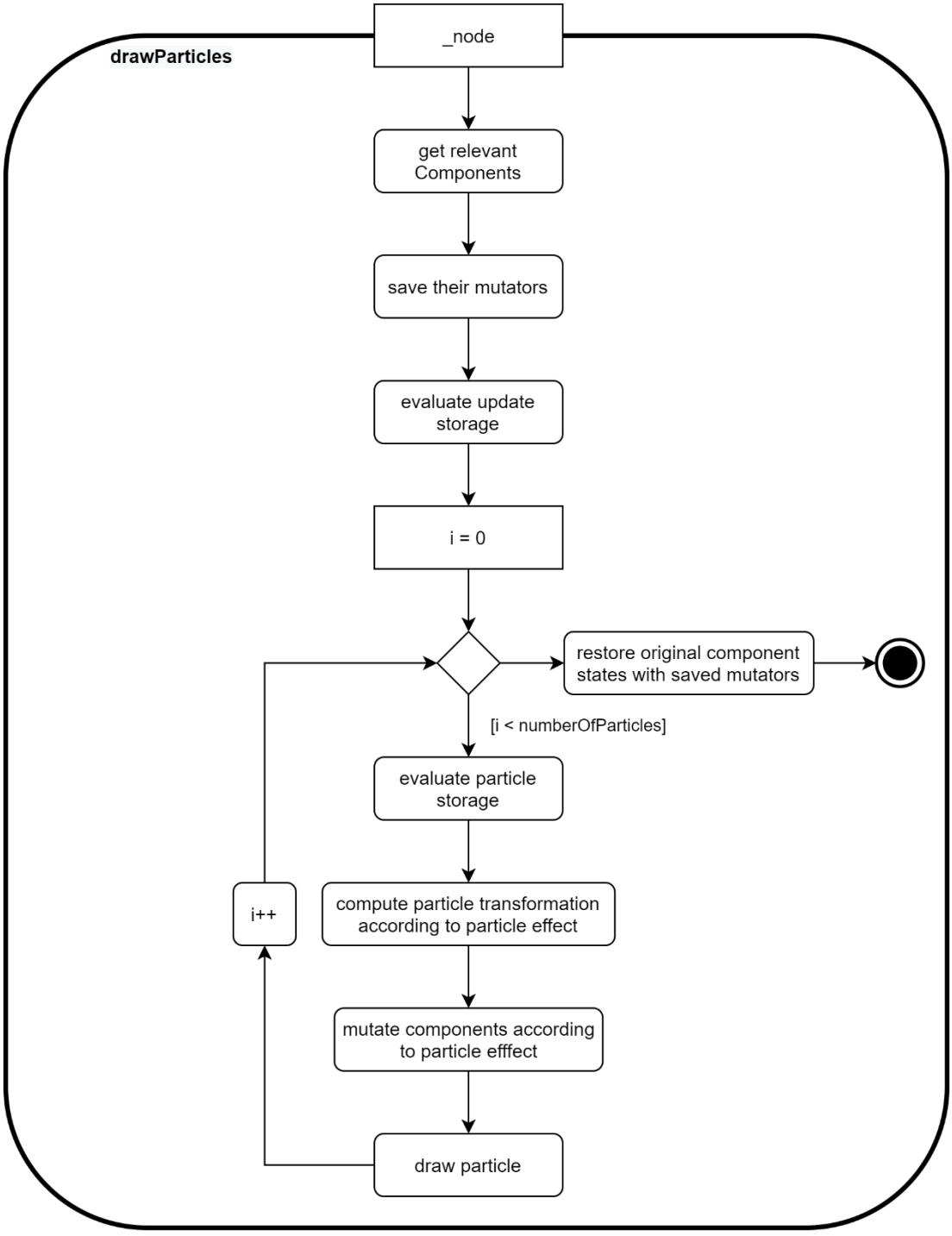

Abbildung 13: Aktivitätsdiagramm drawParticles

Zunächst werden aus dem Knoten alle Komponenten herausgeholt, die vom Effekt verändert werden sollen. Von diesen Komponenten werden dann ihre momentanen Mutatoren zwischengespeichert. Anschließend werden die Closures ausgewertet, welche sich im JSON im Storage-Bereich „update“ befinden. Die Ergebnisse dieser werden zu den in der Partikelsystem-Komponente gespeicherten Variablen geschrieben. Nun beginnt das

Berechnen der Zustände der einzelnen Partikel. Dafür wird eine Schleife von 0 bis zur Anzahl der Partikel (der Variable „size“) ausgeführt. Bei jeder Iteration der Schleife wird zunächst der momentane Laufindex i in die Variable „index“ geschrieben. Dann wird der Storage-Bereich „particle“ ausgewertet, auch dessen Ergebnisse werden über die Variablen verfügbar gemacht. Jetzt wird die Transformation des zu malenden Partikels berechnet. Dazu werden wie in Kapitel 3.4.4 beschrieben zwei Matrizen erstellet: die Lokaltransformations-Matrix und die Welttransformations-Matrix. Auf beiden werden dann die im JSON definierten Methoden aufgerufen um diese zu Manipulieren. Die finale Transformation des Partikels ergibt sich jetzt aus einer Reihe von Matrixmultiplikationen. Multipliziert müssen hierbei werden: Die Welttransformations-Matrix des Knoten, die zwei eben berechneten Transformationsmatrizen des Partikels und die Pivot-Matrix der am Knoten hängenden Mesh-Komponente. Auf die Berechnung der Transformation folgt nun die Manipulation der anderen am Knoten hängenden Komponenten. Dafür werden die im JSON angegebenen Komponenten mit den dazugehörigen Mutatoren mutiert. Jetzt wird der Knoten mit Hilfe des Meshs, des Materials und seiner Transformation gerendert. Nachdem alle Partikel gerendert wurden, werden die am Knoten hängenden Komponenten mit den am Anfang gespeicherten Mutatoren dieser mutiert und damit zurück in ihren Initialzustand überführt. Damit ist das Ende der Methode erreicht.

Beschrieben wurde hier nur der ungefähre Ablauf der Methode. Die konkrete Methode wurde weitgehend optimiert, sodass benötigte Objekte (Matrix4x4, Vektor3x3, Mutator) wiederverwertet werden, anstelle dass diese immer wieder neu erzeugt werden müssen. Auch das mehrmalige Aufsuchen von Informationen (Zugriffe auf assoziative Arrays, Herausholen von Komponenten aus einem Knoten, etc.) wurde vermieden.

3.7 Erzielte Ergebnisse

Von den in Kapitel 3.1 angegeben Anforderungen konnte ein Großteil umgesetzt werden. Folgende Ergebnisse wurden erzielt:

- Ein vollständig zustandsloses Partikelsystem wurde implementiert.

- Die Zustandsveränderungen der Partikel werden in Form von Funktionen gespeichert.
- Die Speicherung erfolgt in einer JSON-Datei.
- Die JSON-Datei kann zur Laufzeit in einen Partikel Effekt importiert werden.
- Die veränderbaren Eigenschaften der Partikel entsprechen allen Eigenschaften, die einem FUDGE Knoten per Komponente angehängt werden können.
- Insbesondere wurde dabei Wert daraufgelegt, dass die Transformation der Partikel genau gesteuert werden kann.
- Das Mesh und die Textur eines Partikels können über die in FUDGE vorhandene Mesh- und Material-Komponente gewählt werden.
- Die Anzahl der zu rendernden Partikel kann in der Partikelsystem-Komponente angegeben werden.

Nicht erfüllt wurden zwei der Anforderungen:

- Die Bewegung der Partikel kann nicht von der Bewegung des Partikelsystems entkoppelt werden. Die Bewegung in Weltkoordinaten, die in einem zustandsbasierten Partikelsystem möglich ist, kann nicht simuliert werden.
- Die Bewegung der Partikel kann nicht von anderen Objekten abhängig gemacht werden. Eine Kollision mit anderen Objekten kann nicht überprüft werden.

Diese Funktionalitäten werden im Kapitel 5 Ausblick nochmals diskutiert.

Die nichtfunktionalen Anforderungen wurden, so gut es möglich war, umgesetzt:

- Die Struktur der JSON-Datei wurde nach Möglichkeit so gehalten, dass die in ihr gewählten Bezeichner (ComponentXY, Transformations, local/world, etc.) den Begrifflichkeiten entsprechen, die auch innerhalb von FUDGE verwendet werden. Anwendern soll es so einfacher fallen, die Daten von Hand zu editieren.

- Durch die Verwendung des Factory-Entwurfsmusters ist die leichte Erweiterbarkeit um neue Funktionen gewährleistet.
- Auch sonst werden möglichst bereits bestehende Strukturen wie die Mutatoren verwendet. Dadurch wird die Wartbarkeit des Systems verbessert.
- Zu der sonstigen Anwenderverständlichkeit können keine konkreten Aussagen getroffen werden, da zum Zeitpunkt des Schreibens noch kein Feedback hierzu eingeholt wurde.

4 Beispieleffekt Flamme

In diesem Kapitel wird Schritt für Schritt ein konkreter Partikeleffekt erstellt. Damit soll veranschaulicht werden, wie das Partikelsystem genutzt werden kann und wie ein Effekt entwickelt und dargestellt wird. Als Beispieleffekt soll eine zweidimensionale Flamme erzeugt werden. Ein vollständiges Diagramm, das den hier entwickelten Effekt beschreibt, befindet sich am Ende des Kapitels.

Für die Textur der Partikel wird ein PNG-Bild verwendet. In PNG-Bildern ist es möglich Farben eine Transparenz zu verleihen. Die Textur ist in Abbildung 14 zu sehen.

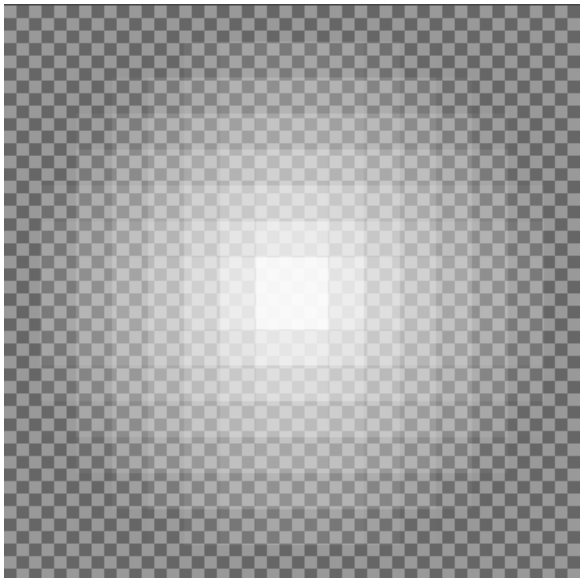

Abbildung 14: Partikeltextur

Sie besteht aus einem weißen Kreis, der nach außen hin durchsichtiger wird. Die Farben von halbdurchsichtigen Flächen, die sich überlagern, werden für den Flammeneffekt addiert. Wenn sich also später mehrere Partikel überschneiden werden die überschneidenden Pixel als heller dargestellt. Weiße Farbe lässt sich im Code über die Material-Komponente zu einer beliebigen anderen Farbe ändern. Für den Beispieleffekt werden die Partikel in einen Orangeton gefärbt. Auf das Festlegen der Textur folgt die Bestimmung der Bewegung der Partikel. Dafür wird zunächst die Bewegung eines einzelnen Partikels beschrieben. Dieser soll auf einer geraden Bahn nach

oben fliegen und nach einer gewissen Zeit wieder von vorne starten. Diese Bewegung lässt sich durch die in Abbildung 15 zu sehende Funktion erreichen.

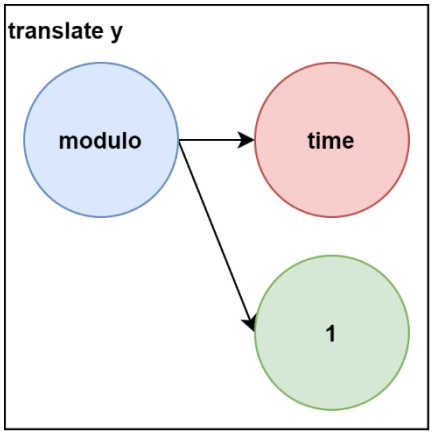

Abbildung 15: Modulo-Funktion

Durch die Modulo-Funktion wiederholt der Partikel seine Bahn einmal pro Sekunde:

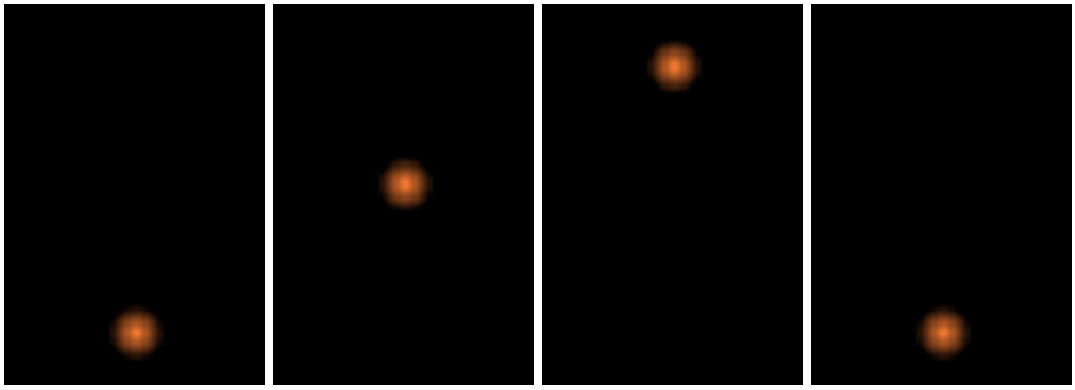

Abbildung 16: Einzelner Partikel zum Zeitpunkt 0, 0.5, 0.9 und 1

Als nächstes muss dafür gesorgt werden, dass jeder Partikel seine eigene Zeit erhält, so dass sich mehrere Partikel gleichzeitig auf der Bahn verteilen. Dafür wird die Zeit in unserer Funktion durch eine individuell berechnete Partikelzeit ersetzt:

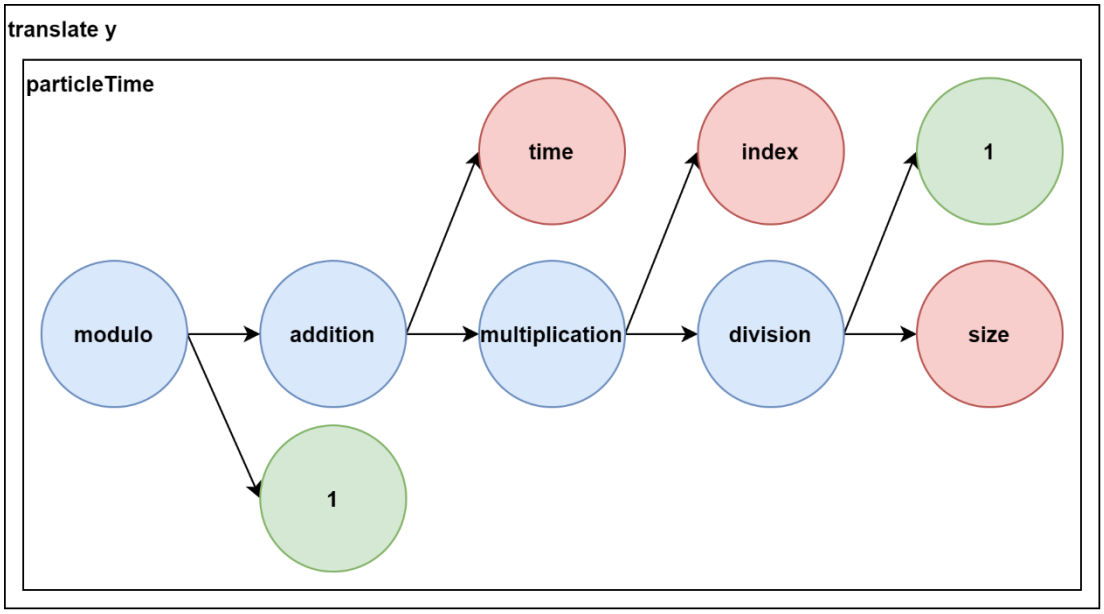

Abbildung 17: Funktion für Translation in Y-Richtung bzw. individuelle Partikelzeit

Für die individuelle Zeit wird zunächst die Zahl 1 durch die Anzahl der Partikel geteilt. Das Ergebnis dieser Division wird dann mit dem momentanen Partikelindex multipliziert, so ergibt sich für jeden Partikel eine seinem Index entsprechende Verschiebung zwischen der Zahl 0 und 1 auf der Y-Achse:

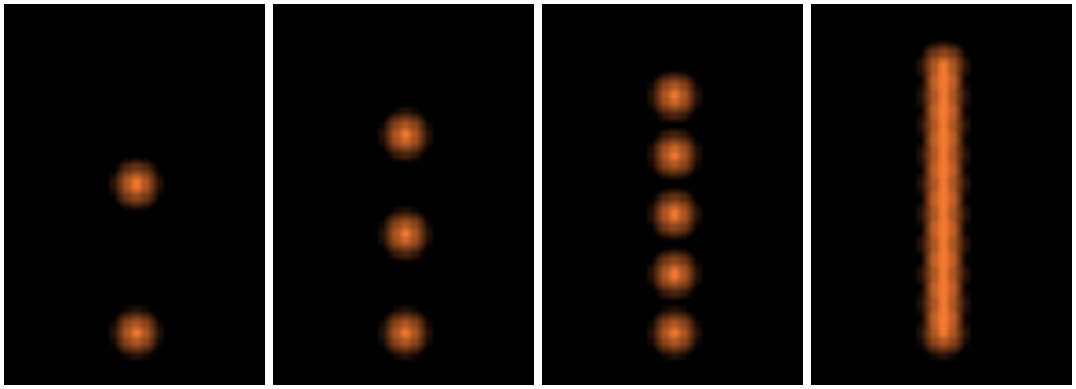

Abbildung 18: Verteilung von 2, 3, 5 und 10 Partikeln zum Zeitpunkt 0

Diese Verschiebung wird dann auf die vergangene Zeit addiert. Dadurch verschieben sich die Partikel in Abhängigkeit von der Höhe ihres Indizes in die Zukunft, bzw. in diesem Fall auf der Y-Achse nach oben. Von der nun berechneten individuellen Partikelzeit muss nur noch der Modulo mit 1 gebildet werden, damit sich die Zeit immer wiederholt:

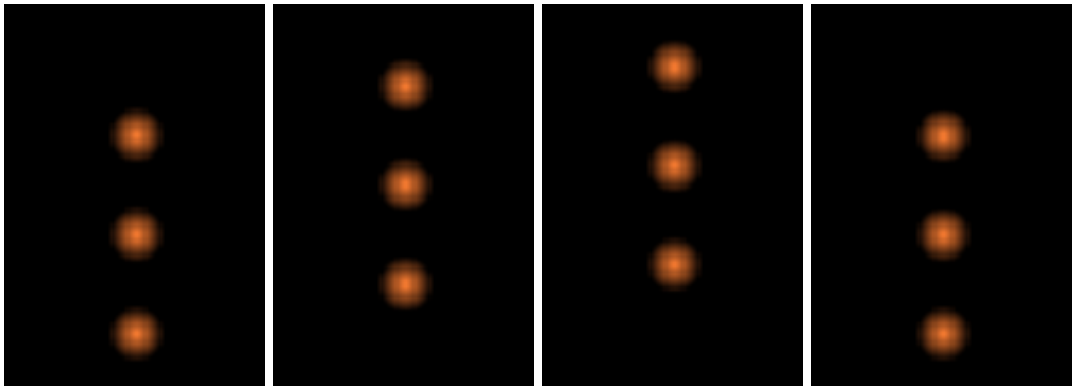

Abbildung 19: Verteilung von 3 Partikeln zum Zeitpunkt 0, 0.5, 0.9 und 1

Der Effekt sieht einer Flamme bisher nicht sehr ähnlich. Dies lässt sich ändern, indem die Partikel auch auf der X-Achse verschoben werden. Um die geschwungene Bahn einer Flamme darzustellen, wird hier eine Polynomfunktion 3. Grades verwendet (siehe Abbildung 20).

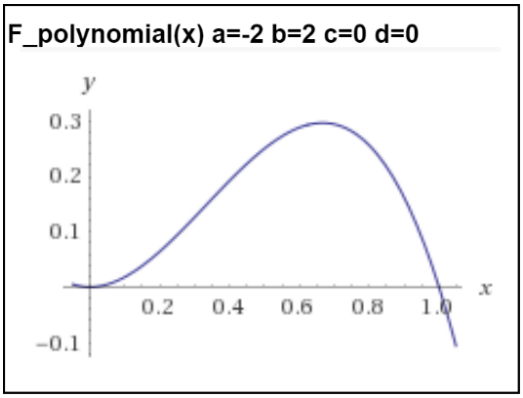

Abbildung 20: Polynomfunktion

Damit sich die Partikel auf der X-Achse entsprechend ihrer auf der Y-Achse zurückgelegten Strecke bewegen, wird hier dieselbe individuelle Partikelzeit als Eingangswert für die Funktion genutzt wie für die Translation in Y-Richtung. Weil sich der Bauch der Flamme aber unten befinden soll, muss diese Funktion in umgekehrter Richtung durchlaufen werden, weshalb zunächst die Subtraktion von 1 und der Partikelzeit gebildet wird:

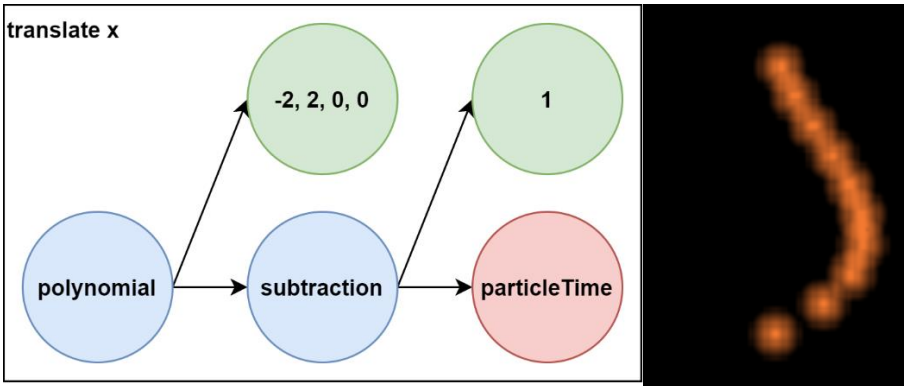

Abbildung 21: Links Funktion für Translation in X-Richtung, rechts Effekt mit 10 Partikeln zum Zeitpunkt 0

Die Partikel bewegen sich jetzt alle auf derselben Kurve. Um die Flamme fertigzustellen, muss jetzt nur noch der berechnete X-Wert mit einer Zufallszahl zwischen -1 und 1 multipliziert werden:

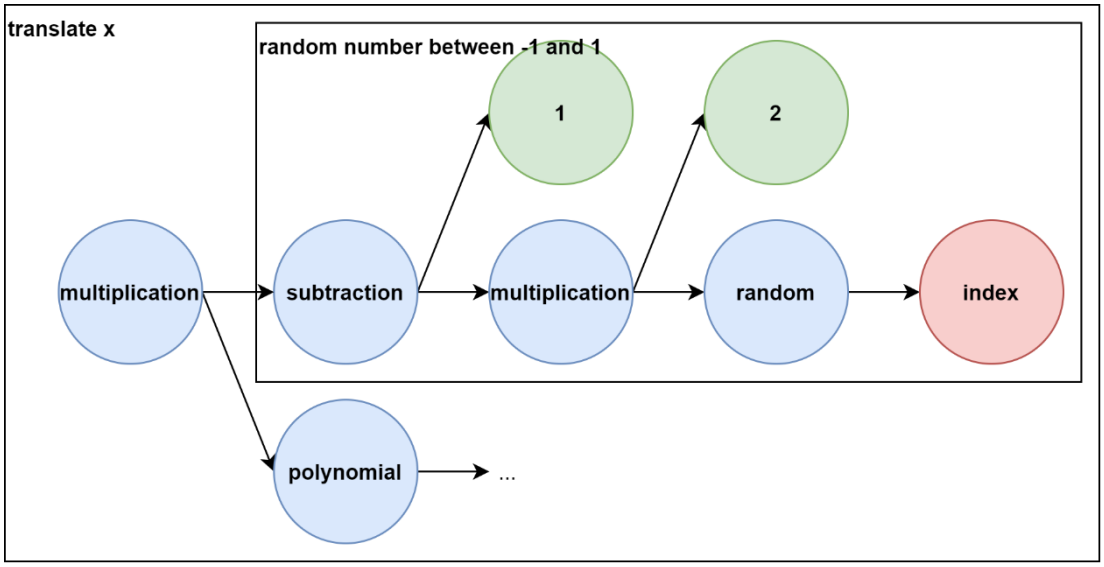

Abbildung 22: Funktion für Zufallszahl zwischen -1 und 1

Die Zufallsfunktion ergibt für jeden Index immer wieder dieselbe Zahl, sodass sich nun jeder Partikel auf einer individuellen Kurve bewegt. Als finale Ergänzung kann dann noch die Transparenz der Partikel von der Partikelzeit abhängig gemacht werden (siehe Abbildung 24). Das Ergebnis und die damit fertige Flamme sieht dann so aus:

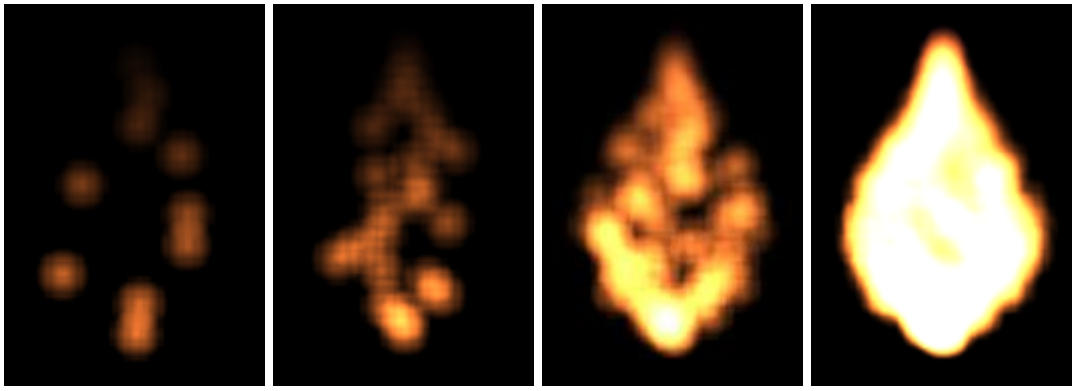

Abbildung 23: Flamme mit 10, 25, 100 und 500 Partikeln zum Zeitpunkt 0

Der hier beschriebene Effekt kann jetzt noch über den Storage-Bereich optimiert werden. Da sich die Anzahl der Partikel, die zur Berechnung der Partikelzeit verwendet wird, nicht ändert, muss die Division von 1 mit dieser nicht immer wieder erneut berechnet werden. Es reicht, wenn diese Berechnung einmal beim Erstellen des Partikelsystems ausgeführt wird. Die Partikelzeit wird zum Berechnen des Zustandes eines einzelnen Partikels an mehreren Stellen genutzt. Die Translation in X- und Y-Richtung wie auch die Transparenz der Partikel sind von dieser abhängig. Die Berechnung der Partikelzeit muss also nur einmal pro Partikel ausgeführt werden. Der optimierte und vollständige Effekt ist in Abbildung 24 zu sehen.

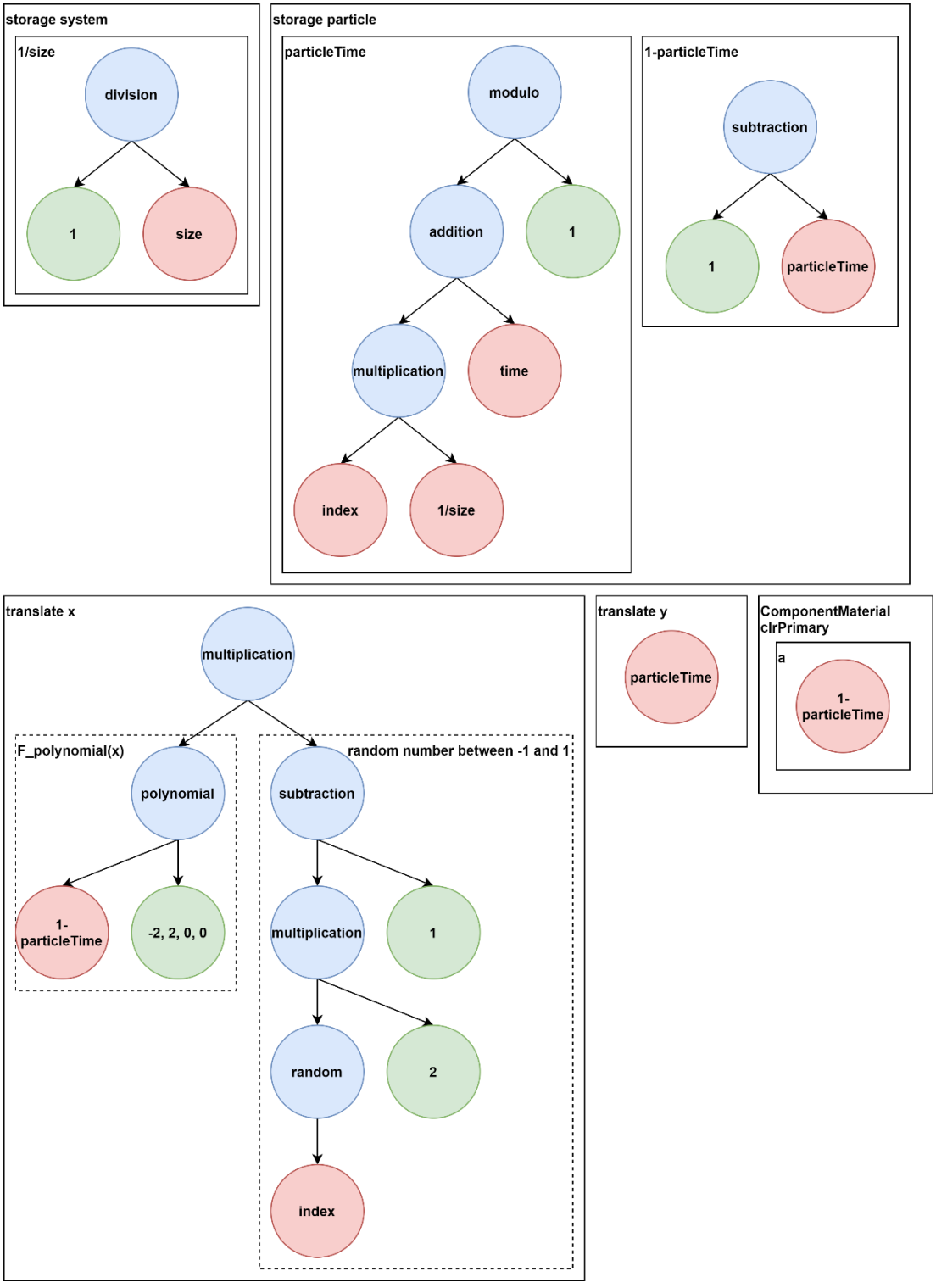

Abbildung 24: Beispieleffekt Flamme

5 Ausblick

Mit einem zustandslosen Partikelsystem lassen sich, wie in Kapitel 2.4.3 bereits erwähnt, nicht alle Anwendungsfälle eines Partikelsystems abdecken, die sich mit einer zustandsbehafteten Lösung abdecken lassen. Für die zwei nicht erfüllten Anforderungen (siehe 3.7) wurden in diesem Kapitel auch schon mögliche Lösungsansätze angesprochen.

Durch die Aufzeichnung der letzten Transformationen des Partikelsystems und das Nutzen dieser zum Berechnen der Partikeltransformationen, kann das Partikelsystem in Zukunft noch um einen Partikel-Nachzieheffekt ergänzt werden. Auf diese Weise kann dann zumindest eine Bewegung der Partikel in Weltkoordinaten simuliert werden.

Die Interaktion von Partikeln mit anderen Spielobjekten gestaltet sich als schwieriger. Sie kann mit einem zustandslosen Partikelsystem auch nur grob simuliert und nicht realistisch abgebildet werden. Da sich die Partikel stets deterministisch verhalten, können auch Interaktionen nur deterministisch eingeplant werden. Das bedeutet, dass, wenn sich weder das Partikelsystem, noch das Objekt, mit dem die Partikel interagieren sollen, bewegen wird, (z.B. ein dauerhafter Regeneffekt, der auf einen ebenen Boden trifft, also eine Kollision der Regentropfen mit einer unbewegten Fläche) kann dies über die Bewegungsfunktionen der Partikel eingeplant werden. Eine wirkliche Interaktion der Partikel mit ihrer Umwelt wird mit einem zustandslosen Partikelsystem auch in Zukunft nicht möglich sein.

Neben den bereits entwickelten Optimierungen sind weitere möglich. Partikelsysteme mit über 1000 Partikeln lasten auch Rechner mit einer modernen CPU stark aus. Über eine weitere Optimierung des Codes lässt sich hier höchstwahrscheinlich auch keine große Verbesserung mehr erzielen. Der Grund für die nicht so gute Performance ist aber nicht unbekannt. Alle Berechnungen der Partikelzustände werden auf der CPU ausgeführt. Da die Rechenzeit mit jedem weiteren Partikel ansteigt, wird die CPU schnell ausgelastet und wird zum Flaschenhals. Dass die Perfomance des

implementierten Systems nicht so gut sein wird, wurde aber von Anfang an vorausgesehen.

Das in dieser Arbeit implementierte System soll nur eine vorrübergehende Lösung sein. In der Zukunft soll auf Grundlage dessen ein System entwickelt werden, das die Rechenarbeit auf die GPU auslagert. In diesem sollen die Berechnungen der Partikelzustände auf die Hardware-Shader der Grafikkarte ausgelagert werden. Da die GPU solche Berechnungen viel schneller durchführen kann als die CPU, verspricht sich von einem GPU basierten Partikelsystem eine erhebliche Performanceverbesserung. Dies ist neben den sowieso schon bestehenden Performancevorteilen eines zustandslosen Systems gegenüber eines zustandsbehafteten (siehe 2.4.3) auch der Hauptgrund ,warum ein zustandsloses System für FUDGE entwickelt wurde. Zustandslose Partikelsysteme können viel leichter auf eine GPU implementiert werden als zustandsbehaftete.

Die Ergebnisse dieser Arbeit sollen also auch insbesondere als Grundlage für eine zukünftige Arbeit dienen, in der ein GPU basierendes zustandsloses Partikelsystem für FUDGE entwickelt werden soll.

6 Fazit

Das Ziel dieser Arbeit, ein zustandloses Partikelsystem für FUDGE zu entwickeln, wurde erreicht. Zum Erreichen dieses Zieles wurden die Methoden der funktionalen Programmierung wie auch allgemeine Entwurfsmuster verwendet.

Als menschenlesbares Format dient JSON zum Abspeichern und Definieren der Funktionen, mit welchen die Partikelzustände ausgerechnet werden. Innerhalb der JSON-Datei können zur Performancesteigerung Variablen definiert werden. In diesen Variablen können Ergebnisse von Funktionen zwischengespeichert werden und stehen dann anderen Funktionen zur Verfügung. Aus den JSON Daten werden zur Laufzeit Closures erstellt. Diese müssen nur einmal erstellt werden und können dann immer wieder genutzt werden. Die Closures werden in einer Ressource gespeichert, die gemäß dem Design von FUDGE von einem oder mehreren Partikelsystem-Komponenten referenziert werden kann. Durch dieses Design werden Datenredundanzen vermieden. Beim Rendern wird die Partikelsystem-Komponente zusammen mit den anderen an einen Knoten angehängten Komponenten genutzt, um das Partikelsystem darzustellen. Die anderen Komponenten enthalten dabei die Eigenschaften der Partikel. Durch Nutzung des Mutatoren-Konzeptes von FUDGE können so alle Eigenschaften, die ein Partikel über Komponenten erhalten kann, über die Definitionen in der JSON-Datei manipuliert werden. Selbst Komponenten, die erst in der Zukunft geschrieben werden, können dann direkt vom Partikelsystem manipuliert werden.

Über mehrere Iterationen hinweg entstand so ein System, das der FUDGE Kompositionsarchitektur entspricht und sich diese gleichzeitig zu Nutzen macht. Das resultierende System ist vielseitig einsetzbar, leicht erweiterbar und kann aufgrund seiner funktionalen Natur als Grundlage für das zukünftig geplante GPU basierende zustandslose Partikelsystem verwendet werden.

Literaturverzeichnis

[1] game – Verband der deutschen Games-Branche e.V., „Jahresreport der deutschen Games-Branche 2019," [Online]. Available: https://www.game.de/publikationen/jahresreport-der-deutschen-games-branche-2019/. [Zugriff am 12 08 2020].

[2] J. Dell'Oro-Friedl, „FUDGE wiki," [Online]. Available: https://github.com/JirkaDellOro/FUDGE/wiki/. [Zugriff am 18 07 2020].

[3] J. Harband und K. Smith, „ecma-international.org," [Online]. Available: https://www.ecma-international.org/ecma-262/11.0/index.html#title. [Zugriff am 27 7 2020].

[4] ecma international, „ecma-international.org," 12 2017. [Online]. Available: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. [Zugriff am 27 7 2020].

[5] Microsoft, „typescriptlang.org," [Online]. Available: https://www.typescriptlang.org/docs/home. [Zugriff am 27 7 2020].

[6] Khronos Group, „khronos.org," [Online]. Available: https://www.khronos.org/webgl/wiki/Getting_Started. [Zugriff am 27 7 2020].

[7] TheWusa, „Wikimedia.org, Lizenz: CC BY-SA 3.0, abgeändert," [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=4749571. [Zugriff am 20 07 2020].

[8] K. Jones, „Unity Blog," 20 12 2016. [Online]. Available: https://blogs.unity3d.com/2016/12/20/unitytips-particlesystem-performance-culling/. [Zugriff am 20 07 2020].

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

_____

Furtwangen, 21.08.2020 Jonas Plotzky