FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Steigerung der Performance in Videospielen durch Datenorientierte Programmierung

-BACHELORARBEIT-

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

im Studiengang Informatik

Friedrich-Schiller-Universität Jena

Fakultät für Mathematik und Informatik

eingereicht von Dennis Untiet

geboren am 01.10.2001 in Esslingen

Matrikelnummer: 192151

Erstgutachter: Joachim Giesen

Zweitgutachter: Julien Klaus

Jena, den 30. August 2023

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Performancesteigerung in Videospielen durch Datenorientierte Programmierung. Mit dem datenorientierten Ansatz wird versucht, die objektorientierte, nicht Cache freundliche Verarbeitung von Daten zu ersetzen. Der Fokus wird auf die Art und Weise gelegt, wie Daten im Speicher liegen, gelesen, geschrieben und verarbeitet werden. Hierdurch soll eine schnellere Verarbeitung der Daten erreicht werden. Eine mögliche Implementierung ist dabei der *Datenorientierte Technologie-Stack* von Unity. In dieser Arbeit wird die Leistungssteigerung betrachtet, die ein datenorientierter Ansatz bietet. Dazu wurde eine Spielesimulation in Unity einmal objektorientiert und einmal datenorientiert entwickelt, welche anschließend gebenchmarkt wurde. Die Umsetzung des datenorientierten Spiels gestaltete sich schwieriger, als die des objektorientierten Spiels. Die Benchmarks ergaben, dass durch einen datenorientierten Ansatz eine enorme Performancesteigerung möglich ist.

Inhaltsverzeichnis

1 Einleitung

Die Videospielbranche ist einer der umsatzstärksten Marktzweige der Welt. Laut dem Marktforschungsunternehmen Newzoo hatte die Gaming-Industrie 2021 einen Gesamtumsatz von rund 192,7 Milliarden Dollar weltweit. Dies ist mehr Umsatz, als Filme-, Serien- und aufgenommene Musikindustrie zusammengenommen [6]. Auch in Deutschland liegt der Umsatz der Gaming-Industrie knapp vor dem Umsatz des Videostreamings und der Musik [2].
Videospiele, welche heutzutage auf den Markt kommen, werden immer komplexer. Es werden immer größere Spiele entwickelt, welche mit immer mehr Inhalt gefüllt werden. Diese müssen aber natürlich trotzdem, oder vielleicht auch gerade deshalb Leistungsstandards entsprechen. Um diesen Anforderungen gerecht zu werden, sind fortschrittliche Programmieransätze erforderlich, die eine Leistungssteigerung in komplexen Videospielen ermöglichen. Zwar steigt die Rechenleistung in modernen Computern weiter an, dennoch passiert es aber immer wieder, dass Spiele mit einem objektorientierten Ansatz an ihre Grenzen stoßen, die Rechenleistung vollständig zu nutzen. Auch ist es sinnvoll, Videospiele energieeffizienter zu gestalten um Stromeinsparungen zu erhalten und nachhaltiger zu werden. Das heißt, es wird in Zukunft wesentlich wichtiger sein, einen alternativen Lösungsansatz anzustreben, statt die immer größer werdende Rechenleistung auszunutzen. Dabei können ein datenorientierter Ansatz bei der Programmierung und eine Parallelisierung des Programms mögliche Lösungsansätze sein.
Die vorliegende Arbeit geht insbesondere auf den datenorientierten Ansatz von Unity, das *Entity Component System* (*ECS*), ein. Die Spiele-Engine Unity wurde aus den folgenden Gründen gewählt:
1. Unity ist eine Spiele-Engine, welche führend in der Gaming-Industrie ist. Mit ihr wurden schon viele große Spiele entwickelt[1].
2. Das *ECS* von Unity wird momentan aktiv weiterentwickelt. Unity setzt viel daran, den *Datenorientierten Technologie-Stack* auszubauen und es kamen viele Neuerungen in den letzten Monaten auf den Markt[2].
Im Verlauf der Arbeit werden Eigennamen und englische Begriffe *kursiv* geschrieben. Begriffe, die sich auf den Programmtext beziehen oder Variablen, Funktion etc. beschreiben, werden im `Typewriter` Format im Fließtext dargestellt.

[1] https://www.thegamer.com/unity-game-engine-great-games/
[2] https://unity.com/de/roadmap/unity-platform/dots

2 Datenorientierte Programmierung

Das folgende Kapitel zur Datenorientierten Programmierung basiert auf dem Artikel „Data-oriented design“ von Noel Llopis [3], dem Buch „Data-Oriented design“ von Richard Fabian [1] und der Präsentation „Entity Component Systems & Data Oriented Design“ von Unity [4].

2.1 Problemstellung

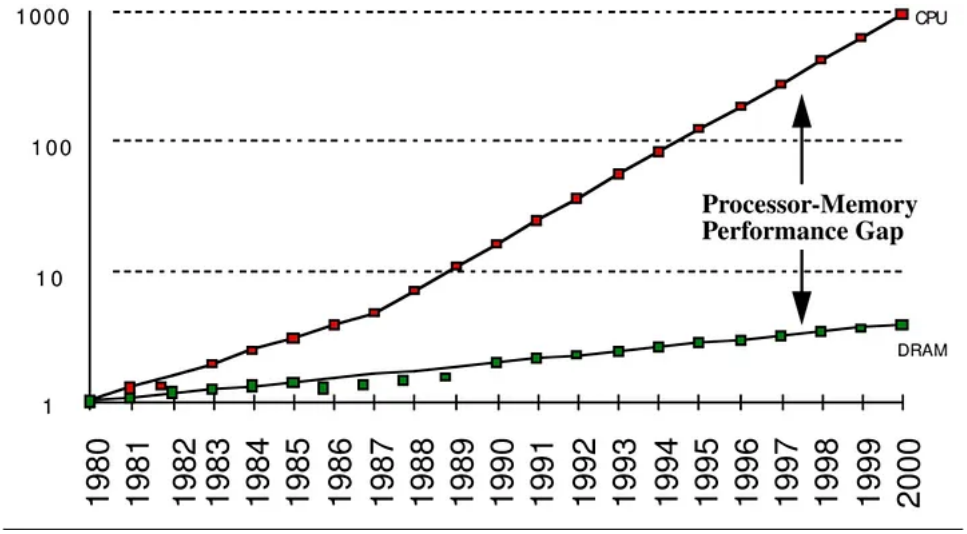

Abbildung 1: Performance Unterschied von CPU zu Speicher von 1980 bis 2000.
Quelle: `http://gec.di.uminho.pt/discip/minf/ac0102/1000Gap_Proc-Mem_Speed.pdf`

Abbildung 1 zeigt, dass der Performance Unterschied von CPU und Speicher über die Jahre immer größer geworden ist. Dieser Trend setzt sich fort. Die zeitlichen Kosten auf dem Hauptspeicher Daten zu lesen sind wesentlich größer, als etwas auf der CPU zu berechnen. Daher wurden auch CPU's über die Zeit mit kleinen Speichereinheiten ausgerüstet um den Performanceunterschied auszugleichen - die Caches. Diese bringen den Vorteil, oft genutzte Daten direkt an der CPU zu speichern. Wenn die angefragten Daten schon im Cache liegen (Cache-Hit), benötigt der Zugriff eine wesentlich kürzere Zeit. Dennoch kommt es vor, dass die Daten nicht in dem Cache liegen (Cache-Miss). Dann braucht ein Speicherzugriff viel mehr Zeit. In der Objektorientierten Programmierung, wo meist ein Objekt nach dem anderen in den Cache geladen und verarbeitet wird, kommt dies oft vor.
Damit die Performance der Anwendung nicht durch Speicherzugriffe behindert wird und die gesamte Rechenleistung des Prozessors genutzt werden kann, wurde eine effizientere Lösung, das datenorientierte Design, entwickelt.

2.2 Datenorientiertes Design

Im datenorientierten Design geht es vor allem um die gegebenen Daten. Wichtig hierbei sind die Fragen nach den ideale Daten für ein Problem, wie diese Daten im Speicher liegen und wer diese Daten liest oder schreibt. Der Fokus wird bei dem datenorientierten Design auf die Daten gelegt, da der Sinn von Programmen einzig und allein das Transformieren von Daten ist.
Die Wichtigkeit einer guten Datenstruktur wird an einem Beispiel deutlich gemacht. Listing 1 zeigt, wie im objektorientierten Design eine Person aufgebaut wäre.

```
class Person {
    string name;
    Wohnort wohnort;
    int alter;
}
```

Listing 1: Person im objektorientierten Design. Objekte sind im objektorientierten Design oft Baumstrukturen, erkennbar an dem Unterobjekt `wohnort`.

Angenommen man möchte in einem Programm aus gegebenen Personendaten das Durchschnittsalter berechnen, dann wären die gegebenen Personendaten im objektorientierten Ansatz meistens in einem Array gespeichert.

```
Person[] personen;
```

Um damit das Durchschnittsalter zu berechnen wäre im objektorientierten Ansatz der Zugriff auf jedes einzelne Personenobjekt nötig, um sich aus dem Objekt das Alter zu holen. Der Weg über jedes einzelne Objekt ist aber mühselig, erst recht wenn man eine größere Baumstruktur hat. Möchte man beispielsweise die Straße und Hausnummer jeder Person wissen geht der Zugriff der Daten erst über die Person und dann auf den Wohnort. Dies ist bei einer oder wenigen Personen kein Problem. Da man aber in seltenen Fällen nur wenige Objekte hat, erst recht nicht in Videospielen, ist ein anderer Ansatz hier sinnvoller. Listing 2 zeigt, wie im datenorientierten Design die Personen gespeichert wären.

```
class Personen {
    string[] namen;
    Wohnort[] wohnorte;
    int[] alter;
}
```

Listing 2: Personen im datenorientierten Design. Die Objekte werden in ihre einzelnen Komponenten zerlegt, welche dann in Arrays gespeichert werden.

Wie man sieht, wurden die Personenobjekte in ihre Komponenten zerlegt. Im datenorientierten Ansatz gibt es Objekte nur implizit. Alle Daten der Personen sind in Arrays gespeichert. Die erste Person, welche im objektorientierten Ansatz `personen[0]` ist, setzt sich im datenorientierten Ansatz aus `namen[0]`, `wohnorte[0]` und `alter[0]` zusammen. Das Durchschnittsalter

lässt sich hier wesentlich einfacher berechnen. Das Array für das Alter wird in den Cache geladen, sequenziell aufaddiert und durch die Anzahl der Personen geteilt. Bei Daten, welche immer zusammen gebraucht werden ist es sinnvoll diese zusammen zu speichern, damit diese gemeinsam in den Cache geladen werden. Damit kann man sich leichter die Straße und die Hausnummer aller Personen holen, um diese weiter zu verarbeiten, falls gewünscht.
Das nächste Beispiel zeigt den Cachingvorteil und die damit einhergehende Performanceverbesserung.

Cache Beispiel: Angenommen es gibt Daten von zehn Personen. Der Cache ist in diesem Beispiel 64 Bytes groß. Ein Personenobjekt braucht 32 Bytes an Speicher und ein Integer (das Alter) 4 Bytes. Würde man objektorientiert vorgehen, müsste man $10 \cdot 32 = 320$ Bytes an Daten in den Cache laden und hätte dabei zehn Cache-Misses. Betrachtet man aber den datenorientierten Ansatz, müsste man nur das Alter der Personen in den Cache laden, was $10 \cdot 4 = 40$ Bytes braucht. Das gesamte Array passt in den Speicher und man hat dabei nur einen Cache-Miss.

Parallelisierung: Nicht nur die Nutzung des Caches ist ein klarer Vorteil der Datenorientierten Programmierung. Durch das Speichern einzelner Komponenten in Arrays, lässt sich die Transformierung der Daten einfacher parallelisieren. In der Objektorientierten Programmierung ist es durch Abhängigkeiten sehr umständlich Code zu parallelisieren. Im datenorientierten Ansatz kann man das Array einfach auf mehrere *Threads* aufteilen und die Verarbeitung parallel ausführen.

Cache Affinität: Zusätzlich zur Parallelisierung wird der Cache bei sequenzieller Verarbeitung sehr effizient genutzt, da derselbe Code immer wieder ausgeführt wird. Wenn die Daten sequenziell verarbeitet werden resultiert das in sehr guter Performance und fast perfekter Cache Nutzung.

Modularität: Wenn objektorientierter Code zur Verbesserung der Performance angepasst wird, resultiert das oft in einem schlechter lesbarem und schlechter wartbarem Code. Das liegt an Abhängigkeiten die ein objektorientierter Ansatz oft mit sich bringt. Bei Konzentration auf die Transformierung der Daten hat man am Ende kleinere Funktionen mit weniger Abhängigkeiten zu anderem Code. Dadurch lässt sich der Code besser warten und verbessern.

Testen: Unit Tests für Objektinteraktionen können kompliziert sein. In einem objektorientierten Ansatz braucht man oft *setup* und *mocking* um gut testen zu können. Im datenorientierten Design sind Unit Tests jedoch einfacher. Man erstellt Eingabedaten, ruft damit die Funktion zum Testen auf und verifiziert die Ausgabedaten.

Schwierigkeiten: Das datenorientierte Design ist nicht die Lösung für alles. Es ist schwierig zu erlernen, wenn man die objektorientierte Denkweise gewohnt ist. Zusätzlich lässt es sich schwer mit bestehendem prozeduralen oder objektorientierten Code verbinden.

3 Unity's Datenorientierter Technologie-Stack[3]

Die vorliegende Arbeit basiert auf dem *com.unity.entities* Package mit der Version 1.0.0-pre.65[4]. Dies ist, Stand dem 01.04.2023, die aktuellste Version von Unity's *Enitity Component System*. Da die Version nicht final fertiggestellt wurde und sich noch im Entwicklungsstadium befindet, kann sich mit der Zeit viel an der Art und Weise ändern, wie Unity Datenorientierte Programmierung umsetzt. Das Grundkonzept der Datenorientierung Programmierung ist allerdings mit dem *Datenorientierten Technologie-Stack* festgeschrieben.
Unity's datenorientierter Ansatz wird durch Unity's *Datenorientierten Technologie-Stack* umgesetzt. Dieser besteht aus drei Teilen:
1. Das *Entity Component System* (*ECS*) für Unity. Dies ist ein datenorientiertes Framework für Unity. Mit dem *ECS* lässt sich der datenorientierte Ansatz umsetzen.
3. Das C# Job System. Das Job System von Unity erlaubt es parallelen Code zu schreiben, welcher sicher und schnell läuft.
2. Der Burst Compiler. Der Burst Compiler übersetzt von IL Code zu optimiertem nativem Code. Er nutzt die LLVM Compiler Infrastruktur.
Diese drei Teile werden in den folgenden Katpiteln 3.2, 3.3 und 3.4 erklärt.

3.1 Objektorientierte Programmierung in Unity

Bevor das *Entity Component System* betrachtet wird, ist ein kleiner Exkurs zu der herkömmlichen Weise, wie in Unity gearbeitet wird, sinnvoll. Dies ist hilfreich um die folgenden Kapitel besser zu verstehen. In Unity basiert alles auf *GameObjects*, die das Objektorientierte schon im Namen haben. Die *GameObjects* werden immer in einer Szene erstellt, wobei es mehrere Szenen in einem Projekt geben kann. Es kann beispielsweise eine Szene für den Startbildschirm geben und eine Szene für das eigentliche Spiel. Alles was man in einer Szene erstellt, ist zunächst ein *GameObject* welches man dann mit Funktionalitäten füllt. Durch Komponenten, die man dem *GameObject* gibt, kann man sie zu Licht, Charakteren oder Gegenständen werden lassen. Die Komponenten sind nicht zu verwechseln mit den *Components* aus dem *Entity Component System*. Diese werden in Kapitel 3.2.2 erläutert. In den folgenden Abschnitten erkennt man hierbei jedoch viele Parallelen zu dieser Arbeitsweise. Die Art wie man Code schreibt ist aber unterschiedlich. Zugriffe auf Daten anderer *GameObjects* funktionieren, indem man erst auf das *GameObject* zugreift und dann darüber auf seine Komponenten. Also ein typischer objektorientierter Ansatz.

[3] `https://unity.com/de/dots`
[4] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/changelog/CHANGELOG.html`

3.2 Das *Entity Component System*

Das *Entity Component System* besteht aus *Entities*, *Components* und Systemen. *Entities* zeigen auf, welche *Components* zusammen gehören und Systeme verarbeiten die *Components*. Sie sind also für das Transformieren der Daten zuständig. Abbildung 2 zeigt das Konzept des *Entity Component Systems*.

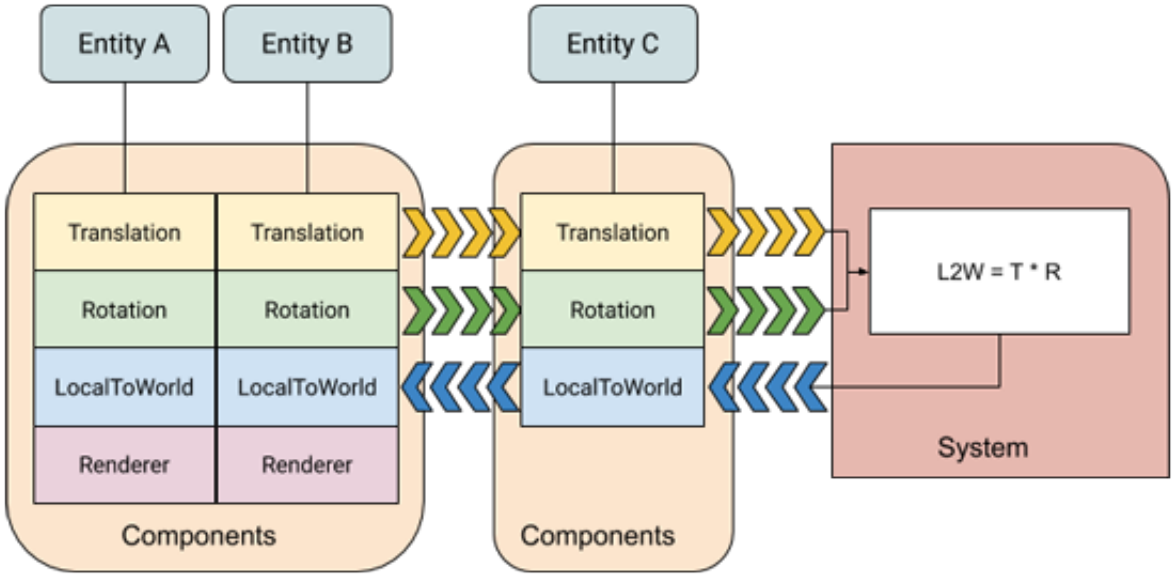

Abbildung 2: Zusammenspiel von *Entities*, *Components* und Systemen. Beispielhaft werden hier drei *Entities* gezeigt, wobei *Entity* A und B vier *Components* haben und *Entity* C nur drei *Components* hat. *Entity* C fehlt das `Renderer` *Component*. Das System nimmt als Eingabe alle `Translation` und `Rotation` *Components* und modifiziert damit das `LocalToWorld` *Component* der *Entities*.
Quelle: `https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/ecs_core.html`

3.2.1 *Entities*[5]

Entities repräsentieren meist Dinge in einem Unity Spiel. Das können der Spielcharakter, Gegenstände, oder beliebige Gegner sein. Sie können aber auch abstrakte Dinge, wie beispielsweise Events repräsentieren. Ein *Entity* ist vergleichbar mit einem *GameObject* im objektorientierten Ansatz von Unity. *Entities* besitzen hierbei jedoch weder Daten noch ein Verhalten, sondern zeigen lediglich auf, welche Daten beziehungsweise *Components* zueinander gehören. Alle *Entities* in einer Spielwelt gehören zu einer sogenannten Welt[6]. Zu dieser Welt gehört genau ein *EntityManager*[7]. Der *EntityManager* organisiert alle *Entities* in dieser Welt. Mit ihm lassen sich *Entities* erstellen, zerstören, *Components* zu *Entities* hinzufügen, entfernen oder verändern.

[5] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-entities.html`
[6] `https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/world.html`
[7] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/api/Unity.Entities.EntityManager.html`

3.2.2 *Components*[8]

Components speichern die Daten eines *Entity*. Diese Daten werden von Systemen genutzt und verarbeitet. Dabei unterscheidet man zwischen verwalteten *Components* und unverwalteten *Components*. Unverwaltete *Components* werden in Unity als C# Strukturen implementiert, welche leichtgewichtiger als Klassen sind. Diese Strukturen können auch nur unverwaltete Daten speichern. Unverwaltete Daten sind beispielsweise Integer, Boolean, Bytes, Chars, oder andere Strukturen. Verwaltete *Components* werden hingegen als Klassen definiert und können alle Daten halten. Es ist jedoch üblich unverwaltete *Components* zu verwenden, da diese nicht so resourcenintensiv im Speichern und Zugreifen sind. Um ein unverwaltetes Component zu erstellen, kann man das `IComponentData` Interface verwenden. Listing 3 zeigt ein unverwaltetes *Component*.

```
1 public struct ExampleComponent : IComponentData
2 {
3     public int2 position;
4     public float speed;
5 }
```

Listing 3: Beispiel eines unverwalteten *Components*. Das *Component* speichert die Position und die Geschwindigkeit eines *Entity*.

Es gibt aber auch andere Arten von *Components*. Es gibt *Shared Components*, *Buffer Components*, etc[9]. Es sind hier nicht alle *Components* relevant, da manche lediglich die Spieleentwicklung mit dem datenorientierten Ansatz erleichtern, aber keine neue Funktion bieten. Zudem gibt es alle Arten von *Components* sowohl als verwaltete als auch unverwaltete *Components*.
Shared Components: *Shared Components* sind, wie der Name schon sagt, unter den *Entities* geteilt. Sie gruppieren die *Entities* nach dem Wert des *Shared Component*. *Shared Components* werden abseits anderer *Components* gespeichert und sind ein Weg um Datenduplizierung zu vermeiden. Unity speichert zusätzlich alle *Entities*, welche die gleiche Kombination aus *Component* Typen haben und das gleiche *Shared Component* besitzen, also auch den gleichen Wert, gemeinsam. Dies hat zwar Vorteile, aber auch den großen Nachteil, dass das Ändern von Werten des *Shared Component* ein Verschieben der *Entities* im Speicher zufolge hat. Die Probleme hierbei findet man in Kapitel 3.2.5. Einen sehr großen Vorteil haben *Shared Components* in jedem Spiel das mit Unity's *ECS* entwickelt wird. Das `RenderMesh` *Component*, also das *Component* welches für das Aussehen der *Entities* zuständig ist, ist immer ein *Shared Component*. Dies ist sinnvoll, da sich dieses *Component* sehr selten im Wert ändert und viele *Entities* das selbe `RenderMesh` *Component* besitzen. Das kann ganz simpel einfach das Aussehen eines Baumes in einem Spiel sein. Oft gibt es viele Bäume in dem Spiel und diese ändern auch ihr Aussehen nicht.
Buffer Components: Falls man mehrere *Components* der selben Art für ein *Entity* braucht,

[8] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-components.html`
[9] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-type.html`

sind *Buffer Components* sehr hilfreich. Diese agieren wie ein Array von *Components*. Das ist besonder hilfreich, wenn man einem *Entity* ein Inventar geben möchte, da dieses aus mehreren Gegenständen bestehen kann. Listing 4 zeigt, wie man ein Inventar entwerfen würde.

```
//Array Kapazität auf acht festlegen
[InternalBufferCapacity(8)]
public struct ItemAmount : IBufferElementData
{
    public int itemID;
    public int amount;
}
```

Listing 4: Beispiel eines *Buffer Components*. Das *Buffer Component* stellt ein Inventar dar, welches verschiedene Items mit ihrer Anzahl speichert. Das Array ist maximal acht Elemente groß.

Wie man sieht, leitet die Struktur diesmal von `IBufferElementData` ab, welche den Typ *Buffer Component* angibt. Zusätzlich wird hier das Attribut `InternalBufferCapacity` genutzt. Damit legt man die Kapazität des *Components* fest. Standardmäßig ist die Kapazität die Anzahl an Elementen, welche in 128 Bytes passen. Also hätten wir bei diesem Beispiel eine Kapazität von 16, da in einem *Component* zwei Integer à 32 bit gespeichert werden. Jedoch braucht man für das Inventar beispielsweise nur 8 zu speichernde Gegenstände und kann so den benötigten Speicherplatz reduzieren.

3.2.3 Systeme[10]

Systeme beschreiben das Verhalten und beinhalten die Logik zum Transformieren der Daten. Die Systeme werden ein Mal pro ausgegebenem Bild mithilfe der `OnUpdate` Funktion auf dem *Main Thread* ausgeführt. Genau wie im objektorientierten Ansatz von Unity gibt es auch hier mehrere Methoden, die zum Start oder Ende ausgeführt werden. Zusätzlich kann man unter den definierten Systemen eine Reihenfolge festlegen, in der diese ausgeführt werden sollen. So wie bei den *Components* gibt es auch bei den Systemen eine Klasse für verwaltete Daten (welche in diesem Fall von der Klasse `SystemBase` erbt) und eine Struktur für unverwaltete Daten (welche in diesem Fall das Interface `ISystem` implementiert). Zusätzlich sind Systeme immer an eine Welt gebunden. Listing 5 zeigt, wie ein System für unverwaltete Daten und ohne implementierte Logik aussieht.

```
public partial struct ExampleSystem : ISystem, ISystemStartStop
{
    //Wird beim Erstellen des Systems ausgeführt
    public void OnCreate(ref SystemState state){}
    //Wird vor dem ersten OnUpdate des Systems ausgeführt
    public void OnStartRunning(ref SystemState state){}
    //Wird für jedes ausgegebene Bild ausgeführt
```

[10] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-systems.html`

```
8       public void OnUpdate(ref SystemState state){}
9       //Wird beim Stoppen des Systems ausgeführt
10      public void OnStopRunning(ref SystemState state){}
11      //Wird beim Zerstören des Systems ausgeführt
12      public void OnDestroy(ref SystemState state){}
13  }
```

Listing 5: Beispiel eines Systems. Das System hat verschiedene Methoden, welche zum Start beziehungsweise Ende ausgeführt werden und eine `OnUpdate` Methode. Diese wird einmal pro Bild ausgeführt.

Dabei ist das Interface `ISystemStartStop` optional und bietet die Möglichkeit beim Starten und Stoppen des Systems zusätzliche Logik auszuführen. Wie man sieht, wird allen Methoden auch eine Referenz des `SystemState` übergeben. Darüber kann man auf verschiedene nützliche Dinge zugreifen, wie beispielsweise die Welt, den *EntityManager*, oder aber auch alle *Components* eines Typs. Eine sinnvolle Herangehensweise ist es, in der `OnUpdate` Methode Jobs zu schedulen. Dies wird in Kapitel 3.3 weiter beschrieben.

3.2.4 Archetypen[11]

Ein Archetyp ist eine gewisse Zusammenstellung aus *Components*. Jedes *Entity* kann somit einem Archetyp zugeordnet werden. Beispielsweise sind alle *Entities*, welche nur das `Translation` *Component* (Position) haben, einem Archetyp zugeordnet. *Entities*, welche zusätzlich *Component* A besitzen, gehören zu einem anderen Archetyp. Abbildung 3 zeigt das Konzept von Archetypen.

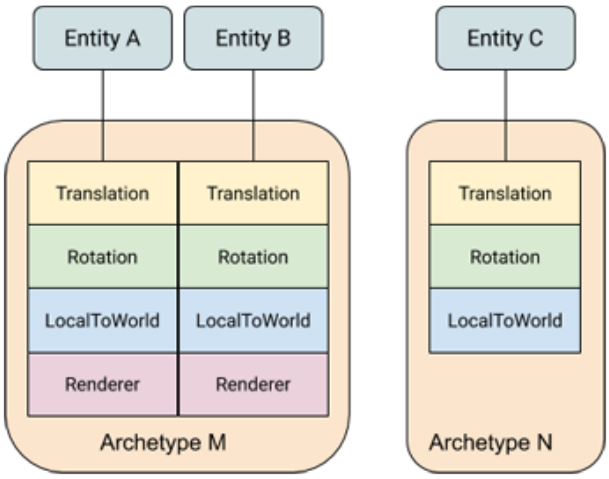

Abbildung 3: Konzept von Archetypen. *Entities*, welche die gleiche Zusammenstellung von *Components* haben gehören einem Archetyp an. *Entity* A und B gehören durch die gleiche Zusammenstellung an *Components* also dem Archetyp M an. *Entity* C hat eine andere Zusammenstellung und gehört Archetyp N an.
Quelle: `https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/ecs_core.html`

[11] `https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-archetypes.html`

Durch Archetypen ist es möglich, sehr performant, datenorientiert zu arbeiten. Möchte man in einem System auf verschiedenen *Components* Operationen ausführen, kann man alle Archetypen nach diesen *Components* durchsuchen und muss nicht alle *Entities* durchsuchen. Zusätzlich kann man diese Anfragen an Archetypen cachen um noch mehr Performance zu erreichen. Unity speichert alle *Components* von *Entities* für einen gewissen Archetyp in einem Block. Dieser wird auch *Chunk* genannt. Abbildung 4 zeigt, wie Archetypen mit *Chunks* zusammenhängen.

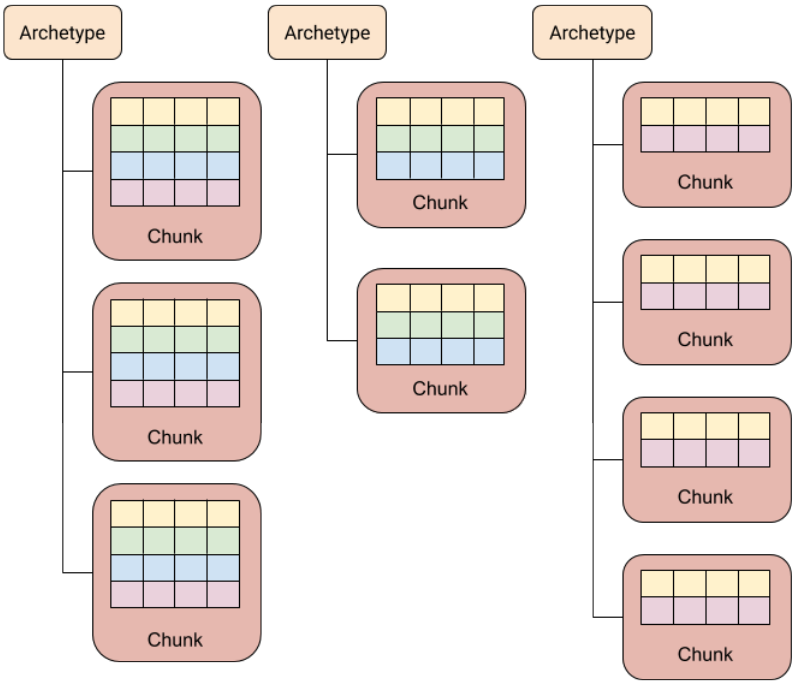

Abbildung 4: Konzept von *Chunks*. *Chunks* sind Speicherbereiche für Archetypen. In dem Bild ist zu erkennen, dass der Archetyp links vier *Components* hat (Anzahl an Reihen), der Archetyp in der Mitte drei *Components* und der Archetyp rechts zwei *Components*. In einen *Chunk* passen hier jeweils vier *Entities*, wobei dies von der Anzahl und Größe der jeweiligen *Components* abhängig ist. Wenn ein *Chunk* voll ist, muss ein neuer *Chunk* erstellt werden. Daher kann es, abhängig von der Anzahl der *Entities* in einem *Chunk*, unterschiedlich viele *Chunks* pro Archetyp geben.
Quelle: `https://docs.unity3d.com/Packages/com.unity.entities@0.50/manual/ecs_core.html`

Jeder dieser *Chunks* ist 16 KiB groß. Demnach hängt es von den *Components* ab, wie viele *Entities* in einen *Chunk* passen. Der *Chunk* beinhaltet ein Array für jeden Typ der *Components* und zusätzlich ein Array für die ID's der *Entities*. Pro Arrayindex wird je ein *Entity* gespeichert. Im Index 0 aller Arrays werden die Daten des ersten *Entities* gespeichert. Falls ein *Entity* zerstört, oder in einen anderen *Chunk* bewegt wird (falls ein *Component* hinzugefügt oder entfernt wird), wird das letzte *Entity* an seine Stelle bewegt. Falls ein *Chunk* voll ist, erstellt der *EntityManager* einen neuen *Chunk*, wenn ein *Entity* hinzukommt. Leere *Chunks* werden gelöscht.

3.2.5 Strukturelle Änderungen

Strukturelle Änderungen sind eines der wenigen unperformantem Operationen in Unity's *ECS*. Strukturelle Änderungen können das Erstellen und Zerstören eines *Entities*, das Hinzufügen und Entfernen von *Components* oder das Ändern von Daten eines *Shared Components* sein[12]. Also im Grunde Operationen, welche das Ändern eines, oder mehrerer *Chunks* erfordern. Solche Änderungen könnten andere zur selben Zeit ausgeführte Aktionen invalidieren und müssen deshalb auf dem *Main Thread* ausgeführt werden. Um dennoch Änderungen dynamisch an beliebiger Stelle ausführen zu können, nutzt man den *Entity Command Buffer* (*ECB*). Mit dem *ECB* lassen sich strukturelle Änderungen sammeln und zu einem späteren Zeitpunkt in einer festgelegt Reihenfolge ausführen. So lassen sich problemlos aus einem Job strukturelle Änderungen sammeln und nach Beendigung des Jobs, diese auf dem *Main Thread* ausführen. Listing 6 zeigt, wie ein *ECB* funktioniert.

```
public void OnUpdate(ref SystemState state)
{
    //Neuer ECB wird erstellt
    EntityCommandBuffer ecb = new EntityCommandBuffer(Allocator.TempJob);
    //Job wird erstellt und der ECB wird übergeben
    new ExampleJob
    {
        ecb = ecb
    //Mit Schedule wird der Job gestartet
    }.Schedule();
    state.CompleteDependency();
    //Strukturelle Änderungen werden abgespielt
    ecb.Playback(state.EntityManager);
    //ECB muss auch wieder disposed werden
    ecb.Dispose();
}
```

Listing 6: Beispiel eines *Entity Command Buffers*. Der *ECB* wird erstellt, es werden strukturelle Änderungen vorgenommen und diese werden auf dem *Main Thread* abgespielt.

Man erstellt einen *ECB*, reiht verschiedene Aktionen in die Schlange ein und spielt diese auf dem *Main Thread* wieder ab. Danach sollte man den *ECB* wieder disposen. Das gibt den Speicher für das Objekt wieder frei und räumt gegebenenfalls weitere Ressourcen auf. In dem Job kann man mit dem übergebenen *ECB* verschiedene Aktionen durchführen. Diese können beispielsweise so aussehen:

```
//Ein Entity erstellen:
Entity newEntity = ecb.Instantiate(e);
//Dem Entity ein Component hinzufügen:
ecb.AddComponent<ExampleComponent>(newEntity);
```

Wie ein Job genau aussieht und funktioniert wird in Kapitel 3.3 beschrieben.

[12] https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-structural-changes.html

3.3 Job System

Das Job System von Unity ist ein leichter Weg um parallelen Code auszuführen. Jobs werden meist in der `OnUpdate` Funktion erzeugt und ausgeführt. Dabei kann man entscheiden, ob der Job auf dem *Main Thread*, einem *Worker Thread*, oder mehreren *Worker Threads* ausgeführt werden soll. Die Anzahl an verfügbaren *Worker Threads* wird dabei von der Anzahl an verfügbaren Kernen im Prozessor bestimmt. Nachdem der Job erstellt wurde, kann man ihn mit dem Funktionsaufruf `Run` (Ausführung auf dem *Main Thread*), `Schedule` (Ausführung auf einem *Worker Thread*), oder `ScheduleParallel` (Ausführung auf mehreren *Worker Threads*) ausführen. Zusätzlich gibt es noch verschiedene Jobarten, welche auf die Arbeit mit dem *ECS* angepasst wurden.

3.3.1 Job mit einem *Entity*

Der Job mit einem *Entity* ist der Standardjob, um über Daten von *Components* zu iterieren. Er implementiert das Interface `IJobEntity`. Mit ihm lässt sich leicht definieren, welche Daten man lesen oder schreiben möchte. Zusätzlich kann man noch weitere Attribute, an die in dem Job befindlichen `Execute` Methode übergeben lassen. Listing 7 zeigt einen fertig implementierten Job, der das `IJobEntity` Interface implementiert.

```
1  //Beispiel Component
2  public struct ExampleComponent : IComponentData { public float Value; }
3  //Entities mit dem DontWantComponent werden ausgeschlossen
4  [WithNone(typeof(DontWantComponent))]
5  //Job ist eine Struktur und implementiert das IJobEntity Interface
6  public partial struct ExampleJob : IJobEntity
7  {
8      //Execute Funktion wird für jedes ExampleComponent ausgeführt
9      void Execute(Entity e, ref ExampleComponent component)
10     {
11         //Addiert eins zu jedem ExampleComponent Wert.
12         component.Value += 1f;
13     }
14 }
15 //Beispiel System welches den Job verwendet
16 public partial class ExampleSystem : ISystem
17 {
18     protected override void OnUpdate()
19     {
20         //Job wird erstellt und auf mehreren Worker Threads ausgeführt
21         new ExampleJob().ScheduleParallel();
22     }
23 }
```

Listing 7: Beispiel für einen Job mit einem *Entity* für eine einfache Addition. Der `Execute` Methode wird das *Entity* und das `ExampleComponent` übergeben.

Wie man in dem Beispiel sieht, ist der Job auch wieder eine Struktur und implementiert das `IJobEntity` Interface. Durch dieses Interface kann man eine eigene `Execute` Methode erstellen und anpassen. Je nach dem, welche *Components* man der `Execute` Methode übergibt, werden alle *Entities*, welche diese *Components* besitzen, an die Funktion übergeben. Die übergebenen *Components* sind dabei die des *Entity*. Falls man die *Entities* weiter einschränken oder lockern will, welche der Funktion übergeben werden, kann man dies mit den Attributen `WithNone( typeof(ComponentX) )` (*Entities* mit dem ComponentX werden ausgeschlossen), `WithAll( typeof(ComponentX) )` (*Entities* müssen das ComponentX besitzen), oder `WithAny( typeof( ComponentX) , typeof(ComponentY) )` (*Entities* müssen das ComponentX, oder das ComponentY besitzen) über dem Job definieren.
In der `Execute` Funktion lässt sich dann mit den *Components* arbeiten. *Components*, welche man nur lesen will, sollte man mit dem Schlüsselwort `in` übergeben, *Components*, welche man lesen und schreiben möchte, sollte man mit den Schlüsselwort `ref` (Zeile 9) übergeben. Dann kann man seine Logik so definieren, wie man sie braucht.

3.3.2 Job mit einem *Chunk*

Der Job mit einem *Chunk* implementiert das Interface `IJobChunk` und behandelt ganze *Chunks* an Daten. Dem Job wird nicht nur ein einzelnes *Entity* mit seinen *Components* übergeben, sondern ein ganzer *Chunk*, mit allen *Components*, die darin enthalten sind. Er kann aus dem *Chunk* das komplette Array eines *Component* Typs holen und dieses dann in einer Schleife bearbeiten. Die Schleife geht die Anzahl an *Entities* in dem *Chunk* durch und indiziert das *Component* Array. Der Job mit einem *Chunk* ist komplexer zu implementieren, als ein Job mit einem *Entity*. Er wird jedoch immer durch Codegenerierung von einem `IJobEntity` Job erzeugt, also ist in Wirklichkeit jeder Job mit einem *Entity* eigentlich ein Job für den ganzen *Chunk*. Listing 8 zeigt einen implementierten `IJobChunk` Job, welcher dieselbe Funktion hat, wie der `IJobEntity` Job in Listing 7.

```
1 //Job der das IJobChunk Interface implementiert
2 public struct ExampleJob : IJobChunk
3 {
4     //ComponentTypeHandle erlaubt es in der Execute Funktion auf die
    Components im Chunk zuzugreifen. Man braucht für jeden Component Typ
    einen eigenen ComponentTypeHandle
5     public ComponentTypeHandle<ExampleComponent> exampleTypeHandle;
6
7     //Execute Funktion, welche den kompletten Chunk übergeben bekommt. Der
    Integer unfilteredChunkIndex enthält den Index des Chunks welcher
    bearbeitet wird, da es, je nach Anzahl an Entities, auch mehr als einen
    geben kann. chunkEnabledMask ist eine Bitmaske. Falls das n'te Bit
    gesetzt ist, passt das n-te Entity in die Anforderungen des Jobs und
    sollte verarbeitet werden. Das Attribut useEnabledMask vereinfacht den
    Nutzen von chunkEnabledMask. Falls useEnabledMask false ist passen alle
    Entities zu den Anforderungen.
```

```
    public void Execute(in ArchetypeChunk chunk, int unfilteredChunkIndex,
        bool useEnabledMask, in v128 chunkEnabledMask)
    {
        //Man erhält ein Array aller Components von einem Chunk
        //durch den ComponentTypeHandle
        NativeArray<ExampleComponent> exampleComponents =
            chunk.GetNativeArray(ref exampleTypeHandle);
        //Der ChunkEntityEnumerator gibt mittels NextEntityIndex das nächste
        //zu bearbeitende Entity unter der Berücksichtigung von
        //chunkEnabledMask wieder.
        var enumerator = new ChunkEntityEnumerator(useEnabledMask,
            chunkEnabledMask, chunk.Count);
        //Schleife über alle Entities, die verarbeitet werden sollen
        while(enumerator.NextEntityIndex(out var i))
        {
            float3 newValue = exampleComponents[i].Value + 1;
            exampleComponents[i] = newValue;
        }
    }
}
```

Listing 8: Beispiel für einen Job mit einem *Chunk* für eine einfache Addition. Dieser ist analog zu dem Beispiel für ein Job mit einem *Entity*.

Wie man sieht, ist der `IJobChunk` Job deutlich länger und komplexer als der `IJobEntity` Job, obwohl beide das Gleiche machen. Jedoch kann man viel besser erkennen, was wirklich bei der Ausführung des Jobs passiert. Die *Chunks*, welche getrennt im Speicher liegen, werden der Execute Funktion übergeben. Welche das sind, kann man mit einer *EntityQuery* bestimmen, welche man bei dem `IJobEntity` Job nicht unbedingt benötigt. Diese kann so aussehen:

```
EntityQuery query = GetEntityQuery(typeof(ExampleComponent));
```

Mit dieser *EntityQuery* kann man dann den Job ausführen:

```
protected override void OnUpdate(){
    var job = new ExampleJob();
    job.exampleTypeHandle = GetComponentTypeHandle<ExampleComponent>(false);
    //Dependency muss mit dem Job aktualisiert werden
    this.Dependency = job.ScheduleParallel(query, this.Dependency);
}
```

Zusätzlich braucht man, da man *Components* auch deaktivieren kann, eine Bitmaske, welche verhindert, dass *Entities* bearbeitet werden, deren *Components* deaktiviert sind. Ein `IJobEntity` generiert den `IJobChunk` so, dass automatisch darauf geachtet wird. Deshalb wird empfohlen den Job mit einem *Entity* zu nutzen, welcher einfacher zu implementieren ist.

3.4 Burst Compiler

Der Burst Compiler dient der Performance Optimierung von beschränktem Code, weshalb er besser ist als andere Compiler. Dabei sind die Schritte, wie man zu ausführbarem Code kommt, wie folgt: Zunächst wird der C# Code von dem Roselyn C# Compiler in die Intermediate Language (IL) übersetzt. Die IL ist dabei eine Zwischensprache auf die nun verschiedene Compiler angewendet werden können. Normalerweise nutzt Unity die Mono Runtime[13]. Diese wiederum nutzt bei der Ausführung einen Just-In-Time (JIT) Compiler, welcher die gegebene IL in ausführbaren Code umwandelt, sobald er gebraucht wird. Eine Alternative zu der Mono Runtime ist die IL2CPP Runtime. Diese übersetzt die IL Ahead-Of-Time (AOT), also vor der Ausführung, zunächst in C++ Code. Dann kann der Code von einem C++ Compiler übersetzt und ausgeführt werden.
Burst bekommt auch die IL. Diese wird von Burst, welcher den LLVM Compiler[14] nutzt, in ausführbaren Code übersetzt. Der große Vorteil den Burst hierbei hat, ist, dass Burst lediglich beschränkten Code unterstützt. Burst kann beispielsweise nicht mit verwalteten Daten umgehen, dafür braucht es keine Garbage Collection. Burst hat zusätzlichen einen großen Überblick über große Teile des Codes und kann somit beispielsweise auch einen Job mit seiner Implementierung kompilieren. Durch diese Beschränktheit im Programmieren, kann Burst mehr Annahmen über den Code machen und dadurch wesentlich schnelleren Code produzieren. Da man mit dem *ECS* sowieso auf leichtgewichtigen Code wert legt, lässt sich dies gut mit Jobs oder auch ganzen System verbinden. Damit Burst schneller kompilierbaren Code findet, nutzt man das `[BurstCompile]` Attribut über Jobs, oder Funktionen von Systemen innerhalb des Codes:

```
1 //BurstCompile Attribut
2 [BurstCompile]
3 private struct ExampleJob : IJobEntity{}
```

3.4.1 Burst Kompilierung

Burst hat zwei Arten wie es den Code kompiliert:
1. **Just-In-Time Kompilierung**: Diese Methode wird im Unity Editor verwendet. Das heißt der Compiler kompiliert den Code dann, wenn er verwendet wird. Der Code läuft zunächst mit der normalen Mono Runtime, bis Burst im Hintergrund den Code kompiliert hat. Das bedeutet es wird asynchron kompiliert. Man kann Unity mit dem Attribut `CompileSynchronously` jedoch auch dazu zwingen, den Code vor dem ersten Ausführen der Coderegion mit Burst zu kompilieren. Dies ist beispielsweise für Laufzeitanalysen sehr vorteilhaft.
2. **Ahead-Of-Time Kompilierung**: Diese Methode wird beim Bauen des Spiels verwendet. Bauen meint hierbei das Spiel von dem Unity Editor in ein ausführbares Programm umzuwan-

[13] `https://docs.unity3d.com/Manual/Mono.html`
[14] `https://llvm.org/`

deln. Dabei speichert Burst den kompilierten Code in eine Bibliothek, welche mit dem Spiel ausgeliefert wird. Zur Laufzeit wird dann der kompilierte Code verwendet.

3.4.2 Beispiel Addition[15]

Um ein kleines Beispiel zu geben, wie der Burst Compiler Code optimieren kann, gibt es in Listing 9 eine Beispielmethode, welche zwei Integer addiert und das Ergebnis zurückgibt. Das Beispiel wird einmal mit dem Burst Compiler kompiliert und dann noch einmal mit dem Mono Compiler kompiliert um das Ergebnis zu vergleichen.

```
public static int Add(int left, int right){
  return left + right;
}
```

Listing 9: Beispiel einer Addition, ausgelagert in eine Methode.

Mit Burst kompiliert ergibt sich dieser Assembly Code:

```
Add: (LLVM x64)
  lea eax, [rcx, rdx]
  ret
```

Listing 10: Addition Burst kompiliert

Mono kompiliert die Methode zu diesem Code:

```
Add: (mono 5.16)
  sub    $0x18, %rsp
  mov    %rdi, (%rsp)
  mov    %rsi, 0x8(%rsp)
  mov    %rdi, %rax
  add    0x8(%rsp), %eax
  add    $0x18, %rsp
  retq
```

Listing 11: Addition Mono kompiliert

Wie man sieht gibt es einen großen Unterschied zwischen den kompilierten Ergebnissen. Während beim Mono Compiler die Variablen zunächst ein paar mal verschoben werden, führt der Burst Compiler direkt die Addition (`lea`) aus und gibt den Wert zurück. Unity Entwickler Alexandre Mutel erwähnt dazu noch, dass dieses Beispiel so in der Realität nicht vorkommt, da man eine Addition nicht in eine Funktion auslagern würde, jedoch trotzdem ein großer Unterschied erkennbar ist (original: *But this particular example would not really happen because most of the time this function is going to be in line by a call at the call site so you won't get this kind of code in Mono, but still the way they are parsing of arguments and so on are still here ...*) [5].

[15] `https://www.youtube.com/watch?v=QkM6zEGFhDY`

4 Fabrik Spiel in Unity

Um zu sehen, welches Potential die Datenorientierte Programmierung in der Spieleentwicklung bietet, wurde eine Spielsimulation sowohl datenorientiert als auch objektorientiert erstellt[16]. Die Spielesimulation ist eine Art Aufbauspiel, welche mit dem populären Spiel Factorio[17] verglichen werden kann. Hier wird es jedoch etwas schlichter und einfacher gehalten. Um beide Programmierparadigmen miteinander zu vergleichen, wird eine kleine Fabrik, welche Stahl produziert, in der Spielwelt generiert und vervielfältigt. Dies soll einen Spieler simulieren, der mehrere Fabriken in die Spielwelt platziert. In Abbildung 5 sieht man wie eine Produktionsstraße für Stahl in dem Spiel aussehen kann.

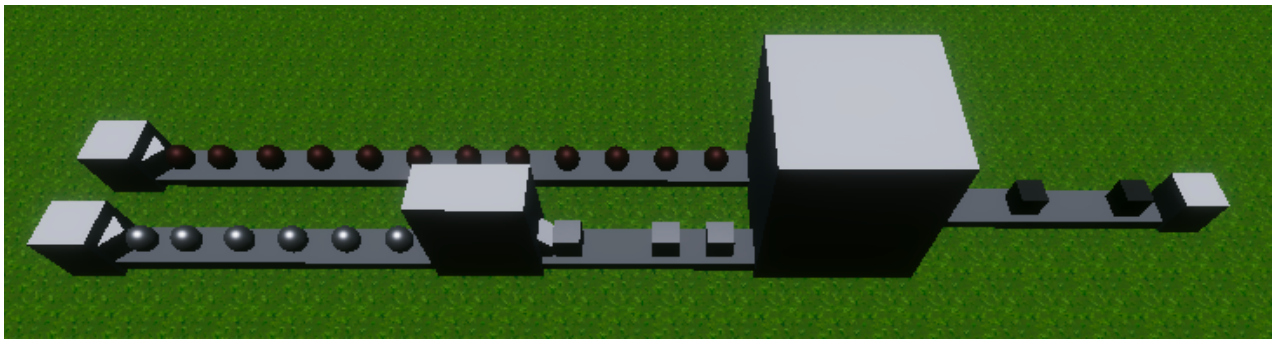

Abbildung 5: Eine Stahl Fabrik in der Unity Spielesimulation. Links wird Kohle und Eisenerz gefördert. Die Items werden immer von links nach rechts mit Förderbändern zu dem nächsten Kasten weiterbewegt. Das Eisenerz wird zu Eisenbarren geschmolzen (siehe unteres linkes Förderband). In dem großen Kasten wird aus der Kohle und den Eisenbarren Stahl produziert und in dem rechten Kasten verbraucht.

Ganz links befinden sich zwei Kästen. Diese sollen Erzbohrer darstellen, welche Eisenerz (silbern) und Kohle (braun) aus der Erde befördern und rechts auf ein Förderband legen. Die beiden Förderbänder transportieren die Kohle und das Eisenerz nach rechts weiter. Das Eisenerz wird als nächstes in Eisenbarren geschmolzen und ist danach rechteckig. Die Kohle und die Eisenbarren kommen dann gemeinsam in den großen Würfel, wo sie zu Stahl verarbeitet werden. Dieser Stahl wird dann wieder auf ein Förderband gelegt und in dem kleinen Würfel anschließend verbraucht. Die ganze Produktionskette soll einem Videospiel nahekommen, damit es eine möglichst realistische Simulation bietet. Nachfolgend wird gezeigt, wie die Spielsimulation objektorientiert und datenorientiert umgesetzt und gebenchmarkt wird.

[16] Der Quellcode für die objektorientierte Umsetzung kann zur Reproduktion der Experimente unter `https://github.com/dennis12493/FactoryGameSimulationOOP` gefunden werden. Der Quellcode für die datenorientierte Umsetzung findet man unter `https://github.com/dennis12493/FactoryGameSimulationDOP`.

[17] https://www.factorio.com/

4.1 Objektorientierte Programmierung

In der Objektorientierten Programmierung mit Unity dreht sich alles um *GameObjects*. Jedes Objekt in der Szene, egal ob sichtbar oder nicht, ist ein *GameObject*. *GameObjects* können verschiedene Komponenten haben, welche Eigenschaften definieren, also Daten speichern. Diese Komponenten sind jedoch nicht zu verwechseln mit den *Components* aus dem datenorientierten Ansatz von Unity. Die Komponenten im objektorientierten Ansatz haben jeweils eine `Start` und eine `Update` Methode welche das Verhalten definieren. Die `Start` Methode wird einmalig vor dem ersten Aufruf der `Update` Methode ausgeführt. Die `Update` Methode läuft einmal pro ausgegebenem Bild. Listing 12 zeigt die Item Komponente des Item *GameObjects*.

```
1  public class Item : MonoBehaviour {
2      private int2 pos;
3      //Serialisiertes Feld für den Unity Editor
4      [SerializeField] private int itemID;
5
6      void Update() {
7        //Gegenstand wird zu der übergebenen Position pos bewegt
8          transform.position = Vector3.Lerp(transform.position,
9              new Vector3(pos.x, pos.y, -0.5f), Time.deltaTime * 2f);
10     }
11
12     public void SetPosition(int2 pos) {
13         this.pos = pos;
14     }
15 }
```

Listing 12: Item Komponente im objektorientierten Ansatz. Es speichert die Position und seine ID. In der `Update` Methode bewegt sich das Item linear zu der übergebenen Position.

Wie man sieht, beinhaltet das `MonoBehaviour` nicht nur die Daten, sondern auch die Logik. Für ein Item benötigen wir zum einen die Position, wohin sich das Item bewegen soll, zum anderen speichern wir auch eine ID über die wir das Item ganz einfach identifizieren können. Das Attribut `SerializeField` (Zeile 5) zwingt Unity dazu, ein editierbares Feld im Editor zu erstellen an dem man die `itemID` setzen kann. Dadurch lassen sich vorgefertigte Items erstellen, welche unterschiedlich aussehen und man zusätzlich die Item ID setzen kann. Beispielsweise ist ein Eisenbarren quadratisch, hat ein silbernes Material und bekommt die ID drei. Diese vorgefertigten Items kann man dann zur Laufzeit erstellen.

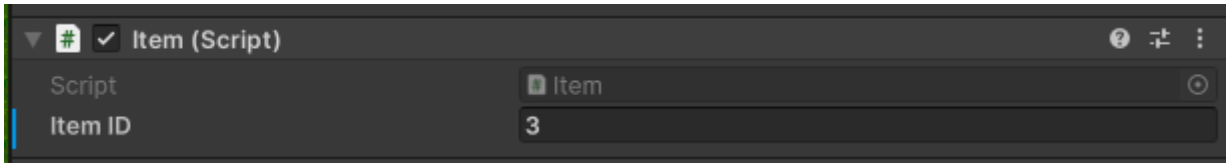

Abbildung 6: Ein editierbares Feld in dem Unity Editor. Das Feld wird durch ein `public` Attribut, oder dem `SerializeField` Flag erzeugt.

Die `Start` Methode ist in diesem Fall nicht notwendig. Die `Update` Methode bewegt das Item langsam an die übergebene Position. `Time.deltaTime` (Zeile 11) gibt die Zeit in Sekunden von dem letzten Bild bis zu dem momentanen Bild an. Dadurch wird die Bewegung linear.
Ein weiterer Teil der Spielesimulation ist die Bewegung der Items über das Förderband. Das Förderband übergibt die Position an das Item und bestimmt daher, in welche Richtung sich das Item bewegen soll. Listing 13 beschreibt den Aufbau des Förderbandes.

```
1  public class BeltPath : MonoBehaviour
2  {
3      private List<ConveyorComponent> beltPath = new ();
4      //Referenzen auf die Komponenten des GameObjects werden geholt
5      private InputConveyorComponent input
6          = GetComponent<InputConveyorComponent>();
7      private OutputConveyorComponent output
8          = GetComponent<OutputConveyorComponent>();
9      private float timeToMove = 2f;
10
11     public void Update()
12     {
13         timeToMove -= Time.deltaTime;
14         //Bewegung der Items wird alle 2 Sekunden durchgeführt
15         if(timeToMove > 0) return;
16         var lastBelt = beltPath[^1];
17         if (!ReferenceEquals(lastBelt.item, null)
18             && ReferenceEquals(output.GetItem(), null)) {
19             //Gegenstand von dem letzten Förderbandsegment
20             //auf die Ausgabe legen
21             var item = lastBelt.item;
22             var itemComponent = item.GetComponent<Item>();
23             itemComponent.SetPosition(output.GetPosition());
24             output.SetItem(item);
25             lastBelt.item = null;
26         }
27         //Gegenstände von hinten nach vorne um ein Segment verschieben
28         for (int i = beltPath.Count-2; i >= 0; i--) {
29             var thisConveyor = beltPath[i];
30             var lastConveyor = beltPath[i + 1];
31             if (!ReferenceEquals(thisConveyor.item, null)) {
32                 if (ReferenceEquals(lastConveyor.item, null)) {
33                     //Item kann verschoben werden
34                     var item = thisConveyor.item;
35                     var itemComponent = item.GetComponent<Item>();
36                     lastConveyor.item = item;
37                     //Position des Items aktualisieren
38                     itemComponent.SetPosition(lastConveyor.pos);
39                     thisConveyor.item = null;
40                 }
41             }
42         }
43         var firstConveyor = beltPath[0];
```

```
44          input.SetOccupied(!ReferenceEquals(firstConveyor.item, null));
45          if (!ReferenceEquals(input.GetItem(), null)
46              && ReferenceEquals(firstConveyor.item, null)) {
47              //Item wird von der Eingabe auf das erste Segment gelegt
48              firstConveyor.item = input.GetItem();
49              input.RemoveItem();
50          }
51          //Zeit wird wieder hochgestellt
52          timeToMove += 2f;
53      }
54  }
```

Listing 13: Förderband Komponente im objektorientierten Ansatz. Es speichert die Ein- und Ausgabe des Förderbandes und alle Segmente die zu dem Förderband gehören. Das Förderband sorgt dafür, dass alle Items entlang des Förderbandes weiterbewegt werden.

Für das Förderband wird eine Liste mit vorhandenen Segmenten, die Eingabe, die Ausgabe und eine Zeit gespeichert (Zeile 3 - 8). Die Zeit wird in der `Start` Methode initialisiert und die Einbeziehungsweise Ausgabe wird über die Funktion `GetComponent` von dem *GameObject* geholt. In der `Update` Funktion wird zunächst nur die `timeToMove` Variable herunter gezählt (Zeile 13). Sollte diese Variable unter Null fallen, werden alle Items von hinten nach vorne ein Segment weiter bewegt, sofern dies möglich ist. Ist die Ausgabe nicht belegt, wird ein vorhandenes Item in die Ausgabe gelegt (Zeile 17 - 26). In der Schleife werden die Items auf den einzelnen Segmenten weiterbewegt (Zeile 28 - 42) und wenn ein Item in der Eingabe liegt wird dieses auf das erste Segment weiterbewegt (Zeile 45 - 50). Immer wenn ein Item weitergegeben wird (egal ob an ein Segment, oder an die Ausgabe) wird auch die neue Position an das Item weitergegeben (Zeile 38). Durch das Bewegen der Items von hinten nach vorne verhindert man, dass sich Items nicht bewegen, obwohl sie es könnten.
Auf den Förderbändern sind Items sichtbare *GameObjects*. In Gebäuden wird lediglich mit den ID's der Items gearbeitet, da man hier keine Objekte braucht. Items werden in der Ausgabe eines Gebäudes erstellt und von dort an das Förderband übergeben. Die `Update` Methdode dafür zeigt Listing 14.

```
1  void Update()
2  {
3      //Wenn die Ausgabe leer ist gibt es nichts zu tun
4      if(itemID == -1) return;
5      outputGameObject ??= BuildingDictionary.Instance
6          .GetGameObjectAtPosition(pos);
7      if (ReferenceEquals(outputGameObject, null) ||
8          !outputGameObject.TryGetComponent(out InputConveyorComponent input))
9          return;
10     //Wenn die Eingabe des Förderbandes belegt ist wird auch keine
11     //Änderung vorgenommen
12     if(input.IsOccupied() || !ReferenceEquals(input.GetItem(), null))
13         return;
14     var itemGameObject = Items.INSTANCE.GetItem(itemID);
```

```
15      //Item wird erstellt
16      var item = Instantiate(itemGameObject, new Vector3(pos.x, pos.y, -0.5f),
        Quaternion.identity);
17      item.transform.localScale = new Vector3(0.5f, 0.5f, 0.5f);
18      var itemComponent = item.GetComponent<Item>();
19      itemComponent.SetPosition(pos);
20      var inputConveyorComponent = outputGameObject
21          .GetComponent<InputConveyorComponent>();
22      //Der Eingabe wird das Item zugewiesen
23      inputConveyorComponent.SetItem(item);
24      //Item wird aus der Ausgabe entfernt
25      itemID = -1;
26      itemCreated = true;
27  }
```

Listing 14: Erstellung eines Items im objektorientierten Ansatz. Wenn ein Förderband verbunden ist und ein Item erstellt werden soll, wird das passende *GameObject* instanziert. Dieses wird dann an das Förderband übergeben.

Da man sich im objektorientierten Ansatz, ohne weitere Vorkehrungen, immer auf dem *Main Thread* befindet, lassen sich die Items direkt erstellen und der Eingabe übergeben. Das Zerstören von Items, also der Fall wenn Items von einem Förderband in ein Gebäude übergeben werden, funktioniert sehr ähnlich zu dem Erstellen von Items.

4.2 Datenorientierte Programmierung

In der Datenorientierten Programmierung werden statt der GameObjects *Components* und Systeme entworfen. Dabei werden in der Implementierung des Factory Spiels alle Aspekte des *Datenorientierten Technologie-Stack's* von Unity eingesetzt. Es werden die Daten in *Components* gespeichert, welche von Systemen verarbeitet werden. Die Systeme werden, soweit möglich, so umgesetzt, dass der Burst Compiler verwendet werden kann. Zusätzlich werden Jobs erstellt, welche aus Systemen ausgeführt werden.
Das Item *Component* des datenorientierten Ansatzes zeigt Listing 15.

```
1  public struct ItemComponent : IComponentData
2  {
3      public int2 pos;
4      public int itemID;
5  }
```

Listing 15: Item *Component* im datenorientierten Ansatz. Es werden nur die Daten des Items gespeichert. Die Logik des Items befindet sich in einem System.

In dem `ItemComponent` wird die Position und die ID des Items gespeichert. Was direkt auffällt, ist die klare Trennung der Daten von der Logik, welche sich hier in einem System befindet. Die Logik des objektorientierten Items (Listing 12) befindet sich im datenorientierten Ansatz in dem `ItemSystem` in Listing 16.

```
1  [BurstCompile(CompileSynchronously = true)]
2  public partial struct ItemSystem : ISystem
3  {
4      [BurstCompile(CompileSynchronously = true)]
5      public void OnUpdate(ref SystemState state)
6      {
7        //Job erstellen, Zeit übergeben und schedulen
8          new ItemMoveJob
9          {
10             deltaTime = SystemAPI.Time.DeltaTime
11         }.ScheduleParallel();
12     }
13
14     [BurstCompile(CompileSynchronously = true)]
15     public partial struct ItemMoveJob : IJobEntity
16     {
17         public float deltaTime;
18
19         [BurstCompile(CompileSynchronously = true)]
20         private void Execute(ref LocalTransform transform,
21             in ItemComponent item)
22         {
23           //Gegenstand wird zu der übergebenen Position pos bewegt
24             transform = transform.WithPosition(
25                 Vector3.Lerp(transform.Position.xyz,
26                     new Vector3(item.pos.x, item.pos.y, -0.5f),
27                 deltaTime * 2f));
28         }
29     }
30 }
```

Listing 16: Item System zum Bewegen von Items im datenorientierten Ansatz. Aus der `OnUpdate` Methode wird ein Job geschedult, welcher die Items bewegt.

Das `ItemSystem` ist durch das implementierte Interface `ISystem` als System defniniert und kann daher die `OnUpdate` Methode nutzen. Bei dem Item System kommt zusätzlich auch der Burst Compiler und das Job System zum Einsatz. Das Attribut `BurstCompile` hilft dem Burst Compiler Methoden zu finden, welche mit Burst kompiliert werden sollen. `CompileSynchronously` dient dem Testen und besagt, dass erst das System durch den Burst Compiler kompiliert werden muss, bevor es zum ersten Mal ausgeführt wird. Andernfalls könnte das System schon laufen, ohne von dem Burst Compiler profitiert zu haben. Die Methode `OnUpdate` wird ein mal pro Bild aufgerufen (Zeile 5). Sie ist zu vergleichen mit der `Update` Methode in einem `MonoBehaviour`. In der `OnUpdate` Methode wird der `ItemMoveJob` erstellt (Zeile 8 - 11). Dieser Job funktioniert mit dem `ItemComponent` und `LocalTransform` *Component* eines *Entities*. Das `LocalTransform` *Component* gibt die Position des *Entities* an. Das `ItemComponent` *Component* ist in Listing 15 definiert und speichert die Daten des Items. In dem Job wird das `LocalTransform` *Component* zum Lesen und Schreiben verwendet (erkennbar durch das Key-

wort `ref`) und das `ItemComponent` lediglich zum Lesen (erkennbar durch das Keywort `in`) (Zeile 20 - 21). Auch hier wird, wie in dem `MonoBehaviour`, nun die Position des Items mithilfe der `Lerp` Funktion von `Vector3` und den Daten im `ItemComponent` verändert (Zeile 24 - 26). Dieser Job, welcher die tatsächliche Logik für das Item enthält, wird in der `OnUpdate` Methode parallel gescheduled (Zeile 11). Dies ist hier speziell sehr vorteilhaft, da es sehr viele Items auf dem Spielfeld geben kann. Dadurch werden nicht tausende Items nacheinander bewegt, sondern alle parallel.
Die übergebene Position kommt, wie auch in dem objektorientierten Ansatz, von dem Förderband. Auch hier ist die Förderbandlogik in einem Job implementiert. Die Vorgehensweise, dass man die Items von hinten nach vorne ein Förderbandsegment weiterbewegt bleibt jedoch gleich. Ganz anders ist die Funktionsweise, wenn man Items erstellen will. Da man sich mit einem geschedulten Job nicht mehr auf dem *Main Thread* befindet, kann man Items nicht mehr direkt erstellen.

```
1  [BurstCompile(CompileSynchronously = true)]
2  [UpdateAfter(typeof(ProcessingBuildingSystem))]
3  public partial struct CreateItemSystem : ISystem {
4      //Lookup um Daten eines anderen Entity auszulesen
5      private ComponentLookup<InputConveyorComponent> inputLookup;
6
7      [BurstCompile(CompileSynchronously = true)]
8      public void OnCreate(ref SystemState state) {
9          //Für die OnUpdate Funktion wird das ItemEntitiesComponent gebraucht
10         //Darin sind alle Items für das Erstellen gespeichert
11         state.RequireForUpdate<ItemEntitiesComponent>();
12         inputLookup = state.GetComponentLookup<InputConveyorComponent>();
13     }
14
15     [BurstCompile(CompileSynchronously = true)]
16     public void OnUpdate(ref SystemState state) {
17         inputLookup.Update(ref state);
18         var ecbSingleton = SystemAPI.GetSingleton
19             <BeginSimulationEntityCommandBufferSystem.Singleton>();
20         //Entity Command Buffer wird erstellt
21         var ecb = ecbSingleton.CreateCommandBuffer(state.WorldUnmanaged);
22         var itemEntities = SystemAPI.GetSingleton<ItemEntitiesComponent>();
23         // Job wird erstellt und gescheduled
24         new CreateItemJob {
25             inputLookup = inputLookup,
26             ecb = ecb,
27             itemEntities = itemEntities
28         }.Schedule();
29         state.CompleteDependency();
30     }
31 }
```

Listing 17: Erstellung eines Items im datenorientierten Ansatz. Zur Erstellung eines Items aus einem Job, wird ein *Entity Command Buffer* verwendet.

Hier sieht man eine Besonderheit des *Entity Component Systems*, der *Entity Command Buffer* (*ECB*). Dadurch, dass strukturelle Änderungen nur auf dem *Main Thread* passieren dürfen, braucht man einen *ECB* um diese Änderungen zu sammeln und an späteren Stelle auf dem *Main Thread* auszuführen. Dafür wird ein *ECB* erstellt (Zeile 21) und dieser dem Job übergeben (Zeile 26). Listing 18 zeigt den Job, der diese strukturellen Änderungen vornimmt.

```
1  [BurstCompile(CompileSynchronously = true)]
2  [WithNone(typeof(OutputNotFoundTag))]
3  public partial struct CreateItemJob : IJobEntity {
4      public EntityCommandBuffer ecb;
5      //Wenn nur von gelesen wird kann ReadOnly verwendet werden
6      [ReadOnly] public ComponentLookup<InputConveyorComponent> inputLookup;
7      [ReadOnly] public ItemEntitiesComponent itemEntities;
8
9      [BurstCompile(CompileSynchronously = true)]
10     private void Execute(ref OutputProcessingBuildingComponent output) {
11         //Wenn die Ausgabe leer ist gibt es nichts zu tun
12         if(output.itemID == -1) return;
13         var itemID = output.itemID;
14         var input = inputLookup[output.outputEntity];
15         //Wenn die Eingabe des Förderbandes belegt ist wird auch keine
16         //Änderung vorgenommen
17         if(input.occupied || input.item != Entity.Null) return;
18         //Item wird aus der Ausgabe entfernt und mittels ECB
19         //wird ein neues Item erstellt
20         output.itemID = -1;
21         output.itemCreated = true;
22         var itemEntity = itemEntities.GetEntityWithID(itemID);
23         var item = ecb.Instantiate(itemEntity);
24         //Position und Item Component des Items werden gesetzt
25         ecb.SetComponent(item, LocalTransform.FromPositionRotationScale(
26             new float3(output.pos.x, output.pos.y, -0.5f),
27             quaternion.identity, 0.5f));
28         ecb.SetComponent(item,
29             new ItemComponent{pos = output.pos, itemID = itemID});
30         //Der Eingabe wird das Item zugewiesen
31         ecb.SetComponent(output.outputEntity,
32             new InputConveyorComponent {
33             item = item, pos = input.pos, occupied = true});
34     }
35 }
```

Listing 18: Job in dem ein *ECB* verwendet wird. Der *ECB* sammelt die Änderungen aus dem Job und spielt sie nach dem Job wieder ab.

Der *ECB* erstellt das Item (Zeile 23) und ändert noch in den *Components* die Position des Items (Zeile 25 - 29). Zusätzlich wird der Eingabe des Förderbandes das erstellte Item zugewiesen (Zeile 31 - 33). Diese Änderungen werden in dem Job aufgenommen und nach dem Job in der richtigen Reihenfolge angewendet.

5 Benchmark

Um die Performance von dem in Kapitel 4 vorgestellten Videospiel zu testen, wird dieses gebenchmarkt. Ein Benchmark ist ein Maßstab um Leistungen zu vergleichen. Hier wird das datenorientierte Spiel (im Folgenden *ECS* genannt) mit dem objektorientierten Spiel (im Folgenden Mono genannt) verglichen. Der Unterschied in der Leistungsfähigkeit zwischen *ECS* und Mono ist besonders interessant, weshalb in diesem Benchmark die Zeit gemessen wird, die ein Bild zur Berechnung gebraucht hat. Eine andere, oft verwendete Einheit, sind die durchschnittlichen Bilder pro Sekunde (fps).
In diesem Kapitel wird nun der direkte Vergleich von *ECS* und Mono angestellt. Dazu stellt Unity einige hilfreiche Tools zu Verfügung, um Aspekte in einem Spiel zu messen. Auch kann dadurch gut nachvollzogen werden, wie lange eine Aufgabe, wie zum Beispiel die Ausführung der `Update` Funktionen, benötigt hat. Bei den nachfolgenden Tests wird jeweils das implementierte Spiel aus Kapitel 4 verwendet. Bei dem Benchmark werden immer 2000 Bilder aufgezeichnet. Sobald ein Stahlbarren hinten in dem Kasten angekommen ist (Abbildung 5), wird das Spiel pausiert. Die letzten 2000 Bilder werden dann zur Auswertung verwendet. Zusätzlich werden mehrere Durchläufe gemessen. Angefangen wird mit 1, 10 und 100 Fabriken. In 100er Schritten wird dann die Anzahl der Fabriken erhöht, bis zu 1000 Fabriken.
Die Benchmark Tests werden auf einer Maschine mit einer NVIDIA GeForce RTX 3070, einem Intel Core i7 12700 und 32 Gb DDR 4 RAM ausgeführt. Die Tests liefen außerdem auf einem Windows 11 Betriebssystem.

5.1 *Profiler*

Der Unity *Profiler* ist ein Tool, mit dem man Performance Informationen über sein Spiel bekommt. Mit dem *Profiler* kann man sein Spiel aufzeichnen und Daten über eine festgelegte Anzahl von Bildern sammeln. Dazu gehören die CPU Auslastung, der Speicherverbrauch, wie viele Objekte gezeichnet werden und noch vieles mehr. Während das Spiel ausgeführt wird, werden diese Daten live angezeigt. Zusätzlich lassen sich die Daten auch beim Pausieren des Spiel abspeichern und andere vorher gespeicherte Daten wieder laden um sie zu betrachten. Abbildung 7 zeigt den *Profiler* mit schon aufgezeichnete Daten.

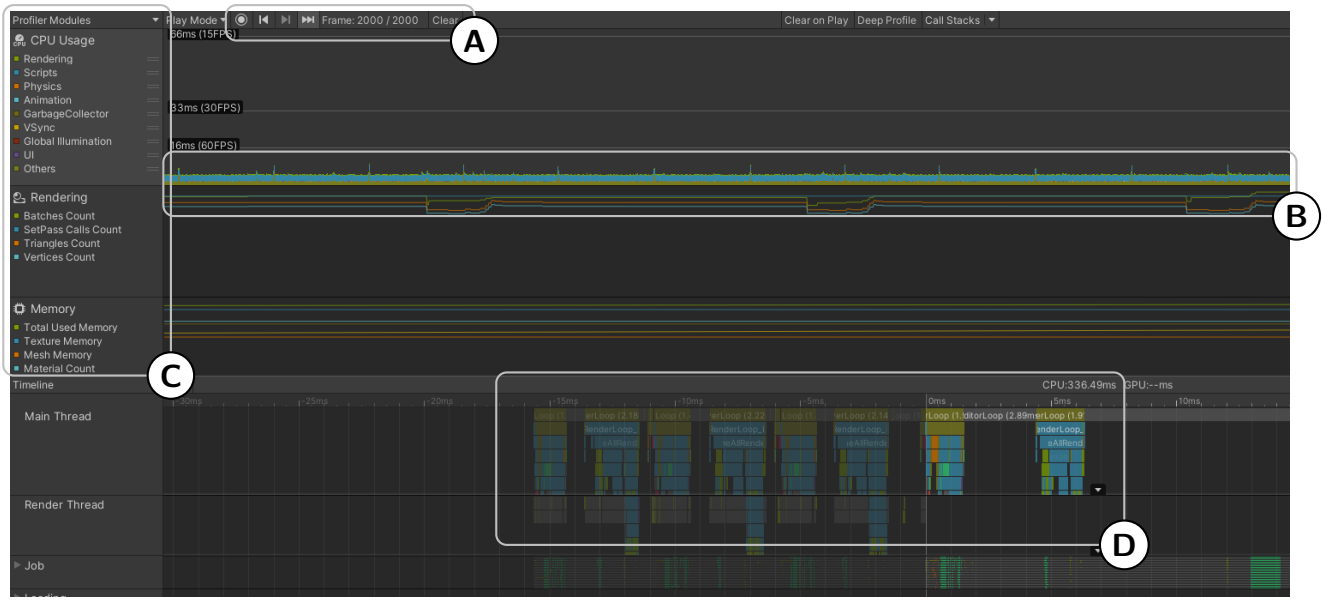

Abbildung 7: Der *Profiler* im Unity Editor. Mit ihm lassen sich die Auslastung des Spiels während des Spielens aufzeichnen und die Daten zur weiteren Verwendung abspeichern. In der oberen Leiste (A), lässt sich das Spiel wie gewünscht aufzeichnen und man sieht die Anzahl an aufgezeichneten Bildern. Man kann auch einzelne Bilder zur weiteren Inspektion auswählen. Darunter (B) wird der Verlauf der aufgezeichneten Bilder dargestellt. Hier sieht man für jedes Modul eine Auslastung über den Verlauf der 2000 Bilder. Links (C) werden die einzelnen Module angezeigt. Sichtbar sind hier die Module CPU Usage, Rendering und Memory, wobei man für weitere im Editor nach unten scrollen muss. Unten (D) ist eine genauere Timeline sichtbar, da hier nur einige Bilder abgebildet werden.

In der Abbildung 7 erkennt man die einzelnen Auslastungen und ihre Kategorien. Zusätzlich kann man ein bestimmtes Bild auswählen und erhält dazu weitere Details in der Timeline weiter unten. Diese aufgezeichneten Daten lassen sich dann in dem *Profile Analyzer* weiter analysieren.

5.2 *Profile Analyzer*

Der *Profile Analyzer* dient der weiteren Analyse von aufgenommenen Profilen mit dem *Profiler*. Zusätzlich hat man mit dem *Profile Analyzer* die Möglichkeit zwei Profile direkt miteinander zu vergleichen. Abbildung 8 zeigt den Vergleich zweier Profile.

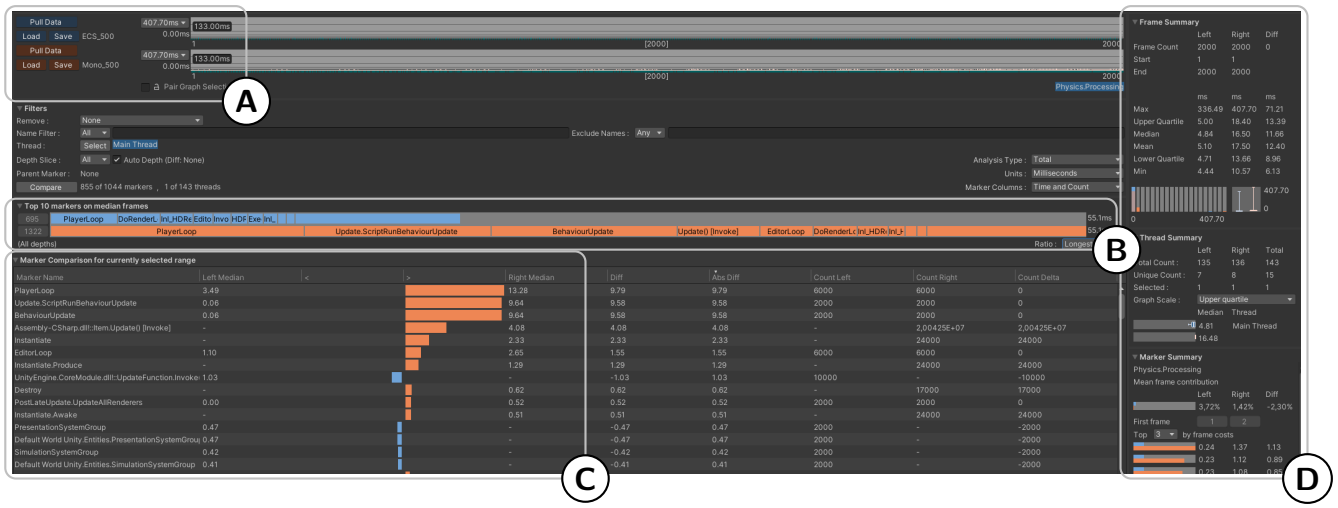

Abbildung 8: Der *Profile Analyzer* im Unity Editor. Es werden die Mono- und die *ECS* Messdaten des Spiels miteinander verglichen. Links oben (A) erkennt man, welche Daten man gerade vergleicht und sieht dazu eine Timeline beider Daten. Die *ECS* Daten sind im Bild blau dargestellt, die Mono Daten rot. In der Mitte des Bildes (B) hat man den Vergleich des Median Bildes. Deutlich zu erkennen ist die wesentlich kürzere Laufzeit des *ECS* Bildes. In dem unteren Teil (C) werden wichtige Spielfunktionen im Vergleich dargestellt. Beispielsweise ist ganz oben der *PlayerLoop* (gesamte Berechnungszeit eines Bildes). Dazu erkennt man an dem Balken, welches der Spiele eine höhere Berechnungszeit hatte. Rechts (D) gibt es zusätzlich noch eine Zusammenfassung über die gesamte Anzahl an Bildern und Threads in beiden Spielen.

Oberhalb erkennt man die zu vergleichenden Daten und wie viele Bilder die beiden Daten enthalten. Hier ist erkennbar, dass einmal die Daten des *ECS* mit 500 Fabriken (blau dargestellt) und des Mono's mit 500 Fabriken (rot dargestellt) verglichen werden. Die wichtigen Teile sind hierbei die zusammenfassende Übersicht (rechts), welche eine Übersicht über alle Bilder oder Threads bietet, und den Vergleich von Kernfunktionen der Spiele, wie beispielsweise der `Update` Funktionen und dem Erstellen, oder Zerstören von Objekten. Das Problem hierbei ist jedoch, dass manche der Kernfunktionen von dem objektorientierten Ansatz anders heißen als die vom datenorientierten Ansatz und man sie somit schlechter gegenüberstellen kann. Dennoch kann man die Zusammenfassung der Bilder gut nutzen, um die Spiele zu vergleichen.

5.3 Vergleich

Abbildung 9 zeigt den direkten Vergleich der Laufzeiten von Monobehaviour zu dem *ECS*. Dabei ist Mono rot und *ECS* blau markiert.

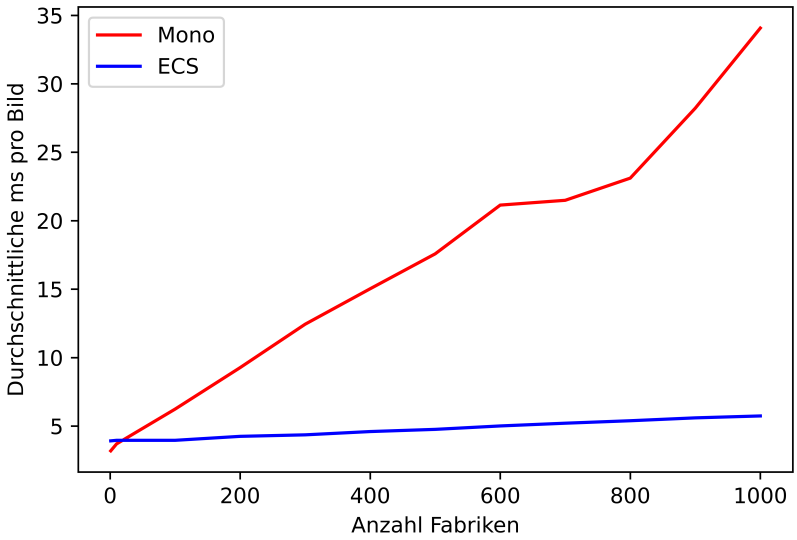

Abbildung 9: Der Vergleich von Mono und dem *Entity Component System*. Es werden die durchschnittlichen Millisekunden, die ein Bild zur Berechnung gebraucht hat, für verschiedene Anzahlen von Fabriken, angezeigt.

	ECS			Mono		
Anzahl Fabriken	Min	Mean	Max	Min	Mean	Max
1	2.42 ms	3.92 ms	10.94 ms	2.77 ms	3.19 ms	8.65 ms
10	2.46 ms	3.96 ms	10.47 ms	3.0 ms	3.7 ms	9.33 ms
100	3.57 ms	3.96 ms	9.03 ms	5.27 ms	6.24 ms	13.08 ms
200	2.76 ms	4.25 ms	10.87 ms	7.87 ms	9.27 ms	19.16 ms
300	3.94 ms	4.36 ms	9.37 ms	10.44 ms	12.44 ms	25.37 ms
400	4.18 ms	4.6 ms	9.82 ms	10.61 ms	15.03 ms	33.69 ms
500	4.3 ms	4.76 ms	9.81 ms	11.12 ms	17.58 ms	39.04 ms
600	4.52 ms	5.01 ms	10.56 ms	10.66 ms	21.14 ms	61.65 ms
700	4.48 ms	5.21 ms	10.16 ms	9.33 ms	21.49 ms	53.76 ms
800	4.48 ms	5.39 ms	10.89 ms	8.51 ms	23.11 ms	65.15 ms
900	4.35 ms	5.6 ms	10.6 ms	8.86 ms	28.22 ms	76.37 ms
1000	4.6 ms	5.74 ms	10.99 ms	9.65 ms	34.07 ms	90.31 ms

Tabelle 1: Die genauen Daten des Benchmarks. Es werden das Minimum, Maximum und die durchschnittlichen Berechnungszeiten für alle Fabrikanzahlen gezeigt.

Ganz deutlich erkennbar schneidet Mono wesentlich schlechter ab. Mono braucht deutlich länger um ein Bild zu verarbeiten, vor allem wenn sich die Anzahl der Fabriken erhöht. Dies bestätigt die Annahme, dass das *ECS* wesentlich besser darin ist große Mengen an Objekten zu verarbeiten. Auch hier steigt die durchschnittliche Zeit ein Bild zu verarbeiten, jedoch nur sehr gering. Eine Ausnahme bildet hier jedoch eine sehr geringe Anzahl an Fabriken. Vor allem bei einer Fabrik ist Mono schneller als das *ECS*. Tabelle 1 zeigt dazu genauere Werte. Das liegt an dem Overhead den das *ECS* mit sich bringt. Für eine geringe Anzahl an *Entities* ist es ein großer Overhead, Jobs zu erstellen und diese auf mehreren *Threads* auszuführen. Hier wäre es schneller die Aktionen direkt auf dem *Main Thread* auszuführen, statt einen Job dafür zu schedulen.

6 Fazit

Zusammenfassend lässt sich sagen, dass die Datenorientierte Programmierung großes Potenzial in der Spieleentwicklung hat, insbesondere bei Videospielen mit vielen Objekten in der Spielwelt. Das *Entity Component System* hat sich in der Auswertung als deutlich überlegen gegenüber dem Monobehaviour herausgestellt. Insbesondere bietet die Cache Freundlichkeit des datenorientierten Ansatzes eine Möglichkeit, die Daten zu verarbeiten, sodass weniger Cache-Misses entstehen. Dadurch wird eine schnellere Verarbeitung ermöglicht. Durch das Zusammenspiel von Daten und Systemen wird zudem eine effiziente Parallelisierung ermöglicht, wodurch Prozessoren mit mehreren Kernen effizienter genutzt werden können.
Es ist jedoch wichtig zu betonen, dass die Vorteile der Datenorientierten Programmierung stark von der spezifischen Anwendung abhängen. Bei Spielen mit insgesamt wenig Objekten bietet ein datenorientierter Ansatz möglicherweise nicht mehr Performance.
Auch ist die Implementierung des *Entity Component Systems* von Unity derzeit noch aufwändig und mit einem gewissen Overhead bei der Programmierung verbunden, insbesondere bei der Erstellung von Objekten zur Laufzeit. Es ist jedoch zu erwarten, dass Unity in Zukunft die Umsetzung des *ECS* vereinfachen wird.
Die Nutzung von Unity's Jobs mit mehreren Threads bringt ebenfalls einen Overhead mit sich. Dadurch muss man abwägen, ab wann sich Jobs, insbesondere parallelisierte Jobs, lohnen. Bei einer hohen Anzahl an *Entities* mit wenig Abhängigkeiten ist das Job System sehr vorteilhaft, auch weil es sehr einfach umzusetzen ist.
Die Nutzung des Burst Compilers ist immer vorteilhaft, jedoch ist er nur mit unverwalteten Daten und einem beschränkten C# Set kompatibel. Dadurch ist es schwierig so viele Funktionalitäten wie möglich kompatibel zu machen. Wenn jedoch eine Funktionalität mit Burst kompatibel ist, bietet der Burst Compiler nur Vorteile.
Insgesamt kann Datenorientierte Programmierung in der Spieleentwicklung eine leistungsstarke Alternative zu der Objektorientierten Programmierung sein. Wie man in der Auswertung gesehen hat, kann bei geeigneten Anwendungen eine erheblich erhöhte Performance möglich sein.

Literaturverzeichnis

[1] Richard Fabian. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*. DataOrientedDesign.com, 2018. https://www.dataorienteddesign.com/dodbook/ (zuletzt besucht am 28.08.2023).

[2] Petra Fröhlich. Umsatz-vergleich 2021: Games vor musik, kino und bundesliga. *www.gameswirtschaft.de*, 2021. https://www.gameswirtschaft.de/wirtschaft/umsatz-vergleich-games-film-musik-2021/ (zuletzt besucht am 28.08.2023).

[3] Noel Llopis. Data-oriented design (or why you might be shooting yourself in the foot with oop). *www.gamedeveloper.com*, 2009. https://gamesfromwithin.com/data-oriented-design (zuletzt besucht am 28.08.2023).

[4] Aras Pranckevičius. Entity component systems & data oriented design, 2018. https://aras-p.info/texts/files/2018Academy%20-%20ECS-DoD.pdf (zuletzt besucht am 28.08.2023).

[5] Unity. Ecs track: Deep dive into the burst compiler - unite la, 2018. https://youtu.be/QkM6zEGFhDY?t=1633 (zuletzt besucht am 28.08.2023).

[6] Florian Zandt. Videospiele sind das lukrativste unterhaltungsmedium. *de.statista.com*, 2022. https://de.statista.com/infografik/28970/geschaetzter-weltweiter-jahresumsatz-mit-videospielen-buechern-film-serie-musik/ (zuletzt besucht am 28.08.2023).

Abbildungsverzeichnis

Listingverzeichnis

Selbstständigkeitserklärung

Ich, Dennis Untiet, erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.
Seitens des Verfassers bestehen keine Einwände die vorliegende Bachelorarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jena, 30. August 2023