FH University of Applied Sciences TECHNIKUM WIEN

Bachelorarbeit

Zur Erlangung des akademischen Grades
„Bachelor of Science in Engineering“
im Studiengang Informatik/Computer Science

Schwächen von Game Engines

Ausgeführt von: Stefan Alfons
Personenkennzeichen: 1810257115

BegutachterIn: Dipl.-Ing. Dr. Gerd Hesina

Wien, 2022

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idgF sowie Satzungsteil Studienrechtliche Bestimmungen
/ Prüfungsordnung der FH Technikum Wien idgF).
Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).
Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Wien, 30.November 2022 Unterschrift

Abstract

In dieser Arbeit wird untersucht, welches Verbesserungspotential moderne Game Engines besitzen beziehungsweise in welchen Bereichen es nötig ist, als Entwicklungsteam selbst Lösungen zu entwerfen. Moderne Game Engines sind starke und vielseitige Werkzeuge, die es Menschen erlauben, mit relativ wenig Aufwand ein fertiges Videospiel zu kreieren. Jedoch ist es dennoch ein Leichtes, in eine Falle zu tappen und ein Produkt herauszubekommen, bei dem die Performance beeinträchtigt wird. In dieser Arbeit werden einige dieser Fallen diskutiert, und an einem praktischen Beispiel gezeigt, dass für manche Art von dreidimensionaler Software es dennoch nicht reicht, Game Engines zu verwenden.

This study discusses possible improvements of modern game engines in the context of performance. It also discusses various areas in which it might be more advisable for developers to roll their own solutions. Modern game engines are powerful and versatile tools, which allow humans to build a completed video game while requiring a relatively low amount of effort. Regardless, there are a lot of traps developers might step into, that might compromise the performance of the final product. This study goes over some of these traps, and also shows, in a small case study, that it can even be necessary to develop separate components on your own for certain kinds of three-dimensional software.

Danksagung

Ich möchte hiermit Dipl.-Ing. Dr. Gerd Hesina herzlichst für seine nette und fachlich kompetente Betreuung bedanken. Außerdem geht mein Dank an sehr viele Menschen, die mir nahestehen – Familie, Freunde und Bekannte, die mir sowohl emotional als auch fachlich tatkräftig zur Seite standen, und ohne die ich es nicht geschafft hätte.

FH University of Applied Sciences TECHNIKUM WIEN

Inhaltsverzeichnis

1 Einleitung

Die Videospielindustrie ist ein milliardenschwerer Bereich der Softwareindustrie. Um die Kosten und Entwicklungszeiten von Videospielen im Zaum halten, war ein logischer Schritt, konzeptuell immer wiederkehrende Probleme nicht für jedes Projekt separat zu lösen. Unter diese wiederkehrenden Probleme fällt unter anderem die Darstellung von dreidimensionalen Objekten auf einem Bildschirm, aber auch die Wiedergabe von Ton sowie der Verarbeitung der Signale von Eingabegeräten. Nicht zuletzt ist es auch immer häufiger eine Voraussetzung, dass mehrere Spieler an verschiedenen Endgeräten an verschiedenen Orten in ein und derselben Spielwelt miteinander agieren sollen. Da diese Aufgaben von jedem einzelnen Spiel wieder und wieder erfüllt werden müssen, wurden nach und nach Softwarepakte entwickelt, die diese Aufgaben so allgemein wie möglich lösen.

Aus diesen Softwarepaketen hat sich das entwickelt, was heutzutage als Game Engine bekannt ist. Diese Game Engines wurden genau dafür konzipiert, diese immer wiederkehrenden Probleme effektiv und möglichst allgemein zu lösen. Mit der Zeit erhielten diese Softwarepakete mehr und mehr Features. Mittlerweile sind diese Engines schon so weit entwickelt, dass es selbst für Menschen mit kaum bis keiner Programmiererfahrung möglich ist, ein funktionales Spiel zu produzieren.

Gerade aus diesem Grund stellt sich die Frage, wo die Schwächen dieser Game Engines liegen, und welche Fallen es beim Verwenden dieser gibt.

1.1 Motivation

Diese Arbeit versucht einen kleinen Einblick darüber zu geben, wo sich möglicherweise Schwächen von Game Engines – in dieser Arbeit am konkreten Beispiel der Unity Game Engine - befinden, und in welchen Anwendungsgebieten es doch überlegenswert wäre, andere, separate Software zu verwenden – oder sogar selbst zu schreiben.

1.2 Methodik

Im Rahmen dieser Arbeit wurde einerseits die Methode der Literaturrecherche angewandt. Da es den Rahmen dieser Arbeit sprengen würde, auf alle möglichen Schwächen und Fehler einzugehen, wurden hier ein paar wenige Beispiele exemplarisch hergenommen, um die Schwächen von Game Engines – beziehungsweise die Grenzen ihrer Anwendbarkeit – darzustellen. Es wurde jeweils ein kleines Spiel sowohl ohne, als auch mit Game Engine gebaut – und dann wurden diese unter Beachtung von verschiedenen Parametern verglichen.

2 Game Engines

Ein Videospiel ist definiert als „ein Spiel, das durch elektronisches Manipulieren von Bildern, die von einem Computerprogramm oder anderem Bildschirm produziert werden, gespielt wird“ [1]. Historisch gesehen waren diese Computerprogramme in sich abgeschlossene Konstrukte, die nur den Zweck erfüllt haben, ein einziges Spiel darzustellen. Um nicht für jedes Spiel – oder auch andere dreidimensional darstellende Software – das metaphorische Rad neu zu erfinden, stellte sich im Laufe der Geschichte heraus, dass es effizienter ist, gewisse Teile eines Spiels – spezifisch diese Teile, die Probleme lösen, die von jedem Spiel gelöst werden müssen – so zu gestalten, dass diese auch für andere Videospiele wiederverwendet werden können.

Eines dieser Probleme ist die Darstellung dreidimensionaler Objekte auf einem zweidimensionalen Bildschirm. Obwohl es mit Videospielen möglich ist, die verschiedensten Szenarien und Fantasien darzustellen, ist der Mensch in seiner Vorstellungskraft doch an die drei Dimensionen der echten Welt gebunden. So läuft es darauf hinaus, dass sich ein Großteil der Videospiele in eben diesen drei Dimensionen abspielt.

Damit dieses und ähnliche Probleme nicht immer von einem weißen Blatt Papier neu gelöst werden müssen, hat sich in der Industrie der Software-Entwicklung das Prinzip bewährt, Teile von Quellcode so allgemein wie möglich zu konzipieren, und diese dann in so genannte Bibliotheken zu verpacken. Diese Bibliotheken können dann von anderen Software-Entwicklern benutzt werden, um effizienter an ihren eigenen Projekten zu arbeiten, und nicht für gut gelöste Probleme unnötigerweise viel Zeit aufbrauchen müssen.

Durch die enorme Größe der Spieleindustrie und der immer größer werdenden Komplexität von modernen Videospielen wird immer mehr Zeit und Energie in Entwicklung dieser Game Engines investiert. Dies führte dazu, dass moderne Exemplare dieser Software leistungsstark und vielseitig geworden sind. Diese modernen Game Engines sind nicht nur in der Lage diese besagten, immer wiederkehrenden Probleme effizient und sauber zu lösen, sondern bieten mittlerweile auch mächtige Werkzeuge an, um im Rahmen dieser Game Engines komplette Spiele zu produzieren – in manchen Fällen ist dies sogar ohne Programmiertätigkeit möglich. [2]

Ein Problem, dass die immer größer werdende Komplexität von modernen Videospielen mit sich bringt ist – neben den immer größer werdenden Entwicklungskosten – dass das technische Verständnis von Menschen, die an Spielen arbeiten, immer breiter und größer werden muss. Da allerdings, wie erwähnt, Game Engines immer bessere und besser zu bedienende Werkzeuge beinhalten, wird es mittlerweile auch Menschen das Arbeiten an Videospielen ermöglicht, die nicht die Begeisterung oder das Interesse für die technischen Aufgaben – wie zum Beispiel Programmieren – mit sich bringen.

Ein großer Vorteil von modernen Game Engines zeigt sich auch dadurch, dass die Videospiele, die mit diesen Game Engines produziert werden, mit sehr vielen Plattformen kompatibel sind. Historisch gesehen war es üblich, sehr viel Plattformspezifischen Code für jede Zielplattform schreiben zu müssen, da sich die Architektur zwischen zum Beispiel eines Heimcomputers mit Windows als Betriebssystem und den verschiedenen Spielekonsolen doch sehr stark unterscheiden. Durch die Art und Weise wie moderne Game Engines aufgebaut sind, fällt dieser Entwicklungsschritt weg, und ermöglicht es den Entwicklern mit wenig Aufwand ihr Produkt an eine größere Zielgruppe verkaufen zu können.

2.1 Was ist eine Game Engine?

Laut Oxford Dictionary ist eine Game Engine die „grundlegende Software eines Computer- oder Videospiels.“ Dadurch, dass historisch gesehen, Videospiele meist ein kohärentes Stück Software waren, stellt sich die Frage: Wo hört ein Spiel auf und wo fängt eine Game Engine an?

Jason Gregory beantwortet diese Frage folgendermaßen [3]: „Datengesteurte Architektur ist wohl der Unterschied zwischen einer Game Engine, und einem Stück Software, das ein [Video]spiel aber keine Game Engine ist.“ Er geht weiters darauf ein, dass Software dann in die Kategorie Game Engine fällt, wenn es ein leichtes ist, diese Software für andere, möglichst viele verschiedene Videospiele wiederzuverwenden.

2.2 Geschichte von Game Engines

2.2.1 Doom

Historisch gesehen war es üblich, ein Spiel in der Hinsicht zu entwickeln, sodass das gesamte Spiel als eigenständiges Softwarepaket verpackt war. Eines der ersten Spiele, das mit dem Gedanken entwickelt wurde, die Software der Kernkomponenten (wie zum Beispiel das dreidimensionale darstellen von Inhalten auf dem Bildschirm sowie das Audio-system oder die Kollisionsabfrage) eindeutig von dem Teil zu trennen, der unter anderem die Texturen und Modelle sowie die Spiellogik selbst beinhält, war id Software’s Doom.[3]

Diese Trennung ermöglichte es anderen Entwicklern durch die Lizensierung der Engine von Doom die Kernkomponenten für ihre eigenen Projekte zu verwenden. Diese Entwickler mussten dann nur noch Modelle, Texturen und Level erstellen, und ersparten sich somit Entwicklungszeit und -kosten. Auf der anderen Seite stellte sich diese Lizensierung für id Software auch als nützliche Nebeneinnahmequelle heraus. Auf der anderen Seite wurde für den privaten Gebrauch auch Werkzeuge veröffentlicht, die von den Entwicklern selbst verwendet wurden. Dies führte zum Beginn der „Modding Community“ – eine Ansammlung an Leuten, die in ihrer Freizeit Videospiele modifizierten und es dadurch schafften, noch mehr Spielstunden aus den schon beliebten Videospielen zu extrahieren.

2.3 Unity

2005 wurde die Unity Game Engine veröffentlicht. Anfangs nur für Mac verfügbar, wurde die Engine jedoch ein Jahr später auch für Windows und verschiedene Internet-Browser veröffentlicht. Über die Jahre sind immer mehr Features und unterstütze Plattformen hinzugefügt worden.

Unity bietet Entwicklern die Möglichkeit, relativ einfach mit Hilfe des UnityEditors Welten (sogenannte Szenen) zu erstellen. In diese Szenen können Entwickler dann Objekte platzieren, und mit relativ wenig Aufwand direkt anfangen, Game-Logik zu implementieren. Unity übernimmt sehr viele der Schritte, die bei der Entwicklung eines Spiels vom leeren Blatt Papier sehr viel Entwicklerzeit und damit Geld kosten würden. Unter diese Schritte fallen unter anderem das Management jeglicher Objekte in der Spielwelt, die visuelle aber auch auditive Darstellung dieser, die Interaktionen zwischen diesen Objekten aber auch das Management von dem Speicher. Da diese Aufgaben wie erwähnt für jedes Videospiel gelöst werden müssen, aber die Möglichkeit besteht, diese Aufgaben sehr allgemein zu lösen, ergibt es Sinn, an einer Game Engine zu arbeiten, die es Entwicklern dann immens erleichtert, die Aufgaben zu lösen, die die Software, die von diesen Entwicklern produziert wird, spezifisch lösen muss.

Diese Game Engine bietet außerdem die Möglichkeit, Scripts in C# sowie in JavaScript zu verwenden, um Game-Logik zu implementieren. Dies erlaubt es dem Programmierer, gewisse Vorteile dieser Programmiersprachen zu benutzen, wie zum Beispiel automatisiertes Memory Management oder aber auch schon existierende Softwarebibliotheken als Plug-Ins zu verwenden. Dadurch dass aber Unity selbst in C bzw. C++ geschrieben ist, kann die Game Engine für ihre interne Funktionsweise die Performancevorteile dieser nativen Sprachen nutzen.

2.4 Kernkomponenten

2.4.1 Renderer

Zu einer der Hauptaufgaben von Game Engines zählt die Darstellung von dreidimensionalen Objekten auf einem zweidimensionalen Bildschirm. Im übertragenen Sinne handelt es sich bei den menschlichen Augen allerdings auch nur um Leinwände, auf die Licht trifft. Aus dieser Wechselwirkung des Mechanismus der Augen und dem einfallenden Licht kann das Gehirn nun Informationen extrahieren.

Da es für die dreidimensionale Darstellung in der Computergrafik genauso notwendig ist, Informationen aus einer modellierten, dreidimensionalen Welt auf eine Art Leinwand zu projizieren, bietet es sich an, sich diesen natürlichen Prozess als Vorbild für diese Darstellung herzunehmen. So besteht nun üblicherweise die Darstellung einer solchen Szene aus einer Kamera, auf die Licht projiziert wird, verschiedenen Lichtquellen, die Licht aussenden und Objekten, die über ihre Oberflächen mit diesem Licht interagieren. [1] Diesen Oberflächen werden bestimmte Effekte zugeordnet, die beschreiben, wie sich dieses Licht verhält, wenn es auf diese Oberfläche trifft. Dieses Licht kann nun entweder absorbiert, reflektiert, transmittiert oder abgelenkt werden.

Bestimmte Oberflächen absorbieren meist nur einen gewissen Wellenlängenbereich, während der Rest reflektiert wird. Fällt zum Beispiel ein weißer Lichtstrahl auf ein rotes Objekt, werden alle Wellenlängen absorbiert – außer die roten. Dies hat zur Folge, dass dieser Gegenstand dann für das menschliche Auge rot erscheint. Wellenlängen, die reflektiert werden, können dies auf unterschiedliche Arten tun; werden diese diffus reflektiert, wird ein eintreffender Strahl gleichmäßig in alle Richtungen wieder weg von der Oberfläche gestreut. Werden diese allerdings spiegelnd reflektiert, bedeutet das, das ein eintreffender Strahl entweder direkt wieder zurückreflektiert wird, oder dies zumindest in einem engen Kegel passiert. Weiters können Reflektionen auch anisotrop passieren, was zur Folge hat, dass die Wahrnehmung dieser Reflektion vom Betrachtungswinkel des Beobachters abhängt.

Da diese Oberflächen maßgebend für das letztendliche Aussehen dieser Szene ist, wird die Geometrie der darzustellenden Objekte durch deren Oberflächen und die Eigenschaften dieser modelliert. Diese Oberflächen werden üblicherweise als sogenannte Vertices dargestellt. Da sich diese Vertices immer an einem diskreten Punkt im Raum befinden, bieten diese eine praktische Möglichkeit um auch Eigenschaften der angrenzenden Oberfläche in der jeweiligen Datenstruktur zu speichern.

Damit ein Videospiel für Menschen flüssig erscheint, ist es nötig, dass zumindest 30 – im Idealfall 60 oder mehr – fertige Bilder pro Sekunde am den Bildschirm erreichen. Dies bedeutet, dass der Hardware im Durchschnitt nur 33ms Zeit zur Verfügung gestellt werden, um jeweils ein Bild – ein „Frame“ – zu berechnen. Außerdem ist es für eine ruckelfreie und flüssige Erfahrung wichtig, dass die Zeit, die zwischen zwei verschiedenen angezeigten Bildern vergeht, relativ konstant ist. Ist diese Konstanz der Zeit zwischen Frames nicht gegeben, kann es bei manchen Spielern zu Unwohlsein bis hin zur Übelkeit kommen.

Ursprünglich wurden all diese Berechnungen direkt auf der CPU linear hintereinander durchgeführt. Jedoch wurde mit der Zeit Hardware immer besser, und die Idee wurde geboren, die verschiedenen Schritte der Render-Pipeline zu parallelisieren. So entstanden spezialisierte Chipsätze, die es ermöglichten sehr viele Dreiecke in sehr kurzer Zeit auf den Bildschirm zu bringen.

2.4.2 Physik

Selbst Computerspiele haben im meisten Fall einen Bezug zur realen Welt. Selbst wenn absoluter Realismus nicht das Ziel einer Simulation oder eines Computerspiels darstellt, ist es dennoch meist nötig bestimmte Interaktionen zwischen verschiedenen Objekten zu beschreiben.

Ist aber das tatsächliche Ziel des Spiels oder der Simulation, Situation so darzustellen, wie sie auch in der echten Welt passieren würden, müssen bei der Entwicklung dieser Simulation einige grundlegende Konzepte bedacht werden. Um überhaupt auch nur annähernd eine Spiegelung der Realität zu erreichen muss zunächst die Realität in einer Form beschrieben werden, in der es möglich ist, eben diese Situation virtuell darzustellen. Schon bei diesem Schritt müssen Abstriche an Realismus gemacht werden, da jede Beschreibung von physikalischen Gegebenheiten Vereinfachungen erhält – aus dem simplen Grund, dass es unmöglich ist, physikalische Gegebenheiten absolut akkurat mathematisch zu beschreiben.

Selbst wenn es nun gelingt, Interaktionen zwischen Objekten vollkommen vorauszusagen, ist dies meist nicht auf eine Art möglich, die es heutigen Rechnern erlaubt, diese Interaktionen in Echtzeit zu berechnen und danach darzustellen. Da es bei Videospielen aber unabdingbar ist, Ergebnisse von Interaktionen mit dem Spieler und der Spielwelt in Echtzeit zu erreichen, ist in diesem Belangen eine Gradwanderung zwischen Realismus und Performance nötig – je nach gewollter Genauigkeit.

2.4.3 Audio

Um Spielern von Videospielen einerseits ein Erlebnis zu bieten, das möglichst viele Sinne beinhaltet, aber andererseits auch um diesen Spielern so viele Informationen wie möglich zu liefern, verwenden Videospiele viele verschiedene Arten von Geräuschen. Auch wird sehr oft Hintergrundmusik verwendet, um das Spielerlebnis abzurunden.

Durch die große Anzahl der möglichen physikalischen Interaktionen von Objekten miteinander, aber auch den schieren Umfang von Dialog sowie Umgebungsgeräuschen, kann es auch dazu führen, dass unglückliches Management der Audiodaten Performanceprobleme verursacht.

Jeder Audio-Effekt der im Spiel abgespielt wird, muss in den Speicher geladen werden. Würde man aber nun als Entwickler entscheiden, jede Audiodatei für die ganze Dauer des Spiels im Speicher zu lassen, wäre in diesem Speicher kein Platz mehr für andere Dinge, wie zum Beispiel die grafische Darstellung von Objekten, aber auch Daten, die für die Spiellogik notwendig sind. Der Zugriff auf die Festplatte stellt sich üblicherweise aber als der eine Schritt heraus, der am meisten Zeit in Anspruch nimmt. Demnach wollen die Entwickler sichergehen, dass Daten zwar nur so lange wie nötig im Speicher bleiben, aber es dennoch vermieden wird, Daten immer wieder neu von der Festplatte in den Speicher zu laden.

Um dieses Problem zu umgehen, wird in manchen Videospielen auf eine Technik zurückgegriffen, die es den Entwicklern erlaubt, nur eine Audiodatei in den Speicher zu laden, aber dennoch verschiedene Effekte damit zu hinterlegen. Durch geschicktes Verstellen von Abspielgeschwindigkeit sowie Tonhöhe und Frequenz lassen sich einzelne Audio-Dateien für verschiedenste Zwecke wiederverwenden – und das, ohne diese mehrfach in den Speicher zu laden.

2.4.4 Netzwerk

Es liegt in der Natur des Menschen, sich mit anderen Menschen zu messen. Aus diesem Grund, und dem Grundbedürfnis der Menschen nach sozialer Interaktion, hat sich – sobald die technischen Möglichkeiten dafür gegeben waren – ziemlich schnell etabliert, dass Videospiele oft mit Modi geliefert werden, in denen sich verschiedene Spieler entweder an einander messen können, oder – je nach Spielmodus und Spiel – miteinander versuchen können, verschiedene Ziele zu erreichen. Mit der Zeit und der Technologie haben sich diese Modi so entwickelt, dass es nun nicht nur möglich ist, miteinander zu spielen, wenn man sich im selben Raum, an derselben Konsole befindet, sondern heute ist es auch möglich, Spiele gleichzeitig zu spielen, obwohl man sich tausende von Kilometern entfernt voneinander aufhält.

Damit diese Aktivität sowohl Spaß macht, als auch überhaupt durchführbar ist, ist es notwendig, die Verzögerung zwischen den Spielern so gering wie möglich zu halten. Auch ist es von Nöten, den Spielverlauf an allen Enden miteinander konstant synchronisiert zu halten, sodass es nicht zu unvorhersehbaren Ereignissen oder im schlimmsten Fall zu Desynchronisierungen kommt.

Schon 2006 erkannten Claypool und Claypool die Wichtigkeit einer geringen Verzögerung von Videospielen im Netzwerk [3].

3 Case Study (praktischer Teil)

3.1 OpenGL

Um ein Spiel ohne Game Engine zu programmieren, wird in der Regel jedoch dennoch zumindest eine Grafik-API verwendet. Wie schon erwähnt, erlauben es Game Engines über eine Abstraktsionsschicht mit ein und demselben Programm, mehrere verschiedene Grafik-APIs anzusprechen. Dies erlaubt einen Level an Portabilität, der ohne einer Game Engine nicht so nachproduziert werden kann.

Für die Beispiele in dieser Studie wurde auf die offene Grafik-API OpenGL zurückgegriffen. Als Leitfaden wurde ein (inoffizielles) OpenGL-Tutorial [11] verwendet – diese bietet einen leicht zu folgenden Einstieg in die Kommunikation mit OpenGL.

Da der Weg, direkt mit OpenGL ein Spiel zu programmieren sich konzeptuell auf einer niedrigeren Abstraktionsschicht abspielt, wirkt der Einstieg um einiges komplexer. Alleine um ein Fenster auf den Bildschirm zu zeichnen, bedarf es einiges an Vorbereitung, und so-genanntem Boilerplate-Code.

Im Vergleich dazu bietet aber die direkte Interaktion mit einer Grafik-API den Vorteil, dass die Programmierer alle Dateien, die verwendet werden auch selbst schreiben. Der Vorteil hierbei kommt dann zum Vorschein, wenn ein Versionierungssystem wie zum Beispiel git verwendet wird.

Eine Game Engine produziert in der Regel immer zusätzliche Dateien, die, unter anderem, diverse Metadaten beinhalten. Dies erschwert es einerseits, Änderungen nachzuvollziehen, und erhöht andererseits den Speicherbedarf des Quelltextes.

3.1.1 Aufbau

Wenn ein Spiel von Grund auf ohne Game Engine programmiert wird, ist einiges an Aufbau und Vorbereitung notwendig. Um überhaupt etwas auf den Bildschirm zu zeichnen, muss zuerst ein Fenster erstellt werden. Die Vorgehensweise dies zu tun ist für jedes Betriebssystem

unterschiedlich – unter Windows kann direkt die WinAPI angesprochen werden, bei Apples Mac-Systemen wird die Cocoa-API[16] zur Verfügung gestellt, und unter Linux ist es sowohl von der Distribution als auch von der verwendeten Desktop-Umgebung abhängig.

Hier zeigt sich ein weiteres Mal ein Vorteil bei der Verwendung von Game Engines – es ist nicht notwendig, diese Umgebung aufzusetzen, da sie bereits zur Verfügung gestellt wird.

Des Weiteren bieten Game Engines wie Unity auch direkt die Möglichkeit, mit einer simplen Einstellung, das Spiel auf mehreren Systemen lauffähig zu bekommen. Für Spiele, die ohne die Verwendung jeglicher Softwarebibliotheken programmiert wurden, bedeutet es also einen Mehraufwand – und das für jede Plattform, die als Zielplattform anvisiert wird.

Im Beispiel dieser Studie wird allerdings die Bibliothek glfw [17] verwendet. Diese erlaubt es, plattformagnostisch ein Fenster zu erstellen, dieses zu manipulieren sowie mit (unter anderem) OpenGL in diesem Fenster etwas zu zeichnen. Da in der Regel ein Fenster nur einmal – am Start des Programms – erstellt werden muss, ist die zusätzliche Rechenzeit, die so eine Bibliothek mit sich bringt, vernachlässigbar. Schlussendlich werden durch glfw auch nur die Funktionen der jeweiligen API des gerade verwendeten Systems aufgerufen – der Vorteil liegt aber darin, dass unter anderem etwaige Fehler abgefangen werden. OpenGL selbst dient nur dazu, um in einen Framebuffer zu schreiben, und aus diesem zu lesen, und bietet keinen Mechanismus um Benutzereingabe zu verarbeiten. [9]

3.1.1.1 Matrizen

Ein dreidimensionales Objekt wird normalerweise als Liste von sogenannten Vertices beschrieben. Diese werden immer relativ zu einem lokalen Ursprung angegeben. Damit dieses Objekt allerdings auf einem zweidimensionalen Bildschirm dargestellt werden kann, müssen diese Vertices allerdings einige Transformationen durchlaufen.

Damit ein und dasselbe Objekt nicht nur an einer Position und in einer Orientierung dargestellt werden kann, werden alle Vertices dieses Objekts zuerst mit einer sogenannten Model-Matrix multipliziert. Diese Model-Matrix ist wiederum das Ergebnis einer Reihe an Multiplikationen. Die Vertices werden zuerst skaliert, dann rotiert und danach transformiert. Das Ergebnis dieser Multiplikation liefert eine Matrix, die das Objekt von einer relativen Darstellung zu sich selbst – dem sogenannten Model-Space – in eine Beschreibung relativ zur gesamten Szene – dem sogenannten World-Space – umwandelt.

Damit alle Objekte aber gezeichnet werden können, müssen die Vertices noch so transformiert werden, dass diese relativ zur Kamera beschrieben werden. Dies geschieht durch eine Multiplikation mit der sogenannten View-Matrix. Die View-Matrix selbst ist die inverse der Model-Matrix der Kamera.

Nachdem nun alle Vertices relativ zur Kamera beschrieben werden, müssen diese noch in den sogenannten Clip-Space transformiert werden. Dies beschreibt nichts anderes als die Projektion aller sichtbaren Punkte aus der Welt auf die Kamera. Sobald diese Transformation vollzogen wurde, lässt sich die Szene auf dem Bildschirm zeichnen.

Während es einfacher ist, mit einer Game Engine ein Projekt aufzusetzen, bietet der in diesem Kapitel beschriebene Ansatz den Vorteil, direkte Kontrolle über viele Aspekte des Programms zu haben. Dies erlaubt es, den Programmierer*innen die Aufgabe sowohl Objektorientiert, als auch mit komplett anderen Programmierparadigmen zu lösen. Dies ist natürlich immer ein zweischneidiges Schwert, da für viele Menschen eine enge Vorgabe auch eine Hilfestellung darstellen kann.

Für das Spiel, welches für diese Studie mit Hilfe von OpenGL programmiert wurde, wurde dennoch ein objektorientierter Ansatz gewählt.

3.1.2.2 Modelle und Texturen

Um die für das Spiel notwendigen Modelle zu importieren, musste für diese Implementation ein eigener Programmteil entwickelt werden, der es erlaubt aus einem vordefinierten Modell Daten in das Programm zu laden. Hierfür wurde sich am OpenGL-Tutorial [11] orientiert. Mit dem quelloffenen 3D-Modellierungsprogramm Blender [12] wurde eine einfache Kugel erstellt, und dann als trianguliertes Mesh in Form einer .obj-Datei gespeichert. Dieses lässt sich nun mit Hilfe der selbst geschriebenen Logik in das Spiel laden. Eine .obj-Datei [13] ist prinzipiell nur eine textuelle Darstellung der Vertices des darzustellenden Objektes, was es erlaubt, dieses Format mit relativ wenig Aufwand zu interpretieren und in einem Programm zu verwenden.

Diese selbst geschriebene Logik hat nun den Nachteil, dass sie nur ein bestimmtes Format unterstützt – und dieses auch nur, unter der Voraussetzung, dass es mit dem richtigen Programm mit den richtigen Parametern exportiert wurde. Für diesen Schritt besteht auch die Möglichkeit, eine Softwarebibliothek wie zum Beispiel ASSIMP [14] zu verwenden. Mit dieser oder einer ähnlichen Lösung ist es nun wieder möglich, alle gängigen Formate für Modelle zu verwenden. Da ASSIMP – im Gegensatz zu Unity – auch quelloffen ist, besteht die Möglichkeit auch für die entwickelnde Person selbst diese zu erweitern, falls es gewünscht ist, Modelle in nicht weit verbreiteten Formaten zu verwenden.

Damit das nun in das Spiel geladene Modell nicht nur einfärbig ist, wird für die Kugel auch eine Textur verwendet. OpenGL bietet nicht nur die Möglichkeit unkomprimierte Texturen in Form einer Bitmap zu verwenden, sondern erlaubt es auch komprimierte Texturen in Form einer .dds-Datei [21] hineinzuladen. Hierfür wurde ebenfalls eine eigene Methode geschrieben, die es erlaubt, eine .dds-Datei zu lesen, und direkt in ein für OpenGL sinnvolles und verständliches Format zu bringen. Auch für das Laden von Texturen in jeglichen gängigen Formaten bieten Bibliotheken wie ASSIMP eine vorgefertigte Lösung. Das Schreiben der Methoden, die die Texturen und Modelle in ein

Programm laden bieten besonders Studierenden und Menschen, die in ihrer Freizeit programmieren einen größeren und breiteren Lerneffekt, als wenn diese Menschen eine Engine verwenden würden.

3.1.2 Das Spiel selbst

Das Spiel besteht aus einer Kugel, die sich auf einer Ebene im Raum befindet. Es kann dieser Kugel über einen Tastendruck einen Impuls gegeben werden, sodass sich diese auf dieser Ebene in eine bestimmte Richtung bewegt. Durch das Betätigen einer anderen Taste wird eine neue Kugel geladen. Nachdem geprüft wurde, dass diese neue Kugel nicht mit einer schon bestehenden Kugel kollidieren würde, erscheint diese nun im Spiel. Es wird immer nur die zuletzt geladene Kugel kontrolliert.

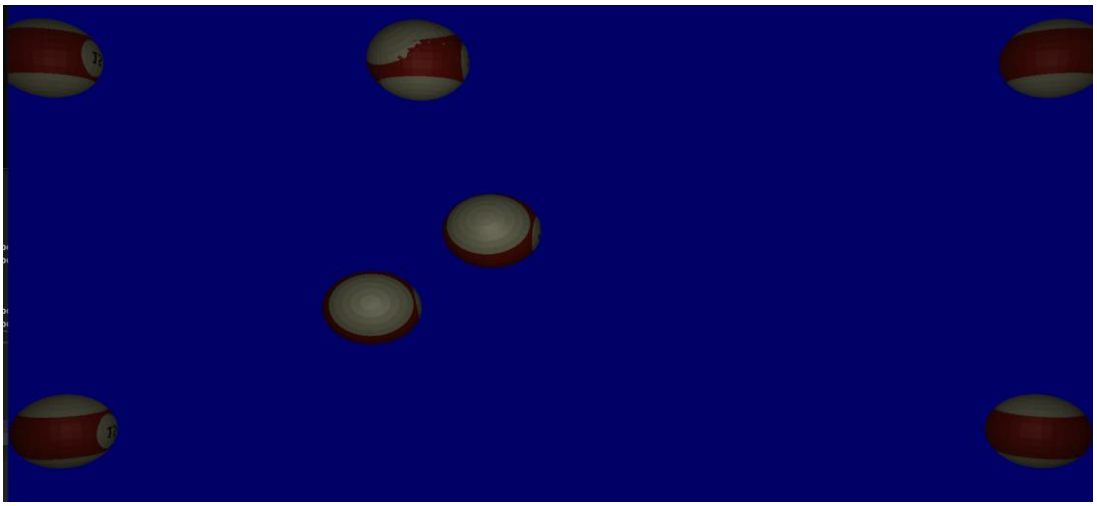

Abb 1.: Das Spiel, das mit OpenGL programmiert wurde

Sobald der kontrollierten Kugel ein Impuls mitgegeben wurde, rollt diese Kugel auf ihrer Ebene solang weiter, bis ihre Geschwindigkeit durch einen eingebauten, konstanten Widerstandsfaktor auf 0 reduziert wird. Des Weiteren ist die Ebene, auf der sich die Kugel bewegen kann, begrenzt. Kollidiert eine der Kugeln nun mit der Begrenzung dieser Ebene, prallt sie mit einem elastischen Stoß zurück.

An den vier Ecken dieser Ebene befinden sich, ähnlich wie bei einem echten Billardtisch, vier Löcher. Sobald eine Kugel mit einem dieser Löcher kollidiert, simuliert das Spiel das Versinken dieser Kugel, und diese Kugel verschwindet von der Spielfläche.

Für die Erkennung der Kollision wird ein naiver Algorithmus verwendet: Jede Kombination zwischen zwei Objekten wird geprüft. Dies führt zu einer ineffizienten Laufzeit von $O(n^2)$. Für eine Anwendung mit wenig Objekten, für die eine Kollisionsprüfung notwendig ist, ist das bei moderner Hardware zwar möglicherweise ausreichend, skaliert allerdings nicht zufriedenstellend mit

größerer Anzahl an Objekten. Um die Anzahl an Vergleichen zu reduzieren gibt es mehrere Lösungsansätze, wie zum Beispiel Binary Space Partitioning. [10]

Um einen Parameter zu erhalten, die Leistungsfähigkeit des Spiels zu messen, wird für jede Kugel, die neu erscheint, dasselbe Modell neu geladen. Dies ermöglicht es, Laborbedingungen zu schaffen, um die Skalierbarkeit dieses kleinen Experiments auf mehrere Modelle zu messen. Würde man diese Anwendung auf ein Spiel erweitern, würde es wenig Sinn machen, ein und dasselbe Modell für mehrere Instanzen nicht wiederzuverwenden.

3.1.3 Messmethode

Glfw bietet die Möglichkeit, mit der Funktion glfwGetTime die aktuelle Systemzeit auszugeben. Vergleicht man nun die Zeit zwischen zwei direkt nacheinander gezeichneten Bildern, erhält man die Bildwiederholungsrate. Diese Bildwiederholungsrate wird über 5 Sekunden summiert und danach durch diese verstrichene Zeit dividiert um einen Durchschnitt zu erhalten. Diese ist ein wichtiger Indikator für die Leistungsfähigkeit eines Spiels. Befindet sich die Bildwiederholungsrate konstant unter 60 Bildern pro Sekunde, wird das Spiel von manchen Spieler*Innen nicht mehr als flüssig wahrgenommen. [8]

Des Weiteren wurde auf die Profiling Tools von Visual Studio zurückgegriffen. Diese erlauben es unter anderem, einen Einblick darauf zu erhalten, welche Methoden am öftesten aufgerufen wurden, welche Methoden die meiste CPU-Zeit in Anspruch nehmen und wieviel Speicher das Programm benötigt.

Die Versuche werden auf einem Rechner mit einem Prozessor der Reihe Intel i9-11900k, mit 3.50GHz, einer Grafikkarte der Reihe Nvidia 3080ti und 128GB Arbeitsspeicher durchgeführt.

3.1.4 Ergebnisse

Auffällig ist, dass beim Starten des Spiels, wenn eine geringe Anzahl an Objekten auf dem Bildschirm sichtbar ist, die Bildwiederholungsrate bei über 4500 Bildern pro Sekunde liegt. Dies ist unter Anderem dem relativ Leistungsstarken System zu verdanken, auf dem dieses Experiment durchgeführt wurde.

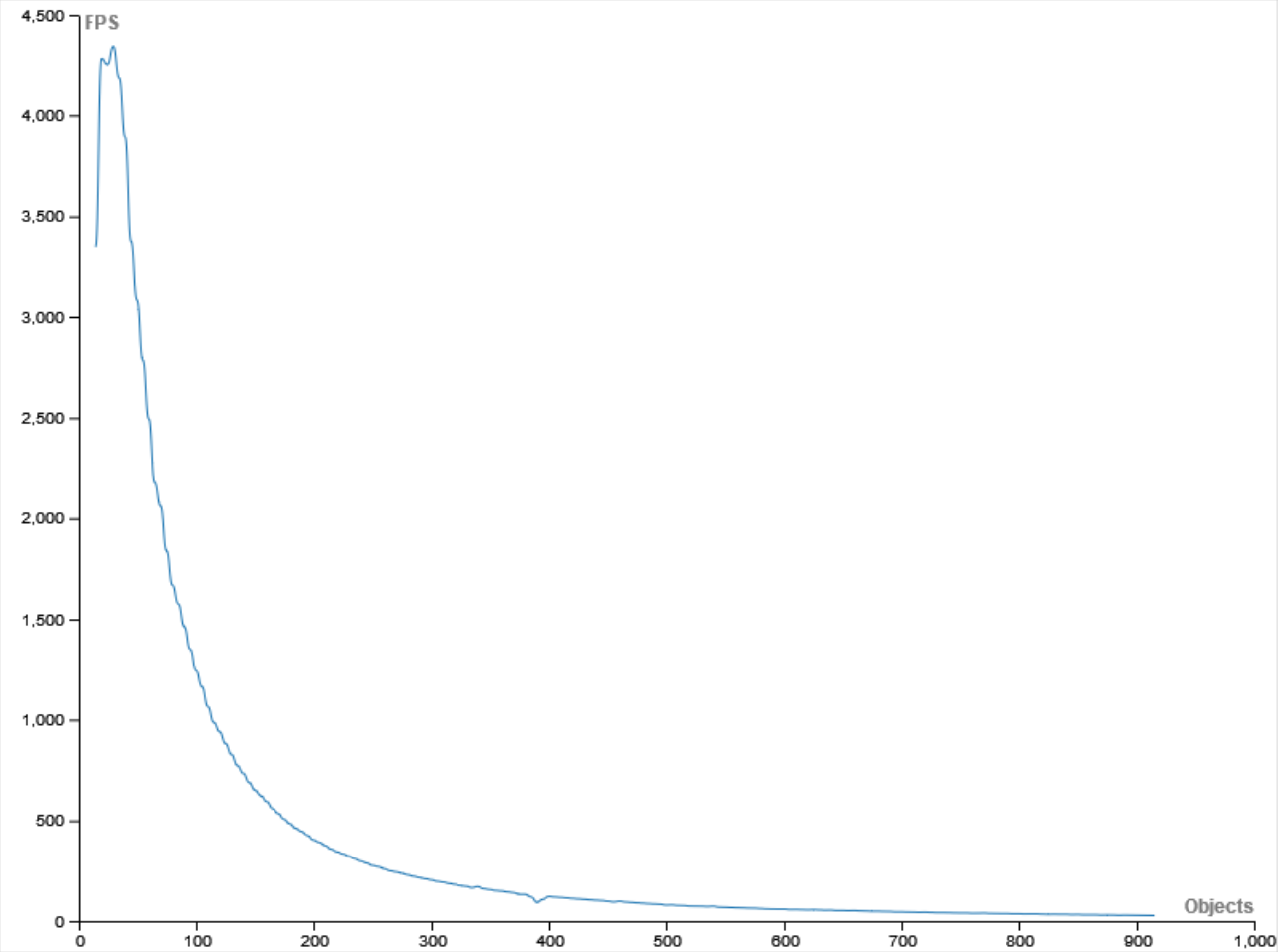

Abb. 2: Bildwiederholrate abhängig von zu zeichnenden Objekten, OpenGL

Wie in Abbildung 2 ersichtlich, läuft das Spiel bis zu einer Anzahl von 200 Objekten noch mit einer Bildwiederholungsrate von über 200 Bildern pro Sekunde. Allerdings verringert sich diese drastisch, je mehr Objekte simuliert werden. Dies hat einerseits den Grund, dass mehr Objekte sowohl im Speicher gehalten werden müssen, aber liegt auch daran, dass für die Implementation der Kollisionsabfrage im OpenGL-Spiel ein naiver Algorithmus verwendet wurde.

Dieser sorgt dafür, dass bei jeder Überprüfung jedes existierende Objekt mit jedem anderen existierenden Objekt verglichen werden muss. Laut Theorie führt das zu einer Rechenkomplexität von $O(n^2)$ – dies ist auch in der Abbildung 1 ersichtlich.

Auch ein Blick in die Auswertung von Visual Studio zeigt, dass die meiste Rechenzeit, damit verbracht wird, zu überprüfen, ob ein Objekt mit einem anderen kollidiert.

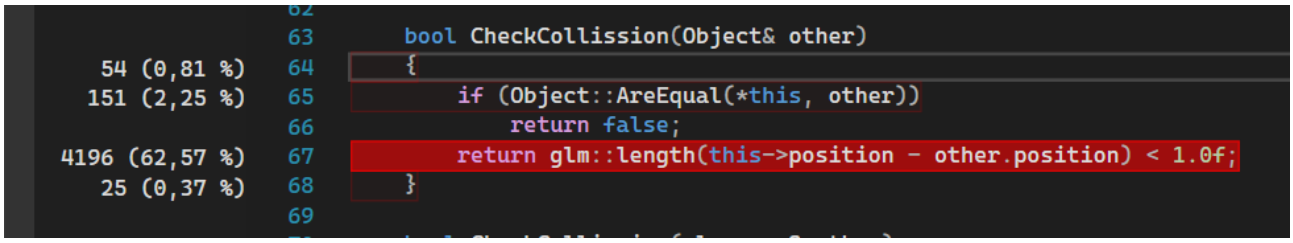

Abb. 3: Auszug aus den Visual Studio Profiling Tools

Dies lässt sich auch mit Hilfe der Visual Studio Profiling Tools erkennen. Diese bieten die Möglichkeit, ausgeben zu lassen, welche Funktionen relativ gesehen wie viel Rechenzeit in Anspruch nehmen. Bei dem oben beschriebenen Versuch werden über 60% der Rechenzeit dafür verwendet, zu überprüfen, ob zwei Objekte miteinander kollidieren. Hierzu ist auch noch zu erwähnen, dass das OpenGL-Spiel eine sehr einfach Kollisionsüberprüfung durchführt. Da bekannt ist, dass jedes Objekt, das im Spiel vorkommt, die Form einer Kugel hat, wird nur überprüft, ob sich zwei Kugeln überschneiden.

3.2 Unity

Sobald Unity von der offiziellen Website [15] heruntergeladen und installiert wurde, bietet ads Programm eine Wahl verschiedener Templates. Für diese Studie wird das Template „3D Core“ verwendet. Dieses stellt eine Szene zur Verfügung, in der eine Kamera und eine Lichtquelle im Voraus platziert wurden. Im Unterschied zum in dem mit OpenGL programmierten Spiel, bietet Unity einen leistungsstarken Editor an. Dieser erlaubt es, verschiedenste Objekte – seien es Modelle, Audioquellen oder andere sichtbar und auch nicht sichtbare Objekte, die sich in der Spielwelt befinden – per einfachem Drag & Drop in die Spielwelt zu ziehen, und diese so intuitiv zu manipulieren. Auch müssen in Unity keine Bibliotheken verknüpft werden, und es muss keine einzige Zeile Code geschrieben werden, um das Spiel starten zu können.

Wurde ein neues Projekt in Unity aus einem Template erstellt, erscheint ein neuer Ordner am gewählten Dateipfad. Eine Auffälligkeit zeigt sich, wenn ein Blick auf die Größe dieses Ordners geworfen wird. Ein leeres Projekt, das mit Unity 2021.3.12f1 aus dem „3D Core“ Template erstellt wurde, benötigt 285 Megabyte Speicherplatz. Dies ist mit der Hardware der heutigen Zeit kein grundsätzlich großes Problem, bringt aber einen beträchtlichen Nachteil mit sich. In der allgemeinen Software-Industrie – besonders für Hobbyprojekte und kleine Unternehmen – ist es üblich, für die Versionierung einen Dienst wie GitHub [18] zu verwenden. GitHub erlaubt es allerdings nicht – zumindest nicht ohne den Zusatzdienst Large File Storage [19] zu bezahlen -

Dateien mit einer Größe von über 50MB zu verwenden. Ein weiterer Aspekt, den es zu beachten gibt, ist, dass jede Änderung in einem Unity-Projekt nicht nur eine Änderung an einer Datei bedeutet – Unity schreib sehr viele Metadaten mit, die intern benötigt werden. Versucht man allerdings nun in einer Versionierungssoftware (wie zum Beispiel git) die Entwicklungshistorie des Spiels an Hand von einzelnen Änderungen nachzuvollziehen, stellt sich das als relativ schwierige Aufgabe dar, da, wie erwähnt, bei jeder Änderung von Unity selbst benötigte Änderungen in vielen anderen Dateien mitgeschrieben werden.

Unity bietet zwar selbst eine Integration mit den Versionierungssystemen Perforce und PlasticSCM [20] an. Dies benötigt allerdings einen vorkonfigurierten Server mit einem dieser Systeme, was wiederum ein Hindernis für Hobbyprojekte darstellen könnte.

Unity verwendet ein streng hierarchisches System für die Darstellung seiner Daten. In der Praxis bedeutet dies, dass es in einem Projekt eine beziehungsweise mehrere Szenen gibt. Jeder diese Szenen kann eine Art Level darstellen, und größerer Projekte werden üblicherweise in mehrere Szenen geteilt. Diese Szenen selbst beinhalten dann mehrere Objekte, die in Unity gameObjects genannt werden. Diese Objekte wiederum speichern einerseits allgemeine Informationen – wie zum Beispiel die Position, die Rotation oder auch Eigenschaften über den Zustand des Objekts, wie zum Beispiel ob es gerade sichtbar ist oder nicht – und andererseits die Komponenten dieses Objekts.

Typische Komponenten für Objekte sind zum Beispiel ein sogenanntes Mesh, das angibt, welche Vertices gezeichnet werden, eine Physik-komponente, die beschreibt, wie dieses Objekt mit Kräften und anderen Objekten interagiert, und Scripts, die es erlauben, Logik zu implementieren. Unity bietet für viele Anwendungszwecke – vor allem Spiele und Simulationen, die sich an der echten Welt orientieren, und deren Ziel es ist, diese vereinfacht darzustellen – vorgefertigte Lösungen beziehungsweise Lösungsansätze.

Wird der Play-Button betätigt, werden zunächst die Assets kompiliert, und dann wird das Spiel gestartet.

3.2.1 Aufbau

Unity bietet direkt über den Editor die Möglichkeit, eine Kugel in die Spielwelt zu ziehen. Dadurch ist es nicht nötig, für sehr einfache Formen auf externe 3D-Modellierungsprogramme zurückzugreifen. Des Weiteren bietet Unity die Möglichkeit, sogenannte Prefabs zu verwenden. Ein Prefab ist ein Objekt, das im Vorhinein konfiguriert und angepasst werden kann, um dann im weiteren Verlauf der Entwicklung des Spiels immer wieder und wieder platziert werden kann. Für die Zwecke dieses Spiel wurde eine Kugel erstellt. Dieser Kugel wurde ein sogenanntes Physics Material als Komponente mitgegeben. Im Vergleich zur Entwicklung mit OpenGL lässt sich hier

wieder ein immenser Vorteil erkennen: Unity bietet von sich aus Wege und Möglichkeiten an, simple physikalische Interaktionen zwischen Objekten zu modellieren. Während es bei der OpenGL-Umsetzung noch notwendig ist, jede noch so kleine Interaktion selbst zu definieren und durchzudenken, ist es bei der Verwendung von Unity mit relativ geringem Aufwand möglich, simple physikalische Interaktionen im Spiel darzustellen. Dies ermöglicht es auch Menschen, die mit den mathematischen Grundlagen der Darstellung von Objekten im Raum nicht vertraut sind, einfache Spiele zu programmieren, und stellt dadurch wieder eine niedrigere Eintrittsbarriere dar.

Weiters lässt sich auf die vorhin erwähnte Kugel ein Material anwenden. Dieses Material beschreibt verschiedene optische Eigenschaften des Objektes, wie zum Beispiel die Farbe, den Glanz aber auch die Lichtdurchlässigkeit.

Nachdem die Kugel nun in der Welt sichtbar war, wurde es Zeit, die simple Spiellogik hinzuzufügen. Dies ist wiederum recht einfach, da Unity auf dem physikalischen Objekt die Möglichkeit bietet, dieses Objekt als Auslöser (Trigger) zu definieren. Dies signalisiert der Game Engine, dass dieses Objekt nicht für physikalische Berechnungen in Betracht gezogen werden soll. Kollidiert jedoch ein anderes Objekt damit, wird stattdessen im anderen Objekt die Funktion OnTriggerEnter aufgerufen. In einem Skript-Objekt wird nun definiert, dass die Spielkugel, wenn sie mit dem Auslöser kollidiert – ähnlich wie im mit OpenGL programmierten Spiel - verschwinden soll.

3.2.2 Messmethode

Auch in diesem Spiel wurde eine Funktion eingebaut, die es erlaubt, per Tastendruck neue Kugeln in der Spielwelt erscheinen zu lassen. Dies dient dazu, zu prüfen, wie die Leistungsfähigkeit eines Spiels, das mit der Game Engine Unity programmiert wurde, mit der Leistungsfähigkeit des oben beschriebenen Spiels ohne Game Engine zu vergleichen, und beobachten zu können, wie der Speicherverbrauch und die CPU-Auslastung mit größer werdender Anzahl an Objekten skaliert.

Auch im mit der Game Engine Unity programmierten Spiel wurde die Bildwiederholungsrate dadurch bestimmt, dass eine Differenz zwischen zwei Zeitpunkten gewählt wurde, dann die Anzahl der gezeichneten Bilder genommen wurde, und diese durch die verstrichene Zeit geteilt wurde.

3.2.3 Ergebnisse

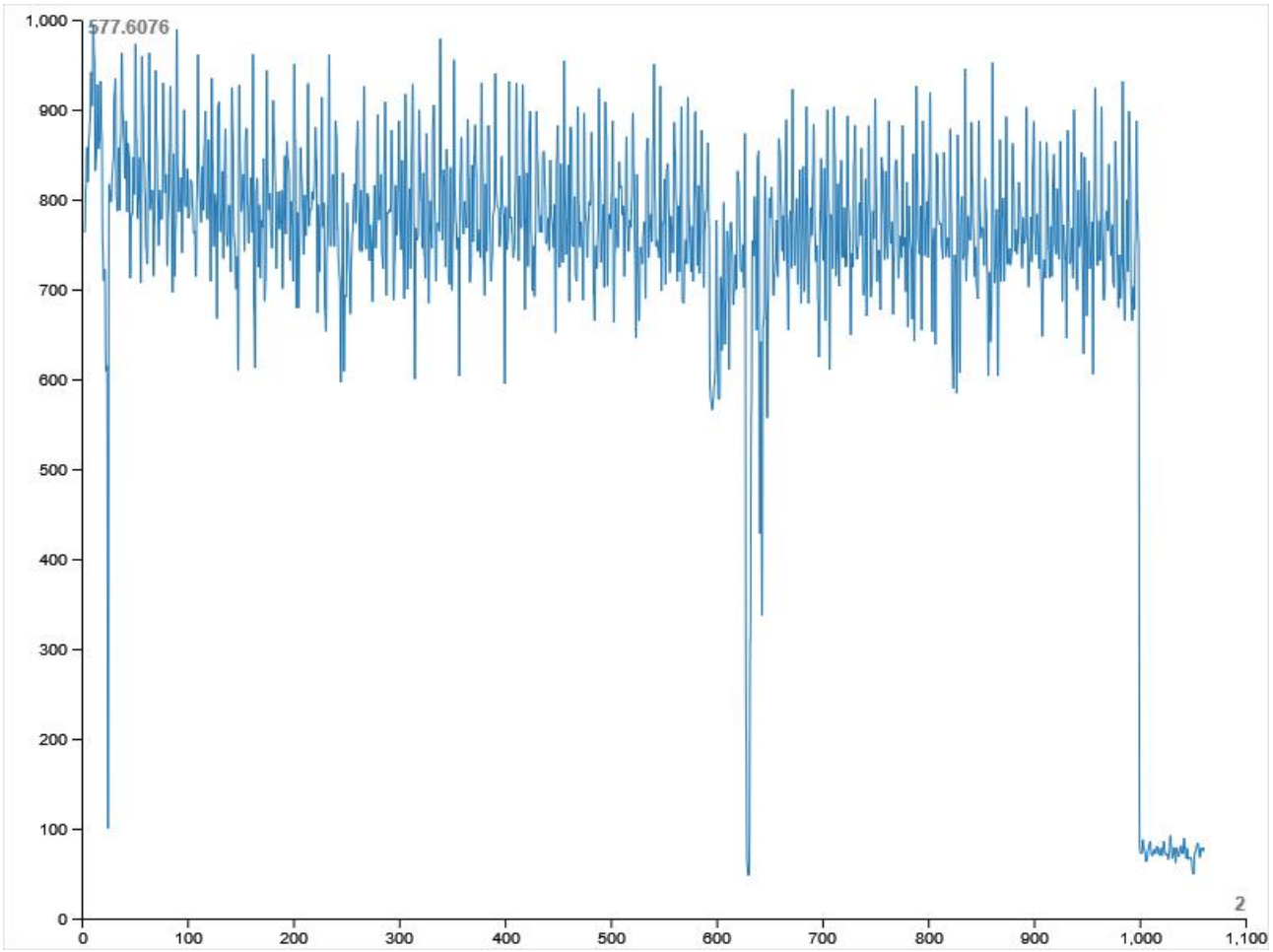

Abb. 4: Bildwiederholrate in Abhängigkeit von dargestellten Objekten, Unity

Bei dem mit Unity programmierten Spiel fallen sofort zwei Dinge in das Auge: Einerseits fluktuiert die Bildwiederholungsrate zwischen einzelnen Messpunkten viel stärker. Dies ist unter anderem darauf zurückzuführen, dass bei Spielen, die mit Unity programmiert werden, viele Systeme automatisch im Spiel inkludiert sind, und diese im Hintergrund Berechnungen vollziehen.

Die andere Auffälligkeit besteht darin, dass bei diesem Experiment nicht auf den ersten Blick eine Korrelation zwischen der Anzahl an Objekten, die gezeichnet werden und der Bildwiederholungsrate feststellbar ist.

Schlussendlich lässt sich daraus erkennen, dass bei Spielen, die mit Game Engines wie Unity entwickelt werden, nicht direkt ein Leistungsnachteil feststellbar ist – zumindest, wenn ähnlich vorgegangen wird, wie in dieser Studie.

4 Conclusio

Dank der immensen Größe der Spieleindustrie wird immer mehr Energie und Zeit in die Entwicklung von Game Engines hineingesteckt. Diese Tatsache führt dazu, dass moderne Game Engines unglaublich starke und vielseitige Werkzeuge geworden sind, mit denen sich sehr viele Aufgaben effizient lösen können. Auch hat diese Entwicklung dazu beigetragen, dass Menschen mit weniger Ressourcen (sei es Zeit oder auch Programmierkenntnisse) es leichter bewerkstelligen können, fertige Produkte auf den Markt der Videospiele zu bringen.

Obwohl Game Engines eben diesen leichteren Einstieg bieten, werden selbst bei einer einfachen Anwendung viele leistungsstarke System in Anspruch genommen. Dies lässt vermuten, dass ein Spiel, das ohne diese zusätzlichen Systeme auskommt, die Leistung eines Systems besser ausnutzt, und dadurch mit einer höheren Bildwiederholungsrate laufen kann.

Eine kurze praktische Studie zeigt allerdings, dass dies nicht notwendigerweise der Fall sein muss. Da Game Engines von großen Entwicklungsteams betreut werden, werden hier viele Optimierungen schon angewandt, ohne dass die Entwickler*innen über diese Bescheid wissen müssen oder diese gar selbst implementieren müssen. Dies führt dazu, dass ein simples Spiel, das von Hand und mit wenig Rücksicht auf mögliche Optimierungsmaßnahmen programmiert wird, schon einen merkbaren Einbruch der Bildwiederholungsrate erleidet, wenn sich die Anzahl der zu zeichnenden Objekte erhöht.

Schlussendlich zeigt dies, dass es zwar einen erheblichen Lerneffekt bietet, Spiele selbst und ohne die Verwendung von Game Engines zu programmieren, es sich aber üblicherweise nicht lohnt, wenn das Hauptziel darin besteht, in wenig Zeit ein fertiges Spiel zu bauen. Die Vorteile, vollständige Kontrolle über die gesamte Software zu haben, überwiegen nicht, und im Allgemeinen ist es zu empfehlen, nicht auf die Verwendung von Game Engines zu verzichten – außer das Ziel besteht darin, die komplette darunterliegende Technologie zu verstehen – der besagte Lerneffekt.

Im Bereich der Kommunikation im Netzwerk zwischen mehreren Spielern sind moderne Game Engines schon auf einer Stufe, wo es sich mehr lohnt über Aspekte wie Sicherheit nachzudenken, da der Austausch von Daten zwischen Spielern meist eher durch die physikalische Distanz zwischen denjenigen Spielern limitiert ist.

Literaturverzeichnis

[1] Oxford English Dictionary. (2021) reference, v. 3. Oxford: Oxford University, https://www.oxfordlearnersdictionaries.com/definition/english/video-game, Aufgerufen am 20.11.2021

[2] Jason Gregory. 2014. Game Engine Architecture, Second Edition (2nd. ed.). A. K. Peters, Ltd., USA.

[3] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. 2018. Real-Time Rendering, Fourth Edition (4th. ed.). A. K. Peters, Ltd., USA.

[4] Mark Claypool, Kajal Claypoo, Latency and player actions in online games, 2005, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.216.9904

[5] Lars Tijsmans, Collaborative work with Augmented and Virtual Reality – A secure network connection in Unity, https://homepages.staff.os3.nl/~delaat/rp/2019-2020/p06/report.pdf

[6] Comparison of Unity and Unreal Engine, Antonín Smíd, 2017, https://core.ac.uk/download/pdf/84832291.pdf

[7] Memory Management for Game Audio Development, Rahmin Tehrani, 2020, https://digitalcommons.csumb.edu/cgi/viewcontent.cgi?article=1881&context=caps_thes_all

[8] Claypool, Mark & Claypool, Kajal & Damaa, Feissal. (2006). The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games. Proceedings of SPIE - The International Society for Optical Engineering. 6071. 10.1117/12.648609.

[9] Mark Segal, Kurt Akeley, The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile) - May 5, 2022)

https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf

[10] Ganter, M. A., and Isarankura, B. P. (March 1, 1993). "Dynamic Collision Detection Using Space Partitioning." ASME. **J. Mech. Des**. March 1993; 115(1): 150–155. https://doi.org/10.1115/1.2919312

[11] http://www.opengl-tutorial.org/, aufgerufen am 1.11.2022

[12] https://www.blender.org/, aufgerufen am 1.11.2022

[13] https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml#notes, aufgerufen am 26.11.2022

[14] https://github.com/assimp/assimp, aufgerufen am 1.11.2022

[15] https://unity.com/, aufgerufen am 26.11.2022

[16] https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CocoaApplicationLayer/CocoaApplicationLayer.html, aufgerufen am 1.11.2022

[17] https://www.glfw.org/, aufgerufen am 1.11.2022

[18] https://github.com/, aufgerufen am 1.11.2021

[19] https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-git-large-file-storage, aufgerufen am 26.11.2022

[20] https://docs.unity.cn/2021.1/Documentation/Manual/Versioncontrolintegration.html, aufgerufen am 2.11.2022

[21] https://learn.microsoft.com/en-us/windows/win32/direct3ddds/dx-graphics-dds-pguide, aufgerufen am 20.11.2022

Abbildungsverzeichnis

Anhang

OpenGL-Implementation: https://github.com/PowerSlaveAlfons/OpenGL

Unity-Implementation:
https://drive.google.com/file/d/17ayow_lfs_Wo6l8OSRoTKzaXmDxwAskA/view?usp=share_link