Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Visualisierung von Partikeldaten in Unreal

Rene Richard Tischler

Studiengang:	Medieninformatik
Prüfer/in:	Prof. Dr. Thomas Ertl
Betreuer/in:	M.Sc. Michael Becher, M.Sc. Ludvig Mangs, Dipl.-Inf. Christoph Müller, Dr. Guido Reina
Beginn am:	4. Oktober 2022
Beendet am:	13. April 2023

Kurzfassung

Durch den vermehrten Einsatz von Computersimulationen in den Forschungsgebieten der Physik, Chemie und Biologie werden die zu visualisierenden Datenmengen immer größer. Diese Arbeit gibt einen Einblick auf die aktuelle Nutzung von Game Engines im Bereich der wissenschaftlichen Visualisierung.

Hierfür wurde mithilfe der Unreal Engine ein eigenes Visualisierungstool entwickelt, das bei Datensätzen mit mehreren Millionen Partikeln immer noch eine nutzbare Bildwiederholungsrate aufweist. Es wurde ein Importmechanismus für MMPLD-Dateien implementiert und ein spezielles Daten-Interface für das Niagara Partikelsystem geschrieben.

Nach der Entwicklung wurde der Prototyp mit unterschiedlichen Datensatzgrößen getestet und die Ergebnisse im Vergleich zu MegaMol, einem Prototyping Framework für interaktive Visualisierungen, evaluiert. Dabei wurden unter anderem auch eine erhöhte Ladezeit und ein erheblicher Arbeitsspeicherverbrauch festgestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AR Augmented Reality. 15

CPU Central Processing Unit. 19, 21, 23

DLL Dynamic Link Library. 15

FPS Frames Per Second. 31

GPU Graphics Processing Unit. 14, 19, 23, 39

HLSL High Level Shading Language. 21

SIMD Single Instruction Multiple Data. 21

VR Virtual Reality. 15

1 Einleitung

Dieser Abschnitt erklärt das Thema dieser Arbeit, gewährt Einblicke in dessen Relevanz und stellt Literatur vor, die mit dieser Arbeit zusammenhängt oder verwandt ist.

1.1 Motivation

In den verschiedenen Forschungsgebieten der Physik, Chemie und Biologie kommen immer häufiger Computersimulationen zum Einsatz. Die Ergebnisse dieser Simulationen sind Zeitreihen von dreidimensionalen Partikeldaten, deren Umfang immer größer wird und die es zum besseren Verständnis zu visualisieren gilt.

Dies führt zu einer zunehmenden Diskrepanz zwischen Datensatzgröße und Renderleistung, obwohl sich die Rechenleistung moderner Desktop-PCs stetig verbessert. Bei einer kosmologischen Simulation wie der DarkSky-Dark-Matter-Simulation kann laut Wald et al. [WWB+14] jeder Zeitschritt die Größe von 30 Terabyte überschreiten. Klassische Visualisierungssysteme kommen bereits bei Datensätzen mit einem Bruchteil der Größe, beispielsweise mit mehreren Millionen Partikeln, an ihre Grenzen.

Des Weiteren werden die wissenschaftlichen Forschungsfragen immer komplexer, weshalb der Bedarf an effizienten, leicht bedienbaren und einfach erweiterbaren Visualisierungstools so hoch ist wie noch nie.

Die Videospielindustrie stand laut Lv et al. [LTD+13] vor ganz ähnlichen Problemen in Bezug auf Komplexität und Effizienz. Weshalb Game Developer über viele Jahre immer fortschrittlichere Technologien entwickelten, wiederverwendbare Frameworks sowie effiziente Routinen aggregierten und sie zu einfach verwendbaren Toolkits, sogenannten Game Engines, zusammengeschlossen haben.

Die daraus entstandenen Tools ermöglichen der Branche einen schnellen Entwicklungszyklus, wodurch Sie die neueste Hardware und Software optimal ausnutzen können. Mittlerweile unterstützen Game Engines quasi-fotorealistisches Rendering in Echtzeit und bieten darüber hinaus sprite-basierte Partikelsysteme, die oft für visuelle Effekte wie Feuer, Gas und Rauch eingesetzt werden.

Ziel dieser Arbeit ist es zu evaluieren, ob die partikelbasierte Visualisierung von den Entwicklungen in der Videospielbranche profitieren kann. Hierzu soll mithilfe der Unreal Engine eine pixelgenaue, raycasting-basierte Darstellung von Partikeldaten implementiert werden und anschließend deren Performanz mit MegaMol einem Prototyping Framework für interaktive Visualisierungen verglichen werden.

1.2 Verwandte Arbeiten

Die partikelbasierte Visualisierung kommt häufig bei der interaktiven und explorativen Analyse von molekularen Daten zum Einsatz. Das Prototyping Framework MegaMol aus [GKM+15; WWB+14] wurde vom Visualisierungsinstitut der Universität Stuttgart in Zusammenarbeit mit dem Sonderforschungsbereich 716 entwickelt und ist auf die Visualisierung von molekularer Daten zugeschnitten.

Bei der Entwicklung des Frameworks standen neben Anpassbarkeit und Erweiterbarkeit auch die Effizienz im Fokus, weshalb MegaMol als dünne unterstützende Abstraktionsschicht über der OpenGL-API konzipiert wurde. Um zu gewährleisten, dass MegaMol mehrere Millionen Partikeln problemlos visualisieren kann, setzen Grottel et al. [GKM+15] auf das GPU-beschleunigte Glyphen-Raycasting. Die Grundidee dabei ist es laut Kozlíková et al. [KKF+17] eine Projektion des ursprünglichen Elements zu rendern welche die implizite Oberfläche umschließt. Für jedes Fragment des Grundelements wird dann der Schnittpunkt des Sichtstrahls mit der impliziten Oberfläche im Fragment-Shader berechnet und so der jeweilige Farbwert des Pixels ermittelt. Dieses Verfahren ist deutlich effizienter und erreicht eine höhere Darstellungsqualität als das dreieckbasierte Rendering.

Der modulare Aufbau des Prototyping Frameworks ermöglicht die Erweiterung von MegaMol durch Plugins und begünstigt so die Entwicklung neuartiger, effizienter Visualisierungsmethoden, wodurch die Anwendung in der Visualisierungsforschung vielseitig einsetzbar wird. Dies wurde in [WWB+14] eindrucksvoll bewiesen als Wald et al. OSPRay-Raytracing implementierten um In-situ-Visualisierungen auf High-Performance Compute-Cluster zu unterstützen.

Die zugrunde liegende Kernbibliothek des Frameworks unterstützt die Nutzerin oder den Nutzer laut Kozlíková et al. [KKF+17] mit grundlegenden Funktionen, schränkt sie oder ihn jedoch nicht im Bezug auf Datenstrukturen oder Technologien ein, wie es bei einigen anderen Visualisierungstools der Fall ist.

Die extreme Renderleistung von MegaMol auf normalen Desktop-PCs, die mit einer leistungsstarken Grafikkarte ausgestattet sind, macht das Prototyping Framework zum ideale Vergleichsmaßstab für diese Arbeit.

In der Visualisierungsforschung werden immer häufiger Game Engines erfolgreich für die Entwicklung von Visualisierungssysteme eingesetzt. Eins Beispiel hierfür ist UnityMol, ein Visualisierungstool das von Lv et al. [LTD+13] als Proof of Concept entwickelt wurde, um zu evaluieren, ob Game Engines bei der Entwicklung von neuartigen Visualisierungen und deren Prototypen helfen können. Während dieser Arbeit entstand ein eigenständiger Viewer, der in der Lage ist, molekulare Strukturen, Oberflächen und biologische Netzwerke zu visualisieren.

Bei der Entwicklung wurde von Lv et al. [LTD+13] festgestellt, dass originelle Features wie animierte elektrostatische Feldlinien oder HyperBalls-Shader schneller und leichter von Wissenschaftlerinnen und Wissenschaftlern mithilfe der Unity Game Engine implementiert werden können. Der dabei entstandene Prototyp ist laut Lv et al. [LTD+13] leicht modifizier- und erweiterbar und kann so anderen als Ausgangspunkt oder Plattform für ihre eigenen Entwicklungen dienen.

Die Performanz der selbst implementierten HyperBalls-Visualisierung wurde mit den molekularen Viewern VMD und PyMol verglichen. Welche laut Kozlíková et al. [KKF+17] neben Chimera, YASARA View und CAVER Analyst zu den robusten und beliebtesten Visualisierungstools für molekulare Daten gehören. Sie sind für nicht-kommerzielle Zwecke frei verfügbar und werden daher im großen Umfang von der wissenschaftlichen Gemeinschaft genutzt.

Die damals getestete Version des Prototypen konnte mit der Leistung der zuvor genannten Anwendungen nicht mithalten. Wurde zur Darstellung der molekularen Daten nicht die HyperBalls-Visualisierung verwendet, sondern das von Unity bereitgestellte Partikelsystem, so konnte eine merklich bessere Framerate erreicht werden. Die begrenzte Renderleistung sei laut Kozlíková et al. [KKF+17] auf den Overhead der Game Engine, die erhöhte Anzahl an Dreiecken und den erforderlichen Draw Calls, die zur Darstellung der molekularen Objekte benötigt werden, zurückzuführen.

Im Bereich der wissenschaftlichen Visualisierung wird meistens auf Game Engines zurückgegriffen, wenn neue Plattformen wie beispielsweise Augmented Reality (AR) oder Virtual Reality (VR) erforscht werden sollen. Nanome aus [KBL+19] ist eine VR-Anwendung die mit Unity entwickelt wurde und die es ermöglicht dass mehr als 10 Personen gleichzeitig in einer virtuellen Welt physisch mit Molekülen interagieren können. Nanome soll die Kommunikationsbarriere innerhalb von Wirkstofforschungsteams verringern und damit den Ideenfluss und die Zusammenarbeit verbessern.

Die Anwendung wurde von Kingsley et al. im Bezug auf Ladezeiten, Speicherverbrauch und Renderleistung mit klassischen Visualisierungstools wie PyMOL, Discovery Studio und Chimera verglichen. Des Weiteren wurde die Performanz auch im Vergleich zu Chimera X einem von Goddard et al. [GBS+18] entwickelten VR-fähigen Visualisierungssystem evaluiert. Nanome übertraf beim Oberflächen-Rendering alle getesteten Anwendungen um ein Vielfaches, benötigte dafür beim Laden der Partikeldaten etwas länger und hatte den größten Speicherverbrauch. Dennoch demonstrierten damit Kingsley et al. [KBL+19], dass die Visualisierungsforschung auch in Bezug auf Renderleistung von Game Engines profitieren kann.

In [WMW+18] wurde die Unity Game Engine zur Implementierung von Molecular Dynamics Visualization (MDV) genutzt, einem Visualisierungstool zur stereoskopischen 3D-Darstellung von biomolekularen Strukturen. Die Besonderheit bei dieser Arbeit ist, dass sich Wiebrands et al. bei der Entwicklung dazu entschlossen haben, die biomolekularen Modellierungs- und Visualisierungssysteme zu trennen.

Das Laden und Analysieren der biomolekulare Modelle wurde als eigenständige C#-Bibliothek realisiert und wird als Dynamic Link Library (DLL) in Unity geladen. Die Trennung ermöglicht eine einfachere Entwicklung von Tests zur unabhängigen Validierung der molekularen Modelle. Die Anwendung ermöglicht es einer Gruppe von bis zu 50 Personen, biomolekulare Modelle auf dem HIVE-Zylinder-Display anzusehen.

2 Implementierung

Dieses Kapitel dreht sich um die Realisierung der pixelgenauen, raycasting-basierten Darstellung von Partikeldaten mithilfe der Unreal Engine. Die Engine wurde von Epic Games mit dem Fokus auf fotorealische und interaktive 3D-Umgebungen entwickelt und ist für Ihre Renderleistung bekannt. In den nachfolgenden Abschnitten werden die technischen Details der Implementierung dokumentiert und teilweise die Design-Entscheidungen des Entwicklungsprozess begründet.

2.1 Importmechanismus

Da die Unreal Engine den Import von Partikeldaten nicht unterstützt musste zunächst einmal ein Importmechanismus für diese Daten implementiert werden. Da MegaMol für partikelbasierte Daten das MMPLD-Dateiformat verwendet, lag es nahe, ebenfalls auf dieses Format zu setzen.

Zunächst sollte hierfür mithilfe der MMPLD Header-only Library ein Modul in Unreal erstellt werden, welches dann das Einlesen und Konvertieren der Daten in ein Game Engine kompatibles Format handhabt. Bei der Entwicklung wurde festgestellt, dass die Unreal Engine aus Kompatibilitätsgründen nicht die vollständige C++-Standardbibliothek unterstützt, wodurch es zu Compilerfehlern kam.

Um dennoch die MMPLD Bibliothek verwenden zu können, mussten die dafür benötigten Funktionen in eine Dynamic Link Library ausgelagert werden. Diese wird dann von der Game Engine zur Laufzeit als Modul geladen. Um Probleme mit dem Garbage Collector zu vermeiden, muss die Engine den benötigten Speicherplatz für die Partikeldaten reservieren und der DLL als Referenz übergeben. Die DLL stellt wiederum die nachfolgenden Funktionen der Unreal Engine bereit.

openFile(): Beim erfolgreichen Öffnen der als Pfad angegebenen Datei wird ein Handle zurückgegeben, ansonsten gibt diese Funktion einen Null Pointer zurück.

closeFile(): Diese Funktion schließt die zum Handle zugehörige Datei.

getBoundingBox(): Die Funktion gibt ein Float Array mit den Eckpunkten der Boundig Box zurück.

getClippingBox(): Gibt ebenfalls die Eckpunkte der Clipping Box als ein Float Array zurück.

getMaxParticleCount(): Diese Funktion gibt die maximale Anzahl an Partikel in der zur Handle zugehörigen Datei als Integer-Wert zurück.

getNumberOfFrames(): Gibt die Anzahl an Frames als Integer zurück, die die MMPLD-Datei beinhaltet.

getParticleDataAtFrame(): Diese Funktion gibt für den übergebenen Frame die konvertierten Partikeldaten als Float Vektor zurück.

getRequiredMemory(): Gibt für den übergebenen Frame die zu reservierende Größe des Float Vektors und die Anzahl der Partikel zurück.

Die Partikeldaten aus einer MMPLD-Datei müssen vor der Verwendung konvertiert werden, da nicht sichergestellt ist, dass alle Daten im richtigen Format vorliegen. MMPLD-Dateien sind in Frames unterteilt, welche wiederum mehrere Listen beinhalten können. In diesen Listen sind die eigentlichen Partikelwerte gespeichert.

Positionswerte können als Double, Float oder Short vorliegen. Bei Float Werten kann zusätzlich noch der Partikelradius enthalten sein. Farbwerte können in RGB 8, RGB 32, RGBA 8, RGBA 16 oder RGBA 32 gespeichert sein. Alternativ zu Farbwerten können auch Intensitätswerte vorliegen.

Damit in der Unreal Engine Partikel per Sprite Renderer visualisiert werden können, müssen die Positionen, Sprite Größen und Farben inklusive Alphakanal als Float Werte vorliegen. Weshalb in der DLL die Positionswerte immer in Float Werte mit Radius und die Farbwerte bzw. die Intensitätswerte in RGBA 32 konvertiert werden. Nach der Konvertierung werden die Werte aller Listen als ein Vektor an die Game Engine übergeben.

Dort werden die Werte vor der Übernahme in ein TArray in das von Unreal genutzte linkshändige Koordinatensystem, welches als Up-Vektor die Z-Achse nutzt, überführt. Dies geschieht, indem die X-Achse negiert und die Y- und Z-Achse miteinander vertauscht werden. Anschließend werden die Werte zusammen mit der Partikelanzahl in einem Struct namens FMMPLDFrame gespeichert. Zur leichteren Umsetzung lädt der Prototyp alle Frames der ausgewählten MMPLD-Datei in den Arbeitsspeicher.

2.2 Partikel-Rendering

Für die Visualisierung der Partikeldaten boten sich zwei Vorgehensweisen an. Die Implementierung eines speziellen Daten-Interface für das Niagara Partikelsystem oder eine an die existierende Implementierung in MegaMol angelehnte Portierung als Unreal-Actor.

Das zuvor verwendete Cascade Partikelsystem wurde in der Unreal Engine durch das Niagara VFX System ersetzt. Es ermöglicht die Erstellung von visuellen Effekten mit Millionen von Partikeln, die in Echtzeit berechnet und gerendert werden. Um von diesem System und dessen Optimierungen zu profitieren, wurde ein Daten-Interface entwickelt.

2.2.1 Daten-Interface

Das Niagara VFX System besteht aus vier grundlegenden Komponenten:

Systeme Beinhalten alle Komponenten eines Effekts.

Emitter Steuern die Erstellung, das Verhalten und das Aussehen der Partikel.

Module Beinhalten die Logik und Funktionen der Simulation.

Parameter Repräsentieren die Daten innerhalb des Systems.

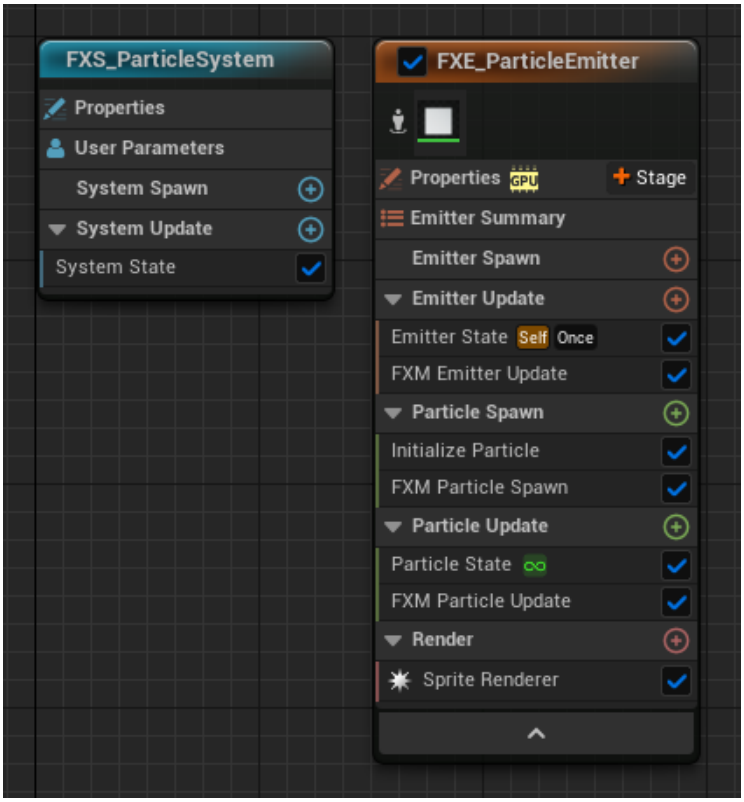

Abbildung 2.1: Aufbau eines Niagara Systems mit einem Emitter und Sprite Renderer.

Niagara Systeme stellen eine Art Container dar, die mehrere Emitter beinhalten können, um einen Effekt innerhalb des Spiels zu ermöglichen. Neben Emittern können Systeme auch User Parameter enthalten, welche zur Laufzeit vom Game Thread manipuliert werden können. Diese User Parameter werden häufig in Modulen als Input Parameter übergeben, um so dynamisch die Simulationen beeinflussen zu können.

In Listing 2.1 wird der im Niagara System verwendete User Parameter dafür genutzt das entsprechende Daten-Interface des jeweiligen Emitters zu finden und ihm seinen MMPLDCache, welcher die zugehörigen Partikeldaten beinhaltet, zuzuweisen.

Des Weiteren können im Niagara System globale Einstellungen, wie beispielsweise das Festlegen einer statischen Bounding Box, vorgenommen werden. Diese Einstellungen gelten dann für alle im System enthaltenen Emitter. Abbildung 2.1 zeigt das als Asset gespeicherte Niagara Partikelsystem.

Niagara Emitter wiederum sind Sammlungen von Modulen und können in drei Abschnitte unterteilt werden. Der orange Abschnitt in Abbildung 2.1 enthält die Eigenschaften des Emitters sowie seine Spawn- und Update-Group. Alle im orangen Abschnitt enthaltenen Module werden generell auf der CPU ausgeführt.

Der grüne Abschnitt enthält die Spawn- und Update-Group der Partikel. Alle Module in diesem Abschnitt werden für jeden Partikel separat aufgerufen. Je nach eingestelltem Sim Target werden die Module auf der CPU oder GPU ausgeführt.

```
void ANiagaraSystemManager::SpawnInstances()
{
  if (MMPLDCaches.IsEmpty())
  {

    for (int32 i = 0; i < InstanceCount; i++)
    {
      MMPLDCaches.Add(NewObject<UMMPLDCache>());
      MMPLDCaches[i]->SetAnimated(Animated);
      MMPLDCaches[i]->Name = FileName + TEXT("_") + FString::FromInt(i);
      MMPLDCaches[i]->NiagaraSystem = UNiagaraFunctionLibrary::SpawnSystemAtLocation(GetWorld
(), NiagaraSystemAsset,
        FVector::ZeroVector, FRotator::ZeroRotator, FVector::OneVector, true, true,
ENCPoolMethod::None, true);
    }

    float Vertices[6];
    getBoundingBox(Handle, Vertices);

    BoundingBox = FBox(FVector(-Vertices[0], Vertices[2], Vertices[1]), FVector(-Vertices[3],
Vertices[5], Vertices[4]));

    ArmLength = CalculateArmLength(BoundingBox.GetSize());
    SpringArm->TargetArmLength = ArmLength;

    Center = BoundingBox.GetCenter();
    SetActorLocation(Center);

    ...

    for (int32 i = 0; i < InstanceCount; i++)
    {
      MMPLDCaches[i]->NiagaraSystem->SetEmitterFixedBounds(TEXT("FXE_ParticleEmitter"),
BoundingBox);
      UNiagaraDataInterfaceMMPLD* MMPLDInterface = Cast<UNiagaraDataInterfaceMMPLD>(
UNiagaraFunctionLibrary::GetDataInterface(UNiagaraDataInterfaceMMPLD::StaticClass(),
MMPLDCaches[i]->NiagaraSystem, TEXT("User.MMPLD")));
      MMPLDInterface->SetMMPLDCache(MMPLDCaches[i]);
    }
  }
}
```

Listing 2.1: Funktion zur Erstellung der Niagara Systeme, MMPLDCache Objekte sowie deren Zuweisung zu den Daten-Interfaces.

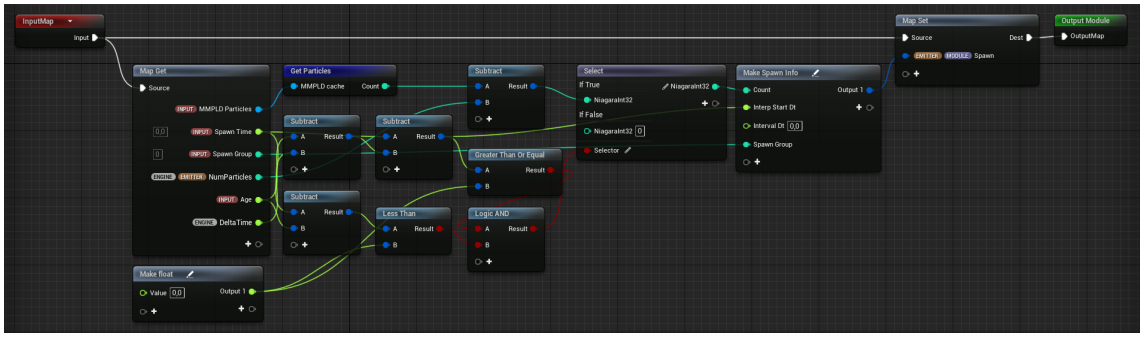

Abbildung 2.2: Aufbau des FXM Emitter Update Moduls.

Alle Module, die sich in einer Spawn-Group befinden, werden einmalig bei der Erstellung ausgeführt. Module, die wiederum in der Update-Group sind, werden bei jedem Simulationsschritt ausgeführt. Die generelle Ausführungsreihenfolge der Module ist von oben nach unten. Der in Abbildung 2.1 gezeigt Emitter enthält drei custom erstellte Module.

FXM Emitter Update Ist eine vereinfachte Version des Spawn Burst Instantaneous und erstellt die Spawn Info für die Particle Spawn-Group. Hierfür fragt das Modul die benötigte Partikelanzahl des aktuellen Frames mithilfe des im Input Parameter verlinkten Daten-Interface ab. Anschließend wird, wie in Abbildung 2.2 zu sehen, die Differenz mit der von der Engine übermittelten Anzahl an bereits existierenden Partikel berechnet.

Mit Hilfe der Delta Time, Spawn Time und des Loop Age wird überprüft, ob zum aktuellen Zeitpunkt Partikel erstellt werden müssen. Ist dies der Fall, so wird die zuvor berechnete Partikelanzahl an den Make Spawn Info Knoten übermittelt, andernfalls wird immer Null übergeben.

FXM Particle Spawn Wie in Abbildung 2.3 zu sehen ist, fragt das Modul die initialen Farb-, Positions- und Größenwerte der jeweiligen Partikel mit Hilfe der UniqueID und dem Daten-Interface ab.

FXM Particle Update Aktualisiert die jeweiligen Partikelwerte vorausgesetzt, die in Abbildung 2.4 übergebene UniqueID ist kleiner als die Partikelanzahl des aktuelle zu visualisierenden Frames. Andernfalls werden bis auf den Alphakanal alle bisherigen Werte übernommen.

Anstatt den Alphakanal auf Null zu setzen, könnte auch die Lifetime reduziert werden, damit das Partikelsystem die betroffenen Partikel zerstört. Dies wird aber nicht gemacht, da die UniqueIDs nicht wieder freigegeben werden und so die darauffolgenden Partikel eine ID außerhalb des Wertebereichs erhalten würden.

Die mit dem Input Parameter „MMPLD Particles“ verbundenen Knoten in Abbildung 2.2 bis 2.4 werden im Listing 2.2 definiert. Abhängig vom Sim Target werden beim Aufruf der Knoten entweder die in Listing 2.3 oder in Listing 2.4 implementierten Funktionen ausgeführt.

Die mit dem Graph Editor erstellten Module werden von der Unreal Engine in High Level Shading Language (HLSL) Code übersetzt. Im Falle eines CPU-Emitters wird dieser Code auf einer für Single Instruction Multiple Data (SIMD) optimierten virtuellen Maschine ausgeführt.

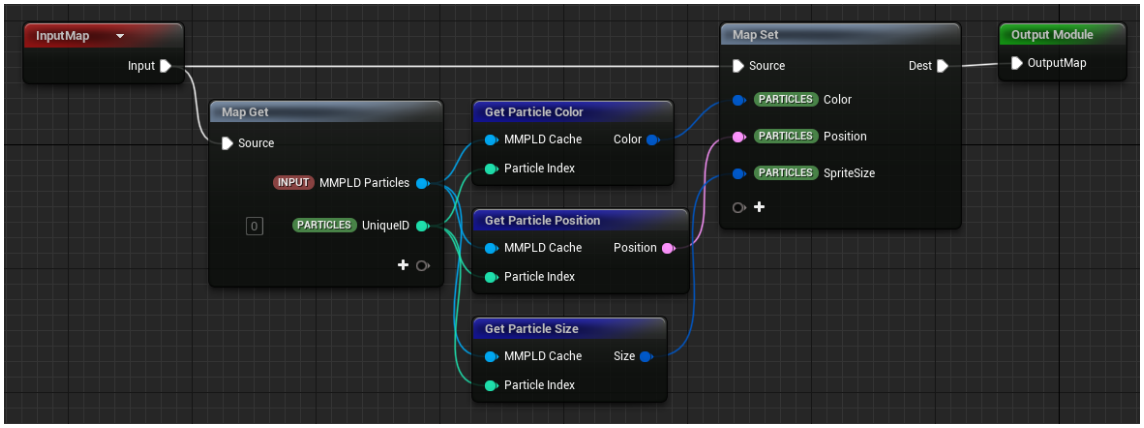

Abbildung 2.3: Aufbau des FXM Particle Spawn Moduls.

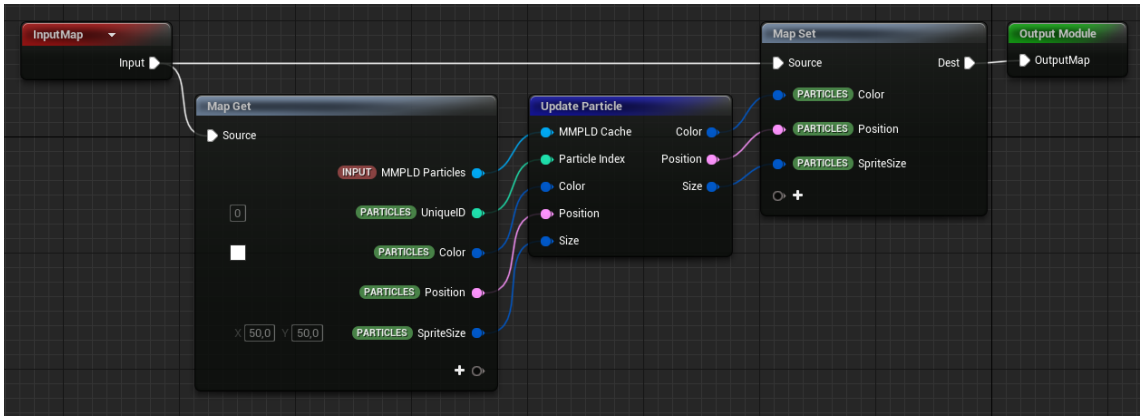

Abbildung 2.4: Aufbau des FXM Particle Update Moduls.

```
void UNiagaraDataInterfaceMMPLD::GetFunctions(TArray<FNiagaraFunctionSignature>& OutFunctions)
{
  {
    // GetParticleColor
    FNiagaraFunctionSignature Sig;
    Sig.Name = GetParticleColorName;
    Sig.bMemberFunction = true;
    Sig.AddInput(FNiagaraVariable(FNiagaraTypeDefinition(GetClass()), TEXT("MMPLD Cache")));
    Sig.AddInput(FNiagaraVariable(FNiagaraTypeDefinition::GetIntDef(), TEXT("Particle Index"))
);
    Sig.AddOutput(FNiagaraVariable(FNiagaraTypeDefinition::GetColorDef(), TEXT("Color")));
    Sig.SetDescription(LOCTEXT("GetParticleColorNameFunctionDescription", "Returns the color
value for a given particle index."));
    OutFunctions.Add(Sig);
  }

  ...
}
```

Listing 2.2: Definition der Daten-Interface Funktionsknoten.

```
void UNiagaraDataInterfaceMMPLD::GetParticleColorVM(FVectorVMExternalFunctionContext& Context)
{
  VectorVM::FUserPtrHandler<FNDIMMPLDInstanceData_GameThread> InstanceData(Context);
  FNDIInputParam<int> ParticleIndex(Context);
  FNDIOutputParam<FLinearColor> OutColor(Context);

  FLinearColor Color;

  for (int32 i = 0; i < Context.GetNumInstances(); i++)
  {
    int Index = ParticleIndex.GetAndAdvance();

    if (Index >= 0 && Index < InstanceData->Particles)
    {
      Color = FLinearColor(InstanceData->ParticleData[(Index * 8) + 4], InstanceData->
ParticleData[(Index * 8) + 5],
        InstanceData->ParticleData[(Index * 8) + 6], InstanceData->ParticleData[(Index * 8) +
7]);
    }
    else
    {
      Color = FLinearColor::Transparent;
    }

    OutColor.SetAndAdvance(Color);
  }
}
```

Listing 2.3: Definition der CPU-Emitter Funktionen.

Trotz ihrer Optimierung ist ihre Performanz signifikant schlechter als eine direkte Ausführung des HLSL Codes auf der GPU, weshalb CPU-Emitter nicht für Visualisierungen mit mehr als 1 Millionen Partikel verwendet werden sollten.

Die GPU-Emitter sind von der Unreal Engine standardmäßig auf 2 Millionen Partikel pro Frame limitiert. Dieses Limit kann zwar per Console-Variable erhöht werden, aber dies kann zu nicht erwünschten Nebeneffekten führen, wie man in Abbildung 2.5 sieht.

Da die Ursache dieses Problems nicht im zeitlichen Rahmen dieser Arbeit gefunden werden konnte wurde ein Mechanismus implementiert der anhand der maximalen Partikelanzahl in der MMPLD-Datei die benötigte Anzahl an Niagara Systemen berechnet. Diese dann im aktuellen Level spawnt und die Partikeldaten auf die korrespondierende Anzahl an MMPLDCache Objekten aufteilt.

Im letzten Abschnitt eines Emitters befinden sich die Niagara Renderer. Zur Visualisierung der zuvor berechneten Werte stehen ein Component, Geometry Cache, Light, Mesh, Ribbon oder Sprite Renderer zur Auswahl.

```
bool UNiagaraDataInterfaceMMPLD::GetFunctionHLSL(const FNiagaraDataInterfaceGPUParamInfo&
ParamInfo, const FNiagaraDataInterfaceGeneratedFunction& FunctionInfo, int
FunctionInstanceIndex, FString& OutHLSL)
{
  FString ParticlesVar = ParticlesBaseName + ParamInfo.DataInterfaceHLSLSymbol;
  FString ParticleDataVar = ParticleDataBufferBaseName + ParamInfo.DataInterfaceHLSLSymbol;

  if (FunctionInfo.DefinitionName == GetParticleColorName)
  {
    // GetParticleColor(int In_ParticleIndex, out float4 Out_Color)
    OutHLSL += TEXT("void ") + FunctionInfo.InstanceName + TEXT("(int In_ParticleIndex, out
float4 Out_Color)\n");
    OutHLSL += TEXT("{\n");
    OutHLSL += TEXT("\tif (In_ParticleIndex >= 0 && In_ParticleIndex < ") + ParticlesVar +
TEXT(")");
    OutHLSL += TEXT("\t{\n");
    OutHLSL += TEXT("\t\tOut_Color.x = ") + ParticleDataVar + TEXT("[(In_ParticleIndex * 8) +
4];\n");  // R
    OutHLSL += TEXT("\t\tOut_Color.y = ") + ParticleDataVar + TEXT("[(In_ParticleIndex * 8) +
5];\n");  // G
    OutHLSL += TEXT("\t\tOut_Color.z = ") + ParticleDataVar + TEXT("[(In_ParticleIndex * 8) +
6];\n");  // B
    OutHLSL += TEXT("\t\tOut_Color.w = ") + ParticleDataVar + TEXT("[(In_ParticleIndex * 8) +
7];\n");  // A
    OutHLSL += TEXT("\t}\n");
    OutHLSL += TEXT("\telse");
    OutHLSL += TEXT("\t{\n");
    OutHLSL += TEXT("\t\tOut_Color.x = 0.0f;\n");
    OutHLSL += TEXT("\t\tOut_Color.y = 0.0f;\n");
    OutHLSL += TEXT("\t\tOut_Color.z = 0.0f;\n");
    OutHLSL += TEXT("\t\tOut_Color.w = 0.0f;\n");
    OutHLSL += TEXT("\t}\n");
    OutHLSL += TEXT("}\n");
    return true;
  }

  ...

  else
  {
    return false;
  }
}
```

Listing 2.4: Definition der GPU-Emitter Funktionen.

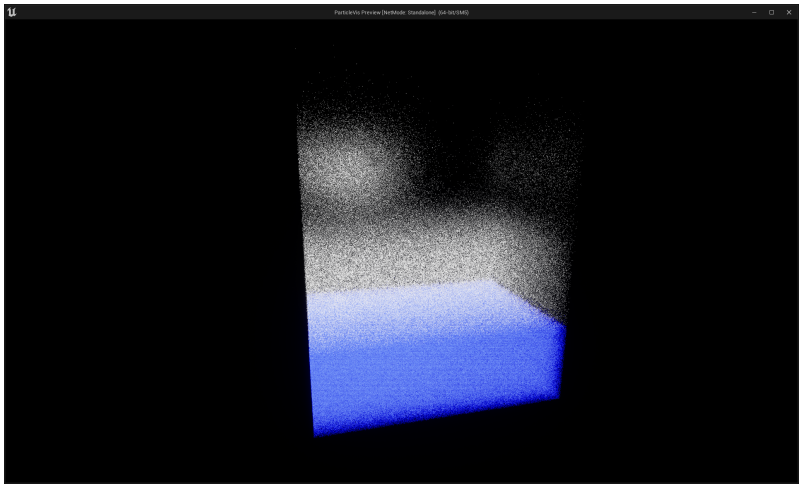

Abbildung 2.5: Fehlerhafte Darstellung bei der Verwendung eines Emitters bei mehr als 2 Millionen Partikeln.

2.2.2 Kugelglyphen

Die raycasting basierte Darstellung der Kugelglyphen konnte auch ohne Implementierung eines Custom Renderers realisiert werden. Zur Visualisierung der Partikeldaten wird ein Sprite Renderer verwendet, dessen standardmäßiges Material durch das in Abbildung 2.6 gezeigte Material ersetzt wurde.

Das erstellte Material erhält vom Niagara Emitter die aktuelle Partikelfarbe, Position und den Partikelradius. Der RayTracedSphere Knoten von Unreal berechnet dann anhand der Partikelposition, dem Radius, der Kameraposition und dem normalisierten Richtungsvektor der Kamera die Oberflächennormale der Kugelglyphe.

Des Weiteren überprüft der Knoten, ob der aktuelle Sprite-Pixel ein Schnittpunkt der Glype mit dem Sichtstrahl ist. Das Ergebnis wird mit dem Alphakanal der Partikelfarbe multipliziert und an die Opacity Mask übergeben. Da die Unreal Engine in Version 5.1.0 bei transparenten Objekten keine globale Beleuchtung unterstützt, muss bei dem Sprite-Material als Blend Mode „Masked“ verwendet werden.

Diese Einstellung hat zufolge, dass Pixel mit einem kleineren Alpha-Wert als 0,33 vom Sprite Renderer verworfen und andernfalls in voller Intensität gerendert werden. Der so im Material Editor erstellte Graph wird im Hintergrund von der Unreal Engine zu HLSL Code übersetzt.

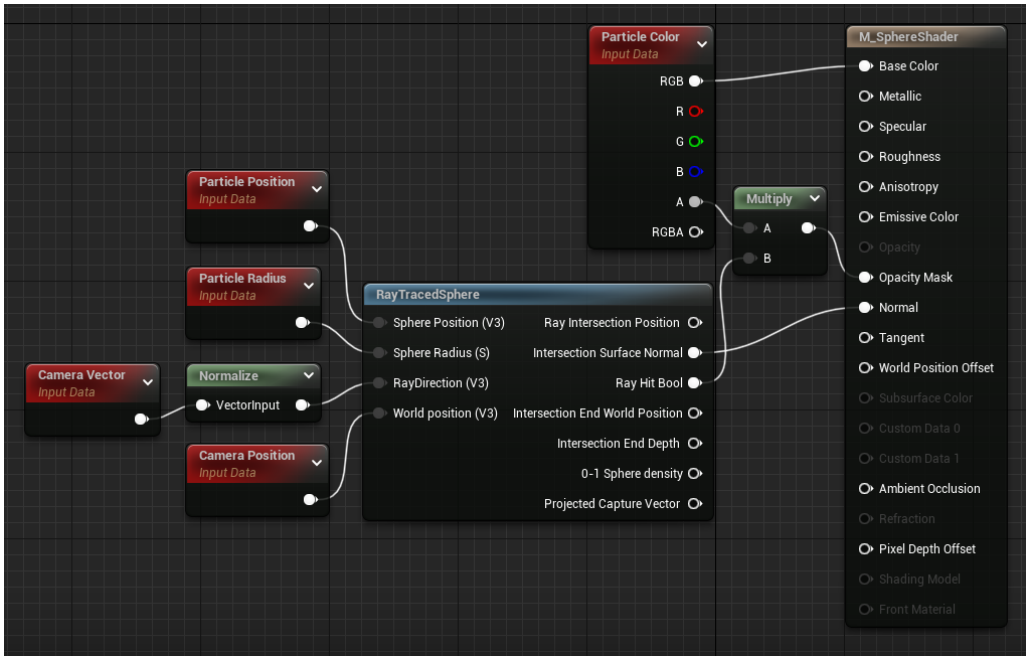

Abbildung 2.6: Aufbau eines im Material Editor erstellten Sphere Shader.

3 Evaluation

In diesem Kapitel geht es um die Evaluation des in dieser Arbeit entstandenen Prototyps. Hierfür wurden mit mehreren Datensätzen Performanztests durchgeführt. Der Aufbau dieser wird im Folgenden beschrieben und die erhaltenen Messergebnisse im Kontext der Forschungsthematik betrachtet.

3.1 Aufbau der Performanztests

Um die Renderleistung des entwickelten Visualisierungstools möglichst realistisch zu testen, wurden drei Kamerafahrten erstellt. Hierfür wurden zwei Orbit-Fahrten und eine direkte Fahrt durch den Datensatz hindurch im Unreal Engine Projekt implementiert. Bei den Orbit-Fahrten umkreist die Kamera die visualisierten Partikeldaten in einem Zeitraum von 10 Sekunden zweimal.

Um diese Fahrten auch in MegaMol reproduzieren zu können, wurden drei Lua-Skripts erstellt. Diese werden zum Starten nach dem Laden der Testumgebung und des jeweiligen Datensatzes per Drag and Drop in das Projektfenster gezogen.

Als Zentrum der Kamerafahrten diente in beiden Fällen der Mittelpunkt der Bounding Box. Neben der Framerate wurden auch die Ladezeiten und der Arbeitsspeicherverbrauch gemessen.

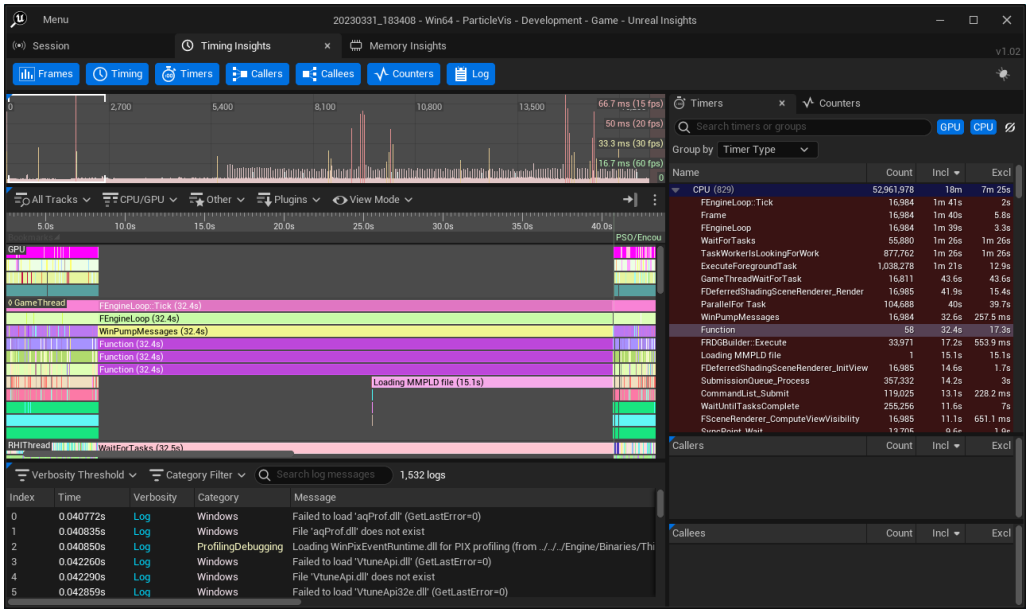

Abbildung 3.1: Das Profiling-Tool Unreal Insights im Viewer-Modus.

Datensatz	1KX2	3OAA	lasercross	exp2mill	n2h2_0000010000
Frames	1	1	401	150	10
Partikel	1.174	98.928	562.500	2.000.000	7.860.108

Tabelle 3.1: Die während der Performanztests visualisierten Datensätze.

Zur Aufzeichnung der Ladezeiten und des Speicherverbrauchs wurde beim Prototypen das Profiling-Tool Unreal Insights, welches in Abbildung 3.1 zusehen ist, verwendet. Für die Messung der Framerate wurden die Engine eigenen Methoden „StartFPSChart“ und „StopFPSChart“ genutzt. Damit dies möglich war, wurde das Visualisierungstool mit Development Einstellungen gebaut und anschließend per Eingabeaufforderung mit dem Befehl „ParticleVis.exe -trace=default,memory implies -llm“ gestartet.

Um die Ladezeiten in MegaMol erfassen zu können, wurde der Code des MMPLDDataSource angepasst und die benötigte Zeit auf der Konsole ausgegeben. Da es für die Erfassung des Arbeitsspeicherverbrauch beim Prototyping Framework keine direkte Möglichkeit gab, wurde hierfür die Anwendung VMMap von Microsoft Sysinternals verwendet. Die benötigte Renderzeit der Frames wurde direkt in den Lua-Skripten erfasst und ebenfalls wie bei der Unreal Engine in CSV-Dateien geschrieben.

Um zu vermeiden, dass die Performanz durch die Hardware limitiert wird, wurden alle Tests auf einem Windows 10 Desktop PC mit einem AMD Ryzen Threadripper PRO 3995WX, mit 265 GB Arbeitsspeicher und einer NVIDIA RTX A6000 ausgeführt.

Als Testdatenstäze wurden die in Tabelle 3.1 aufgelisteten Dateien verwendet. Die Dateien 1KX2 und 3OAA sind Moleküle aus der RCSB Proteindatenbank und ihre Namen repräsentieren ihre PDB-ID. Bei Datensätzen mit Zeitreihen wurde zusätzlich der letzte Frame exportiert und separat getestet, um den reinen Einfluss der Partikelanzahl auf die Renderleistung festzustellen.

3.2 Ladezeiten

Die während den Performanztests gemessenen Wert für das Laden der unterschiedlichen Datensätze werden in Abbildung 3.2 und in Abbildung 3.3 als Balkendiagramme dargestellt. Die Abbildung 3.2 verwendet zur Darstellung der Ladezeiten eine logarithmische Skalierung mit einer 10 Basis damit die Werte des Prototyping Framework MegaMol sichtbar werden.

Die Ladezeiten des in dieser Arbeit entwickelten Prototypen sind in allen Fällen bis auf einen signifikant schlechter. Dies wird darauf zurückzuführen sein, dass die DLL mehrfach über die Frames und Listen der Dateien iterieren muss. Zum einen, um die maximale Anzahl der Partikel im Datensatz zu ermitteln. Zum anderen, um die Partikelwerte in ein einheitliches, von der Game Engine erwartetes Format zu konvertieren.

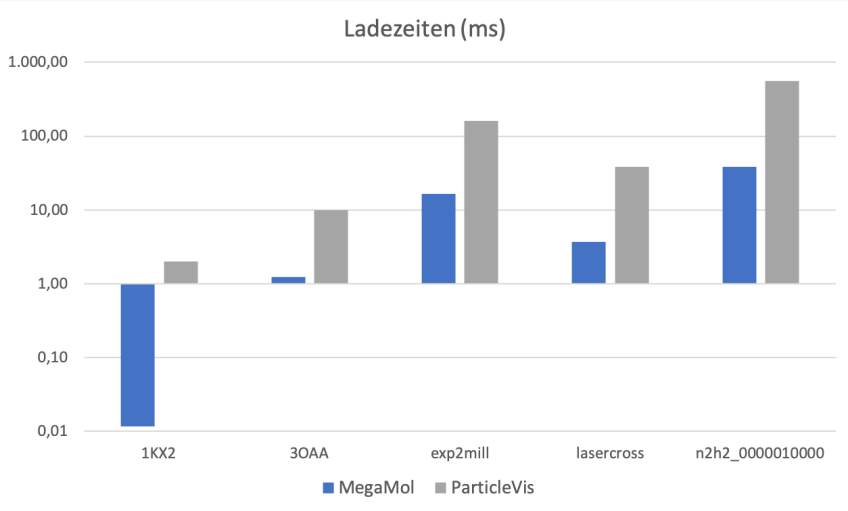

Abbildung 3.2: Die zum Laden der Datensätze benötigte Zeit in Millisekunden.

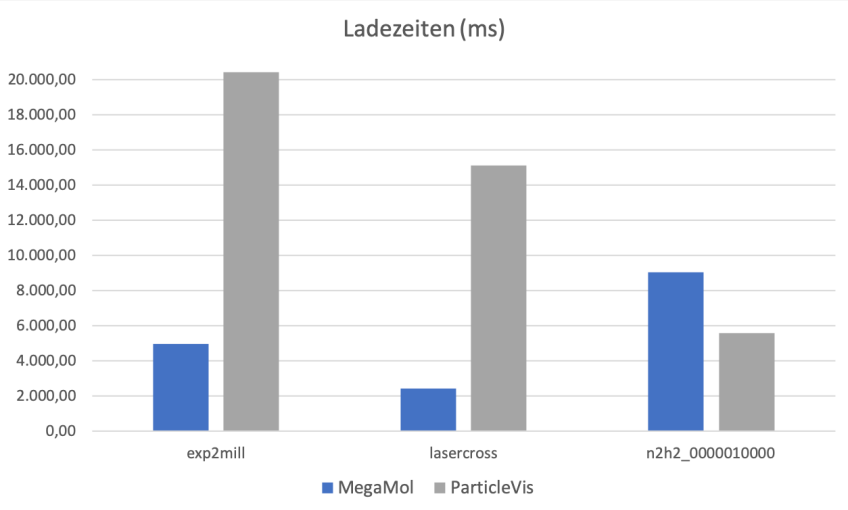

Abbildung 3.3: Die benötigte Ladezeit für die komplette Zeitreihe des jeweiligen Datensatzes in Millisekunden.

3.3 Speicherverbrauch

Wie bei Kingsley et al. [KBL+19] braucht auch hier das mit der Game Engine erstellte Visualisierungstool deutlich mehr Arbeitsspeicher als die Referenzanwendung. In Abbildung 3.4 und Abbildung 3.5 sieht man eindrucksvoll, dass der Prototyp beim bloßen Öffnen der Testumgebung bereits das Zwölffache an Speicherplatz benötigt.

Das liegt daran, dass die Anwendung mit Funktionen wie beispielsweise einer Sound Engine ausgestattet wird, obwohl diese zur bloßen Visualisierung von Partikeldaten nicht benötigt wird. In Abbildung 3.5 wurden ausschließlich Datensätze mit Zeitreihen von den Anwendungen visualisiert. Hier fällt aufgrund ihrer Größe der ursprüngliche Speicherverbrauch nicht ganz so schwer ins Gewicht. Aber auch hier benötigt MegaMol maximal halb so viel Speicherplatz wie der Prototyp.

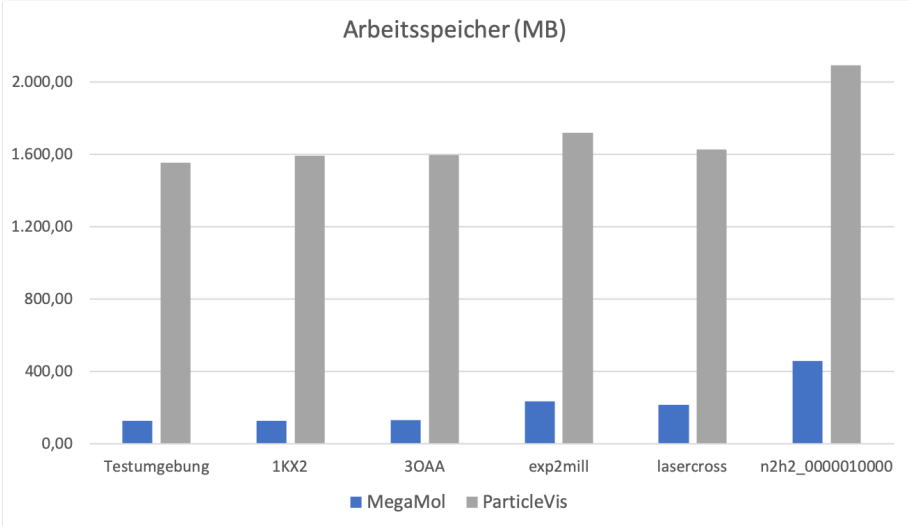

Abbildung 3.4: Der zur Darstellung der Datensätze verbrauchte Arbeitsspeicher in Megabyte.

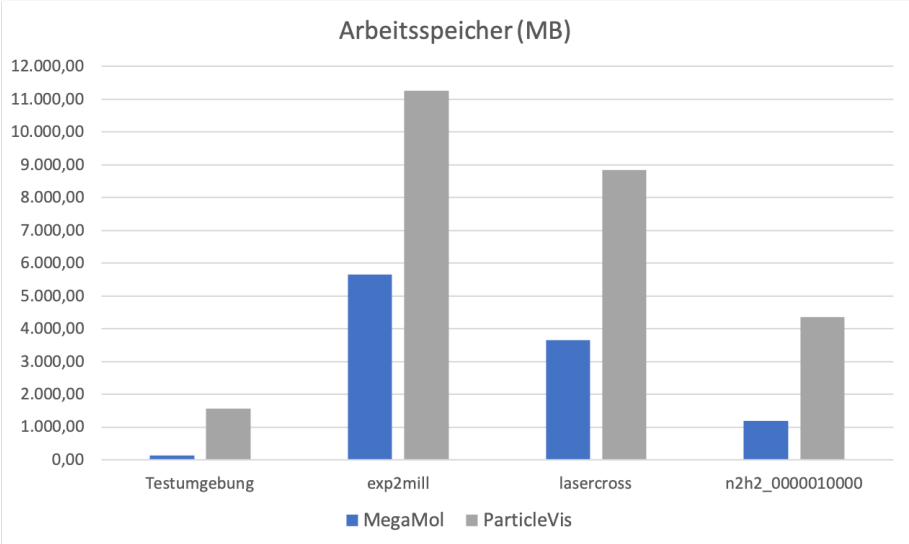

Abbildung 3.5: Der verbrauchte Arbeitsspeicher zur Visualisierung der animierten Datensätze in Megabyte.

Der erhöhte Speicherverbrauch wird hauptsächlich durch die Konvertierung der Partikeldaten in ein einheitliches Format verursacht. Durch Sie werden zahlreiche Duplikate wie beispielsweise Partikelfarbe oder Partikelradius erstellt und im Arbeitsspeicher zwischengespeichert. Des Weiteren muss für die virtuelle Maschine des Niagara Systems immer eine Kopie des aktuellen Frame bereitgestellt werden, was weiteren Speicherplatz benötigt.

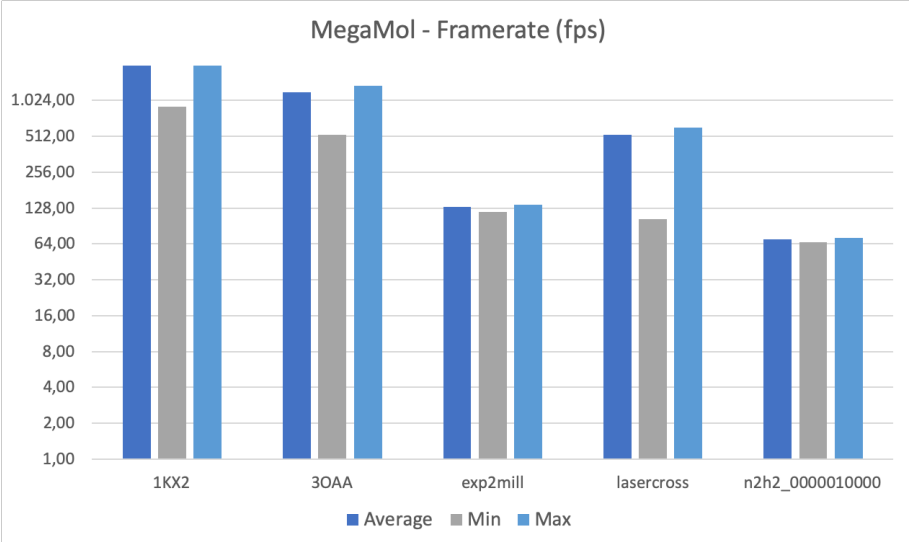

Abbildung 3.6: Die von MegaMol erzielte Framerate beim Durchfliegen der Datensätze.

3.4 Performanz

Die Ergebnisse der Performanztest können in Abbildung 3.6 bis 3.17 ebenfalls als Balkendiagramme betrachtet werden. Alle Diagramme nutzen eine logarithmische Skalierung zur Veranschaulichung der von den Visualisierungstools erreichten Framerates.

Bei der Betrachtung der Abbildung 3.7, 3.9, 3.11, 3.13, 3.15 und 3.17 fällt auf das der in dieser Arbeit entwickelte Prototyp bei allen durchgeführten Tests nie mehr als 410 FPS erreicht. Dies erweckt den Anschein, als wäre die Anwendung entweder durch Software oder durch Hardware limitiert.

Eine Limitierung der Renderleistung aufgrund der verwendeten Hardware kann aber ausgeschlossen werden, da MegaMol bei der Visualisierung der Moleküle 1KX2 und 3OAA deutlich über 1000 FPS erreicht hat.

Um eine softwareseitige Limitierung der Framerate durch die Unreal Engine auszuschließen, wurde die Option der Framerate-Glättung in den Projekteinstellungen deaktiviert und das standardmäßige Limit von 120 FPS auf 1000 FPS erhöht. Weshalb die Begrenzung durch etwas anderes verursacht werden muss.

Ebenfalls auffällig ist, dass der Prototyp eine deutlich bessere Renderleistung bei Datensätzen ohne Zeitreihen aufweist. Dies wurde erreicht, indem beim Laden der MMPLD-Dateien überprüft wird, wie viele Frames im Datensatz vorhanden sind. Sollte nur ein Frame visualisiert werden müssen, so wird das Niagara System nach der erfolgreichen Erstellung der Partikel pausiert und so Aktualisierungen durch unnötige Simulationsschritte vermieden.

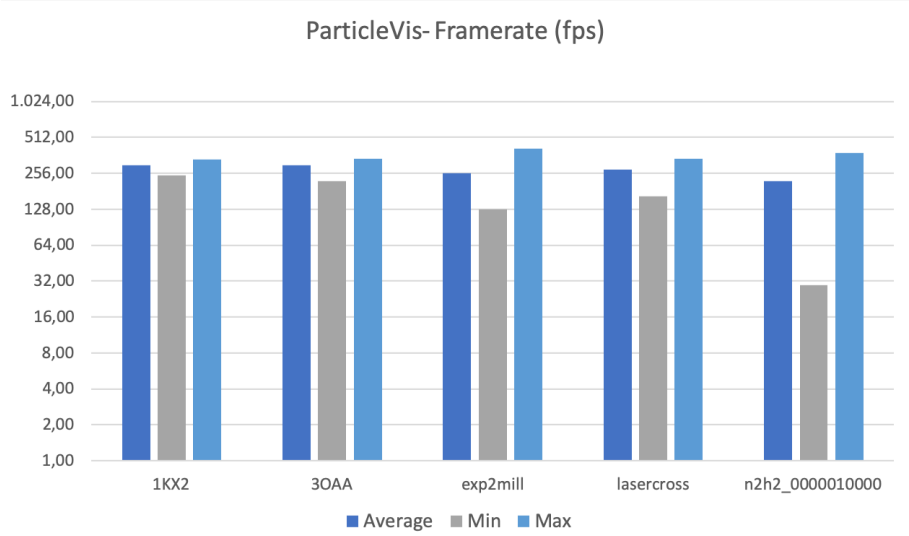

Abbildung 3.7: Die vom Prototyp erzielte Framerate beim Durchfliegen der Datensätze.

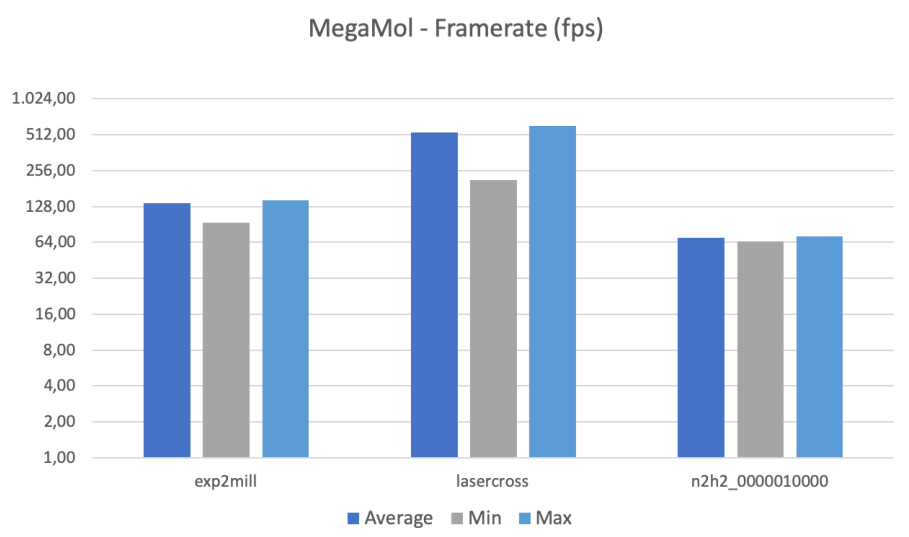

Abbildung 3.8: Die beim Durchfliegen der animierten Datensätze erreichte Framerate.

Wie zu erwarten war, sinkt bei beiden Visualisierungstools die Renderleistung mit steigender Anzahl an Partikeln. Aber bei Datensätzen mit Zeitreihen und mehr als zwei Millionen Partikeln erreicht der entwickelte Prototyp durchschnittliche Framerates, die mit dem Prototyping Framework MegaMol mithalten können oder sogar besser sind. Abbildung 3.9 und 3.17 zeigen aber auch, dass die Differenz zwischen minimalen und maximal erreichten Frames zunimmt.

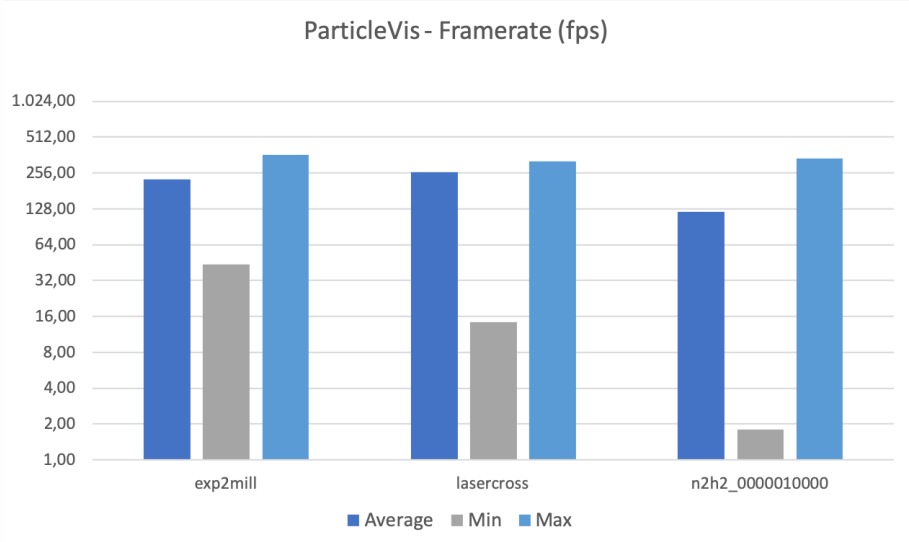

Abbildung 3.9: Die vom Prototyp erreichte Framerate beim Durchfliegen der animierten Datensätze.

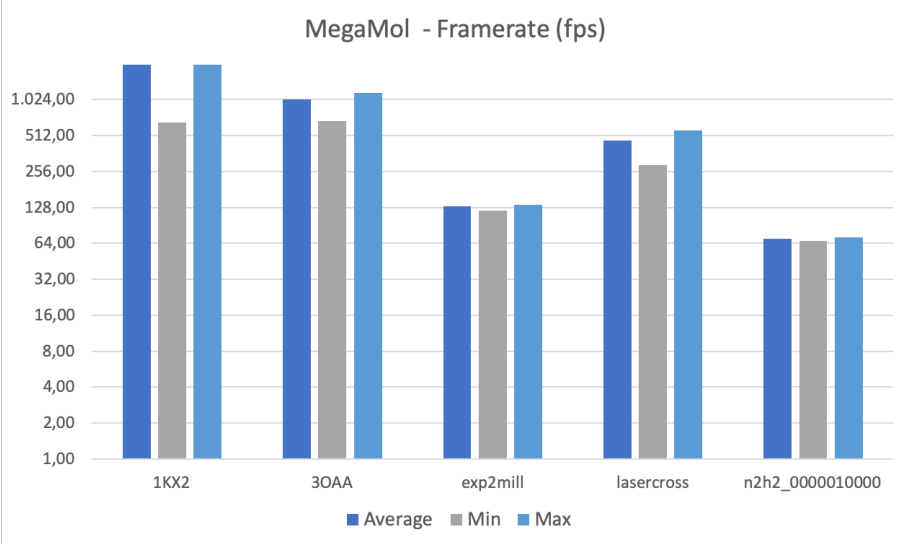

Abbildung 3.10: Die von MegaMol bei der Rotation um den Datensatz erreichte Framerate.

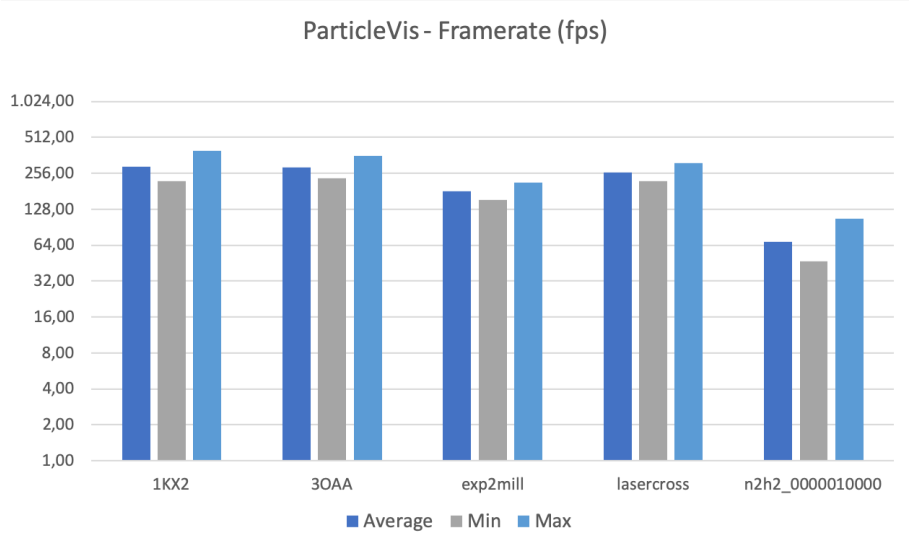

Abbildung 3.11: Die bei der Rotation um den Datensatz erreichte Framerate.

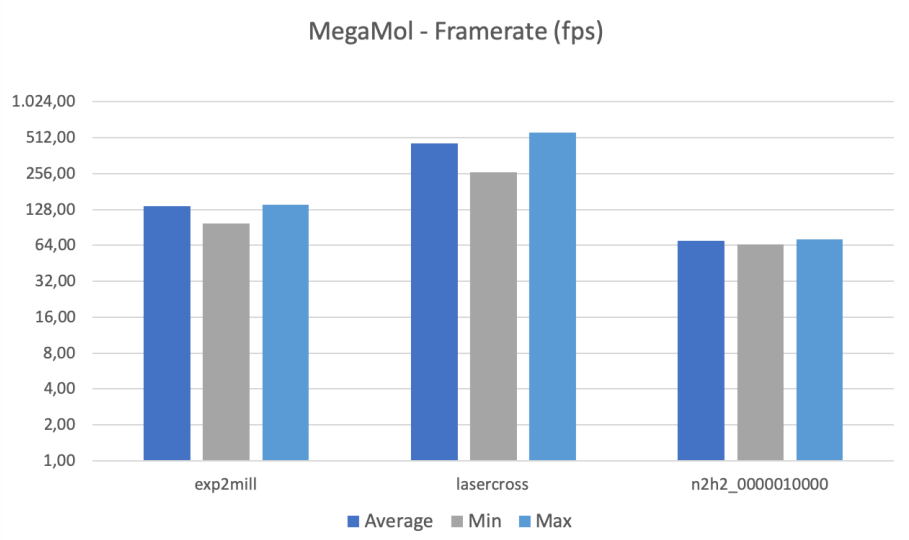

Abbildung 3.12: Die von MegaMol erreichte Framerate bei der Rotation um die animierten Datensätze.

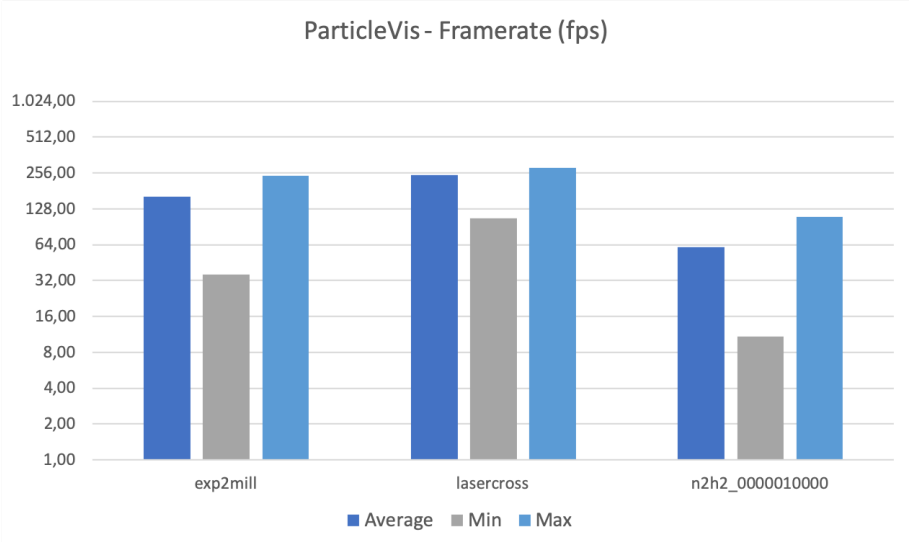

Abbildung 3.13: Die vom Prototyp erreichte Framerate bei der Rotation um die animierten Datensätze.

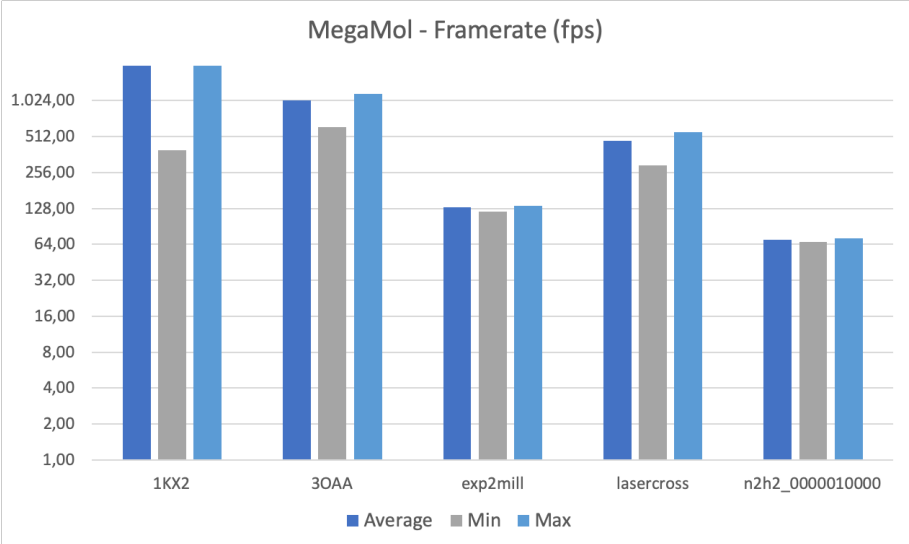

Abbildung 3.14: Die erreichte Framerate bei der Rotation um den Up-Vektor.

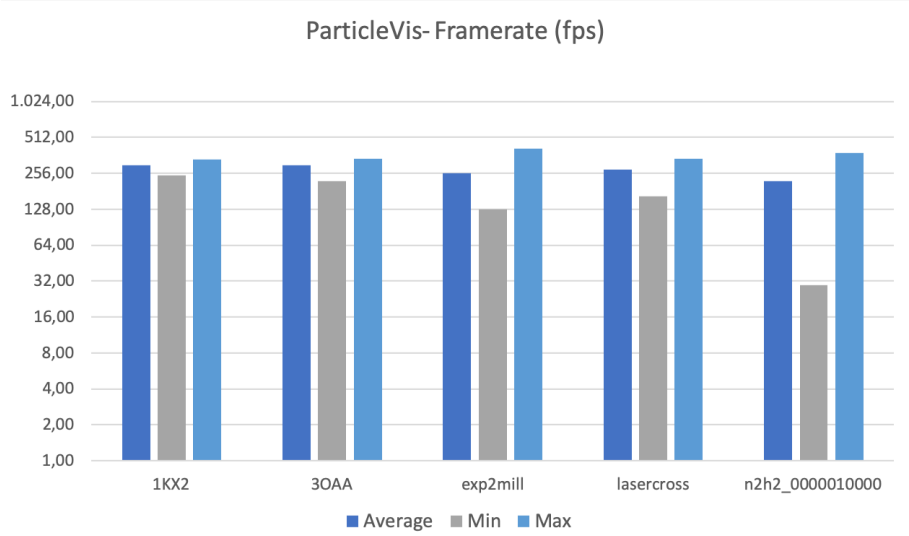

Abbildung 3.15: Die vom Prototyp erreichte Framerate bei der Rotation um den Up-Vektor.

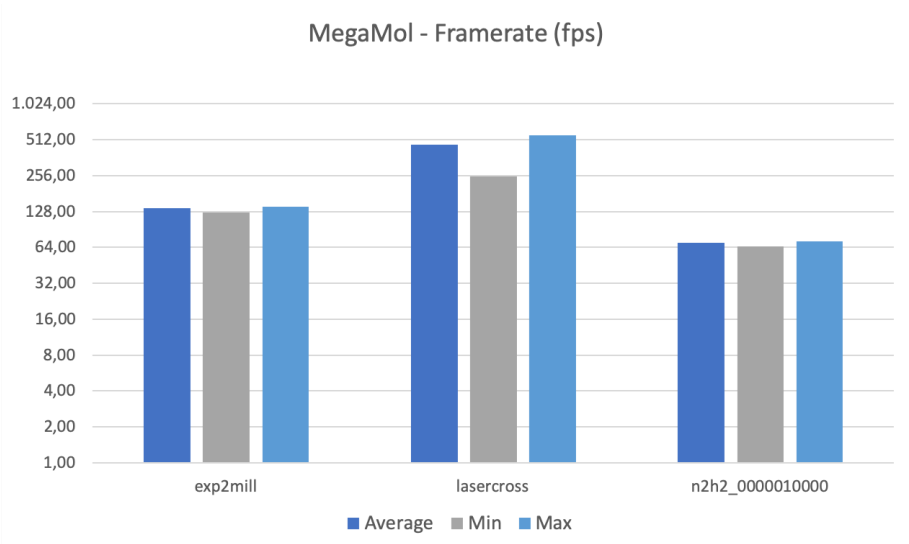

Abbildung 3.16: Die von MegaMol erreichte Framerate bei der Darstellung von animierten Partikeldaten und gleichzeitiger Rotation um den Up-Vektor.

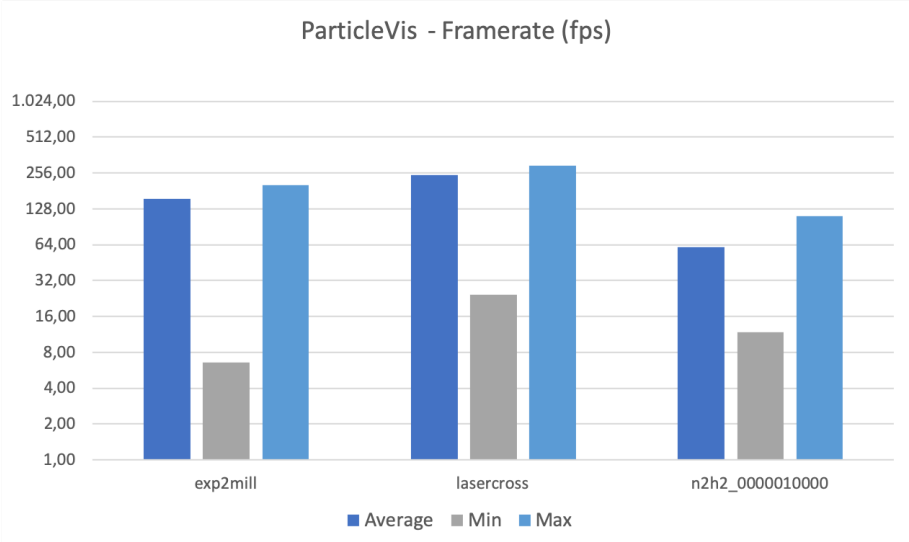

Abbildung 3.17: Die vom Prototyp erreichte Framerate bei der Visualisierung von animierten Partikeldaten und gleichzeitiger Rotation um den Up-Vektor.

4 Zusammenfassung und Ausblick

In diesem abschließenden Kapitel werden nochmals die wichtigsten Erkenntnisse dieser Arbeit thematisiert und ein Ausblick auf die zukünftige Forschung gegeben.

Zusammenfassung

Nach anfänglichen Problemen wurde in dieser Arbeit mithilfe der Unreal Engine ein leistungsstarkes Visualisierungstool entwickelt, das bei Datensätzen mit mehreren Millionen Partikeln immer noch eine gute Performanz bietet.

Durch das Implementieren eines Data-Interface für MMPLD-Dateien wurde ermöglicht das Visualisierungsmethoden wie beispielsweise dem GPU-beschleunigtem Glyphen-Raycasting auch ohne größeren Programmieraufwand durch die Graph Editors hinzugefügt und getestet werden können. Damit wird der entwickelte Prototyp auch für nicht programmier-affine Nutzerinnen und Nutzer leicht erweiter- und modifizierbar. Was die Entwicklung neuartiger sowie effizienter Visualisierungen begünstigen sollte.

Bei der Durchführung der Performanztests wurden ein erheblicher Arbeitsspeicherverbrauch, eine erhöhte Ladezeit und eine deutlich instabilere Framerate bei Datensätzen mit Zeitreihen im Vergleich zum Prototyping Framework MegaMol festgestellt. Trotz der noch bestehenden Probleme zeigt die Arbeit das Potenzial von Game Engines für die wissenschaftliche Visualisierung auf.

Ausblick

Die noch vorhandenen Kinderkrankheiten des in dieser Arbeit entstandenen Prototypen könnten bei der weiteren Entwicklung behoben werden. Durch die Verwendung der zahlreichen Optimierungsoptionen, die die Unreal Engine bietet, sollten mit dem Visualisierungstool noch deutlich bessere Performanzwerte erreicht werden können.

Neben der raycasting-basierten Darstellung von Partikeldaten könnten noch weitere Visualisierungsmethoden wie die HyperBalls-Darstellung aus [KKF+17] implementiert und getestet werden.

Alternativ zum Daten-Interface könnte auch eine an die existierende Implementierung in MegaMol angelehnte Portierung als Unreal-Actor entwickelt werden oder das mit Unreal Engine 5.2 für das Niagara Partikelsystem erscheinende Simulation Caching zur Visualisierung von MMPLD-Dateien verwendet werden.

Des Weiteren sollte die Aussagekraft der Ergebnisse dieser Arbeit durch die Durchführung weiterer Performanztests mit größeren Datensätzen überprüft werden und so die aktuellen Limits des Niagara Partikelsystems untersucht werden.

Literaturverzeichnis

[GBS+18] T. D. Goddard, A. A. Brilliant, T. L. Skillman, S. Vergenz, J. Tyrwhitt-Drake, E. C. Meng, T. E. Ferrin. „Molecular Visualization on the Holodeck“. en. In: *Journal of Molecular Biology* 430.21 (Okt. 2018), S. 3982–3996. ISSN: 0022-2836. DOI: 10.1016/j.jmb.2018.06.040. URL: https://www.sciencedirect.com/science/article/pii/S002228361830696X (besucht am 06. 04. 2023) (zitiert auf S. 15).

[GKM+15] S. Grottel, M. Krone, C. Müller, G. Reina, T. Ertl. „MegaMol—A Prototyping Framework for Particle-Based Visualization“. In: *IEEE Transactions on Visualization and Computer Graphics* 21.2 (Feb. 2015). Conference Name: IEEE Transactions on Visualization and Computer Graphics, S. 201–214. ISSN: 1941-0506. DOI: 10.1109/TVCG.2014.2350479 (zitiert auf S. 14).

[KBL+19] L. J. Kingsley, V. Brunet, G. Lelais, S. McCloskey, K. Milliken, E. Leija, S. R. Fuhs, K. Wang, E. Zhou, G. Spraggon. „Development of a virtual reality platform for effective communication of structural data in drug discovery“. en. In: *Journal of Molecular Graphics and Modelling* 89 (Juni 2019), S. 234–241. ISSN: 1093-3263. DOI: 10.1016/j.jmgm.2019.03.010. URL: https://www.sciencedirect.com/science/article/pii/S1093326318303929 (besucht am 06. 04. 2023) (zitiert auf S. 15, 29).

[KKF+17] B. Kozlíková, M. Krone, M. Falk, N. Lindow, M. Baaden, D. Baum, I. Viola, J. Parulek, H.-C. Hege. „Visualization of Biomolecular Structures: State of the Art Revisited“. en. In: *Computer Graphics Forum* 36.8 (2017). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13072, S. 178–204. ISSN: 1467-8659. DOI: 10.1111/cgf.13072. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13072 (besucht am 05. 03. 2023) (zitiert auf S. 14, 15, 39).

[LTD+13] Z. Lv, A. Tek, F. Da Silva, C. Empereur-mot, M. Chavent, M. Baaden. „Game On, Science - How Video Game Technology May Help Biologists Tackle Visualization Challenges“. en. In: *PLoS ONE* 8.3 (März 2013). Hrsg. von P. Taylor, e57990. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0057990. URL: https://dx.plos.org/10.1371/journal.pone.0057990 (besucht am 18. 02. 2023) (zitiert auf S. 13, 14).

[WMW+18] M. Wiebrands, C. J. Malajczuk, A. J. Woods, A. L. Rohl, R. L. Mancera. „Molecular Dynamics Visualization (MDV): Stereoscopic 3D Display of Biomolecular Structure and Interactions Using the Unity Game Engine“. en. In: *Journal of Integrative Bioinformatics* 15.2 (Juni 2018). Publisher: De Gruyter. ISSN: 1613-4516. DOI: 10.1515/jib-2018-0010. URL: https://www.degruyter.com/document/doi/10.1515/jib-2018-0010/html (besucht am 09. 03. 2023) (zitiert auf S. 15).

[WWB+14] I. Wald, S. Woop, C. Benthin, G. S. Johnson, M. Ernst. „Embree: a kernel framework for efficient CPU ray tracing“. en. In: *ACM Transactions on Graphics* 33.4 (Juli 2014), S. 1–8. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/2601097.2601199. URL: https://dl.acm.org/doi/10.1145/2601097.2601199 (besucht am 05. 03. 2023) (zitiert auf S. 13, 14).

Alle URLs wurden zuletzt am 06. 04. 2023 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift