# Simulation von Gravitation im Sonnensystem

9.July 2024

Hochschule Rhein-Waal
Fakultät Kommunikation und Umwelt
Methoden und Werkzeuge der modernen Astronomie

Till Wegener (28891)

## Abstract

Die Bewegung der Sterne und Planeten fasziniert die Menschheit schon seit Urzeiten. In diesem Paper wird eine Methode zur Simulation der Gravitation im Sonnensystem vorgestellt, welche physikalisch akkurate Daten verwendet, um die Bewegung der Planeten zu berechnen. Die Simulation wird in C++ implementiert und visualisiert.

Schlagwörter: Simulation, Gravitation, Sonnensystem, C++, Visualisierung

# Inhaltsverzeichnis

# 1 Einleitung

Die Planeten in unserem Sonnensystem folgen wie alle anderen Objekt im Universum den physikalischen Grundgesetzen. Für die Bewegungen der Planeten ist primär die Gravitationskraft verantwortlich. Durch das Verstehen dieser Kraft ist es uns möglich, die Bewegungen der Planeten zu simulieren.

Dieses Paper verwendet diese Grundgesetze, um eine Simulation zu entwickeln, welche die Bewegungen der Planeten im Sonnensystem berechnet.

# 2 Historischer Kontext

Im Laufe der Zeit haben viele Physiker und Astronomen sich mit der Bewegung der Planeten beschäftigt. Über die Jahrhunderte wurden viele Modelle entwickelt, welche nah am heutigen Stand der Wissenschaft sind. Im folgenden werden einige dieser Modelle vorgestellt.

1. Im 16ten Jahrhundert entwickelte Nikolaus Kopernikus mit „Über die Umlaufbahnen der Himmelssphären“ [1] das erste Modell, welches den heutigen Erkenntnissen nahe kommt. Er beschrieb die Bewegung der Planeten dadurch, dass alle Planeten aufeinander Gravitation ausüben.
2. Zu Beginn des 17ten Jahrhunderts entwickelte Johannes Kepler mit Astronomia nova [2] ein Modell, welches die Bewegung der Planeten genauer beschrieb. Seine Axiome besagten, dass die Planeten sich auf Ellipsen bewegen, und Körper sich gegenseitig anziehen.
3. 1644 entwickelte Isaac Newton mit Philosophiæ Naturalis Principia Mathematica [3] das Modell, welches bis heute zur Beschreibung der Gravitation verwendet wird. Seine Gravitationsgesetze beschreiben, wie sich Körper aufeinander anziehen.

# 3 Mathematische Grundlagen

Wie durch Newton beschrieben, wirkt auf jeden Körper eine Gravitationskraft, welche durch die Masse der Körper und den Abstand zwischen den Körpern bestimmt wird. Die Gravitationskraft zwischen zwei Körpern kann durch die Formel

$$F_G = G * \frac{m_1 * m_2}{r^2}$$

beschrieben werden. Hierbei ist $F_G$ die resultierende Gravitationskraft, $G$ die Gravitationskonstante ($6.67 * 10^{-11} \frac{m^3}{\text{kg} * s^2}$), $m_1$ und $m_2$ die Massen der Körper und $r$ der Abstand zwischen den Körpern. Die Kraft wirkt hierbei entlang der Verbindungslinie der Körper.

Für ein 2-Körper-System kann die Bewegung der Körper durch diese Formel bestimmt werden. Die Gravitationskraft, welche die Körper aufeinander ausüben, kann mit der Masse dieser verwendet werden, um die Beschleunigung der Körper mit $F = m * a$ zu bestimmen.

Für 2-Körper-Systeme ist es möglich, numerische Lösungen für die Bewegung der Körper zu finden.

In mehr-Körper-Systemen wird die Bewegung der Körper durch die Summe der einzelnen wirkenden Gravitationskräfte bestimmt. Für Systeme mit 3 oder mehr Körpern ist es nicht möglich, analytische Lösungen zu finden. Wenn wir hierzu Lösungen finden wollen, müssen wir numerische Methoden verwenden.

Im folgenden wird eine Methode vorgestellt, welche Eulers Methode verwendet, um die Bewegung der Planeten im Sonnensystem zu simulieren.

Eulers Methode ist eine numerische Methode, welche ein Differentialgleichungssystem durch diskrete Schritte approximiert. Die Methode ist einfach zu implementieren, kann jedoch aufgrund ihrer Funktionsweise ungenau sein. Als erstes bestimmen wir unseren diskreten Zeitschritt $\Delta t$. Danach bestimmen wir die Startwerte für die Position und Geschwindigkeit der Körper im Raum.

Nun können wir für alle $n$ Körper die Gravitationskräfte berechnen, welche auf die Körper wirken. In diesem Fall ist die Kraft $\overrightarrow{F_G}$ ein Vektor, welcher die Richtung und Stärke der Kraft angibt. Mit dieser Kraft können wir die Beschleunigung der Körper bestimmen. Mit $\vec{a} = \frac{\overrightarrow{F_G}}{m}$ und $\vec{v} = \vec{v} + \vec{a} * \Delta t$ können wir nun die angepassten Geschwindigkeiten der Körper bestimmen. Die Geschwindigkeit ist in diesem Fall ein 3-Dimensionaler Vektor, welcher die Geschwindigkeit in $x$, $y$ und $z$ Richtung angibt. Für einen Körper kann nun die neue Position $\vec{r}$ im Raum mit $\vec{r} = \vec{r} + \vec{v} * \Delta t$ bestimmt werden.

Wenn wir dieses Verfahren nun für alle Körper im System durchführen, können wir die Bewegung der Körper simulieren. Dieses Verfahren von Hand durchzuführen ist sehr aufwendig, weshalb sich die Implementation als Computerprogramm anbietet.

# 4 Vorbereitung

## 4.1 Datensätze

Für die Simulation des Sonnensystems benötigen wir Daten über die Planeten und ihre diversen Satelliten. Die wichtigen Daten für die Simulation sind die Masse der Körper, die Position der Körper im Raum und die Geschwindigkeit der Körper. Diese Daten können aus diversen Quellen bezogen werden. Für die Planeten wurden die Daten aus dem Projekt Nasa Data Scraper von Devstronomy [4] auf Github verwendet. Das gleiche Projekt beinhaltet auch einen Datensatz für die Satelliten der Planeten. Dieser ist allerdings für unsere Simulation nicht geeignet, da keine Informationen zu Position und Geschwindigkeit der Satelliten enthalten sind.

Die Daten der Satelliten wurden vom Solar System Dynamics Projekt des Jet Propulsion Laboratory der NASA [5] verwendet. Vom SSD Projekt können wir sowohl die physikalischen Daten der Satelliten [6], als auch die orbitale Parameter der Satelliten [7] beziehen. Diese Daten beinhalten die Semi Major Axis und weitere Parameter, welche für die Simulation benötigt werden.

## 4.2 Wahl der Programmiersprache

Da für diese Simulation eine große Anzahl an Berechnungen durchgeführt werden muss, stellte die Wahl der Programmiersprache eine wichtige Entscheidung dar. Für diese Simulation wurde deshalb C++ gewählt. C++ ist eine schnelle kompilierte Sprache.

## 4.3 Wahl von Hilfsbibliotheken

Zur Visualisierung unserer Simulation benötigen wir eine Bibliothek, welche die Darstellung von 3D-Objekten ermöglicht. Hierzu wurde die Bibliothek Raylib [8] verwendet. Raylib ist eine C-Bibliothek, welche die Erstellung von 2D und 3D Anwendungen ermöglicht. Neben der Darstellung von 3D-Objekten war es auch noch wichtig, dass wir einfache Interface-Elemente anzeigen können. Hierzu wird die Bibliothek Dear ImGui [9] verwendet. ImGui ist eine C++ Bibliothek, welche in der Spieleentwicklung verwendet wird, um einfache Interface-Elemente anzuzeigen. Um diese beiden Bibliotheken zusammen zu verwenden, wurde das Projekt rlImGui [10] verwendet.

Die Bibliotheken ImGui und rlImGui wurden als Submodule in das Projekt eingebunden. Raylib wird während der Kompilierung des Projektes heruntergeladen und verwendet. Hierzu wurde das Programm Cmake [11] verwendet. Cmake ist ein Build-System, welches es ermöglicht, C und C++ Projekte plattformunabhängig zu kompilieren.

# 5 Implementierung

## 5.1 Grundlegendes

Die Implementierung der Simulation folgt dem zuvor beschriebenen Ablauf. In der Main-Funktion der Simulation werden zuerst die nötigen Bibliotheken initialisiert. Für dieses Projekt war es nötig, einen eigenen Vektor-Datentypen zu verwenden. Die Gründe hierzu, werden in Abschnitt 6.1 erläutert. Dieser SciVec3

verwendet 64-Bit Gleitkommazahlen, um möglichst präzise Berechnungen zu ermöglichen. Der Datentyp und die dazugehörigen Funktionen zur Addition, Subtraktion, Multiplikation und Skalierung können in SciVec3.h gefunden werden. Die Körper des Sonnensystems werden mithilfe eines Datentypen Body abgebildet. In Body.h wird das struct Body wie folgt definiert:

```
struct Body
{
 std::string name;
 double radius;
 double mass;
 Color color{};
 SciVec3 position;
 SciVec3 velocity;
 bool isPlanet;

 int satelliteCount = 0;
 std::string satelliteNames[64] = {""};

 ...
}
```

Listing 1: Definition des _Body_-Datentyps

Die Felder radius, mass, postion, und velocity sind für die Simulation am wichtigsten. Für jedes Objekt wird außerdem ein Name, eine Farbe, und eine Liste an Satelliten gespeichert. Die Simulation verwaltet eine dynamische Liste (std::vector) an Body-Objekten. Diese Liste stellt den aktuellen Zustand der Simulation da. Um diese Liste nun mit Daten zu füllen, müssen wir unsere Datensätze einladen.

## 5.2 Laden der Datensätze

Die Datei Loader.h beinhaltet die Funktionen, welche die Datensätze aus Abschnitt 4.1 lädt. Die Funktion loadPlanets lädt die planets.csv-Datei aus dem Assets-Ordner. Zuerst erstellt diese Funktion allerdings die Sonne, welche nicht im Datensatz enthalten ist. Hierfür wurden die wichtigsten Werte in Consts.h definiert. Danach iteriert die Funktion über die Zeilen der CSV-Datei und erstellt für jede Zeile ein Body-Objekt. Hierbei ist es wichtig, die einzelnen Werte in die korrekten Einheiten umzuwandeln. Für diese Simulation wurde die Entscheidung getroffen, alle Werte in SI-Einheiten zu speichern. Insbesondere für die Datenwerte Mass war es nötig, 64-Bit Gleitkommazahlen zu verwenden. Um die Visualisierung besser zu gestalten, wurden allen Planeten individuelle Farbe zugeordnet.

Neben den bereits genannten Daten, war die Bestimmung der initialen Position und Geschwindigkeit der Körper eine wichtige Entscheidung. Der Datensatz der Planeten beinhaltet sowohl Werte für den Aphelion, den Perihelion, die Inclination und die Exzentrizität der Planeten. Des Weiteren wurde ein Wert für die durchschnittliche orbital_velocity angegeben.

Die Funktion loadSatallites funktioniert ähnlich wie loadPlanets. Hierbei werden die Dateien satellites_data.csv und satellites_orbit.csv kombiniert, um die Satelliten der Planeten zu erstellen. Für jeden Satellit wird zuerst der Parent-Körper gesucht. Danach werden die weiteren planetaren Informationen bestimmt. Dieser Datensatz weist die Besonderheit auf, dass nicht die Masse der Objekte angegeben ist, sondern der Standard Gravitational Parameter $\mu$. Dieser Wert ist definiert als $\mu = G * m$, wobei $G$ die Gravitationskonstante ist. Um die Masse der Satelliten zu bestimmen, wird der Wert $\mu$ durch $G$ geteilt.

Die Bestimmung der initialen Position und Geschwindigkeit der Satelliten ist etwas komplizierter. Hierzu müssen wir nicht nur die orbitalen Parameter des Satelliten, sondern auch die des Parent-Körpers verwenden.

## 5.3 Aufteilung von Simulation und Visualisierung

Die Simulation und Visualisierung sind zwei verschiedene Aufgaben mit unterschiedlichen Zielen. Ziel der Visualisierung ist es, in geregelten Abstände den Zustand der Simulation darstellen. Dieser Vorgang ist größtenteils abhängig von der Grafikkarte. Die Simulation hingegen ist abhängig von der CPU. Idealerweise sind die beiden Prozesse soweit wie möglich voneinander getrennt. Dies wurde in diesem Projekt so realisiert, dass die Simulation in einem getrennten Thread läuft. Die Visualisierung hingegen läuft im Main-Thread der Anwendung. Beide

Threads greifen auf die selbe Liste an Daten zu, weshalb es nötig war, die Zugriffe auf diese Liste zu synchronisieren. Um eine möglichst geringe Latenz für den Nutzer zu ermöglichen, hat die Visualisierung die primäre Kontrolle über die Liste. Sie kann angeben wann die Simulation auf die Liste zugreifen kann, indem die Simulation temporär pausiert wird.

## 5.4 Implementieren der Simulation

Die Simulation ist der Kern der Anwendung. Um möglichst große Zeiträume simulieren zu können, ist es also nötig, diesen Vorgang möglichst effizient zu implementieren. Die Simulation wird in der Funktion simulate in main.cpp implementiert. Diese Funktion wird in einem eigenen Thread ausgeführt. Dieser Thread arbeitet komplett unabhängig von der Visualisierung. Solang die Simulation nicht pausiert ist, entweder durch den Nutzer, oder durch die Visualisierung, wird die Simulation in einer Schleife ausgeführt. Im folgenden wird eine vereinfachte Version der Funktion simulate gezeigt:

```
void simulate()
{
 if(simulationRunning)
 {
  for(body1 in bodies)
  {
   for(body2 in bodies)
   {
    if(body1 == body2)
     continue;

    Force = calculate_gravity(body1, body2);

    body1.acceleration += Force / body1.mass;
    body2.acceleration -= Force / body2.mass;

      body1.velocity += body1.acceleration *
delta_t;
      body2.velocity += body2.acceleration *
delta_t;

        body1.position += body1.velocity *
delta_t;
        body2.position += body2.velocity *
delta_t;
   }
  }
 }
}
```

Listing 2: Vereinfachte Version der Funktion simulate

Die Funktion folgt dem zuvor definierten Aufbau. Für jeden Körper wird die Gravitationskraft berechnet, welche auf den Körper wirkt. Mit dieser Kraft wird die Beschleunigung des Körpers bestimmt. Mit der Beschleunigung wird die Geschwindigkeit, und damit die Position des Körpers bestimmt. Dieser Vorgang wird für alle Körper in der Liste bodies durchgeführt.

Wichtig ist hierbei, dass die Berechnung von der globalen variable delta_t abhängig ist. Diese Variable bestimmt den Zeitschritt der Simulation. Je kleiner dieser Wert ist, desto genauer ist die Simulation, jedoch auch langsamer. Je nach verfügbarer Rechenleistung kann dieser Wert auf bis zu 1s pro Iteration gesetzt werden. Mehr hierzu in Abschnitt 7.

## 5.5 Implementieren der Visualisierung

Mithilfe der Visualisierung können wir den momentanen Zustand der Simulation darstellen.

Die primäre draw-loop der Simulation findet innerhalb der main-Funktion der Anwendung statt. Eine vereinfachte Version dieser Funktion sieht wie folgt aus:

```
void draw()
{
 BeginDrawing3D();
 ClearBackground(BLACK);
 UpdateClipPlane();

 simulationRunning = false;
 for(body in bodies)
 {
  DrawBody(body);
 }
 simulationRunning = true;

 UpdateCamera();
 DrawInterface();

 EndDrawing3D();
}
```

Listing 3: Vereinfachte Version der Funktion draw

Diese Funktion wird solange in einer Schleife ausgeführt, bis die Anwendung geschlossen wird. Nach dem Starten des 3-D-Renderers wird der Hintergrund gelöscht. Als nächstes wird die ClipPlane[12] angepasst. Dies ist der Abstand von Geometrie zur Kamera, ab welchem Objekte nichtmehr dargestellt werden. Aufgrund der großen Werte, welche durch die Verwendung von physikalisch akkuraten Werten auftreten, ist es nötig, diesen Wert anhand der aktuellen Kameraeinstellungen anzupassen.

Nachdem diese Vorbereitungen getroffen wurden, werden die einzelnen planetarischen Objekte gezeichnet. Die Funktion DrawBody nimmt die aktuelle Position des Körpers und stellt diesen mithilfe einer gefärbten Kugel in 3D dar.

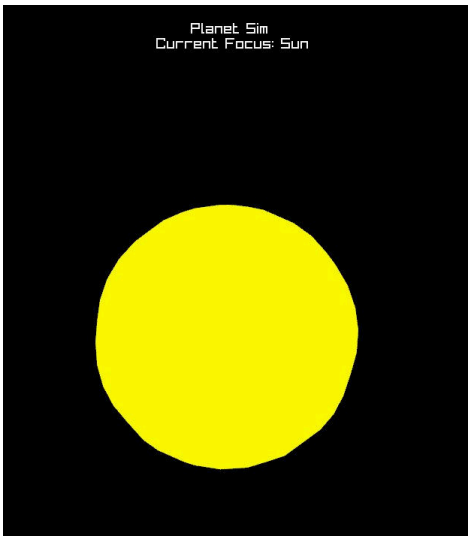


Abbildung 1: Sonne in nSim

Diese Methode kann nun verwendet werden um die Planeten und ihre Satelliten zu visualisieren. Ein Phänomen, welches aufgrund dieser Art der Visualisierung auftritt, ist, dass die relative Bewegung und Position von Körpern zueinander schwer zu erkennen ist.

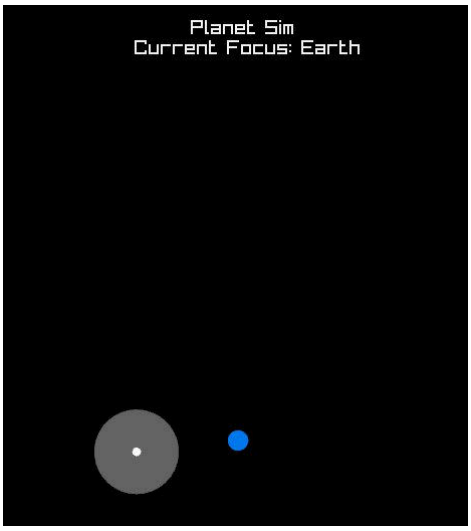


Abbildung 2: Erde und Mond

Wie in Abbildung 2 zu erkennen ist, gibt uns diese Visualisierung keine Anhaltspunkte für die Orientierung und Bewegung der Planeten. Ein weiteres Problem war es, dass aufgrund der Skalierung der Visualisierung einige Satelliten sehr klein, und damit schwer zu erkennen wurden. Um diese Problematiken zu verringern, wurden drei weitere Visualisierungen hinzugefügt.

1. Astronomisches Gitter

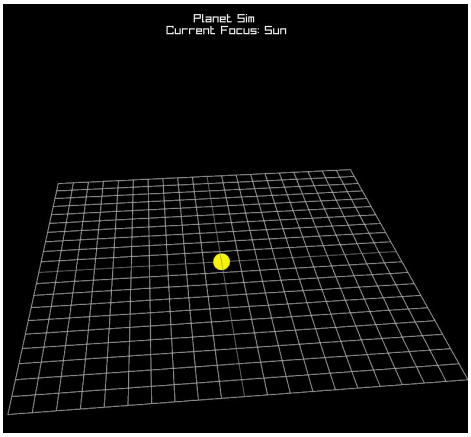


Abbildung 3: Sonne mit astronomischem Gitter

Um ein besseres Gefühl für die relative Position und Rotation von Punkten im Raum zu geben, wurde ein optionales Gitter zur Visualisierung hinzugefügt. Dieses Gitter weist eine Auflösung von $\frac{1}{100}$ AU, oder einem Einhundertstel der astronomischen Einheit da. Über die Einstellung kann auch die Größte des Gitters kontrolliert werden. Das Gitter wird zentriert über dem aktuell fokussierten Körper angezeigt.

2. Transparente Selektoren für Satelliten

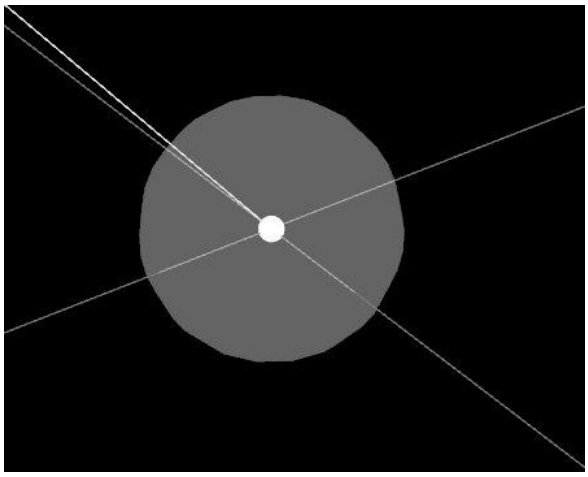

Abbildung 4: Mond mit transparenten Selektor

Abbildung 4 zeigt unseren Mond in Grau mit einem größeren halb-transparenten Selektor darüber. Der Selektor ermöglicht es, auch die kleineren planetarischen Objekte zu selektieren. Die Größe des Selektors ist dabei immer noch abhängig von dem Radius des Objektes, damit die Skalierung trotzdem akkurat bleibt.

3. Historische Pfade

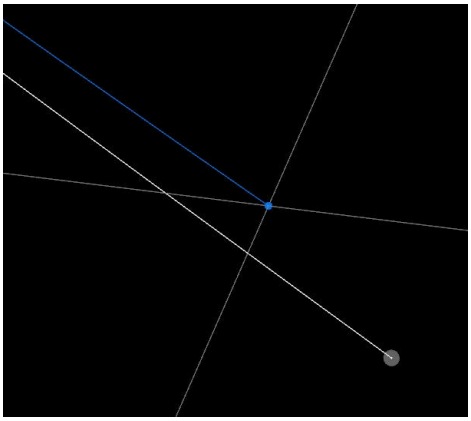

Abbildung 5: Erde und Mond mit historischen Pfaden

Um die Bewegung der Objekte besser zu visualisieren, werden die vergangenen Positionen gespeichert und in Form einer 3D-Kurve durch den Raum angezeigt. So ist es beispielsweise möglich zu sehen, wie der Mond sich relativ zur Erde bewegt.

## 5.6 Implementierung des Interfaces

Um mit der Simulation zu interagieren, wurden einfache Interface-Elemente benötigt. Die Interface-Elemente wurden in Form von verschiedenen ImGui-Fenstern realisiert. Die folgenden Fenster wurden für die Simulation implementiert:

1. Graphics Debugger:
   Fenster zum Einstellen der grafischen Visualisierung. Hier können Dinge wie das Gitter, die Selektoren und Wireframe-Rendering aktiviert werden
2. Camera Settings:
   Fenster, welches die aktuellen Parameter der Camera anzeigt.
3. Simulation Settings:
   Fenster, welches die Konfiguration des Zeitschrittes erlaubt. Über dieses Fenster kann auch die Simulation pausiert werden. Des Weiteren wird angezeigt, wie viele Sekunden, Minuten, Stunden und Tage bisher in der Simulation vergangen sind.
4. Focus Select:
   Fenster, welches verwendet werden kann, um den aktuell fokussierten Körper zu wechseln.

### 5.7 Implementierung der Kamera

Die Kamera selbst ist mithilfe von sphärischen Koordinaten relativ zum aktuell fokussierten Körper ausgerichtet. Es ist möglich, mithilfe der Maus den horizontalen und vertikalen Winkel der Kamera relativ zum Körper anzupassen. Mithilfe des Mausrads kann der Abstand $r$ der Kamera zu diesem kontrolliert werden. Die Kamera dieser Simulation richtet sich am aktuell fokussierten Planeten aus. Dementsprechend ist es nötig, die Kamera in jedem Durchgang der Visualisierungsfunktion zu aktualisieren. Dabei übernimmt die Kamera zuerst die Position des aktuell fokussierten Objektes. Danach wird anhand der sphärischen Koordinaten $\theta$ und $\varphi$ wie folgt die relative Position ermittelt:

$$\Delta_x = r * \sin(\theta) * \cos(\varphi)$$

$$\Delta_y = r * \cos(\theta)$$

$$\Delta_z = r * \sin(\theta) * \cos(\varphi)$$

Die finalen Koordinaten der Kamera können dann mit

$$x = x_{\text{planet}} + \Delta_x$$

$$y = y_{\text{planet}} + \Delta_y$$

$$z = z_{\text{planet}} + \Delta_z$$

bestimmt werden. Hierbei ist wichtig zu beachten, dass OpenGL, und damit auch Raylib, ein rechtshändiges Koordinatensystem verwenden, in welchem $y$ und nicht wie typisch $z$ die vertikale Achse ist.

## 6 Besonderheiten und Lösungen

Die Simulation weist eine vielzahl von Besonderheiten auf, welche die Implementation beeinflussen. Der wichtigste Faktor hierbei ist die Größte der verschiedenen verwendeten Werte. Da die Werte in SI-Einheiten vorliegen, sind die Werte für die Positionen und Geschwindigkeiten sehr groß. Die Masse der Sonne beispielsweise beträgt $1.989 * 10^{30}$kg. Die Distanz zwischen der Erde und der Sonne liegt bei $1.495 * 10^{11}m$. Aufgrund dieser großen Werte ist es nötig, die Visualisierung zu skalieren. Hierzu wurde der Faktor 1:1000000 gewählt. Dieser Faktor wird für alle Größen innerhalb der Visualisierung verwendet. Die Berechnung der Simulation findet allerdings trotzdem mit den ungekürzten Werten statt.

### 6.1 32-Bit Gleitkommazahlen

Die Größenordnung der Werte führt dazu, dass wir Gleitkommazahlen verwenden müssen, um diese akkurat darzustellen. Eine reguläre 64-Bit Zahl hat einen Wertebereich von $-2^{63}$ bis $2^{63} - 1$ oder von $\pm 9.22 * 10^{18}$. Damit wäre es nicht möglich, die meisten Berechnungen durchzuführen. Gleitkommazahlen nach IEEE-754[13] können größere Zahlen darstellen. 32Bit Gleitkommazahlen haben einen Wertebereich von $\pm 3.40 * 10^{38}$. Dieser Wertebereich ist groß genug, um unsere Berechnungen durchzuführen. Für Gleitkommazahlen ist allerdings nicht nur der Wertebereich, sondern auch die Genauigkeit wichtig. 32-Bit Gleitkommazahlen haben eine Genauigkeit von 7 Stellen. Für unsere Berechnungen ist dies nicht ausreichend. Deshalb wurde ein eigener Datentyp SciVec3 implementiert. Dieser Datentyp verwendet 64-Bit Gleitkommazahlen. Diese bieten eine Genauigkeit von 15 Stellen. Der Datentyp SciVec3 wird für alle Berechnungen innerhalb der Simulation verwendet. Nur für die Darstellung wird in den von Raylib verwendeten Vector3-Datentypen umgewandelt.

### 6.2 Floating Origin

Auch nach den Anpassungen in Abschnitt 6.1 treten allerdings noch Probleme bei der Visualisierung auf. Unsere skalierten Werte sind häufig länger als 7 Stellen. Dies führt aufgrund des Aufbaus von IEEE-754 Gleitkommazahlen dazu, dass Genauigkeitsprobleme sichtbar werden. In diesem Fall können wir nicht wie zuvor den Datentypen anpassen, da moderne Grafikschnittstellen wie OpenGL nur 32-Bit Gleitkommazahlen unterstützen.

Um dieses Problem zu lösen, wurde das Konzept des Floating Origin eingeführt. Hierbei werden alle Objekte relativ zum aktuell fokussierten Körper dargestellt. Dies führt dazu, dass der fokussierten Bereich der Visualisierung immer im genausten Bereich der Gleitkommazahlen dargestellt werden. Die Logik hierfür ist wie folgt:

1. Die Position des aktuell fokussierten Körpers wird als Ursprung der Welt gesetzt.
2. Beim Anzeigen der Körper wird die Position des fokussierten Körpers von der Position des Körpers abgezogen.
3. Die Kamera wird relativ zum fokussierten Körper positioniert.

Da die gesamte Positionierung und Berechnung weiterhin mithilfe des SciVec3 in 64-Bit ausgeführt wird, sind die Berechnungen weiterhin akkurat. Erst nachdem diese Berechnungen durchgeführt wurden, werden die Werte in 32-Bit Gleitkommazahlen umgewandelt.

# 7 Ergebnisse

Ziel unserer Simulation war es, dass wir die Bewegung der Planeten simulieren und in 3D visualisieren können. Hierzu wurde das Programm nSim, ein n-Body Simulator, entwickelt.

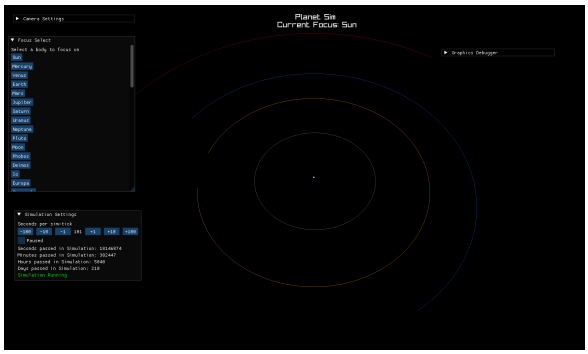


Abbildung 6: nSim Simulation des inneren Sonnensystems

Abbildung 6 zeigt einen weiten Blick auf das innere Sonnensystem. Hier werden die Umlaufbahnen des Merkurs, der Venus, der Erde und des Mars um die Sonne sichtbar.

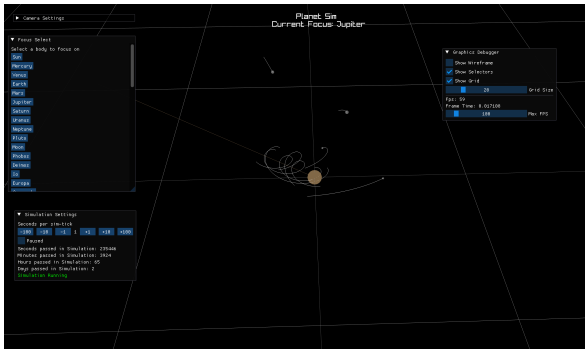


Abbildung 7: nSim Simulation von Jupiter

Abbildung 7 zeigt die Simulation des Jupiter und seiner vier größten Monde. Hierbei ist zu erkennen, wie die Monde um den Jupiter kreisen.

nSim ermöglicht es, innerhalb kürzester Zeit die Bewegung der Planeten über Tage oder sogar Jahre hinweg zu simulieren. Hierzu werden diverse Open-Source Projekte verwendet. Der Code für dieses Projekt kann auf Github [14] eingesehen werden.

# Quellen


[1] Nikolaus Kopernikus, De revolutionibus orbium coelestium. 1543.

[2] Astronomia nova. 1609.

[3] Philosophiæ Naturalis Principia Mathematica. 1644.

[4] „NASA Data Scraper“. 8. Juli 2024. [Online]. Verfügbar unter: https://github.com/devstronomy/nasa-data-scraper/tree/master

[5] „Solar System Dynamics“. [Online]. Verfügbar unter: https://ssd.jpl.nasa.gov/

[6] „Satellite physical data“. [Online]. Verfügbar unter: https://ssd.jpl.nasa.gov/sats/phys_par/

[7] „Satellite orbital data“. [Online]. Verfügbar unter: https://ssd.jpl.nasa.gov/sats/elem/

[8] raysan5, „Raylib“. 8. Juli 2024. [Online]. Verfügbar unter: https://github.com/raysan5/raylib.git

[9] ocornut, „Imgui“. 8. Juli 2024. [Online]. Verfügbar unter: https://github.com/ocornut/imgui.git

[10] „Rlimgui“. 8. Juli 2024. [Online]. Verfügbar unter: https://github.com/raylib-extras/rlImGui.git

[11] „CMake“. [Online]. Verfügbar unter: https://cmake.org/

[12] Khronos Group, „ClipPlane“. [Online]. Verfügbar unter: https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/glClipPlane.xml

[13] „IEEE Standard for Floating-Point Arithmetic“. 2019.

[14] „nSim sourcecode“. 8. Juli 2024. [Online]. Verfügbar unter: https://github.com/tillwege/nSim.git