H O C H
S C H U L E
T R I E R

Hausarbeit in Spieleprogrammierung - Vertiefung

Dokumentation zur Entwicklung eines Rubiks Cubes

Themensteller: Prof. Dr. Christoph Lürig

Name: Gregor Germerodt

Matrikelnummer: 977527

Fachbereich: Informatik

Studiengang: Informatik – Digitale Medien und Spiele

Abgabetermin: 30.06.2025

Inhaltsverzeichnis

1 Einleitung

Diese Dokumentation wurde im Rahmen der Vorlesung „Spieleprogrammierung – Vertiefung“ erstellt. Die Aufgabe war es, einen Rubiks Cube von Grund auf oder anhand gegebener Module aus den Vorlesungseinheiten zu erstellen. Die benutzte Entwicklungsumgebung war Visual Studio Community 2022 mit der Programmiersprache C++.

1.1 Mögliche Noten

- 4.0: Logische Repräsentation, Rendern Würfel, keine Animation, simple Steuerung
- -1: Animation des Würfels
- -1.3: Maussteuerung Steuerung vorne nach Konzept wie in Vorlesung
- -0.3: Maussteuerung für Kamera
- -0.3: Sound
- -0.3 bis -0.7: Aufwendige Grafikprogrammierung

2 Logische Vorgehensweise

2.1 Wintersemester 2024/2025

Zum ersten Mal habe ich mich letztes Semester an dem Projekt versucht. Ich schaute teilweise die Präsenzvorlesungen und die die online verfügbar waren. Weiterhin beteiligte ich mich an wenigen Fragestunden. Schließlich versuchte ich anhand der Module aus den Vorlesungseinheiten den Rubiks Cube zu erstellen, indem ich sie durch meinen Code erweiterte. Zum Ende des Semesters hat das Programm leider noch nicht funktioniert. Aufgrund mangelnden Verständnisses der Matrixberechnungen bzw. in der Reihenfolge der Berechnungen, kam es zu keiner effektiven Rotation der Scheiben des Cube. Zudem hätte der Code als sehr unschön bezeichnet werden können und war damit ebenso wenig effizient. Letztendlich entschied ich mich die Hausarbeit auf das nächste Semester zu legen und komplett von neu zu beginnen, um mögliche fehlerhafte Gedankengänge aufdecken zu können.

2.2 Sommersemester 2025

Zu beginn dieses Semesters habe ich mich erneut an dem Rubiks Cube versucht. Ich schaute mehrmals die online verfügbaren Vorlesungen, um die Konzepte größtenteils zu verstehen. Auch hier habe ich wieder teilweise den Code übernommen und erweitert. Im folgenden Abschnitt sind die übernommenen Module aufgelistet.

2.2.1 Übernommene Module

Header Files:

- CubieRenderer.h
- GameInterface.h
- InputSystem.h
- KeyboardObserver.h
- ShaderUtil.h
- TestCompoundCube.h

Shader Files:

- FragmentShaderColor.glsl
- FragmentShaderSimple.glsl

- VertexShaderColor.glsl
- VertexShaderSimple.glsl

Source Files:

- CubieRenderer.cpp
- InputSystem.cpp
- KeyboardObserver.cpp
- RubixCube.cpp
- ShaderUtil.cpp
- TestCompoundCube.cpp

Diese Module dienten als Grundgerüst und damit als Option das Projekt bei Bedarf zurückzusetzen, was auch mehrmals der Fall war. Das Modul TestCompound-Cube.cpp diente als Hauptentwicklungsumgebung, in der die gesamte Logik des Cube implementiert wurde. Aus diesem Grund wurde sie später zu CubeLogic.cpp umbenannt. Ein weiteres benutztes Modul ist TestKey.cpp, aus der jedoch nur die Logik für das Rotieren des gesamten Cube mittels der Pfeiltasten und die Methodenaufrufe zum Überwachen dieser Tasten ausgeschnitten wurden.

2.2.2 Implementierung und Konzeption

Zur Ausführung des Programms diente immer das Modul TestCompoundCube.cpp. Zu Beginn entfernte ich den Teil der **Render**-Methode, die die einzelnen Cubies verdreht hat, um einheitlich farbige Seiten des Cube zu visualisieren. Weiterhin fügte ich den, im vorherigen Abschnitt erwähnten Code-Teil aus dem TestKey.cpp-Modul meiner CubeLogic.cpp bzw. dessen **Initialize**-Methode hinzu. Unter Gebrauch von m_orientationQuaternion konnte der Cube bereits vollständig rotiert werden. Dort wurden ebenso die Tasten des Ziffernblocks, die linke Shift-, die Leer- und die R-Taste übernommen. Dabei dient der Ziffernblock zum Rotieren der Scheiben des Cube, die Leertaste zum Ausgeben der Koordinaten eines variabel definierten Cubies und die R-Taste zum Zurücksetzen des Cube auf die Ausgangposition.

Um die einzelnen Cubies ersichtlicher zu visualisieren, wurde eine zusätzliche Lücke zwischen den Cubies (gapBetweenCubies) hinzugefügt. Zu einem späteren Zeitpunkt, teilte ich die **Render**-Methode weiter auf, um die anfängliche Positionierung der Cubies von der **Render**-Logik zu trennen. Die dazu erstellte **SetUpCubies**-Methode

beinhaltet noch weitere später hinzugefügte Funktionalitäten, wie zum Zwischenspeichern der mittleren Cubies der äußeren Scheiben und das Merken der Startposition jedes einzelnen Cubies zwecks Debuggings.

Die **ClearResources**-Methode wurde vollständig übernommen und dient zum Löschen der vertex buffer objects (vbo), der vertex array objects und des shader Programms.

Die übernommene Update-Methode aus dem TestKey.cpp-Modul teilte ich für bessere Übersichtlichkeit in drei Methoden auf. Die **HandleArrowKeys**-Methode, übernahm dabei den Code der **Update**-Methode, welche am Ende die zusätzlich implementierte **RotateCube**-Methode aufruft, die letztendlich den gesamten Cube rotiert. Diese nutzt das m_orientationQuaternion, welches im TestKey.cpp-Modul an die globale Transformation multipliziert wurde, ich jedoch ebenso zur Übersichtlichkeit daraus extrahiert habe. In der **HandleArrowKeys**-Methode wird zudem das erwähnte Quaternion zu Beginn zurückgesetzt, da der Cube sonst ununterbrochen rotiert, sollte eine Pfeiltaste gedrückt werden.

Folgend wird in der Update-Methode die Methode **HandleNumpadKeys** aufgerufen. In dieser werden die Tasten des Ziffernblocks gemappt, um entsprechende Scheiben des Cube zu rotieren. Benutzte Tasten sind dabei alle außer die mit der 0 und 5. Die Tasten mit den Zahlen 1, 4, 7 und 3, 6, 9 dienen zur Rotation auf der Y-Achse und die mit den Zahlen 1, 2, 3 und 7, 8, 9 zur Rotation auf der X-Achse in entsprechender Richtung, wobei die zweite besagte Belegung in Kombination mit der linken Shift-Taste betätigt werden muss (aufgrund der Doppelbelegung der Tasten mit der 1, 3, 7 und 9). Sollte eine dieser Tasten betätigt werden, wird die Methode **RotateLayer** aufgerufen. Sie entscheidet je nach Argumenten, auf welcher Achse (X oder Y), in welche Richtung (1 oder -1) und welche Scheibe[1] (obere (2), mittlere (1) oder untere (0)) rotiert werden soll.

Die dadurch aufgerufene Methode **RotateLayer** enthält die weitere große Logik des Cube. An dieser Stelle entschied ich mich dafür die Scheiben anhand des mittleren Cubies der Scheibe ausfindig zu machen. Dazu wurde die Methode **FindMiddleCubie** definiert. Diese ermittelt je nach Achse und Scheibe, um welchen middleCubie[2] es sich

[1] Die aktuelle Scheibe ist durch einen globalen Pointer-Container (glm::mat4* currentLayer[9]) ansprechbar
[2] Globaler Pointer auf aktuellen mittleren Cubie (glm::mat4* middleCubie)

handelt. Dazu werden die in der **SetUpCubie**-Methode initialisierten Pointer auf die middleCubies[3] gebraucht. Soll die mittlere Scheibe (1) angesprochen werden, so ist es immer der Cubie an der Stelle [1][1][1] in dem Container der Matrizen aller Cubies[4]. Andernfalls wird geschaut, um welche Achse rotiert wird und ob die momentan oberste (2) oder unterste Scheibe (0) angesprochen werden soll. Dafür werden die Positionen der middleCubies betrachtet. Der middleCubie ergibt sich dann aus dem jeweiligen größten bzw. niedrigsten X- oder Y-Wert (je nach Achse) des Positionsvektors.

Danach wird die ebenfalls achsen- und scheibenabhängige **FindLayer**-Methode aufgerufen. Hier werden die Scheibeen einzeln und abhängig vom aktuellem middleCubie definiert. Da durch die Rotation die Cubies neu positioniert und indiziert werden müssen, sind die Scheiben einheitlich zirkulär und im gegengesetzten Uhrzeigersinn definiert, wobei der letzte Cubie der Scheibe immer der entsprechende middleCubie ist. Sollte der aktuelle middleCubie der der mittleren Scheibe sein, so wird die Scheibe über die Methode **FindMiddleLayer** ermittelt. Für die mittlere Scheibe können drei Varianten in Frage kommen (entlang der X-, Y- und Z-Achse). Dafür wurden drei Repräsentanten und drei Kombinationen mit ihnen erstellt. Die Repräsentanten sind die middleCubies an den positiven Enden der drei globalen Achsen. Die Kombinationen ergeben die gesuchte Scheibe, indem die absoluten Positionswerte (X oder Y) der jeweiligen Repräsentanten addiert werden. Die Kombination, die am nächsten an Null liegt, ergibt die gesuchte Scheibe. Der Repräsentant der nicht teil der Kombination war, bestimmt weiterhin die Scheibendefinition, genauer ob der Würfel geflippt ist.

Nachdem die Scheibe gefunden wurde, müssen die einzelnen Cubies rotiert werden. Dieser Vorgang besteht aus vier Schritten: Relative Rotationsachse jedes Cubies bestimmen; Die Cubies rotieren; Die Position und die Indizes der Cubies aktualisieren.

Die Rotationsachse der Cubies einer Scheibe stimmen schnell durch mehrere Rotationen der Scheiben nicht mehr überein. Aus diesem Grund muss für jeden Cubie die relative Achse um die rotiert werden soll gefunden werden. Dazu wurde die Methode **FindRotationAxis** implementiert. Als Argumente nimmt sie ebenfalls die Achse entgegen und weiterhin den aktuellen Cubie der über die Schleife in der **RotateLayer**-

[3] Globaler Pointer-Container, um auf mittlere Cubies zugreifen zu können (glm::mat4* middleCubies[6])
[4] Globaler Container zur Speicherung sämtlicher Cubie-Matrizen (glm::mat4 m_Cubies[3][3][3]), wobei hinterster Cubie links unten aus Blickrichtung und Ausgangsposition der erste ist.

Methode angesprochen wird. Um die relative Achse zu bestimmen, werden die ersten drei Spaltenvektoren (erster entspricht der X-Achse, zweiter der Y-Achse und dritter der Z-Achse) auf ihre X- und Y-Werte (je nach Achse) geprüft. Der Spaltenvektor mit dem größten absoluten Wert entspricht dann der Rotationsachse, wobei das Präfix bestimmt, ob die Rotationsrichtung geflippt werden muss. Danach wird in der **RotateLayer**-Methode der aktuelle Cubie rotiert.

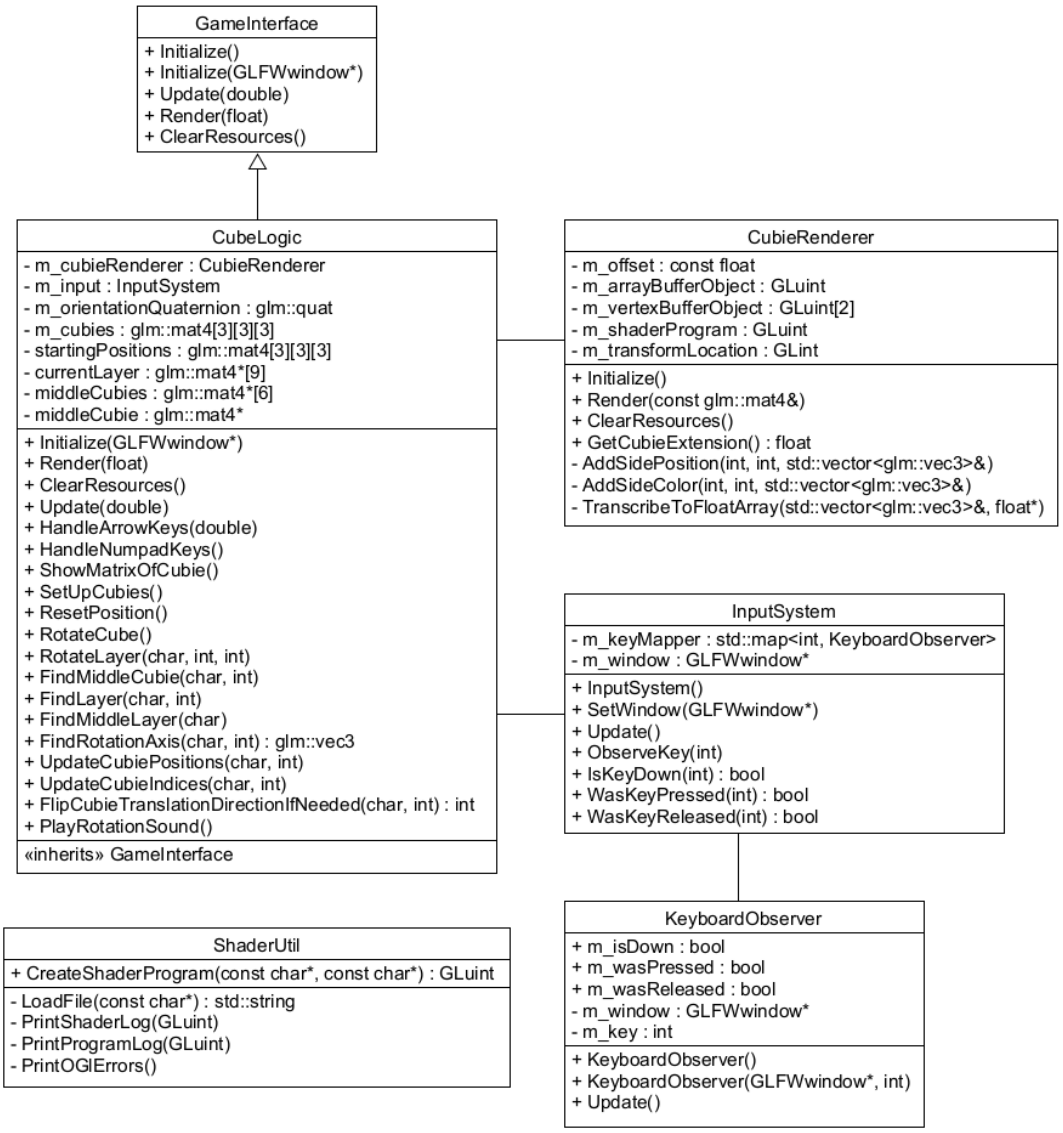
Nach der Rotation müssen die Cubies verschoben werden, da das ursprüngliche Ziel die Rotation einer Scheibe und nicht der einzelnen Cubies war. Um dies zu erreichen, ermittelt die Methode **UpdateCubiePositions** unter Nutzung der Achse und der Richtung die neuen Positionen. Hier werden die um jeweils zwei verschobene Positionen der Cubies zwischengespeichert und dann die aktuellen Positionen damit überschrieben. Hierfür waren die erwähnten zirkulären und im gegengesetzten Uhrzeigersinn definierten Scheiben notwendig. Ferner wurde eine Hilfsmethode **FlipCubieTranslationDirectionIfNeeded** implementiert, welche die Richtung in die Cubies verschoben werden sollen, flippt, je nachdem ob sie im jeweilig positiven oder negativen globalen Achsenbereich liegen.

Einen sehr ähnlichen Aufbau und Ablauf hat die Methode **UpdateCubieIndices**, welche aufgrund kleiner Logikunterschiede nicht mit der vorherigen Methode vereinheitlicht wurden konnte.

Im Verlauf des Projekts implementierte ich diverse Debugging-Methoden, welche teilweise wieder entfernt wurden. Geblieben sind die Methoden **ResetPosition** und **ShowMatrixOfCubie**, welche die Ausgangsposition des gesamten Cube wiederherstellt, und die Matrix eines variablen Cubies ausgibt.

Als letztes wurde die kurze Methode **PlayRotationSound** entwickelt, welche bei der Rotation einer Scheibe einen aus fünf Sounds zufällig abspielt.

2.2.3 UML-Klassendiagramm



3 Fazit und Reflexion

Im Laufe des Projekts sind mir einige Schwierigkeiten unterlaufen, weshalb ich auch im ersten Versuch das Projekt nicht erfolgreich abschließen konnte. Auch in diesem Versuch hatte ich einige Probleme mir die benötigten mathematischen Konzepte stets vor Augen zu halten und herauszufinden welche vielen einzelnen Schritte (nicht) notwendig sind, um ans Ziel zu kommen. Auch im Umgang mit C++ war ich nicht sehr agil. Aus diesen Gründen hatte ich mich dafür entschlossen das Projekt bei der 3.7 Konstellation abzuschließen und mich anderen Projekten zu widmen. Wahrscheinlich ist die Code Qualität auch nicht ganz den Erwartungen entsprechend. Trotzdem bin ich für meine Verhältnisse zufrieden und sehe das Projekt als einen Erfolg an.