Informatik
Hauptcampus

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

Rubiks Cube Dokumentation

Florian Wößner

Hausarbeit

Betreuer: Prof. Dr. Christoph Lürig

Trier, 02.02.2025

Inhaltsverzeichnis

1 Einleitung 1

1.1 Steuerung 1

2 Umsetzung 2

2.1 Klassenübersicht 2

2.2 main.cpp 3

2.3 RubiksGameInterface und IGameInterface 4

2.4 InputSystem und KeyboardObserver 5

2.5 CubieRenderer und ShaderUtil 8

2.6 RubiksCube und Cubie 9

3 Zusammenfassung 10

1

Einleitung

Das Rubik's Cube Projekt wurde im Rahmen des Moduls Spieleprogrammierung - Vertiefung entwickelt. Ziel des Projekts war es, einen interaktiven Rubik's Cube zu simulieren, der mithilfe von Maus- und Tastatureingaben gesteuert werden kann. Die technische Umsetzung erfolgte unter Verwendung von C++ und OpenGL, wobei mathematische Konzepte wie Quaternionen und Transformationen zur Anwendung kamen.

In dieser Hausarbeit werden die Funktionsweisen der wichtigsten Klassen und die grundlegenden Lösungsideen dokumentiert. Der Fokus liegt auf den entwickelten Ansätzen sowie einer oberflächlichen Beschreibung der Implementierung.

1.1 Steuerung

Der Rubik's Cube wird vollständig durch Maus- und Tastatureingaben gesteuert:

- Rechte Maustaste gedrückt: Durch Ziehen wird der gesamte Cube rotiert.
- Linke Maustaste gedrückt: Ermöglicht das Drehen einzelner Slices (Zeilen und Spalten).
- Leertaste: Setzt den Würfel in den Ausgangszustand zurück.
- Scrollrad: Kann benutzt werden, um mit der Kamera herein- oder herauszuzoomen.

2

Umsetzung

In diesem Kapitel werden die wichtigsten Klassen und deren Funktionsweisen erläutert. Jede Klasse wird kurz beschrieben, wobei der Fokus auf ihrer Rolle im Gesamtsystem und ihrer Implementierung liegt.

2.1 Klassenübersicht

Das Klassendiagramm zeigt die Beziehungen der wichtigsten Komponenten des Rubik's Cube Projekts. Die Basis bildet das Interface `IGameInterface`, das von `RubiksGameInterface` implementiert wird. Diese zentrale Klasse koordiniert das Zusammenspiel von Eingaben (`InputSystem`, unterstützt durch `KeyboardObserver`), der Logik (`RubiksCube`) und der Darstellung (`CubieRenderer`).

Der Würfel selbst besteht aus `Cubie`-Objekten, die von `RubiksCube` organisiert werden. Zur Unterstützung von Rendering-Aufgaben wird die Klasse `ShaderUtil` genutzt, welche die Shader-Verwaltung übernimmt. Abbildung 2.1 verdeutlicht die Beziehungen der Klassen.

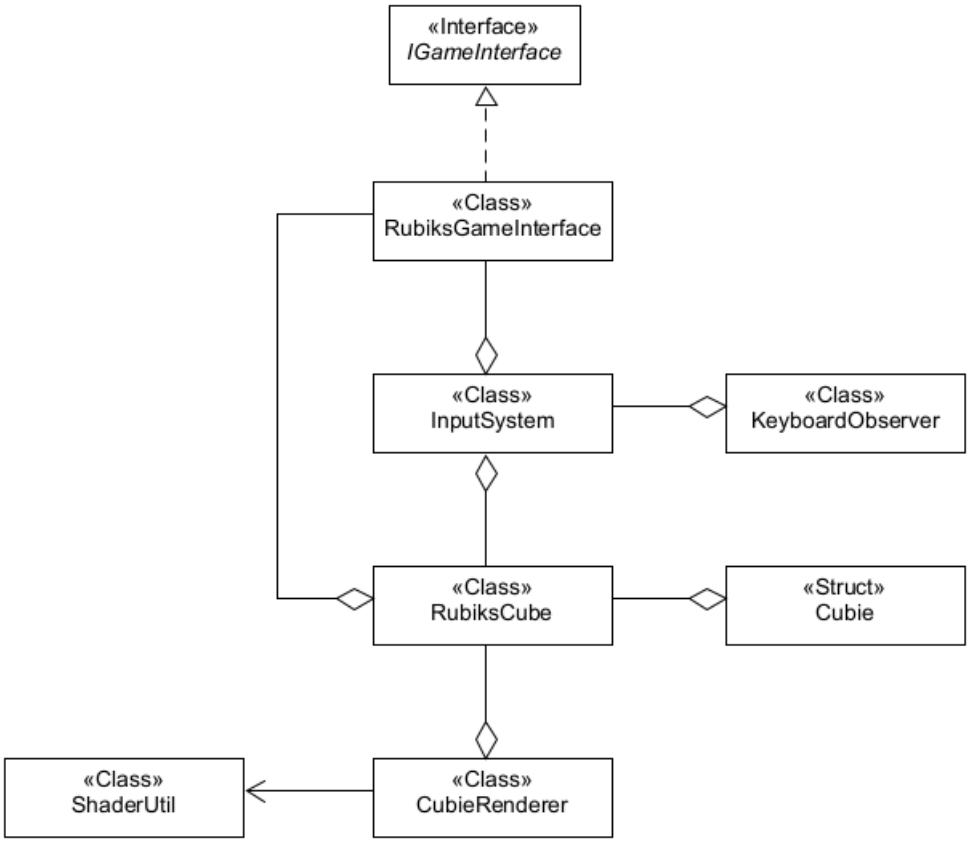

Abbildung 2.1: Vereinfachtes Klassendiagramm

2.2 main.cpp

Der Code in `main.cpp` dient als Einstiegspunkt des Projekts. Hier wird *GLFW* initialisiert, ein Fenster erstellt und die *GLEW*-Bibliothek geladen, um moderne OpenGL-Funktionen nutzen zu können. Ein `RubiksGameInterface`-Objekt wird erstellt und als aktuelle Schnittstelle festgelegt. Der Game-Loop verarbeitet Benutzereingaben und rendert das Fenster. Ein zusätzlicher Mechanismus stellt sicher, dass das Programm bei minimiertem Fenster nicht abstürzt. Nach dem Schließen des Fensters werden mit der Methode `ShutdownSystem()` alle Ressourcen freigegeben und *GLFW* ordnungsgemäß beendet.

2.3 RubiksGameInterface und IGameInterface

Die Klasse `RubiksGameInterface` implementiert das Interface `IGameInterface` und stellt die zentrale Schnittstelle für die Verwaltung des Spiels dar. Die Methode `Initialize(GLFWwindow* window)` kümmert sich um die Initialisierung des Spielfensters und des Eingabesystems. Dabei wird das Fensterobjekt in `m_window` gespeichert und das Eingabesystem durch den Aufruf von `m_input.Initialize(window)` eingerichtet. Der Rubik's Cube wird durch `m_rubiksCube.Initialize(*this)` initialisiert. Zusätzlich wird mit `m_input.ObserverKey(GLFW_KEY_SPACE)` ein Observer für die Leertaste erstellt.

Die Methode `Render(float aspectRatio)` ist für das Zeichnen des Fensters und des Rubik's Cube verantwortlich. Sie berechnet bei Bedarf die Projektions- und View-Matrizen neu, wenn die Kameraposition oder das Seitenverhältnis des Fensters sich geändert haben. Anschließend wird die Renderlogik auf den Rubik's Cube angewendet, um dessen aktuellen Zustand mit dem Aufruf von `m_rubiksCube.Render(m_projection * m_view)` darzustellen.

Die Methode `Update(double deltaTime)` verarbeitet die Benutzereingaben und aktualisiert den Rubik's Cube. Wenn die Leertaste gedrückt wird, erfolgt ein Zurücksetzen des Rubik's Cubes, andernfalls wird `m_rubiksCube.Update(*this)` aufgerufen. Änderungen der Kameraentfernung, die durch das Mausrad bedingt sind, werden in Echtzeit aktualisiert, indem der Wert von `m_CameraDistance` angepasst wird. Der Kameraabstand wird durch `if`-Abfragen auf bestimmte Werte begrenzt. Die Methode `ClearResources()` gibt die Ressourcen des `m_rubiksCube`-Objektes frei.

Zusätzlich verfügt die Klasse über die Methode `QueueMatrixRecalculation()`, die es ermöglicht, das Neuberechnen der Projektions- und View-Matrizen gezielt anzustoßen. Weitere Getter-Methoden erlauben den Zugriff auf `m_deltaTime` und `m_input`.

Abbildung 2.2 zeigt das Klassendiagramm zu diesem Abschnitt.

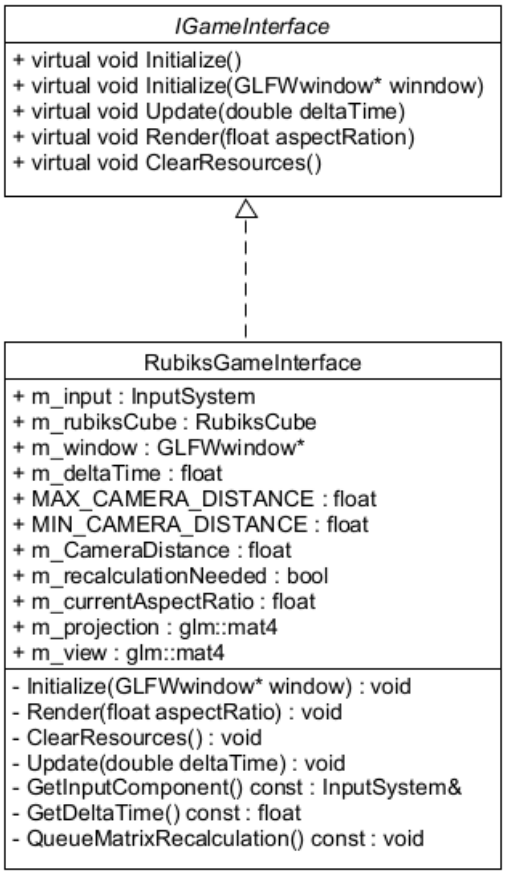

Abbildung 2.2: Klassendiagramm RubiksGameInterface

2.4 InputSystem und KeyboardObserver

Die Klasse `InputSystem` ist eine zentrale Komponente zur Verwaltung von Benutzereingaben, die sowohl Maus- als auch Tastatureingaben verarbeitet.

Ein zentrales Merkmal des `InputSystem` ist die Verwaltung der Maustasten-Zustände. Diese werden durch die Enumeration `ClickState` repräsentiert, die die Zustände `NO_ACTION`, `CLICK`, `HOLD` und `RELEASE` definiert. Diese Zustände werden in den Membern `m_leftClickState` und `m_rightClickState` gespeichert und können über die Methoden `GetLeftClickState()` und `GetRightClickState()` abgefragt werden. Dadurch müssen sich andere Klassen nicht um die Details der Zustandsverwaltung kümmern. Zusätzlich wird der aktive Maustasten-Zustand

durch die Enumeration `MouseButton` verwaltet, die die möglichen Maustasten wie `LEFT_BUTTON`, `RIGHT_BUTTON` und `NO_BUTTON` definiert.

Die Aktualisierung der Maustasten-Zustände erfolgt in der `Update()`-Methode, die regelmäßig aufgerufen wird. Die Methode `UpdateClickState(MouseButton mouseButton, ClickState& clickState)` wird dabei verwendet, um den Zustand einer Maustaste basierend auf den aktuellen *GLFW*-Eingaben zu aktualisieren. Dabei wird auch die aktive Maustaste berücksichtigt, um sicherzustellen, dass nicht beide Maustasten gleichzeitig aktiv sind.

Mit den Methoden `ScreenToWorld(const glm::vec2& screenPosition)` und `WorldToScreen(const glm::vec2& screenPosition)` gibt es auch Möglichkeiten zur Umrechnung zwischen Bildschirmkoordinaten und Weltkoordinaten. Zusätzlich kann mit der Methode `GetPickingRay(glm::vec3& out_origin, glm::vec3& out_direction)` ein Strahl erzeugt werden, der in die Richtung der aktuellen Mausposition in die Szene zeigt, um Interaktionen wie das Ziehen von Objekten zu unterstützen.

Die Behandlung des Mausrads erfolgt über eine statische Callback-Funktion `ScrollCallback(GLFWwindow* window, double xScroll, double yScroll)`, die von *GLFW* aufgerufen wird, wenn der Benutzer das Mausrad bewegt. Der Scroll-Offset wird in dem statischen Member `m_mouseScrollOffset` gespeichert und kann über die Methode `GetMouseWheelScrollOffset()` abgerufen werden.

Für die Tastatureingabe wird eine Map von `KeyboardObserver`-Objekten verwendet, die jeweils den Zustand einer bestimmten Taste verwalten. Der Zustand einer Taste kann über die Methoden `WasKeyDown(int key)`, `WasKeyPressed(int key)` und `WasKeyReleased(int key)` abgefragt werden Die Klasse `KeyboardObserver` speichert den Zustand einer Taste in den Membern `m_wasDown`, `m_wasPressed` und `m_wasReleased`, die in der `Update()`-Methode der `KeyboardObserver`-Klasse aktualisiert werden.

Eine detaillierte Übersicht der Klassenstruktur ist in Abbildung 2.3 dargestellt.

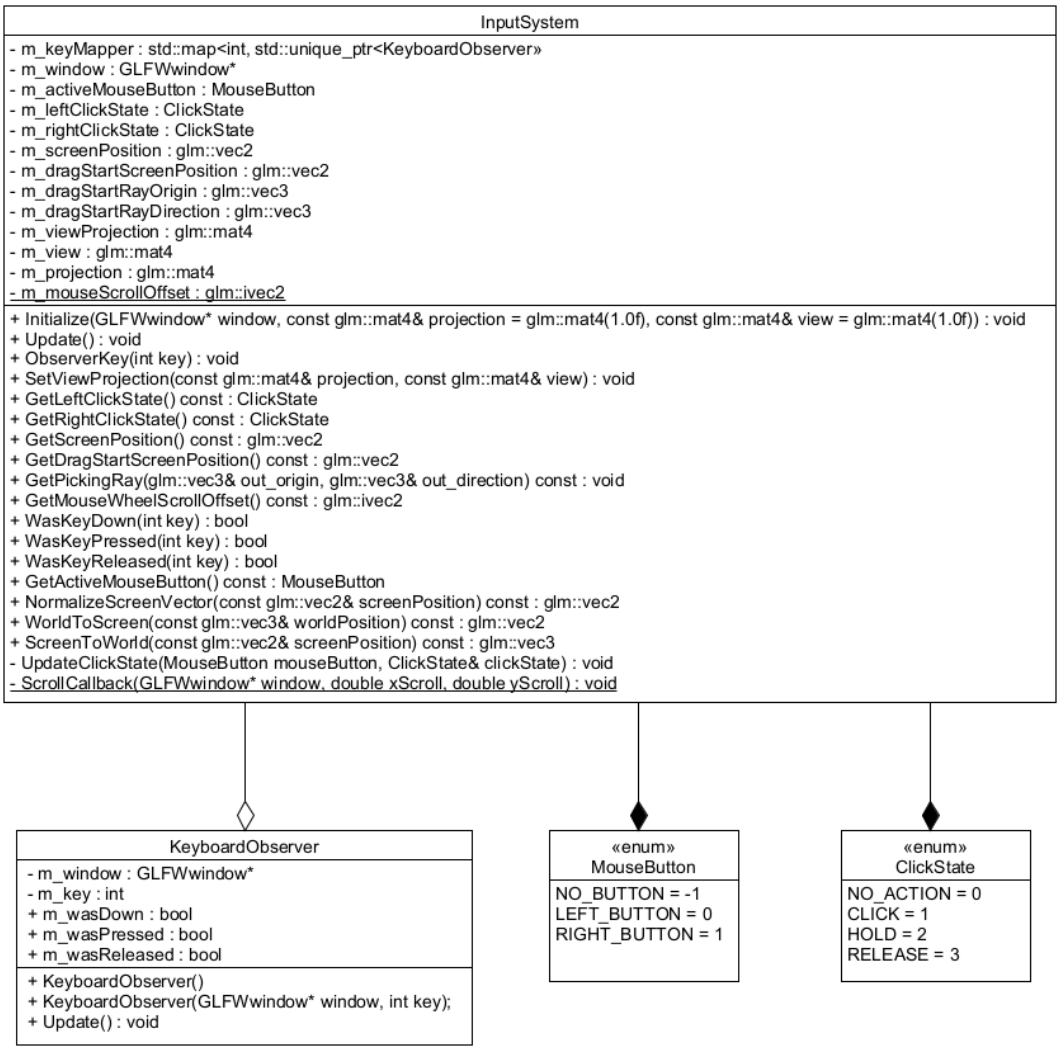

Abbildung 2.3: Klassendiagramm InputSystem und KeyboardObserver

2.5 CubieRenderer und ShaderUtil

Die Klasse `CubieRenderer` ist für das Rendern der einzelnen Cubies eines Rubik's Cube zuständig. Sie verwaltet die OpenGL-Ressourcen wie Vertex-Buffer, Shader und Texturen, die für die Darstellung der Cubies benötigt werden. Die Initialisierung der Render-Ressourcen erfolgt in der Methode `Initialize()`, die die Vertex-Daten für die Positionen, Farben und Texturkoordinaten der Cubie-Seiten generiert und in OpenGL-Buffern speichert. Zusätzlich wird ein Shader-Programm geladen, das für die Transformation und das Texturieren der Cubies verantwortlich ist.

Die Methode `Render(const glm::mat4& viewProjection, const glm::mat4& model)` zeichnet einen einzelnen Cubie unter Verwendung der übergebenen Transformationsmatrizen. Die Cubies bestehen aus 36 Vertices, die die sechs Seiten des Würfels repräsentieren. Eine Seite besteht also aus 6 Vertices(zwei Dreiecke). Die Textur wird aktiviert und gebunden, um die Oberfläche der Cubies zu gestalten. Nach dem Rendern werden die OpenGL-Ressourcen wieder freigegeben, um Konflikte mit anderen Render-Operationen zu vermeiden.

Die Klasse `ShaderUtil` bietet Hilfsfunktionen für das Laden und Kompilieren von Shadern sowie das Laden von Texturen. Dazu gehört `CreateShaderProgram(const char* vertexFilename, const char* fragmentFilename)`, diese Methode lädt mit `LoadFile(const char* fileName)` die Vertex- und Fragment-Shader aus Dateien, kompiliert sie und verknüpft sie zu einem Shader-Programm. Dabei auftretende Fehler werden durch die Methoden `PrintShaderLog(GLuint shader)` und `PrintProgramLog(GLuint shader)` protokolliert.

Mit `LoadTexture(const char* textureFilename)` wird eine 2D-Textur aus einer Bilddatei geladen und die Texturparameter für das Rendering konfiguriert.

Die Struktur und die Beziehungen der Klassen `CubieRenderer` und `ShaderUtil` sind detailliert in Abbildung 2.4 dargestellt.

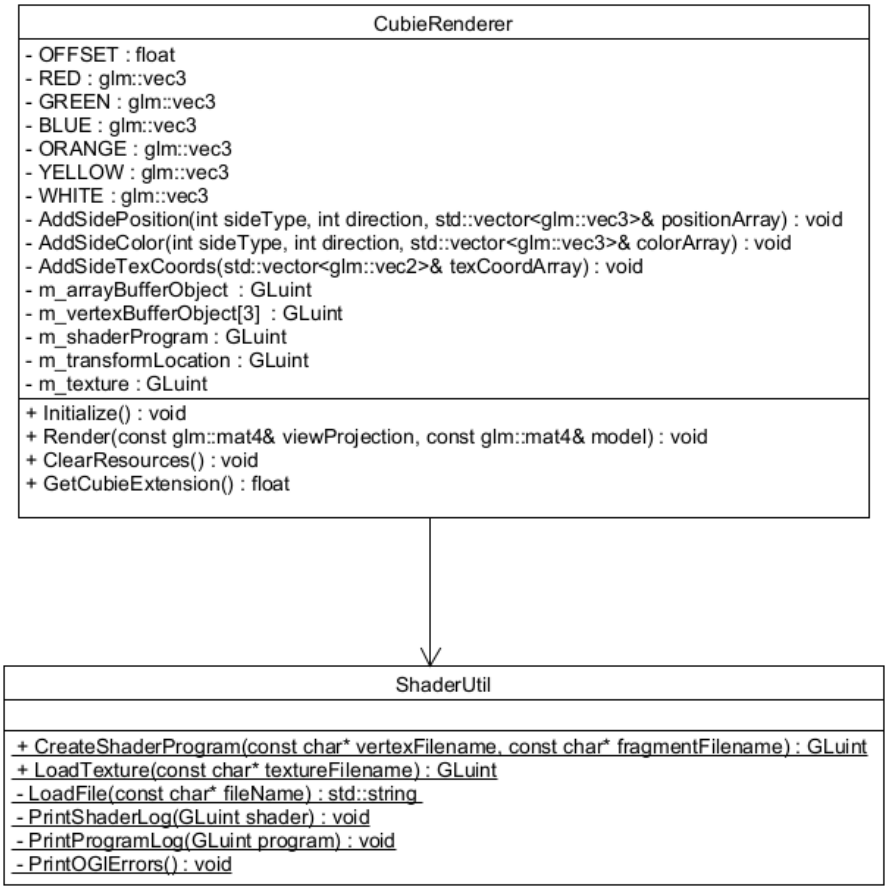

Abbildung 2.4: Klassendiagramm CubieRenderer und ShaderUtil

2.6 RubiksCube und Cubie

3

Zusammenfassung