Wirtschaft
Hauptcampus

HOCHSCHULE TRIER

Hausarbeit im Fach Spielekonsolenprogrammierung

Hausarbeit zum Rubiks Cube Projekt

Themensteller: Prof. Dr. Christoph Lürig

Name: Zishan Ahmed

Matrikelnummer: ***

Fachbereich: Informatik – Digitale Medien und Spiele (Spiele)

Abgabetermin: 04.02.2024

Inhaltsverzeichnis

1 Einleitung

In dieser Hausarbeit werden oberflächliche Funktionsweisen der wichtigsten Klassen und grundlegende Lösungsideen zum Rubiks Cube Spielprogrammierprojekt dokumentiert. Im Fokus stehen die entwickelten Ansätze und eine oberflächliche Darstellung der Implementierung.

2 Klassen

2.1 Klassendiagramm

Die folgende Abbildung stellt ein reduziertes Klassendiagramm mit den wichtigsten Klassen des Projekts dar. Es ist wichtig zu beachten, dass während der Implementierung die ursprünglichen Klassennamen umgestaltet wurden.

- Die Klasse `GameInterface` wurde in `AbstractGameInterface` umbenannt.
- Die Übersetzungseinheit `GameInterface` wurde zur Subklasse von `AbstractGameInterface`
- Die Datei `RubicsCube.cpp` wurde in `main.cpp` umbenannt, wobei `RubicsCube` zu einer eigenen Übersetzungseinheit wurde.

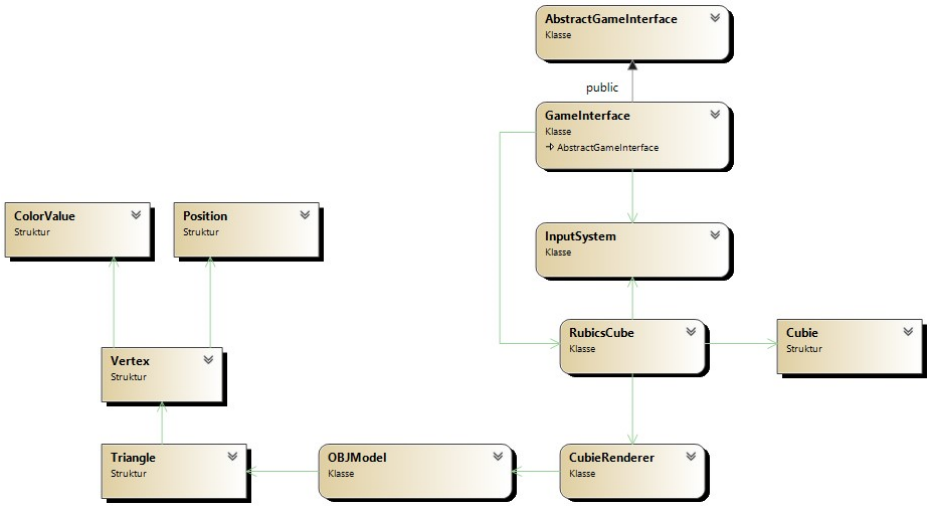

Abbildung 1: Reduziertes Klassendiagramm aller relevanten Klassen

2.2 main.cpp

Der Code in `main.cpp` unterscheidet sich nicht allzu sehr von dem Code aus der Vorlesung. In dieser Übersetzungseinheit wird *GLFW* initialisiert und ein Fenster wird

erstellt. Ein Spielobjekt vom Typ `GameInterface` wird erstellt und als aktuelle Schnittstelle festgelegt. Der Game- und Render-Loop wird gestartet, welcher das Spielfenster rendert und auf Benutzereingaben reagiert. Wenn das Fenster geschlossen wird, werden alle Ressourcen freigegeben und *GLFW* wird beendet.

Zusätzlich zu dem Code aus der Vorlesung habe ich einen Check eingebaut, der überprüft, ob das Fenster minimiert ist, um in diesem Fall einen unerwarteten Absturz zu verhindern. Außerdem habe ich die Bildfrequenz auf 125 FPS (1 Frame pro 8 Millisekunden) begrenzt und die Farben so interpretieren lassen, dass sie in sRGB für ein besseres Aussehen dargestellt werden.

2.3 AbstractGameInterface und GameInterface

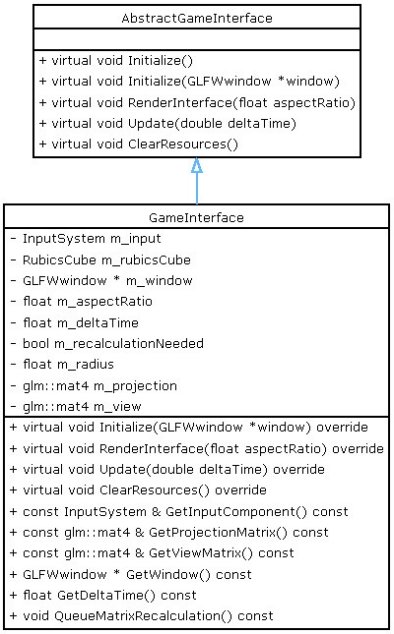

Abbildung 2: Klasse AbstractGameInterface und GameInterface

Diese abstrakte Klasse, mit dem Namen `AbstractInterface`, dient ausschließlich dem Zweck des einfachen Austauschs der Schnittstelle in der `main.cpp`. Diese Klasse besitzt keine sinnvolle Implementierung und sollte ausschließlich zum Vererben genutzt werden, um der Klasse `GameInterface` eine Grundstruktur vorzugeben.

Die `GameInterface`-Klasse repräsentiert eine implementierte Schnittstelle des Spiels und simuliert das Spiel mit einem Rubiks Cube. Die Methode `Initialize` initialisiert das Spielfenster und das Eingabesystem, dabei wird festgelegt, dass die

Leertaste beobachtet werden soll. Die Methode `RenderInterface` berechnet die Projektions- und View-Matrix neu, wenn dies erforderlich ist. Danach erfolgt das Rendern des Rubiks Cube.

Die Methode `Update` aktualisiert das Eingabesystem und den Rubiks Cube. Wenn die Leertaste gedrückt wird, erfolgt ein Zurücksetzen des Rubiks Cubes, andernfalls wird er aktualisiert. Die Kameraentfernung wird ebenfalls basierend auf der Mausradbewegung aktualisiert. Die Methode `ClearResources` gibt die Ressourcen des `RubicsCube`-Objektes frei.

Durch die Methode `QueueMatrixRecalculation` kann das `GameInterface` Objekt benachrichtigt werden, dass bei dem nächsten Update-Tick die Matrizen neu berechnet werden sollten.

Die restlichen Methoden sind alle Getter-Methoden, welche die jeweiligen Member zurückgeben. Die restlichen Memberattribute der Klasse `GameInterface` sind eher selbsterklärend.

2.4 InputSystem

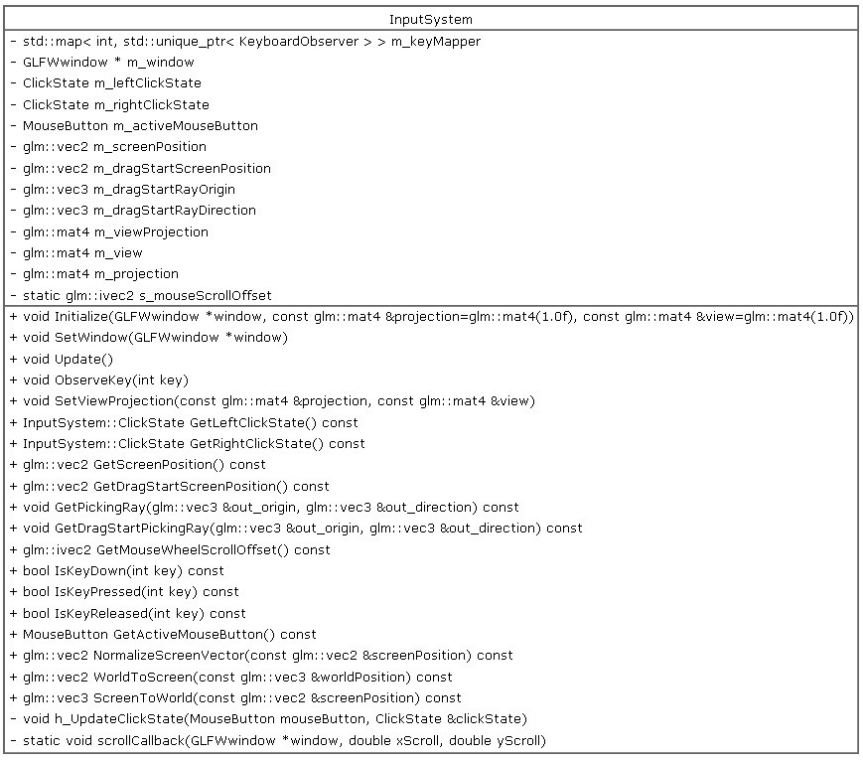

Abbildung 3: Klasse: InputSystem

Die Klasse `InputSystem` dient der Verwaltung von Eingaben durch Maus und Tastatur. Basierend auf dem Vorlesungscode wurde die Klasse angepasst und umgeschrieben, um den Anforderungen bei der Implementierung anderer Klassen gerecht zu werden. Aufgrund der Vielzahl von Membern möchte ich hier nur auf einige wichtige eingehen.

Im Unterschied zum Vorlesungsansatz habe ich versucht, die Maustastenzustände direkt in der Klasse mit der Enumeration `ClickState` zu lösen. Dadurch können die Zustände (`NO_ACTION`, `CLICK`, `HOLD`, `RELEASE`) von jeder anderen Klasse direkt durch die Methoden `GetLeftClickState` und `GetRightClickState` abgefragt werden, ohne weitere Logik in den jeweiligen Klassen implementieren zu müssen.

Die Klickzustände der rechten und linken Maustaste werden in der `Update`-Methode aktualisiert. Um Code-Redundanz zu vermeiden, wurde die Hilfsmethode `h_UpdateClickState` eingeführt, um die Maustastenzustände jeweils durch Parameterübergabe zu aktualisieren. Die Klasse speichert die aktive Maustaste, um sicherzustellen, dass während des Spiels nicht gleichzeitig Funktionen der rechten und linken Maustaste ausgeführt werden.

Das `InputSystem` ist auch für das Mauszug-Verhalten auf dem Bildschirm verantwortlich und merkt sich sowohl die 2D Position auf dem Bildschirm als auch den Strahl in die Tiefe im dreidimensionalen Weltraum, an der das Ziehen begonnen wurde.

Die Hilfsfunktionen `ScreenToWorld` und `WorldToScreen` sind für die Konvertierung zwischen dreidimensionalen Weltvektoren und 2D-Bildschirmvektoren zuständig und können von anderen Klassen von außerhalb verwendet werden.

Die Klasse kümmert sich ebenfalls um das Mausrad. Da *GLFW* eine Callback-Funktion für das Scrollen erfordert, sind die Callback-Funktion `scrollCallback` und das Memberattribut `s_mouseScrollOffset`, der den Scroll-Offset speichert, statisch. Daher sollte nur eine Instanz der `InputSystem`-Klasse existieren, um sicherzustellen, dass die Scrollfunktion ordnungsgemäß funktioniert.

Die Tastatureingabe hat sich im Vergleich zur Vorlesungsvorgabe nicht geändert. Eine Map mit den Werten vom Typ `KeyboardObserver` und Tasten vom Typ `int` (*GLFW Key Codes*) speichert die zu beachtenden Tasten und der Zustand der Taste lässt sich einfach mit den entsprechenden Methoden abfragen. Ein Objekt der Klasse `KeyboardObserver` kümmert sich dabei jeweils nur um den Zustand einer beobachteten Taste.

2.5 OBJModel und VertexStructs

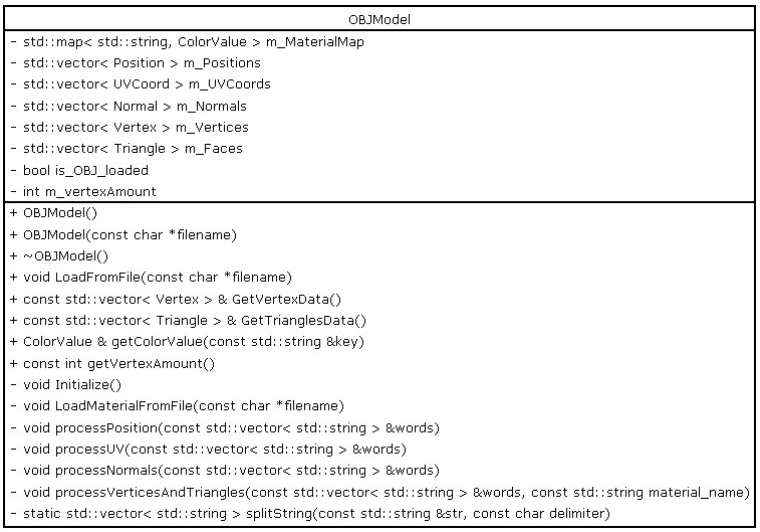

Abbildung 4: Klasse OBJModel

Die `OBJModel`-Klasse ist darauf ausgerichtet, OBJ-Modelle zu repräsentieren und bietet essenzielle Funktionen für das Laden und Verwalten dieser Modelle. Beim Aufrufen des Konstruktors mit dem Pfad oder Namen zur OBJ-Datei (z.B. "Cube.obj") werden sowohl die angegebene OBJ-Datei als auch die zugehörige MTL-Datei geladen.

Die `LoadFromFile`-Methode ist darauf spezialisiert, die eigentlichen Modelldaten aus der angegebenen OBJ-Datei zu laden. Die Methode durchläuft jede Zeile der OBJ-Datei und verarbeitet entsprechende Schlüsselwörter wie "v" für Positionen, "vt" für UV-Koordinaten, "vn" für Normale und "usemtl" für das Festlegen des aktuellen Materials auf ein Dreieck. Zwar werden Daten zu UV-Koordinaten und Normalen gespeichert, jedoch im Spiel selbst nicht verwendet. Alle Informationen werden dann entsprechend den `VertexStructs` Datenstrukturen in Listen gespeichert.

Die Methode `LoadMaterialFromFile` ist verantwortlich für das Einlesen von Materialinformationen aus der MTL-Datei. Die Materialnamen und die dazugehörigen Farbwerte werden in einer Hashmap gespeichert. Dabei werden auf Schlüsselwörter wie "newmtl" für die Definition eines neuen Materials und "Kd" für die dazugehörige Farbe geachtet.

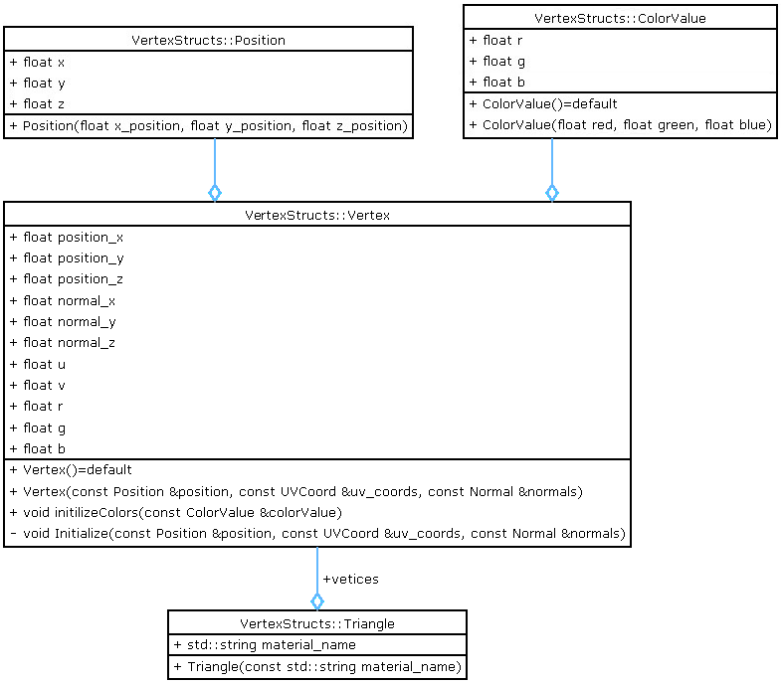

Abbildung 5: Klassen Position, ColorValue, Vertex und Triangle

Auf der *Abbildung 5* ist das Verhältnis der wichtigsten Strukturen, nämlich `Position`, `ColorValue`, `Vertex` und `Triangle`, aus dem Namespace `VertexStructs` dargestellt. Diese Strukturen dienen dazu, die aus den OBJ- und MTL-Dateien eingelesenen Informationen sinnvoll zu verteilen und zu speichern.

2.6 CubieRenderer

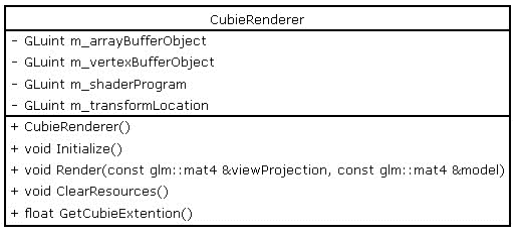

Abbildung 6: Klasse CubieRenderer

Die `CubieRenderer`-Klasse fungiert als Renderer für die Darstellung eines Cubies im Spiel. Der Konstruktor initialisiert verschiedene *OpenGL*-Objekte wie den Array Buffer, Vertex Buffer und Shader-Programm, die für das Rendern des Cubies erforderlich sind.

Statt fest definierter Punkte aus der Vorlesung lädt die `Initialize`-Methode ein dreidimensionales Modell eines Cubies („Cube.obj“) mithilfe der Klasse `OBJModel`. Sie extrahiert die Eckpunkte, Farben, Texturkoordinaten und Normalen des Modells. Dabei werden Texturkoordinaten und Normalen nicht weiter im Spiel verwendet. Zudem erstellt und kompiliert die Methode Shaderprogramme und initialisiert alle *OpenGL*-Objekte, wie Vertex-Arrays und Buffer für die Positionen und Farben eines Cubies.

Die `Render`-Methode nimmt View-Projektions- und Model-Matrix entgegen, berechnet die globale Transformation und verwendet das Shader-Programm sowie *OpenGL*-Objekte, um die Dreiecke eines Cubies basierend auf dem dreidimensionalen Modell zu rendern. Durch die `ClearResources`-Methode werden *OpenGL*-Objekte, darunter Buffer und Vertex Arrays, sowie Shaderprogramme freigegeben.

Die `GetCubeExtenstions`-Methode gibt eine konstanten `float` zurück, welche die Größe eines Cubies darstellt.

2.7 RubicsCube und Cubie

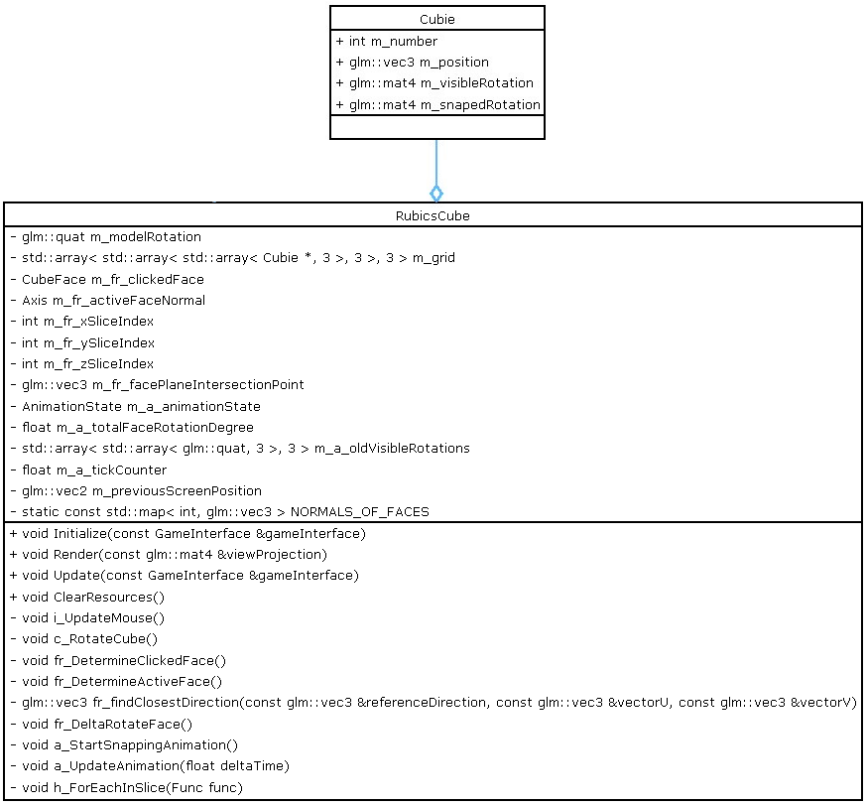

Abbildung 7: Klassen RubicCube und Cubie

Die Klasse `RubicsCube` implementiert einen Rubiks Cube im Spiel. In der `Initialize`-Methode wird die `CubieRenderer`-Instanz eingerichtet und ein dreidimensionales `Cubie`-Gitter wird erstellt. Dieses Gitter besteht aus `Cubie`-Objekten, die entsprechend ihren Positionen und Nummerierungen initialisiert werden. Die Methode extrahiert zudem ein `InputSystem`-Objekt aus dem `GameInterface`-Objekt, welches durch Parameter übergeben wird.

Die `Cubie`-Klasse ist eine blutleere Klasse, die Informationen über die Position, die sichtbare Rotation und die eingerastete Rotation eines `Cubie`-Objekts enthält. Die sichtbare Rotation repräsentiert die Rotation, welche für das Rendern eines `Cubie`-Objekts verwendet wird, während die eingerastete Rotation mit der jeweiligen Position im dreidimensionalen `Cubie`-Gitter verknüpft ist. Zu Debugging-Zwecken besitzt jeder `Cubie` außerdem eine Nummer.

Die `Render`-Methode ist für die visuelle Darstellung des Rubiks Cube verantwortlich. Sie verwendet die übergebene View-Projection-Matrix, um die

Rotation des gesamten Rubiks Cubes und die Position jedes Cubies korrekt zu berücksichtigen. Das `CubieRenderer`-Objekt wird genutzt, um jeden Cubie einzeln zu rendern.

Die `Update`-Methode verarbeitet Benutzereingaben, insbesondere Mausinteraktionen, mit Hilfe der Methode `i_UpdateMouse`. Während Animationen laufen, wird die `Update`-Methode genutzt, um die Animation des Rubiks Cube basierend auf `deltaTime` zu aktualisieren. Auch das Mausverhalten wird überwacht, um die Klickzustände und Positionen zu erfassen und die entsprechenden Aktionen einzuleiten. Beim Loslassen der linken Maustaste wird zusätzlich ein Sound abgespielt.

Die `ClearResources`-Methode gibt alle Ressourcen des Rubiks Cube frei. Dazu gehört die Freigabe des `CubieRenderer`-Objekts, sowie das Löschen aller `Cubie`-Pointer im dreidimensionalen `Cubie`-Gitter.

Die Methode `fr_DetermineClickedFace` in der `RubicsCube`-Klasse wird aufgerufen, wenn der Benutzer auf das Spielfeld klickt. Ihr Zweck besteht darin, die angeklickte Fläche des Rubiks Cube zu identifizieren. Dies geschieht durch die Bestimmung des Strahls in die Tiefe basierend auf der Mausposition. Die Methode iteriert dann durch die verschiedenen Flächen des Rubiks Cubes, überprüft, ob der Strahl eine Fläche schneidet und bestimmt schließlich durch Vergleich der Skalarprodukte die angeklickte Fläche sowie den Schnittpunkt.

Die Methode `fr_DetermineActiveFace` bestimmt die zu drehende Fläche des Rubiks Cubes. Sie wird aufgerufen, wenn der Spieler mit gedrückt gehaltener Maustaste eine bestimmte Strecke auf dem Bildschirm zurücklegt und kurz vor einer Drehung einer Fläche des Rubiks Cubes steht. Zunächst berechnet die Methode den Schnittpunkt zwischen dem Strahl beim Zugstart und der geklickten Fläche des Rubiks Cubes im Objektraum. Anschließend werden die Slice-Indices für jede Koordinatenachse (`m_fr_xSliceIndex`, `m_fr_ySliceIndex` und `m_fr_zSliceIndex`) ermittelt, indem geprüft wird, in welchem Bereich des Rubiks Cubes der berechnete Schnittpunkt liegt. Die Methode ermittelt ebenfalls den Schnittpunkt des aktuellen Strahls auf der geklickten Ebene im Objektraum und berechnet die Zug-Richtung auf der geklickten Fläche. Hierbei wird die Zug-Richtung

als die nächstgelegene Richtung zu den beiden verbleibenden orthogonalen Objektachsen bestimmt, welche nicht die Normale der geklickten Fläche sind. Die Methode `fr_findClosestDirection` gibt den Vektor zurück, welcher einer der übergebenen Referenzrichtung am nächsten liegt, indem sie das maximale Skalarprodukt zwischen der Referenzrichtung und den beiden gegebenen Vektoren auswählt. Abschließend erfolgt die Festlegung der aktiven Flächen-Normale (`m_fr_activeFaceNormal`), die orthogonal zur Normale der geklickten Fläche und der Zug-Richtung steht. Die Ergebnisse werden in den entsprechenden Member der Klasse gespeichert.

Die Methode `fr_DeltaRotateFace` implementiert die Teilrotation einer Fläche des Rubiks Cube basierend auf der Mausbewegung. Sie wird aufgerufen, wenn sich die linke Maustaste im Haltezustand und der Rubiks Cube sich im Rotationszustand befindet. Dabei wird die Änderung des Bildschirmpositionen-Vektors seit dem vorherigen Frame verwendet, um die Rotation zu berechnen. Die Methode berechnet den gezogenen Vektor, welcher sich auf der Ebene der geklickten Fläche befindet, und projiziert ihn auf einen Vektor, welcher orthogonal zu den Normalen der aktiven und geklickten Fläche steht, dabei wird hier die Arithmetik mit Indices ausgenutzt, um diesen Prozess effektiv durchzuführen. Es werden verschiedene Fälle berücksichtigt, um sicherzustellen, dass die Rotation in die richtige Richtung erfolgt. Die Rotationsänderungen werden dann auf jedes Cubie-Objekt der betroffenen Fläche angewendet.

Nachdem der Nutzer die Fläche ausreichend rotiert hat und die linke Maustaste loslässt, wird die `a_StartSnappingAnimation` aufgerufen. Diese ist für den Start einer Einrast-Animation verantwortlich. Die Methode normalisiert zunächst den Gesamtrotationsgrad der Fläche und rundet sie auf die nächsten 90 Grad ab, um eine definierte Position zu erreichen. Anschließend wird eine Rotationsmatrix erstellt, die die Gesamtdrehung der Fläche nach dem Einrasten darstellt. Diese Matrix wird dann auf jeden `Cubie` in der betroffenen Fläche angewendet, indem sie mit der aktuellen sichtbaren Rotation des `Cubie` multipliziert wird. Die alten sichtbaren Rotationen werden ebenfalls in den `Cubie`-Objekten gespeichert.

Die Methode `a_UpdateAnimation` wird aufgerufen, wenn sich der Rubiks Cube in einem einrast-animierten Zustand befindet. Diese Methode führt, basierend auf der

vergangenen Zeit (`deltaTime`), die Aktualisierung der Animation durch. Wenn der Animationsticker (`m_a_tickCounter`) nahezu 1 erreicht, was darauf hindeutet, dass die Animation abgeschlossen ist, werden die sichtbaren Rotationen der Cubies auf ihre eingerasteten Rotationen gesetzt. Zudem erfolgt eine Array-Rotation des dreidimensionalen `Cubie`-Gitters im Gegenurzeigersinn durch Transponieren der gesamten Fläche und Invertieren von Zeilen in einem Slice, abhängig von der Anzahl der von dem Nutzer durchgeführten Flächenrotationen. Nach Abschluss der Animation werden verschiedene interne Variablen zurückgesetzt, einschließlich des angeklickten Gesichts, des aktiven Achsnormalen und der Animationseinstellungen. Die Animation wird auf den stabilen Zustand zurückgesetzt und der Gesamtrotationsgrad wird auf 0 zurückgesetzt. Wenn die Animation noch nicht abgeschlossen ist, erfolgt eine schrittweise Aktualisierung der sichtbaren Rotationen der Cubies durch eine *Spherical Linear Interpolation* zwischen den alten und neuen Rotationen. Der Animationsticker wird entsprechend erhöht. Die spezielle Vorsichtsmaßnahme diente nur dazu, während der Testung, auch bei sehr kleinen Tickerhöhraten, Fortschritte in der Animation zu erzwingen.

Die `h_ForEachInSlice`-Methode ist eine geschickte Funktion, die abhängig vom aktuellen Slice des Rubiks Cube eine übergebene Funktion auf jedes `Cubie` anwendet. Diese Methode ermöglicht eine effiziente Iteration durch die `Cubie`-Objekte in dem aktuell aktiven Slice. Durch diesen Ansatz konnte ich insbesondere das Speichern von zusätzlichen Teilmengen des `Cubie`-Gitters für jeden Slice vermeiden.

3 Sonstiges

Die Abgabe meines Projektes war mein zweiter Versuch bei der Umsetzung dieses Projekts. Dieser Versuch basierte auf der Erkenntnis, dass mein erster Ansatz, bei dem ich die Rotation um den Rubiks Cube mit Kreiskoordinaten gelöst hatte, nicht zielführend war. Aufgrund der Schwierigkeiten, den Code entsprechend zu ändern, entschied ich mich für einen neuen Ansatz. In diesem zweiten Versuch begann ich direkt mit der Implementierung einer Klasse namens `LineRenderer`. Diese Klasse sollte mir beim Debuggen helfen und ein klareres Feedback zu den mathematischen Berechnungen liefern. Die `LineRenderer`-Klasse habe ich nicht mit in meiner Abgabe inkludiert.