HOCHSCHULE **TRIER**
Trier University of Applied Sciences
Informatik - Computer Science

Spielekonsolenprogrammierung

Rubik's Cube

Daniel Schreiber

Ausarbeitung

Betreuer: Prof. Dr. Christoph Lürig

Trier, Montag, 23 Januar 2017

Inhaltsverzeichnis

1

Einleitung

Die vorliegende Arbeit dient als Dokumentation der Semesterabgabe für das Modul Spielekonsolenprogrammierung im Wintersemester 2016/2017. Ziel der Abgabe war es, einen funktionsfähigen Rubik's Cube für die Playstation Vita zu realisieren. Die Umsetzung begann mit dem Entwurf und der Implementierung der Spiellogik. Die logische Repräsentation erfolgte mithilfe eines Arrays und wurde auf die später erstellte grafische Repräsentation gemapped. Im Laufe des Semesters wurden weitere Features wie simple Beleuchtung, Animationen, Eingabe über Buttons, sowie die Nutzung des Backtouch Panels implementiert.

Abb. 1.1. Rubik's Cube

2

Spiellogik

2.1 Struktur

Die Spiellogik besteht aus 4 Dateien:

1. Side.h (Header-File)
2. Side.cpp
3. Logic.h (Header-File)
4. Logic.cpp

Die Klasse Side dient der logischen Repräsentation einer Würfelseite und enthält ein Struct face, welches einen Integer Wert für die Platzierung des faces auf einer Seite enthält. Zudem wird die Farbe in genannten struct angegeben. Desweiteren existieren ein Integerwert, der die logische Seite darstellt, sowie ein zweidimensionales Array vom Typ face, um alle faces einer Seite abzuspeichern. Weiterhin enthält sie eine Getter-Methode, welche die Farbe als uint32_t zurückliefert, sowie den Enum-Typ Color (siehe Abb. 2.1).

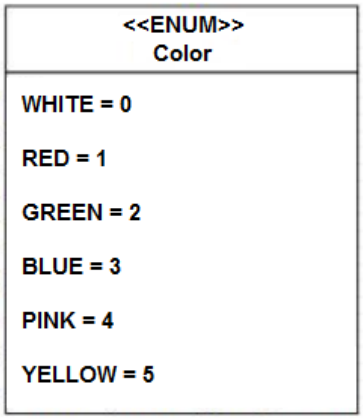

Abb. 2.1. Enum-Typ Color

Die Klasse Logic enthält:

1. Enums für die Seiten(Top, Bottom, Left, Right usw.)
2. ein Array, welches alle 6 Würfelseiten enthält
3. Methoden für die 9 logischen Rotationen

Um die Seiten anzusprechen und eine Fallunterscheidung durchzuführen wird der Enum-Typ SIDE_INDEX zur Hilfe genommen (siehe Abb. 2.2).

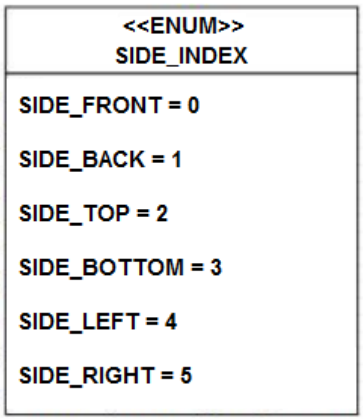

Abb. 2.2. Enum-Typ Side_Index

Die Methoden für die logischen Rotationen werden von der grafischen Repräsentation aus aufgerufen. Diese tauschen ausschließlich die Werte innerhalb der Arrays aus. Anschließend erhält die grafische Repräsentation die aktualisierten Werte und färbt den Würfel dementsprechend ein.

3

Grafische Repräsentation

Der komplette Rubik's Cube wird aus 27 Würfeln zusammengesetzt. Diese werden einzeln erstellt und so im Raum positioniert, dass sich ein gesamter Cube ergibt.

3.1 Der einzelne Würfel

Um einen Würfel zu erstellen werden die Klassen Cube.cpp sowie Cupe.h (Header-File) verwendet. Der Konstruktor ruft automatisch die Methode InitializeBuffer()

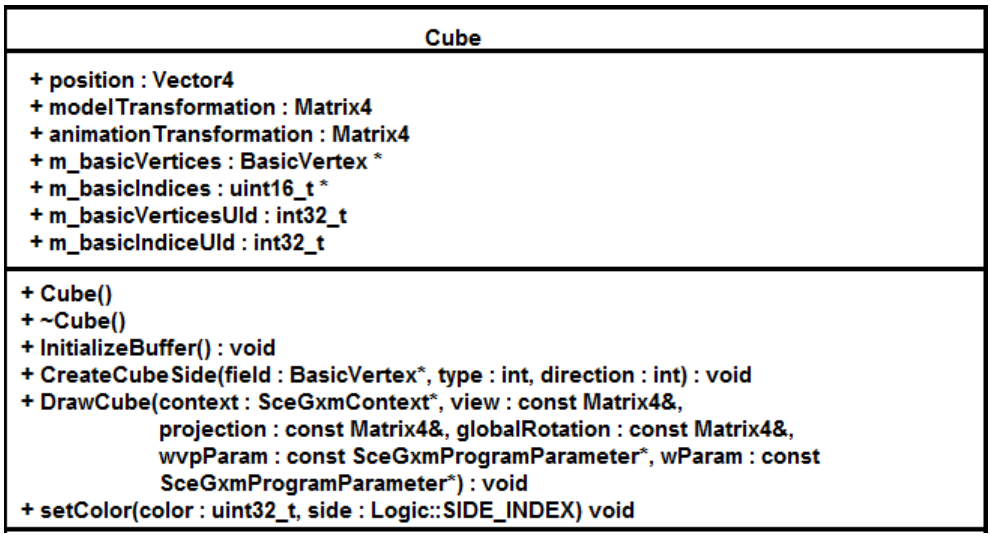

Abb. 3.1. Diagramm der Klasse Cube

auf, um entsprechend großen Speicher für einen einzelnen Würfel zu allokieren. Zudem findet hier auch der Methodenaufruf statt, um die einzelnen Seiten eines Würfels zu erstellen. Jede der Würfelseiten steht orthogonal zu einer der Achsen. Die Mitte des Würfels liegt dabei im Ursprung des Weltkoordinatensystems. Für jede Seite des Würfels sind vier Vertices mit sechs Indizes verantwortlich.

3.2 Der zusammengesetzte Würfel

Nachdem in der Klasse Main alle 27 Cube-Objekte erstellt und in einem Array gespeichert wurden, wird für jeden Würfel eine Translationsmatrix erstellt und in seiner eigenen modelTransformation gespeichert. Um dem gerenderten Würfel seine Gitternetzoptik zu verleihen, werden alle Flächen aller Würfel bei Start des Programms schwarz gerendert. Erst im Nachhinein wird aus der Logik ausgelesen, welche Fläche welche Farbe erhälten soll. Die Methode colorSide() dient dazu, die Logik auf die grafische Repräsentation zu mappen und erhält als Parameter eine Seite. Farbe und der SIDE_INDEX (siehe a4) werden als Parameter an den jeweiligen Würfel weitergereicht, um die passenden Vertices einzufärben.
Dieser Teil der Arbeit war mit großem Aufwand verbunden, da sich im Nachhinein herausgestellt hat, dass die Rotationsmethoden der Logik nicht korrekt waren und es durch fehlerhaftes Setzen der Indizes dazu führte, dass einzelne Flächen nach der Rotation falsch eingefärbt wurden. Diese Fehler konnten nur unter großem Zeitaufwand beseitigt werden.

3.3 Update Methode

Die Update Funktion der Klasse Main wird einmal je Frame aufgerufen. Für das rendern des Würfels ist eine World-View-Projection Matrix notwendig, welche sich aus den folgenden Matrizen zusammensetzt:

1. Perspective-Matrix
2. LookAt-Matrix
3. Rotation-Matrix

Die Perspective-Matrix erhält beim Erstellen als Parameter einen Field-of-View-Radianten, einen skalaren Wert für die Aspect Ratio, welcher sich aus dem Verhältnis der Seiten des PS Vita Displays zusammensetzt, sowie eine near- und far-Plane. Die LookAt-Matrix benötigt die Position der Kamera, die Position in die die Kamera schaut, sowie einen Up-Vektor. Die Rotationsmatrix, welche für die Rotation des gesamten Würfels angewendet wird, wird im nächsten Abschnitt erläutert.

3.4 Rotation des gesamten Würfels

Die Rotation des gesamten Würfels wird mithilfe eines Quaternions vollzogen. Hierzu werden x- und y-Position des linken Analogsticks verwendet um ein Einheitsquaternion zu erstellen. Das Rotations-Quaternion wird mit der in Spieleprogrammierung vorgestellten Formel erstellt und nachnormalisiert. Um dieses Rotations-Quaternion auf die globale Rotation anzuwenden ist es notwendig dies in eine Matrix4 umzuwandeln.

3.5 Beleuchtung

Um ein Beleuchtungsmodell zu implementieren war es notwendig dem Struct BasicVertex ein weiteres Attribut für die Normalen zu geben. Die Initialisierung der Normalen geschieht im Konstruktor der einzelnen Würfel. Diese werden zu Beginn auf 0 gesetzt. Beim Erstellen der einzelnen Würfelseiten werden die entsprechenden Normalen in die passende Richtung gesetzt. Die Berechnung der Normalen, des Lichts und der Farbe im Vertex Shader sieht wie folgt aus:

```
1  float3 normal = mul(float4(aNormal, 0.0f), w).xyz;
2  float light = saturate(-normal.z);
3  vColor = aColor*light;
```

4

Animation und Steuerung

4.1 Animation

Die Animation des Rubik's Cube wird durch folgende drei Schritte vollzogen:

1. Rotation bis Winkel 90 Grad erreicht hat und die Spiellänge der Animation erreicht wurde
2. Rotationsmatrizen zurücksetzen auf Ausgangsstellung
3. Aktualisierung der Logik und Anpassen der Farben

Um dies zu realisieren wurde die Klasse Animator mit passendem Header erstellt. Für die Animation wird ein Struct Animation erstellt, welches über folgende Variablen verfügt:

```
1  struct Animation {
2      float length;          //Playlength of Animation
3      float timeElapsed;
4      int side;
5      float startAngle;      //starts at zero
6      float targetAngle;        //90 degrees
7      bool running;
8      bool isVertical;
9      bool isHorizontal;
10     bool isMidZ;
11 } m_currentAnimation;
```

Die Klasse Animator verfügt über eine eigene Update-Methode, die als Parameter die aktuelle Zeit erhält und von der Main-Klasse aus jeden Frame aufgerufen wird. Sobald der Flag in m_currentAnimation auf true gesetzt wird, wird die Animation angestoßen. Dabei wird der Winkel über Zeit interpoliert. Es folgt ein Beispiel einer Rotation der linken Würfelseite:

```
1  if (m_currentAnimation.running)
2  {
3      float t = m_currentAnimation.timeElapsed / m_currentAnimation.length;
4      float targetAngle = m_currentAnimation.targetAngle * t +
5                          (1.0f - t) * m_currentAnimation.startAngle;
6
7      if(m_currentAnimation.side == Logic::SIDE_LEFT)
8      {
9          animationTransform = Matrix4::rotationX(targetAngle);
10         for(int i = 0; i <= 8; i++) {
11             cubes[i*3]->animationTransformation = animationTransform;
12         }
13     }
14 }
```

Sobald die gewünschte Stellung erreicht ist (in diesem Fall werden die 90 Grad genau dann erreicht, wenn die deltaTime die Spieldauer der Animation überschritten hat) wird die Logik über einen entsprechenden Methodenaufruf aktualisiert. Im nächsten Schritt wird durch ein Array, welches alle Cubes enthält, iteriert und deren Matrix4::animationTransformation auf eine Einheitsmatrix gesetzt, was dazu dient die Rotation rückgängig zu machen um Verzerrungen in der Darstellung zu verhindern. Anschließend werden die Farben so neu gesetzt, wie sie nach einer tatsächlichen Rotation auch angeordnet sein müssten. Der Ablauf der Animation wird hierbei von der Klasse InputSystem angestoßen (siehe Kapitel 4.3 Steuerung).

4.2 Steuerung

Um die Steuerung zu implementieren muss die lib "libSceCtrl_stub.a" inkludiert sowie der Header "ctrl.h" angegeben werden. Das InputSystem aktiviert über den nachfolgendem Aufruf in seinem Konstruktor das Digitalkreuz sowie die Buttons der PS Vita:

```
1  sceCtrlSetSamplingMode(SCE_CTRL_MODE_DIGITALANALOG_WIDE)
```

In SceCtrlData result wird das Ergebnis der Eingabe geschrieben und kann ausgelesen werden. Hierzu wurde im InputSystem eine eigene Update-Methode angelegt, welche jeden Frame Prüfmethoden aufruft, um die Eingabe abzufangen. Aus der Klasse Input heraus wird nach erfolgtem Tastendruck durch den Nutzer die passende Animation gestartet. Dies passiert nur falls keine Animation am abspielen ist.

4.3 Backtouch

Das Backtouch-Panel der PS Vita wird genutzt um den gesamten Würfel zu drehen. Zu Beginn der Main wird das hintere Touch Panel der Playstation Vita initialisiert. In der Update Methode wird in jedem Frame ein Objekt vom Typ SceTouchData, welches die Informationen über die Touchpoints enthält, erstellt. Sobald die Touch Punkte unterschiedlich sind wird xDif und yDif aus der Differenz von aktuellem Punkt und vorherigem Punkt berechnet. Mit diesen wird analog zur Joystick Steuerung die Rotationsmethode mithilfe eines Quaternions aufgerufen. Um die Drehung flüssig und steuerbar wirken zu lassen werden xDif und yDif mit einer Konstanten multipliziert, sobald sie als Parameter an die Rotationsmethode weitergereicht werden.

```
1  //Zu Beginn der Main
2  //Schalte Backtouch Panel ein
3  sceTouchSetSamplingState(SCE_TOUCH_PORT_BACK, SCE_TOUCH_SAMPLING_STATE_START);
4
5  [...]
6  //In der Update()
7  SceTouchData resultTouch;
8  sceTouchRead(SCE_TOUCH_PORT_BACK, &resultTouch, 1);
9  float xDif = 0.0f;
10 float yDif = 0.0f;
11 if(resultTouch.report[0].id != resultTouchStart.report[0].id) {
12 resultTouchStart = resultTouch; }
13
14 if(resultTouch.reportNum) {
15 xDif = resultTouch.report[0].x - resultTouchStart.report[0].x;
16 yDif = resultTouch.report[0].y - resultTouchStart.report[0].y; }
17
18 //Multiplikation mit Konstante
19 UpdateRotation(yDif * 0.01f, xDif * 0.01f);
```