

Computergrafik I - Beleg

Simon Fliegel
M.-Nr.: 53043
s-Nr.: s85344

6. Mai 2024

Inhaltsverzeichnis

1	Aufgabenstellung	2
2	Implementierung	2
2.1	Elementare Objekte	2
2.2	Shader	2
2.3	Kamera	3
2.4	Textur	3
2.5	Komplexe Objekte/ Szenen	3
2.5.1	Sonnensystem	3
2.5.2	Raum mit Lampe	4
2.6	Hauptprogramm	4
3	Ideen zur Verbesserung	5
4	Fazit	6
5	Quellen	7
A	Installations- und Bedienungsanleitung	8
B	Screenshots	10

Abbildungsverzeichnis

1	A screenshot of the full window with light turned off.	10
2	A screenshot of the full window with light turned on.	10

1 Aufgabenstellung

Es ist eine animierte Szene in OpenGL mit C++ zu implementieren. Es sollen mehrere unterschiedliche Lichtquellen verschiedene 3D-Objekte beleuchten. Die Lichtquellen sollen dabei verschiedenartig sein (ambient, diffuse, specular). Die Szene soll gleichzeitig in mehreren Ansichten und Projektionen Viewports dargestellt werden. Außerdem soll eine Nutzerinteraktion möglich sein.

2 Implementierung

Bei der Implementierung habe ich mich stark an [1] und den dazugehörigen Code-Beispielen [2] orientiert. Ich habe vorab die ersten Kapitel des Buches durchgearbeitet um einen Eindruck für den Umfang und die Struktur des Programms zu bekommen. Mein Anspruch war eine möglichst saubere und modulare Architektur, die problemlos eine Weiterentwicklung ermöglichen würde. Besonders bei OpenGL empfand ich es als sehr hilfreich, wenn Kernfunktionalitäten in eigene Klassen ausgelagert werden. Dadurch konnten diese leicht wiederverwendet werden und nach dem Testen als Fehlerquelle ausgeschlossen werden. Die Architektur hat den Großteil der Entwicklungszeit in Anspruch genommen aber ich hätte in Zukunft auch keine Angst davor, das Programm zu erweitern ohne das bestehende Funktionalität kaputt geht.

2.1 Elementare Objekte

Angefangen habe ich mit der Implementierung von elementaren geometrischen 3D-Objekten, wie `Cube`, `Sphere` [6] und `Cylinder` [7]. Diese Herangehensweise war sinnvoll, da man hier noch nicht viel Setup benötigt um die Objekte zu rendern und deren Richtigkeit zu testen. Spätestens beim Anlegen mehrerer verschiedener Objekte bietet sich das Auslagern des *Setups* in eine Basis-Klasse `BaseShape` an, da hier immer wieder die gleichen Schritte vorgenommen werden müssen. Das hat zur Folge, dass sich die konkreten Implementierungen der Objekte auf das algorithmische Erzeugen der Vertices und Indices beschränken und dass zentrale Einstellungen wie der Rendermodus global vorgenommen werden können. Das Ziel war es mehrere dieser Objekte zu verwenden um komplexere Objekte zu erzeugen (s. 2.5).

2.2 Shader

Da die Shader für jeden Bearbeitungsschritt notwendig sind, habe ich diese parallel immer weiter entwickelt. Zunächst habe ich nur statisch eine Farbe vergeben um die korrekte Darstellung der Objekte zu prüfen. Später habe ich Texturen und Beleuchtung hinzugefügt. Außerdem habe ich mich irgendwann für eine eigene Shader-Klasse entschieden (analog zu [3]), die den Umgang mit mehreren Shader-Programmen und vielen Uniforms deutlich vereinfacht. Ich habe mich auch dafür entschieden für komplexe Objekte eigene Shader-Programme zu verwenden. Das erschien mir anfangs sinnvoll, nachdem ich allein für das Sonnensystem neun Texturen brauchte und das zunehmend unübersichtlich wurde. Außerdem habe ich eine ObjektID zur Zuordnung von Objekten und Textur auf Shaderseite eingeführt und mir gefiel der Gedanke nicht, diese ID klassenübergreifend hochzuzählen. Allerdings hat das am Ende die Handhabung mehrerer Lichtquellen erschwert, weil ich sozusagen redundant die Informationen der Lichtquelle aus dem Sonnensystem-Shader (Sonne) an den Shader für die Raum-Szene übergeben musste, damit die Sonne die Beleuchtung des Raumes beeinflusst. Somit habe ich am Ende nicht die saubere Trennung der komplexen Objekte erhalten, wie ich es mir anfangs erhofft hatte. Im Nachhinein bin ich mir nicht mehr sicher, was die bessere Lösung gewesen wäre.

2.3 Kamera

Die Kamera-Klasse `FlyCamera` habe ich weitestgehend aus [4] übernommen und nur leicht angepasst. Die Callbacks für User-Inputs sind natürlich andere durch die Verwendung von `FreeGLUT` statt `GLFW` aber das ist keine große Hürde gewesen. Die Aufteilung in einzelne Kameraklassen kommt ebenfalls der Les- und Wartbarkeit sehr entgegen. Später habe ich dann noch eine zweite Kamera `FixedCamera` hinzugefügt, die statisch durch Position, Blickrichtung und FOV definiert ist. Da ich zur Darstellung der Szene in einem Viewport von der Kamera nur View- und Projection-Matrix sowie die Position benötige, habe ich ein Interface `AbstractCamera` eingeführt, das von beiden Klassen implementiert wird. Damit geschieht das Rendern eines Viewports nun sehr elegant und das Hinzufügen weiterer Viewports wäre problemlos in wenigen Zeilen möglich.

2.4 Textur

Nachdem ich schon erfolgreich Shader und Kamera in Klassen ausgelagert habe, war es mein Anspruch dasselbe auch mit Texturen zu probieren. Das hat es mir ermöglicht, problemlos viele Texturen zu verwenden und zwischen ihnen hin und her zu wechseln. Bei dem Entwurf der Klasse habe ich mich an [8] entlang gehangelt. Ein Fehler, der mich hier fast in den Wahnsinn getrieben hätte, war die korrekte Farbdarstellung der Texturen. Die Farben waren alle invertiert und ich musste statt `GL_RGB` die Konstante `GL_BGR_EXT`, da `FreeImage` die Farben in BGR-Reihenfolge speichert. Eine weitere Frage war der Umgang mit mehreren bis vielen 2D-Texturen. Bei der Recherche sind mir mehrere Möglichkeiten begegnet, wie dem laden mehrerer 2D-Texturen durch eine 3D-Textur um die Begrenzung der Texture-Units zu umgehen. Als mir nach weiterer Recherche klar wurde, dass diese Begrenzung im Bereich von mehreren tausend Texturen liegt, habe ich mich für die herkömmlichen Weg entschieden, die Texturen getrennt als 2D-Texturen zu laden. Am Ende bin ich auch hier nicht ganz zufrieden mit der Umsetzung. Besonders auf Shaderseite hätte ich mir eine generischere Lösung gewünscht ohne für jede Textur eine neue Objekt-ID zu vergeben bzw. auszuwerten. Außerdem funktioniert das Mapping der Texturen auf die Objekte über eine mitgelieferte ID, die über mehrere Klasssen hinweg konsistent sein muss. Eleganter wäre wahrscheinlich eine zentrale Vergabe bzw. Verwaltung von Texturen und IDs, die die programmweite Konsistenz sichert aber das würde den Rahmen dieses Belegs sprengen.

2.5 Komplexe Objekte/ Szenen

Nachdem alles oben genannte funktionierte, hat der Entwurf von zusammengesetzten komplexen Objekte den meisten Spaß gemacht. Da ich nun alles sauber getrennt hatte, konnten Fehler eigentlich nur in der Implementierung des komplexen Objekts liegen und ich musste nicht jedes mal dessen Bestandteile überprüfen.

2.5.1 Sonnensystem

Angefangen habe ich mit der Klasse `SolarSystem`, da ich hier Animation, Texturen und Beleuchtung kombinieren konnte. Durch die Vereinbarung der Strukturen `Planet` und `Sonne` konnte ich angenehm über die Planeten iterieren und die nötigen Transformations-Berechnungen durchführen. Anfangs dachte ich, ein Referenzwinkel für alle Planeten würde ausreichen wenn ich diesen abhängig von der Geschwindigkeit des Planeten um einen Faktor erhöhe. Da der Referenzwinkel aber im Wertebereich zwischen 0 und 2π läuft, wurden alle Planeten nach einer gewissen Zeit wieder auf ihren Ursprungsort zurückgesetzt. Die Lösung war natürlich für jeden Planeten in der Struktur einen eigenen Winkel mitzuführen und diesen dann abhängig von der Geschwindigkeit zu erhöhen, sodass alle Planeten unabhängig voneinander rotieren können. Bei den Texturen für die Planeten habe ich mich von *Solar System Scope* [9] bedient. Bei der Beleuchtung habe ich mich nur für ambient und diffuse entschieden, da mir specular im Sonnensystem (für die Planeten) nicht sinnvoll erschien. Als sowohl Texturen als auch Beleuchtung korrekt funktionierten, war ich sehr zufrieden mit dem Ergebnis.

2.5.2 Raum mit Lampe

Nach der Entwicklung des Sonnensystems (s. 2.5.1) hab ich mir überlegt eine Raum-Szene `RoomWithLamp` mit einer weiteren Lichtquelle zu bauen, die dann durch das Sonnensystem ergänzt werden sollte. Anfangs wollte ich diese Szene genau wie das Sonnensystem in einer Klasse implementieren aber da die Objekte sauber trennbar waren, habe ich sie letztendlich doch in eigene Klassen `Room` und `Lamp` aufgeteilt. Dadurch musste ich zwar mehrere Transformationsschritte durchführen und jeweils die Transformationsmatrix durchreichen aber das Verschieben der Objekte im Raum ist so sehr elegant geworden, da ich z.B. die Lampe als ganzes über eine Matrix transformieren kann. Bei dem Raum hatte ich anfangs nur einen Würfel mit Textur. Später habe ich mich dann doch noch dazu entschieden einen Boden einzufügen. Die Lampe war etwas schwieriger, da sie aus mehreren elementaren Objekten (s. 2.1) besteht und ich den Ort der Lichtquelle mit beachten musste. Bei Lampe hat sich die Verwendung von der gesamten Phong-Beleuchtungsmodells angeboten, da die Lampe durch die Reflexionen im Raum sehr realistisch wirkt. Zuletzt habe ich hier den Shader so angepasst, dass die Sonne als externe Lichtquelle im Shader hinzugefügt werden kann, aber nicht muss. Dadurch beleuchtet die Sonne den Raum auch bei ausgeschalteter Lampe und die Lampe wird als passives Objekt im Raum angestrahlt. Der Fokus bei der Erstellung dieser Szene lag auf der korrekten Anwendung mehrerer Transformationsstufen um die Objekte korrekt im Raum anzutragen und der korrekten Beleuchtung mit zwei (unterschiedlichen) Lichtquellen.

2.6 Hauptprogramm

Das Hauptprogramm ist aufgrund der sauberen Trennung der Klassen sehr übersichtlich geblieben und besteht im Wesentlichen aus der Initialisierung, dem Rendern der Szenen in den Viewports, den Callbacks, einer FPS-Anzeige und der `main`-Funktion. Bei der Begrenzung der FPS bin ich in eine nette Falle getappt. Anfangs hatte ich das Problem, dass meine Grafikkarte bei Ausführung des Programms dauerhaft auf fast 100% Auslastung lief. Grund dafür war, dass die `display`-Funktion durch `glutMainLoop` so oft, wie möglich aufgerufen wurde. Gelöst habe ich das Problem durch eine FPS-Begrenzung über den Callback `glutTimerFunc`, der die `display`-Funktion nur so oft aufruft, dass ca. 60 FPS erreicht werden. Das hat zwar den gewünschten Effekt auf die Performance erzielt (unter 10% Auslastung) aber alle Animationen sind *stehen geblieben*. Zu dem Zeitpunkt, als ich diese Änderung gemacht habe, hatte ich die Animationen noch nicht in Abhängigkeit der `deltaTime` (Zeit zwischen zwei Bildern) implementiert sondern so, dass es im aktuellen Zustand gut aussah. Da jetzt die Anzahl der Bilder um ca. Faktor 100 reduziert wurde, nahm auch die Animationsgeschwindigkeit um diesen Faktor ab und es stand alles still. Leider habe ich eine Weile gebraucht, bis ich auf die Ursache gestoßen bin. Anschließend habe ich die Klasse `AnimatedAbstractShape` eingeführt, welche dafür sorgt, dass immer eine `deltaTime` übergeben werden muss. Hätte ich das Programm auf einem anderen Rechner ausgeführt, hätte ich wahrscheinlich einen ähnlichen Effekt gemerkt, da die Animationsgeschwindigkeit abhängig von der Bildwiederholrate und damit der Leistung des Systems war. Die FPS-Anzeige habe ich dann im gleichen Zug noch mit eingefügt um die Performance in Zukunft besser im Blick zu haben. Dabei ist mir ein Problem aufgefallen, dass ich bisher nicht gelöst habe. Die User-Interaktionen umgehen momentan die FPS-Begrenzung und führen zu einer deutlich höheren Auslastung der Grafikkarte. In 3 ist dieser Punkt mit aufgeführt.

3 Ideen zur Verbesserung

Es gibt noch einige Dinge die ich gerne umgesetzt oder verbessert hätte, aber aufgrund vieler anderer Belege und eines privaten Zwischenfalls wollte ich das Thema jetzt vom Tisch haben. Meine Ideen sind:

- realistische Werte für das Sonnensystem (zumindest in Bezug auf die Größenverhältnisse und Geschwindigkeiten)
- Hinzufügen des Saturn-Rings
- FPS auch bei User-Interaktionen begrenzen
- flackernde Texturen fixen
- mehrere Lichteinstellungsmöglichkeiten (dimmen, Farbe ändern)
- bewegliche Kamera in der Raum-Szene halten
- Mauszeiger immer zentrieren und ausblenden um eine gute Kamerasteuerung zu ermöglichen
- FPS-Kamera

4 Fazit

Ich hatte großen Respekt vor der Bearbeitung dieses Belegs und habe auch über einen sehr langen Zeitraum daran gearbeitet. Allerdings habe ich dabei nicht nur die rudimentäre Anwendung von OpenGL sondern auch viele Dinge über C++ gelernt, sodass ich mich jetzt mit der Sprache deutlich wohler fühle als am Anfang. Teilweise ist mir die Umsetzung einer guten Architektur mit OpenGL nicht leicht gefallen, da OpenGL sehr Zustandsorientiert arbeitet und somit einige Operationen in manchem Kontexten nicht zulässig sind. Das hat die Trennung der Funktionalitäten in möglichst atomare Klassen erschwert und mich machmal zu Kompromissen gezwungen. Dennoch bin ich mit dem Ergebnis sehr zufrieden und kann mir vorstellen in Zukunft noch einige Dinge umzusetzen bzw. dieses Projekt als *OpenGL-Blaupause* für mich zu verwenden.

5 Quellen

Literatur

- [1] Joey de Vries, *Learn OpenGL - Graphics Programming*, Joey de Vries, 2020.
- [2] Joey de Vries, *LearnOpenGL*, 2024, <https://github.com/JoeysDeVries/LearnOpenGL>.
- [3] Joey de Vries, *LearnOpenGL*, 2024, <https://github.com/JoeysDeVries/LearnOpenGL/blob/master/includes/learnopengl/shader.h>.
- [4] Joey de Vries, *LearnOpenGL*, 2024, <https://github.com/JoeysDeVries/LearnOpenGL/blob/master/includes/learnopengl/camera.h>.
- [5] Song Ho Ahn, *Camera*, 2024, https://www.songho.ca/opengl/gl_camera.html.
- [6] Song Ho Ahn, *Sphere, Icosphere, Cubsphere*, 2024, https://www.songho.ca/opengl/gl_sphere.html.
- [7] Song Ho Ahn, *Cylinder, Prism, Pipe*, 2024, https://www.songho.ca/opengl/gl_cylinder.html.
- [8] *Tutorial 14 : Render To Texture*, 2024, <https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/>
- [9] *Solar System Scope*, 2024, <https://www.solarsystemscope.com/textures/>.

Anhang

A Installations- und Bedienungsanleitung

Installationsanleitung

Ich habe zwei Zip-Dateien bereitgestellt. Eine enthält die ausführbare Datei mit den benötigten Texturen und Shadern. Da ich relative Pfade verwende, muss folgende Ordnerstruktur für die ausführbare Version eingehalten werden:

```
s85344-beleg-compiled
├── s85344-beleg.exe
├── textures
│   └── *.png/jpg
└── shaders
    ├── roomWithLamp.vs
    ├── roomWithLamp.fs
    ├── solarSystem.vs
    └── solarSystem.fs
```

Die zweite Zip-Datei enthält das gesamte Projekt einschließlich Quellcode und müsste entsprechend gebaut werden. Ich habe zur Entwicklung *Visual Studio 2022* verwendet aber bin bei den Bibliotheken aus dem Praktikum geblieben. Deshalb hoffe ich, dass es keine Probleme bei der Kompilierung gibt.

Die Ordnerstruktur sollte wie folge aussehen:

```
s85344-beleg
├── s85344_cg1_beleg.sln
├── s85344_cg1_beleg
│   └── s85344_cg1_beleg.vcxproj
├── main.cpp
└── models
    ├── base_models
    │   ├── BaseShape.h
    │   ├── BaseShape.cpp
    │   ├── Cube.h
    │   ├── Cube.cpp
    │   ├── Sphere.h
    │   ├── Sphere.cpp
    │   ├── Cylinder.h
    │   ├── Cylinder.cpp
    │   ├── Plain.h
    │   └── Plain.cpp
    ├── AbstractShape.h
    ├── AnimatedAbstractShape.h
    ├── Lamp.h
    ├── Lamp.cpp
    ├── Room.h
    ├── Room.cpp
    ├── RoomWithLamp.h
    ├── RoomWithLamp.cpp
    ├── SolarSystem.h
    └── SolarSystem.cpp
    └── shaders
        ├── roomWithLamp.vs
        ├── roomWithLamp.fs
        ├── solarSystem.vs
        └── solarSystem.fs
```

```
└── textures
    └── *.png/jpg
└── util
    ├── cams
    │   ├── AbstractCamera.h
    │   ├── FlyCamera.h
    │   └── FixedCamera.h
    ├── Shader.h
    ├── Shader.cpp .4 Texture.h
    └── Texture.cpp
```

Bedienungsanleitung

In der Anwendung ist auf der linken Seite frei im Raum steuerbare Kamera. Die Kamera kann mit **W**, **A**, **S**, **D**, bewegt werden. Außerdem kann man sich mit der Maus umsehen und mit Mausrad zoomen. Auf der rechten Seite ist eine statische Kamera auf das Sonnensystem gerichtet ist. Die Lampe kann mit **Space** an- und ausgeschaltet werden.

B Screenshots

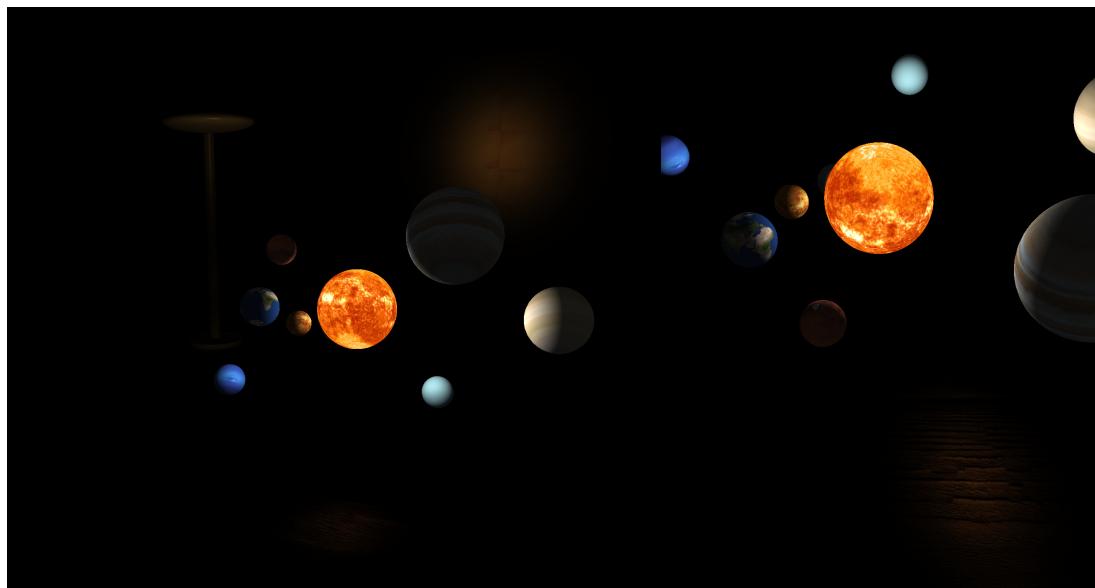


Abbildung 1: A screenshot of the full window with light turned off.

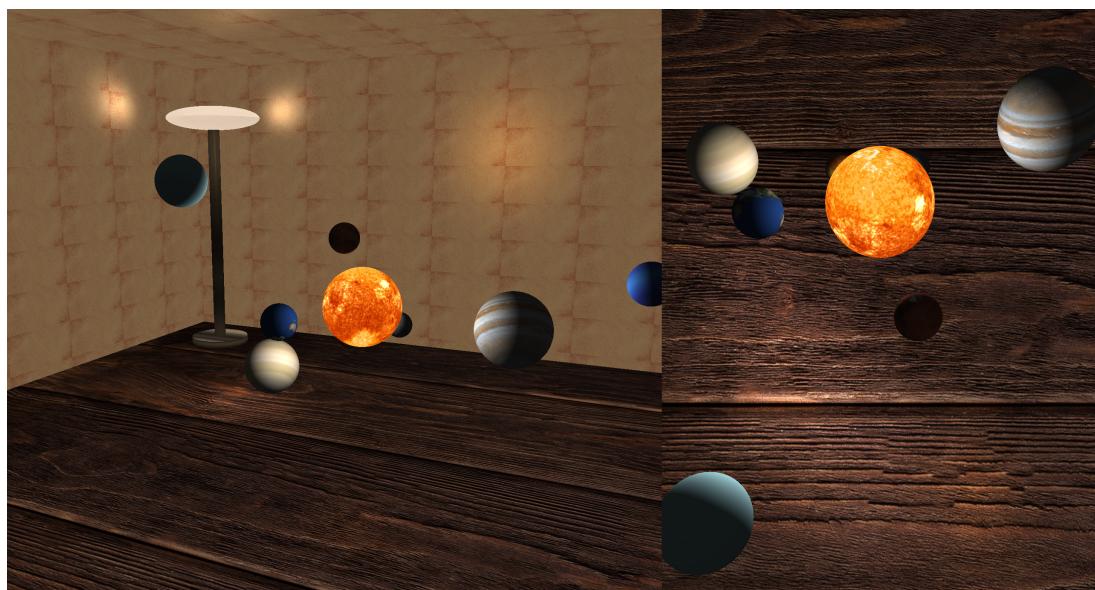


Abbildung 2: A screenshot of the full window with light turned on.