Aufgabe 3: Lichtblick

Martin Aumüller *
Universität zu Köln
basierend auf Aufgabe 4 von GraPA an der Universität Erlangen-Nürnberg,
`http://www9.informatik.uni-erlangen.de/GraPA/`

9. Juli 2013

1 Themen

In dieser Aufgabe werden folgende Themengebiete bearbeitet:

- Raytracing
- Quadriken und polygonale Objekte
- Beleuchtungs- und Farbberechnungen

2 Einführung

Ein Raytracer bestimmt mittels Strahlrückverfolgung, wie die Pixel im Bild einzufärben sind. Ein Raytracer kann in viele Teile strukturiert werden, von denen ein jeder eine sehr spezielle und klar abgegrenzte Aufgabe zu erledigen hat, etwa das Lesen der Szenenbeschreibungsdatei, das Erzeugen von Strahlen, das Bestimmen von Schnittpunkten von Strahlen mit Objekten, die Berechnung von Normalen und gespiegelten Strahlen, das Bestimmen der Farbe am Schnittpunkt, ... Der Schlüssel zum Entwerfen eines guten Raytracers ist es, die miteinander verknüpften Programmteile zu ermitteln und ein objektorientertes Programmskelett zu entwerfen, das eine effiziente Implementation und flexible Erweiterbarkeit erlaubt. In dieser Aufgabe wirst Du das Gerüst eines Raytracers vervollständigen und verbessern.

Die Basisversion des Raytracers kompiliert automatisch nach Eingabe von `qmake` und `make`. Wenn alles gut ging, dann solltest Du jetzt ein Programm namens `raytrace` haben. Sieh Dir die Kommentare in `src.pro` an, um zu erfahren, wie Du Debug- und Release-Varianten erstellst. Startest Du das Programm ohne Argumente, wird es Dir sagen, wie Du ihm den Namen der Datei mit der Szenenbeschreibung bekannt machst und welche sonstigen Optionen es versteht.

Das Programm erzeugt beim Rendern im selben Verzeichnis wie die Eingabedatei eine Bilddatei mit der Erweiterung `.ppm`, in der sich das erzeugte Bild im *Portable Bitmap*-Format befindet. Etwa `gimp` kann diese Bilder anzeigen.

*aumueller@uni-koeln.de

3 Aufbau

Ziel dieser Aufgabe ist das Verstehen eines einfachen Raytracers, so dass Du ihn erweitern kannst. Das Gerüstprogramm ist folgendermaßen aufgebaut:

main.cpp In dieser Datei befindet sich die Steuerung des gesamten Programms. Nachdem die globalen Szenenparamter gesetzt sind und die Szenenbeschreibung gelesen wurde, wird das Ergebnisbild aufgebaut, indem ein rekursiver Raytracing-Algorithmus für jeden Pixel aufgerufen wird. Das Resultat wird in eine Datei geschrieben und das Programm beendet.

input.y Diese Datei beschreibt die Grammatik der Szenenbeschreibungssprache. Hierzu wird `bison` benutzt, welches die Beschreibung der Grammatik liest und C-Programmtext zum Parsen der Szenenbeschreibung erzeugt. Du wirst diese Datei erweitern, so dass Dein Programm die Beispielszenen verstehen kann. Dokumentation zu `bison` findest Du bei [2].

ray.h, ray.cpp Hier ist die Klasse *Ray* zur Darstellung eines Strahls definiert. Ein Strahl kennt seine Position in der Szene und seine Farbe.

geoobject.h, geoobject.cpp Die Klasse *GeoObject* ist die Bassis für geometrische Objekte der Szene. Es greift dazu auf ein Objekt vom Typ *GeoProperties* zum Speichern der Oberflächeneigenschaften zurück. Dadurch können verschiedene Objekte der Szene diese Teilen. Alle Eigenschaften des Objekts können bei dieser Klasse abgefragt werden.

vector.h, vector.cpp Hier ist ein Template für dreidimensionale Vektoren definiert. Es ist bereits vollständig implementiert und verfügt über alle zur Bearbeitung der Aufgabe nötigen Methoden.

Zusätzlich zum Erweitern des Gerüstprogramms besteht Deine Aufgabe auch im vollständigen Dokumentieren des bereits vorhandenen Codes! Benutze dazu `doxygen`, das speziell formatierte Kommentare im Quelltext in eine Dokumentation Deines Programms z. B. im HTML-Format umwandeln kann. Richte Dich nach den Beispielen im Gerüstprogramm und schlage in [1] für genauere Information nach. Kommentiere v. a. auch, was Ihr im weiteren Verlauf programmiert!

Dein fertiger Raytracer sollte über folgende Fähigkeiten verfügen:

- Er liest Szenenbeschreibungen im in Abschnitt 3.1 beschriebenen Format.
- Er stellt Darstellen die durch die Eingabedatei beschriebenen Szene korrekt dar:
  - Quadriken und polygonale Flächen können ohne Einschränkung in derselben Szene verwendet werden,
  - jeder Strahl wird mit dem nächsten Objekt der Szene geschnitten und ein Schattentest wird von dieser Position aus durchgeführt,
  - die Beleuchtung durch jede sichtbare Lichtquelle wird korrekt berechnet,
  - der Strahl wird reflektiert und der gespiegelte Strahl wird richtig bestimmt und
  - das Ergebnisbild wird im *Portable Bitmap*-Format in einer Datei mit der Erweiterung `.ppm` gespeichert.
- Du hast in in einer sinnvollen Weise erweitert, siehe Abschnitt 4.
- Er muss eine eigene Szene, deren Beschreibung in der Datei `my_scene.data` abgelegt ist und die von Deiner Erweiterung Gebrauch macht, richtig rendern. Diese Szene muss aus mindestens fünf Objekten bestehen, von denen mindestens zwei polygonbegrenzt sein müssen. Natürlich muss sich diese Szene von der aller anderen Teilnehmer unterscheiden!

3.1 Die Szenenbeschreibung

Die hier aufgelisteten Parameter der Szenenbeschreibung sind Minimalanforderungen. Dein Raytracer kann darüber hinaus weitere Parameter verstehen, jedoch muss die angegebene Grammatik für Szenenbeschreibungen weiterhin verwendbar sein und korrekte Resultate liefern. Genauso muss Dein evtl. erweiterter Parser noch mit den Referenzdateien umgehen können.

Im nachfolgenden bedeutet `<int>` eine ganze Zahl, `<float>` eine Fließkommazahl und `<0-1>` eine Fließkommazahl mit Wertebereich zwischen 0 und 1. Ist ein solcher Parameter von `[n]` gefolgt, sind n Wiederholungen eines solchen Wertes notwendig.

Bildparameter

`resolution <int>[2]` – die horizontale und vertikale Bildauflösung in Pixeln

`background <0-1>[3]` – die Rot-, Grün- und Blauanteile der Hintergrundfarbe des Bildes (für Strahlen, die ins „Leere“ gehen)

Betracherparameter

`eyepoint <float>[3]` – die x-, y- und z-Koordinate des Betrachters

`lookat <float>[3]` – ein Punkt, der im Zentrum des Blickfelds des Betrachters liegt

`up <float>[3]` – ein Vektor, dessen Projektion im erzeugten Bild senkrecht nach oben gerichtet ist

`aspect <float>` – das Verhältnis aus horizontaler und vertikaler Größe der Szene

`fovy <float>` – der vertikale Öffnungswinkel der Kamera in Grad

Globale Beleuchtung

`ambience <0-1>[3]` – die Rot-, Grün- und Blauanteile des Korrekturewertes für die Beleuchtung, der den Lichtaustausch zwischen den Objekten annähern soll

Geometrien

`numsurfaces <int>` – die Anzahl der Flächen, hauptsächlich zum Überprüfen der Korrektheit der Szenenbeschreibung

`object quadric <float>[10]` – die 10 Koeffezienten A, $\ldots K$, die eine Quadrik durch folgende Gleichung implizit beschreiben:

$$\begin{aligned} Ax^2 + Bxy + Cxz + Dx \\ +Ey^2 + Fyz + Gy \\ +Hz^2 + Jz + K \qquad = 0. \end{aligned}$$

`object poly` – Einleiten eines Polygonobjekts, das durch die folgenden Parameter in der aufgeführten Reihenfolge beschrieben wird

`numvertices <int>` – Anzahl der Knoten

`vertex <float>[3]` – Koordinaten eines Knoten, muss für jeden Knoten wiederholt werden

`numpologyons <int>` – Anzahl der Polygone

`polygon <int> <int>[n]` – Anzahl der Knoten und Liste der Knoten in einem Polygon, muss für jedes Polygon wiederholt werden

Oberflächenparameter

`diffuse <0-1>[3]` – Farbe, unter der das Objekt erscheint, wenn es weiß beleuchtet wird

`ambient <0-1>[3]` – Farbe für ambiente Beleuchtung

`numproperties <int>` – Anzahl der nachfolgenden Eigenschaften

`mirror <float>` – Spiegelungskoeffizient

`specular <float>[2]` – Koeffizient für Stärke und Exponent für Abfallen der Glanzlichter

(Das ist allerdings kein Standard-Beleuchtungsmodell, vgl. [3].)

Lichtquellen Alle Lichtquellen befinden sich im Unendlichen und strahlen das Licht parallel ab.

`numlights <int>` – Anzahl der Lichtquellen in der Szene

`color <0-1>[3]` – Farbe der Lichtemission

`direction <float>[3]` – Richtung, aus der die Lichtquelle beleuchtet

Objekte

`object <int> <int>` – definiert ein Objekt der Szene durch Angabe der Indizies der zu verwendenden Geometrie sowie der Oberflächeneigenschaften

4 Eigene Erweiterung

Deine Erweiterung des Raytracers könnte das Darstellen zusätzlicher Objektarten ermöglichen:

- Weitere Oberflächenformen: z. B. Flächen höherer Ordnung, durch Extrusion entstehende Objekte, ...
- Zusätzliche Arten von Objekten: z. B. Nebel, lokale Lichtquellen, lichtemittierende Oberflächen, Volumen, ...
- Optische Effekte: Brechung, Farbseparation („Prisma“), ...
- *Constructive Solid Geometry* (CSG): Bildung neuer Objekte durch Anwendung von Mengenoperationen auf vorhandene Objekttypen (z. B. Kugel mit Loch, ...)
- flächige Lichtquellen: zum Modellieren realer Lichtquellen ist es erforderlich, Lichtquellen mit Ausdehnung definieren zu können
- Texturierung: das Aufbringen von Bildern auf die Oberfläche ermöglicht das Modellieren feiner Details und Strukturen
- Bump/Displacement Mapping: Verwendung einer Textur als Höhenfeld, mit dem die Reflexionsrichtung bzw. die Höhe der Oberfläche moduliert wird (Orangen, ...)
- Prozedurale Texturen: die Position auf der Oberfläche bestimmt mittels geeigneter Funktionen ihre Farbe (z. B. zur Modellierung von Holz, Stein, ...)
- Adaptives Super-Sampling: reduziere Stufenartefakte durch mehrmaliges Samplen, da das aber nur an Orten mit hohem Detailgrad nötig ist, mache die Anzahl der Samples davon abhängig
- Stochastisches Super-Sampling: moduliere die Richtung der Strahlen mit Zufallsfunktionen – das ermöglicht besseres Darstellen anisotropischer Effekte

Oder Du könntest den Raytracer effizienter machen:

- Hidden Surface Removal: zeichne vom Betrachter angewandte Seiten der Objekte nicht
- Bounding Volumes: vermeide aufwendige Schnittberechnungen durch einhüllende Objekte
- Space Partitioning: vermeide aufwendige Schnitberechnungen durch organisieren der Objekte in geeigneten Beschleunigungsstrukturen wie Octrees, uniformen Gittern, ...

Aber das alles sind nur Beispiele – Deine Erweiterung kann auch ganz anders beschaffen sein. Alle Erweiterungen müssen aber folgende Bedingungen erfüllen:

- Sie erweitert den Raytracer um zusätzliche Funktionalität oder macht ihn effizienter.
- Die Erweiterungen sind in einer Datei Namens `README` dokumentiert und lassen sich – falls sie nicht durch zusätzliche Parameter in der Szenenbeschreibung gesteuert werden – durch Kommandozeilenparameter an- und abschalten. D. h. der Raytracer muss immer noch in der Lage sein, sich wie der zu implementierende Standard-Raytracer zu verhalten.

Laufzeit (s)	dflt	REF1+2	REF3	REF4-9	REF10+12	REF11	dinopet	flamingo
Opteron, 2.2 GHz, 64 bit	2.4	5.7	9.9	1.7	6.3	4.7	314	294
Core 2, 2.53 GHz, 32 bit	2.1	5.2	9.3	1.0	5.1	3.8	162	157
Core 2, 2.53 GHz, 64 bit	1.3	3.5	6.0	0.7	3.5	2.5	101	104
Core i7, 2.66 GHz, 64 bit	1.2	3.2	5.4	0.6	3.5	2.5	88	96

Tabelle 1: Laufzeiten für die Beispielszenen

- Es handelt sich nicht um bloßes Beschleunigen des vorhanden Codes durch Hacks, Assemblerprogrammierung, ...– das ist die Aufgabe Deines Compilers. Das Programm sollte weiterhin verständlich bleiben!

5 Durchschnittliche Laufzeiten

In Tabelle 5 sind die Laufzeiten der Referenzimplementierung des Raytracers für die Testbeispiele angegeben. Deine Laufzeiten sollten nicht um Größenordnungen von den angegebenen abweichen.

6 Zusammenfassung

Hast Du den Raytracer dokumentiert und das Rendern von Polygonen implementiert, dann hast Du jetzt verstanden, wie ein Raytracer prinzipiell arbeitet. Bei der Programmierung von Shadern wird Dir dieses Wissen hilfreich sein.

7 Programmierrichtlinien

- Die abgegebenen Lösungen müssen auf dem Referenzsystem `vis.rrz.uni-koeln.de` am Lehrstuhl mit dem GNU-C++-Compiler der Version 4.4 fehlerfrei kompilieren.
- Von Euch selbst verschuldete Compiler-Warnungen (Kompilation mit `-Wall -O2`) haben einen Punktabzug zur Folge.
- Bitte haltet Euch an die in den Rahmenprogrammen vorgegebenen Formatierungsregeln:
  - Einrückung um 4 Zeichen,
  - keine Tabulatoren,
  - geschweifte Klammern auf separaten Zeilen.
- Kommentiert bei jeder Funktion ihre Aufgabe und ihre Parameter und formatiert diese Kommentare so, dass `doxygen` diese auswerten kann.
- Kommentiert nicht-triviale Codeblöcke.
- Benutzt sprechende Variablennamen: je größer der Sichtbarkeitsbereich, desto wichtiger ist das.
- Haltet Euch an *eine* Konvention für die Namensgebung der Variablen.
- Schränkt den Sichtbarkeitsbereich Euerer Variablen so weit wie möglich ein, recyclet eine Variable nicht sinnlos.
- Wann immer möglich, definiert Variablen nicht bevor Ihr sie initialisieren könnt.
- Keine Zuweisungen innerhalb der runden Klammern nach `if`, möglichst auch nicht nach `while`.
- Belegte Resourcen sind wieder freizugegeben.

8 Bewertungsrichtlinien

Wenn Du die folgenden Bedingungen erfüllst, dann kannst Du die Höchstpunktzahl von 20 Punkten erreichen:

1 Punkt Der Quelltext ist vollständig in objektorientiertem C++ verfasst. Der Code kompiliert auf dem Referenzsystem mit dem GNU C++-Compiler ohne Fehler und erzeugt mit den Optionen `-Wall -O2` keine unnötigen Warnungen. Auch die anderen Programmierrichtlinien wurden befolgt.

3 Punkte Der gesamte Quelltext – d. h. sowohl das Gerüstprogramm als auch Deine Ergänzungen – ist in englischer Sprache kommentiert und die Kommentare erklären die algorithmische Struktur des C++-Codes. Die Kommentare sind so formatiert, dass sie das Erzeugen einer HTML- und LaTeX-Dokumentation mit `doxygen` erlauben.

1 Punkt Der Parser ist in der Lage, Szenen im angegebenen Format zu lesen.

1 Punkt Der Raytracer kann sowohl polygonale Objekte als auch Quadriken verarbeiten.

3 Punkte Schnittpunkte werden sowohl für konkave als auch konvexe Polygonobjekte richtig berechnet und von ihnen reflektierte Strahlen besitzen die korrekte Richtung.

2 Punkte Primärstrahlen werden richtig erzeugt.

1 Punkt Die Objekte verdecken sich entsprechend ihrer Anordnung.

1 Punkt Ambiente Beleuchtung wird berücksichtigt.

1 Punkt Glanzlichter werden richtig berechnet.

2 Punkte Der Raytracer wurde erweitert und die Erweiterung wurde in einem `README` dokumentiert.

2 Punkte Zur Demonstration Deiner Erweiterung hast Du in `my_scene.data` eine vollständig eigene Szene beschrieben, die mindestens fünf Objekte enthält.

2 Punkt Die Laufzeiten Deines Raytracers weichen nicht erheblich von der Beispielimplementation ab.

Literatur

[1] Doxygen manual. `http://www.stack.nl/~dimitri/doxygen/manual.html`, 2010.

[2] Charles Donelly and Richard Stallman. Bison: The yacc-compatible parser generator. `http://www.gnu.org/software/bison/manual/pdf/bison.pdf`, 2005.

[3] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 2nd edition, 1990.

[4] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1990.

[5] Mel Slater, Anthony Steed, and Yiorgos Chrysanthou. *Computer Graphics and Virtual Environments: From Realism to Real-Time*. Addison Wesley, 2002.

[6] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 3rd edition, 1997.

[7] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, 2004.