QRay - Entwicklerdokumentation

Daniel Grün

1 Was ist qray?

QRay ist ein Raytracing-Programm für Linux/X/Qt, geschrieben in C++. Diese Dokumentation bezieht sich auf die Entwicklerversion im Oktober 2003.

2 Funktionsweise

QRay ist objektorientiert programmiert und enthält drei grundlegende Objekte:

- **QRayObject**, eine begrenzte oder unbegrenzte Ebene im Raum, also z.B. ein Parallelogramm. Alle von QRay dargestellten Körper werden aus diesen Objekten zusammengesetzt.
- **QRayLight**, eine punktförmige Lichtquelle von beliebiger Farbe und Intensität.
- **QRayView**, eine „Kamera“, die das auf dem Bildschirm dargestellte Bild aufnimmt.

Es werden beim heutigen Entwicklungsstand folgende Objekte unterstützt:

- **QRayPlain**, unbegrenzte Ebene
- **QRayRhomboid**, Parallelogramm
- **QRayRhomboid3D**, „raumliches Parallelogramm“, z.B. Quader; besteht aus 4 QRayRhomboid-Objekten

Alle diese Objekte sind von QRayObject abgeleitet und haben daher folgende gemeinsame Variablen und Zugriffsfunktionen:

- `vektor a` - Positionsvektor
- `vektor n` - Normalvektor (orthogonal zur Ebene, wird berechnet mit Kreuzprodukt)
- `vektor u, v` - Ebenenvektoren, auf denen die Ebene „liegt“
- `QColor c` - Farbe
- `double s` - Abhängigkeit der Helligkeit vom Blickwinkel: „Spiegel-Faktor“
- `double t` - Abhängigkeit der Helligkeit vom Lichteinfallwinkel

Nach Erzeugung des QRay-Objekts (einem QWidget, in dem die gerenderte Grafik dargestellt wird) und Initialisierung der gewünschten Objekte, Kameras und Lichtquellen kann durch Aufruf der Funktion `QRay::paintView()` das Bild auf dem Bildschirm erzeugt werden. Die Funktion ruft dabei `QRayView::paintView()` auf. Die Funktionsweise dieser Funktion ist schrittweise:

1. Berechnen der Winkel für Pixel $P(x|y)$: $\alpha = tan^{-1}(\frac{x}{\text{Bildbreite}})$, $\delta = tan^{-1}(\frac{y}{\text{Bildhöhe}})$. Durch die Faktoren `width` und `height` kann hier eine Verzerrung des Blickfelds erreicht werden.
2. Berechnen eines Vektors $\vec{r}$ mit Ursprung in der Kamera und Richtung nach o.g. Winkeln mit Komponenten $x_r = sin\alpha$, $y_r = sin\delta$ und $z_r = cos\alpha + cos\delta$.
3. Berechnen der Schnittpunkte zwischen der Verlängerung des Vektors $\vec{r}$ und den Objekten im Raum (Matritzenrechnung siehe Dokumentation Holger Kadau)

4. Auswählen des nächstgelegenene Schnittpunkts

5. Aufruf der Funktion `QRayObject::color()` und Zeichnen des daraus berechneten Farbwerts

Die eigentliche Berechnung des Farbwertes erfolgt also durch `QRayObject::color()`. Die Funktion ruft `QRayObject::pointColor()` auf, in der folgendermaßen vorgegangen wird:

1. Berechnen des „reflektierten Blickes'"

2. Berechnen der Helligkeit für jede Lichtquelle und -farbe (Rot, Grün, Blau) einzeln unter Berücksichtigung von *Entfernung* (Abnahme der Lichtstärke im Quadrat mit der Entfernung) sowohl von der Lichtquelle zum Objekt als auch vom Objekt zur Kamera, *Schattenwurf*, also mögliche Schnittpunkte vom Vektor zwischen Lichtquelle und Objekt mit anderem Objekt, *Intensität* der Lichtquelle, Winkel β zwischen Normalvektor und *einfallenden Lichtstrahl* nach $cos^t\beta$ und Winkel γ zwischen direkt reflektiertem Lichtstrahl und *„reflektiertem Blick"* nach $s^{-\gamma^2}$.

3. Berechnung des Farbwert unter Berücksichtigung des Eigenfarbwerts des Objekts und Überblendung (maximaler Farbwert `0xffffff`)

Dieser Vorgang wird für jedes Pixel wiederholt.
Die Eigenschaften von Lichtquelle und Kamera sind über ein Dialogfenster einstellbar.
Im Quelltext können leicht weitere Objekte und Lichtquellen, z.B. zur Demonstration additiver Farbmischung, hinzugefügt werden.

3 Ausblick

Mögliche Weiterentwicklungen sehe ich in der Darstellung von gekrümmten Oberflächen (z.B. Kugeln), in Spiegeleffekten und in der Generierung von Landschaften (evtl. mit Fraktalen) unter Einbeziehung von Texturen für die Oberflächen der Objekte.