Echtzeitgrafik VU

JustDukeIt

Dokumentation

Granzer Wolfgang
0026013
E-881
woif@woif.org

Praus Fritz
0025854
E-881
fritz@praus.at

1. Shortcuts

Während des Programmablaufes können online einige Einstellungen geändert werden:

‚0'	Pause
‚b'	aktiviert/deaktiviert Backface Culling
‚c'	zeichnet die Bezier Kurve des Bahnverlaufes der grünen Duke
‚d'	Start das Demo, bzw. startet die Kamerafahrt (oder setzt sie zurück)
‚h'	blendet den Himmel aus bzw. ein
‚m'	veranlasst die grüne Duke einen Wheely zu machen (yeah! ;-))
‚o'	zeichnet den Koordinatenursprung
‚p'	aktiviert/deaktiviert Motion Blur
‚l'	ändert Parameter für Motion Blur
‚s'	schaltet den Schatten aus bzw. ein
‚t'	blendet das Terrain aus bzw. ein
ESC	Beendet die Demo

2. Demoablauf

Beim Starten des Programms gelangt man zum Startbildschirm. Durch drücken der Taste ‚d' werden die notwendigen Objektdaten geladen bzw. berechnet. Danach können bereits die fahrenden Motorräder beobachtet werden. Durch drücken der Taste ‚d' wird die Kamerafahrt initialisiert. Dabei synchronisiert sich die Kamera mit dem Ablauf der Motorräder („Synchronize… „ links oben) d.h. es wird gewartet bis die Motorräder einen idealen Zeitpunkt für den Start der Kamerafahrt erreichen. Ist dieser Zeitpunkt erreicht, beginnt die Kamera die vordefinierte Bahn abzufahren („Following…" links oben; man beachte den Überholvorgang in der Kurve, wo die Selbstschattung sehr gut zu erkennen ist). Nach kurzer Zeit erreicht sie nun den Endpunkt („Waiting … „ links oben), wo man noch mal das Vorbeifahren der Motorräder erkennen kann (man kann in diesem Punkt wiederum sehr schön die Selbstschattung erkennen). Anschließend befindet sich die Kamera wieder im Ausgangszustand, wo von neuen eine Kamerafahrt gestartet werden kann.

3. Schatten

Zur Berechnung von dynamischen Schatten wurde ein Shadow Volume Algorithmus verwendet. Beim Laden der 3D Studio Max Objekte werden einmalig pro Fläche die Flächengleichungen berechnet. Diese Flächengleichungen werden später für die Sichtbarkeitsberechnung benötigt. Weiters werden zu Beginn für jede Fläche die angrenzenden, benachbarten Flächen bestimmt. All diese Informationen werden für die Schattenberechnung benötigt. Außerdem werden diese einmalig beim Starten des Programms berechnet und bleiben während des Programmablaufes konstant.
Allerdings ändern sich natürlich die Weltkoordinaten der Objekte, die Schatten werfen. Dadurch würden sich auch die Flächengleichungen ändern. Um nicht bei jedem Frame alle Flächengleichungen neu zu berechen (wären in unserem Programm einige Tauschende), wird die Lichtquellenposition in Objektkoordinaten rück transformiert. Dadurch wird einiges an Rechenaufwand gespart, da somit nur mehr 3 Matrixmultiplikationen für das Rückrechnen der Lichtquellenposition notwendig ist. Als zusätzliche Schwierigkeit trat bei uns das Problem auf, dass sich die Reifen der Motorräder zusätzlich zu den normalen Transformationen noch um die eigene Achse drehen. Daher werden für die beiden Reifen die Lichtquellen extra berechnet.
Anschließend wird bestimmt, welche Flächen sichtbar sind. Diese Information ist notwendig um die Silhouette der Objekte zu berechnen. Ist eine Fläche sichtbar und deren Nachbarfläche hingegen unsichtbar, handelt es sich bei der Berührungskante um eine Silhouettenkante (die Umkehrung gilt natürlich auch). Von diesen Kanten aus, wird nun das Schattenvolumen in die „Unendlichkeit“ gezeichnet. Das Problem, wie weit man zeichnen soll löste sich von selbst, da bei uns die Straße und das Terrain mindestens auf z=0 liegen.
Um nun zu bestimmen, ob sich ein Bildpunkt im oder außerhalb des Schattenvolumen befindet, wurde der Stencil Buffer verwendet. Im ersten Pass wurde bei allen Frontfaces des Schattenvolumens der Stencil Buffer um 1 erhöht. Im zweiten Pass hingegen wurde bei einer Backface der Stencil Buffer erniedrigt.
Nach den beiden Passes wurde nun ein graues Rechteck über den Framebuffer geblendet. Neben dem Tiefentest wurde auch der Stencil Test aktiviert. War der Tiefentest erfolgreich und ist der Stencilwert im entsprechenden Fragment ungleich 0, wurde dieses Fragment überblendet, da dieses Fragment im Schatten liegt.

4. Motion Blur

Der Motion Blur wurde einerseits mit Hilfe des „Accumulation Buffers“, als auch via „Rendering to Texture“ implementiert. Grund dafür war, dass wir auch auf einer Geforce 2 TI entwickelten und diese keinen Accumulation Buffer besitzt. Des Weiteren war die Implementierung mit dem Buffer trivial.

4.1. Rendering to Texture

Der Ablauf der Display Funktion ist wie folgt:

1) Zuerst wird mit prepareMotionBlur() der aktuelle MotionBlurTextur – Inhalt gerendert, die zu blurrenden Objekte hinzugefügt und wieder in die Textur zurück gezeichnet.
2) Nachdem die Szene ein weiteres Mal komplett gezeichnet wurde, wird mit renderMotionBlur() über den ganzen Bildschirm ein Quadrat mit der MotionBlurTextur gelegt.

4.2. Accumulation Buffer

Die Funktion des Blur Effekts läuft in 2 Schritten ab.

1) In den Accumulation Buffer blenden
2) Aus dem Accumulation Buffer in den Frame Buffer blenden

Dies wurde mit diesen 3 Zeilen Code bewerkstelligt.

```
glAccum( GL_MULT, MB_UPDATE ); // multiply acc.buff. by MB_UPDATE
glAccum( GL_ACCUM, 1-MB_UPDATE ); // write to acc.buff.
glAccum(GL_RETURN,1.0); // write back to framebuffer
```

5. Entwicklungsumgebung / Systemvorraussetzungen

Rechner 1:

Intel 3,06 Ghz
1 GB RAM
Radeon 9700 Pro

Effekt	FPS
ohne	220
Schatten + render to texture – Blur	130
Schatten + accumulation Buffer – Blur	100

Rechner 2:

Athlon 1700+
512 MB RAM
Geforce 2 TI

Effekt	FPS
ohne	86
Schatten + render to texture - Blur	43
Schatten + accumulation Buffer – Blur	0

6. Quellen

Open GL Red Book
http://www.gametutorials.com
http://nehe.gamedev.net/