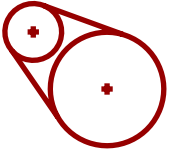






# Schattenvolumen

## Technik und Praxis



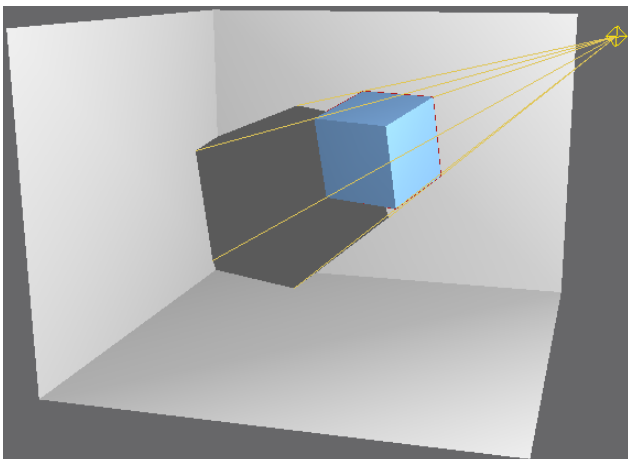
### Vorwort

Heute versuchen wir nicht über unseren Schatten zu springen sondern setzen ihn bewusst in Szene. Eine Szene ohne Schatten wirkt nicht echt. Dem Betrachter fehlen die Tiefeninformationen und Objekte, die eigentlich auf dem Boden stehen, schweben in der Luft. Gouraud Shading vermittelt zwar den Eindruck von Plastik kann aber keinen Schatten werfen. Ein guten Ansatz bieten Schattenvolumen um das es im Folgendem gehen soll. Anschließend werfen wir einen Blick auf das Devil Shadow System was auf diese Technik aufbaut.

### Schattenvolumen

Wir sehen zunächst einmal nicht mit dem Auge der Kamera sondern dem der Lichtquelle, gerichtet auf das schattenwerfende Objekt. Aus dieser Perspektive ist jedes Objekt beleuchtet, denn alle sichtbaren Flächen zeigen zur Lichtquelle hin. Und was ist hinter diesen Flächen? Richtig, Schatten! Der Algorithmus des Schattenvolumens macht sich genau diesen Umstand zur Nutze. Zunächst wird eine Silhouette des schattenwerfenden Objekts erstellt und diese dann in die Szene verlängert (extrudiert). Ein Volumen entsteht und alles, was sich in diesem Volumen befindet, bekommt es mit dem schwarzen Farbpinsel zu tun.

Dieses Verfahren eignet sich für unebene Untergründe und es beherrscht zudem das Selbstschattieren. Der Rechenaufwand steigt allerdings mit erhöhter Polygonzahl was einen Einsatz von LOD (Level of Detail) nötig macht.

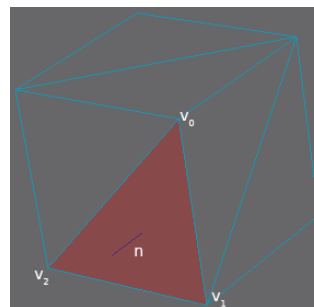


### Ein Blick unter die Motorhaube

Um die Silhouette berechnen zu können, muss zunächst eine Kantenliste des schattenwerfenden Objekts angelegt werden. Jedes Dreieck im Mesh besitzt drei Kanten. In die Kantenliste werden gleiche Kanten (Start- und Endpunkt sind gleich) aufgespürt und der Kantenliste hinzugefügt. So werden, neben Start- und Endpunkt, auch die beiden Nachbarpolygone mit gespeichert. Besitzt eine Kante einmal keine zwei Nachbarpolygone, so spricht man von einem offenen Mesh. Zu jedem Dreieck muss die Flächennormale berechnet werden. Nur anhand dieser kann man ermitteln, ob das Dreieck dem Licht zu- oder abgewandt ist.

$$\vec{n} = (\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)$$

Diese Formel ist dafür zuständig. Der Normalvektor  $n$  steht, dank des Kreuzprodukts, senkrecht auf der Ebene die das Dreieck bildet. Das Erstellen der Kantenliste mit den Flächennormalen erfolgt nach dem Laden des Meshs und ist nur einmal nötig. Die Flächennormalen sind im Objectspace berechnet worden (Transformation des Entities wurde also noch nicht mit einbezogen), daher müssen nach jeder Transformation (Rotation, Skalierung, Translation) diese Normalen in Eyekoordinaten transformiert werden (mittels `TFormNormal`).



len der Kantenliste mit den Flächennormalen erfolgt nach dem Laden des Meshs und ist nur einmal nötig. Die Flächennormalen sind im Objectspace berechnet worden (Transformation des Entities wurde also noch nicht mit einbezogen), daher müssen nach jeder Transformation (Rotation, Skalierung, Translation) diese Normalen in Eyekoordinaten transformiert werden (mittels `TFormNormal`).

gen), daher müssen nach jeder Transformation (Rotation, Skalierung, Translation) diese Normalen in Eyekoordinaten transformiert werden (mittels `TFormNormal`).

Jetzt muss die Silhouette des schattenwerfenden Objekts ermittelt werden. Die Silhouette grenzt, aus Sicht der Lichtquelle, sichtbare- von nicht sichtbaren Polygonen ab. Das Prinzip dahinter ist dem *BackFace-Culling* gleich. Statt die Sichtbarkeit vom Betrachter aus zu ermitteln, wird sie vom Licht aus ermittelt. Eine Kante gehört dann zur Silhouette, wenn jeweils ein Polygon von ihr dem Licht zugewandt und eines dem Licht abgewandt ist. Bei Kanten mit nur einem Polygon gilt: zeigt das Polygon zum Licht, gehört es zur Silhouette. Anhand des Ortsvektor  $l$  der Lichtquelle und des Normalvektors  $n$  im Eyespace wird die Sichtbarkeit wie folgt ermittelt:

$$d = \vec{n} \cdot \vec{l}$$

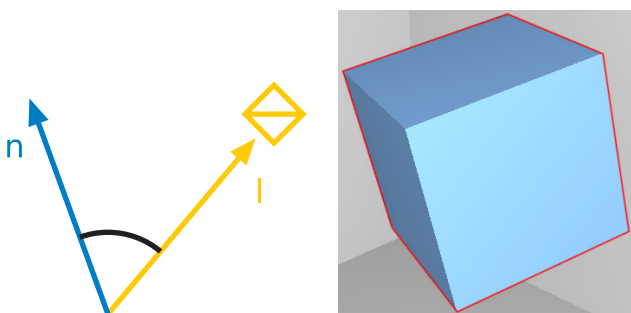
Das Punktprodukt der zwei Vektoren ist definiert durch:

$$\vec{n} \cdot \vec{l} = |\vec{n}| |\vec{l}| \cos \varphi(\vec{n}, \vec{l})$$

Die Normale  $n$  ist bereits normalisiert durch `TFormNormal`, hat also einen Betrag von 1. Wird noch der Ortsvektor  $l$  normalisiert, so ergibt sich folgende Formel:

$$\vec{n} \cdot \vec{l}' = \cos \varphi(\vec{n}, \vec{l}')$$

Damit ist  $d$  der Kosinuswert des eingeschlossenen Winkels der beiden Vektoren. Im Bereich von  $270^\circ$  bis  $90^\circ$  ist das Polygon dem Licht zugewandt ( $d \geq 0$ ) und im Bereich von  $90^\circ$  bis  $270^\circ$  dem Licht abgewandt ( $d < 0$ ). Gilt beim einem Polygon  $d \geq 0$  und beim anderen  $d < 0$ , dann gehört die Kante zur Silhouette.



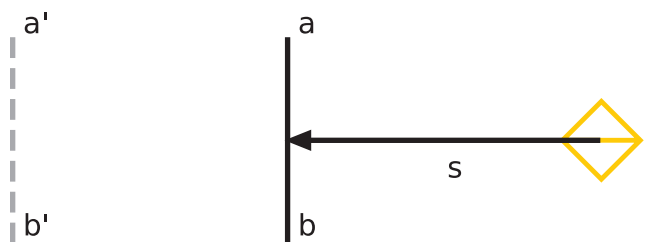
Es gibt zwei Wege die Silhouette zu extrudieren. Ist die Lichtquelle ein gerichtetes Licht (Sonne) werden die Kanten parallel extrudiert, ist sie ein Punktlicht (Glühbirne) werden die Kanten perspektivisch extrudiert. Unabhängig von beiden Varianten erstreckt sich das Schattenvolumen immer in die Unendlichkeit. Zunächst aber müs-

sen Startpunkt  $a$  und Endpunkt  $b$  der Kante ebenfalls in den Eyespace transformiert werden mittels `TFormPoint`.

#### • Parallelprojektion

Wir bilden einen Richtungsvektor  $s$  zwischen Entity und Lichtquelle und normalisieren ihn:

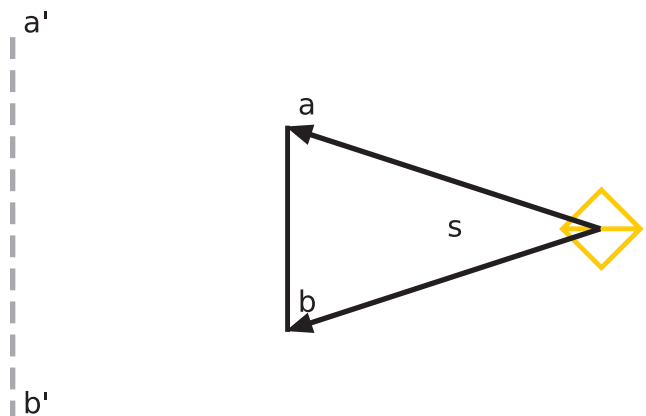
$$\vec{s} = \vec{l} - \vec{e}; \vec{s}' = \frac{\vec{s}}{|\vec{s}|}$$



#### • Perspektivische Projektion

Hier muss der Richtungsvektor  $s$  für jeden Punkt der Silhouette berechnet werden. Statt dem Ortsvektor  $e$  (Entityposition) wird der Ortsvektor des jeweiligen Punkts benutzt.

$$\vec{s} = \vec{l} - \vec{p}; \vec{s}' = \frac{\vec{s}}{|\vec{s}|}$$



Mit Hilfe dieses Richtungsvektors werden die Punkte  $a$  und  $b$  in die Unendlichkeit projiziert:

$$\vec{a}' = \vec{a} + \infty \vec{s}$$

$$\vec{b}' = \vec{b} + \infty \vec{s}$$

Anschließend wird ein Quad aus den Punkten  $a, b, a'$  und  $b'$  gebildet - In Blitz3D also die Dreiecke  $a, b, b'$  und  $b', a', a$ . Bis zu diesem Punkt konnte alles mit Blitz3D ohne Userlibs erreicht werden. Für den Rest ist jedoch der Zugriff auf den Stencilbuffer nötig, was ohne Userlibs nicht möglich ist. Dazu wird eine DLL benötigt, die vorzugsweise in C++ programmiert wird, da

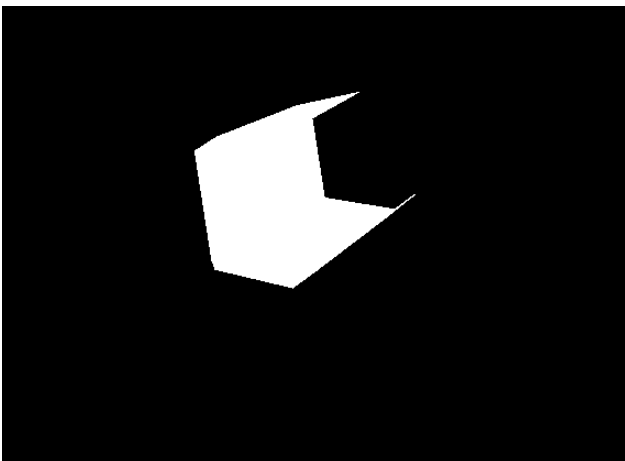


DirectX auf dem COM (Component Object Model) basiert und dieses objektorientiert arbeitet (Blitz3D ist eine prozedurale Programmiersprache)

- Die Szene wird zuerst ohne Schattenvolumen und nur mit Ambientlight gerendert (alle anderen Lichtquellen werden mit `HideEntity` ausgeschaltet). Danach sind vorerst alle Bereiche schattiert. Der Schatten wird also nicht noch später extra verdunkelt sondern behält seine natürliche Erscheinung.

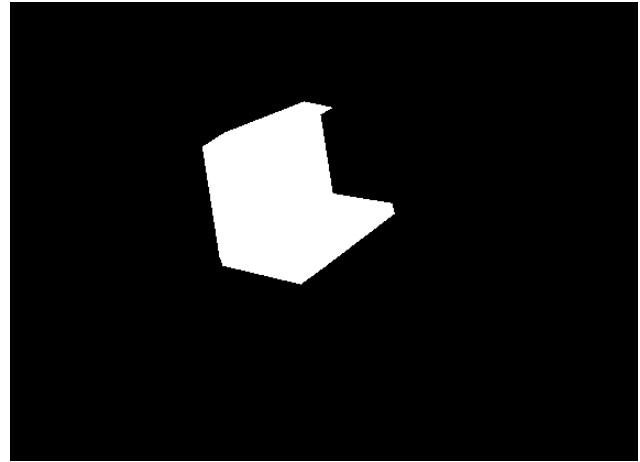


- Jetzt müssen die Fragmente der Szene ermittelt werden, die innerhalb des Schattenvolumens liegen. Hierfür wird der Stencilbuffer gelöscht (er wird auf 0 gesetzt) und das Schreiben in Fragment- und Depthbuffer deaktiviert. Nun wird nur die Vorderseite (`FrontFace = CW`) des Schattenvolumens gerendert. Alle Fragmente, die den Tiefentest bestanden haben, erhöhen den Stencilwert.

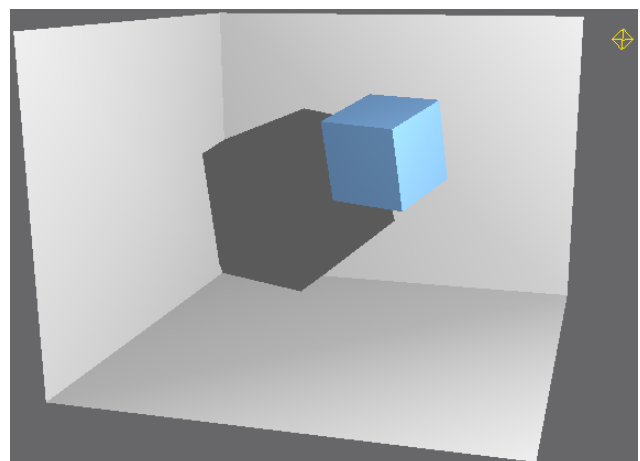


- Danach wird die Rückseite (`FrontFace = CCW`) des Schattenvolumens gerendert.

Alle Fragmente, die den Tiefentest bestanden haben, erniedrigen diesmal den Stencilwert. Die Fragmente mit einem Stencilwert von 0 gehören zum beleuchteten Bereich der Szene.



- Das ambiante Licht wird deaktiviert und das omnidirektionale- oder gerichtete Licht aktiviert (`ShowEntity`). Die Szene wird jetzt nur an den Stellen erneut mit Licht gerendert, deren Stencilwert 0 ist. Dazu bleibt der Stenciltest aktiv. Würde die Szene normal gerendert werden, würden sich Ambient- und Diffuselight addieren. Um das im Nachhinein bei uns zu ermöglichen, verwenden wir additives Blending. Bei dieser Technik wird der schattierte Bereich nicht wirklich schwarz eingefärbt oder zusätzlich abgedunkelt, sondern bekommt, wie in der Natur, bloß kein direktes Licht ab.



Die Kantenliste wird nach laden des Meshs angefertigt oder wird bereits in das Dateiformat integriert. Die Silhouette muss immer dann neu berechnet werden, wenn das Mesh oder die Lichtquelle transformiert wird. Auch mehrere

Lichtquellen sind möglich dank des additiven Blendings. pro Lichtquelle kommen dann 3 Renderpasses hinzu (Rendern der Schattenvolumen Vor- und Rückseite und der gesamten Szene). Bei hochauflösenden Meshs ist es zudem zu empfehlen, mit einem niedrigauflösenden Mesh zum Erstellen der Silhouette zu arbeiten.

### Devil Shadow System

Für die unter uns, die mit Vektoralgebra auf Kriegsfuß stehen, gibt es das einfach zu bedienende *Devil Shadow System*, entwickelt von *DevilsChild*. Wenige Befehle reichen, um seinem Spiel mehr Realismus einzuhauchen. Das ganze ist Open Source und, laut Hersteller, sogar für kommerzielle Nutzung frei. Kantenlisten können in Dateien gespeichert und von ihnen geladen werden und es funktioniert Hand in Hand mit animierten Meshs. Weiterhin bietet das System ENBM (Environmental Bump Mapping).

### Installation

Unter [www.devil-engines.dev-ch.de](http://www.devil-engines.dev-ch.de) gibt es die Aktuelle Version 1.35 als Quellcode und Binary. Öffnet man die Datei *DSS\_1.35u-1.zip* findet man das Unterverzeichnis *Userlibs* vor. Die *DevilShadowSystem\_DX7.dll* ist für den Zugriff auf das Direct3D7 Device von Blitz3D zuständig und *DevilShadowSystem\_B3D.dll* erweitert Blitz3D um ein paar nützliche Funktionen (beide wurden von *Tom Speed* entwickelt). *DevilShadowSystem.decls*

deklariert die Funktionen der beiden DLLs und *highlighted* praktischerweise auch die Funktionen des Devil Shadow Systems in der IDE. Der Inhalt wird in das *Userlibs*-Verzeichnis von Blitz3D kopiert. Die enthaltenen Beispiele und Tutorials sollten danach ohne Probleme laufen.

### Erste Tests

Mit Einbinden von *DevilShadowSystem.bb*, *ShadowVolumes.bb* und *UserInterface.bb* steht einem das System voll zur Verfügung. Mittels *InitShadows* wird es initialisiert. Unter anderem wird hier das Direct3D Device an die DLL übergeben und der Stencil-Buffer abgefragt. Über *FreeShadows* werden die verwendeten Ressourcen wieder freigegeben. Um Schattenvolumen in seinem Spiel dann nutzen zu können, benö-

tigt man nur die Befehle *SetShadowMesh* und *SetShadowLight*. Der Parameter *casting* von *SetShadowMesh* legt fest, ob das Mesh entweder Schatten wirft oder empfängt. Im Fall von *casting = True* kann das Mesh auf sich selbst Schatten werfen. Interessanterweise ist es möglich, auch Quake III BSP Levels, Terrains und MD2 Models als Shadow Receiver (*casting = false*) bei *SetShadowMesh* anzugeben. Über den Parameter *parallel* kann bei *SetShadowLight* bestimmt werden, ob direktionales oder omnidirektionales Licht simuliert werden soll. Die Szene wird mittels *Render* gerendert - das ersetzt *RenderWorld*. Der Parameter *mode* gibt an, ob die Szene normal 0, die Szene mit Schatten 1 oder oder die Szene mit rot-durchscheinenden *ShadowMesh* 2 gerendert werden soll.



Das Beispiel *Sample5 - LOD.bb* zeigt gut, wie man eine hochauflösende Szene mit einem weniger aufgelösten Mesh schattieren kann.