CARL HANSER VERLAG

David Scherfgen

# 3D-Spieleprogrammierung

Modernes Game Design mit DirectX 9 und C++

3-446-22152-2

www.hanser.de

# 9 Fortgeschrittene Techniken

## 9.1 Was Sie in diesem Kapitel erwartet

In diesem Kapitel werde ich einige fortgeschrittene Techniken – den Bereich der 3D-Grafik betreffend – vorstellen, die in unseren beiden Spielen *Breakanoid* und *Galactica* noch nicht zur Anwendung kamen. Außerdem beinhaltet dieses Kapitel viele kleine „Denkanstöße“ (das heißt: nicht immer komplette Quellcodes, sondern vielmehr Ideen und Anregungen). Ich kann Ihnen versprechen, dass es interessant wird!

## 9.2 Schatten mit dem Stencil-Buffer

### 9.2.1 Schatten in der 3D-Grafik

Wenn wir einen Screenshot eines 3D-Spiels betrachten, dann fällt es meistens nicht schwer, diesen von einem realen Foto zu unterscheiden. Einerseits mag das vielleicht an zu wenig detaillierten Modellen liegen, aber der hauptsächliche Grund ist in den meisten Fällen die unrealistische Beleuchtung. Wie bereits im zweiten Kapitel angesprochen, ist eine realistische Beleuchtung in Echtzeit-3D-Polygongrafik kaum zu realisieren. Vor allem *Schatten* lassen sich mit dem Prinzip, jeden Vertex einzeln zu beleuchten, kaum simulieren. Da Schatten aber für den Realismus und vor allem für die Atmosphäre eines Spiels sehr wichtig sind, darf man an dieser Stelle nicht aufgeben, sondern sollte nach anderen Methoden zur Schattenberechnung Ausschau halten.

### 9.2.2 Ansätze

Schatten können auf vielerlei Arten berechnet werden. Die beiden am häufigsten benutzten Methoden sind *Shadow Mapping* und *Shadow Volume Rendering*.

#### 9.2.2.1 Shadow Mapping

Beim *Shadow Mapping* versetzt man sich in die Sicht der Lichtquelle und rendert die gesamte Szene von dort aus – und zwar nicht in den Bildpuffer, sondern in eine eigens dafür angefertigte *Textur* (die so genannte *Shadow Map*). Dabei handelt es sich um eine *Tiefentextur* – das heißt, dass diese Textur nur die *Tiefe* jedes gerenderten Pixels speichert und nicht die Farbe (wie der Z-Buffer). Das Prinzip ist nun folgendes: Wenn ein Objekt aus der Sicht der Lichtquelle durch ein anderes Objekt verdeckt wird, dann wird es im Schatten liegen – die Lichtstrahlen können es ja nicht direkt erreichen.

Shadow Mapping funktioniert auf *Pixelebene*, und die Qualität der Schatten hängt vor allem von der Größe der Shadow Map ab. Zu kleine Shadow Maps führen zu „eckigen“ Schatten, bei denen man jeden Texel einzeln erkennen kann, und zu große Shadow Maps sind beim Rendern zu langsam. Dafür sind die Schatten aber auch angenehm „weich“, wie sie es in der Realität meistens ebenfalls sind.

Leider gibt es Probleme, wenn man Shadow Mapping mit *punktförmigen* Lichtquellen anwenden will, denn in welche Richtung soll das Licht dann zeigen? Diese Richtung muss ja bekannt sein, um sich in die Sicht der Lichtquelle zu versetzen. Man bräuchte eine 360°-Sicht der Szene und müsste wahrscheinlich alles 6 x rendern: nach vorne, hinten, links, rechts oben und unten. Unter Direct3D ist Shadow Mapping leider nur recht schwer zu implementieren, und die Hardwareanforderungen liegen bei einer GeForce 3 (während es mit OpenGL auch mit älteren Grafikkarten funktioniert).

#### 9.2.2.2 Shadow Volume Rendering

Beim *Shadow Volume Rendering* arbeitet man *objektbasierend* – Schatten werden für einzelne Objekte und nicht für die gesamte Szene berechnet. Auch hier versetzt man sich in die Sicht der Lichtquelle und erstellt anhand der Silhouette des Objekts ein *Schattenvolumen*. Das ist der Bereich, der von dem Objekt abgedunkelt wird. Mit einem cleveren Trick nutzt man dann den Stencil-Buffer, um den Bereich auf dem Bild zu finden, der später – um den Schatten darzustellen – verdunkelt werden muss. Diese Schatten sind auf den Pixel genau und sehr scharf.

Weil *Shadow Volume Rendering* recht leicht zu implementieren ist, habe ich mich für diese Methode entschieden.

### 9.2.3 Das Prinzip

#### 9.2.3.1 Die Silhouette für das Schattenvolumen

Der Umriss (Silhouette) des Schattenvolumens, also des Bereiches, der im Schatten liegt, ist gleich der Silhouette des Objekts aus der Sicht der Lichtquelle.

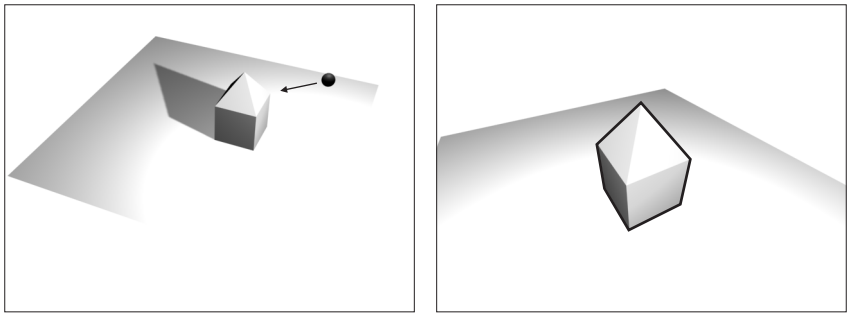

**Abbildung 9.1** *Links*: das Objekt wirft einen Schatten; *Rechts*: die Silhouette des Objekts aus der Sicht der Lichtquelle

Schauen Sie Sich die rechte Abbildung an, und stellen Sie sich vor, die Silhouette (die dicken Linien) in die Tiefe zu ziehen. Genau so entsteht das Schattenvolumen!

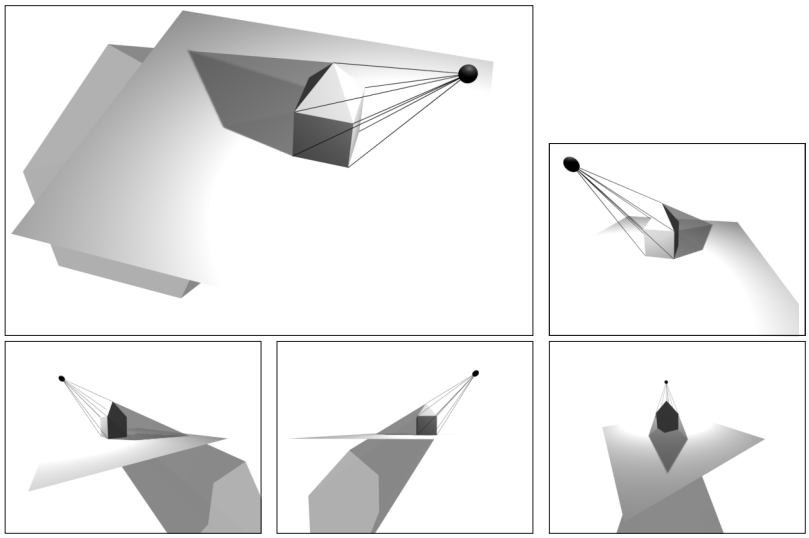

**Abbildung 9.2** Man verlängert die Verbindungslinie zwischen den Vertizes und der Lichtquelle, um das Schattenvolumen zu erhalten. Dort, wo es die Ebene schneidet, ist Schatten.

In Ordnung – wir brauchen also „nur“ die Silhouette des Objekts aus der Sicht der Lichtquelle. Das ist aber nicht so einfach, wie es vielleicht auf den ersten Blick aussieht, denn wir müssen genau die Dreiecksseiten herausfiltern, die *nicht* zur Silhouette gehören.

Hier hilft folgende Überlegung weiter:

Eine Dreiecksseite gehört normalerweise zu einem oder zu zwei verbundenen Dreiecken. Sie gehört genau dann zur Silhouette, wenn *eines* der beiden Dreiecke *sichtbar* ist und das andere nicht oder wenn sie nur zu einem einzigen Dreieck gehört.

Oder anders gesagt: Wenn innerhalb der Menge der für das Licht *sichtbaren* Dreiecke eine Seite *zweimal* vorkommt, dann gehört sie *nicht* zur Silhouette, weil sie ja dann zu *zwei* Dreiecken gehört, die beide sichtbar sind.

Also können wir nach folgendem Prinzip vorgehen:

- Wir durchlaufen alle Dreiecke des Modells.
- Wenn ein Dreieck sichtbar ist, fügen wir seine drei Seiten zu einer Liste hinzu.
- Wenn wir bemerken, dass sich eine Seite bereits in der Liste befindet, dann entfernen wir sie, weil sie ja doppelt vorkommt und daher nicht zur Silhouette gehört. Eine Seite wird anhand ihrer beiden Vertizes identifiziert.

Die Liste enthält nur noch die Dreiecksseiten, die zur Silhouette gehören, und wir wären nun schon in der Lage, das Schattenvolumen zu erstellen. Aber erst einmal werden wir uns um den zeichentechnischen Aspekt kümmern.

### 9.2.3.2 Der Trick mit dem Stencil-Buffer

Auf der Abbildung konnte man das Schattenvolumen *sehen*. Das wird später natürlich nicht mehr der Fall sein (obwohl es kombiniert mit Alpha-Blending sicherlich auch ein schöner Effekt wäre …). Irgendwie müssen wir ja jetzt diejenigen Pixel markieren, die später wirklich

verdunkelt werden sollen. Mit geometrischen Ansätzen kommt man hier nicht weit, aber eine große Hilfe wird uns durch den *Stencil-Buffer* geboten. Wie Sie hoffentlich noch wissen, ist der Stencil-Buffer ein Teil des Z-Buffers (beide zusammen heißen dann Z-Stencil-Buffer). Der Stencil-Buffer speichert für jeden Pixel einen ganzzahligen Wert, der für die Markierung oder Maskierung einzelner Pixel oder ganzer Bereiche verwendet werden kann.

Zu Beginn der Szene wird der Stencil-Buffer auf null zurückgesetzt. Man zeichnet dann zuerst nur die *Vorderseite* des Schattenvolumens. Das heißt mit anderen Worten, dass alle Dreiecke, deren Vertizes gegen den Uhrzeigersinn angeordnet sind, wegfallen (`D3DRS_CULLMODE` auf `D3DCULL_CCW` setzen). Den Stencil-Buffer stellt man so ein, dass der Stencil-Wert bei jedem gezeichneten Pixel um eins *erhöht* wird.

Anschließend zeichnen wir nur die *Rückseiten* des Schattenvolumens (`D3DCULL_CW`) und stellen den Stencil-Buffer so ein, dass die Stencil-Werte um eins *verringert* werden.

Jetzt haben die Pixel, wo sich Vorder- und Rückseite überlappen, einen Stencil-Wert von null (einmal erhöhen und einmal verringern). Aber dort, wo nachher der Schatten sichtbar sein soll, sind die Stencil-Werte nicht gleich null! Das sind die Stellen, an denen die Rückseite des Schattenvolumens zum Beispiel schon in den Boden eindringt und damit unsichtbar wird. Damit haben wir genau die richtigen Pixel markiert.

Vor dem Rendern sorgen wir dafür, dass das Schattenvolumen im Bildpuffer nicht sichtbar wird – es soll seine Spuren nur im Stencil-Buffer hinterlassen. Das kann man durch Alpha-Blending erreichen (`D3DRS_SRCBLEND` auf `D3DBLEND_ZERO` und `D3DRS_DESTBLEND` auf `D3DBLEND_ONE` setzen). Besser ist es, das Render-State `D3DRS_COLORWRITEENABLE` auf null zu setzen.

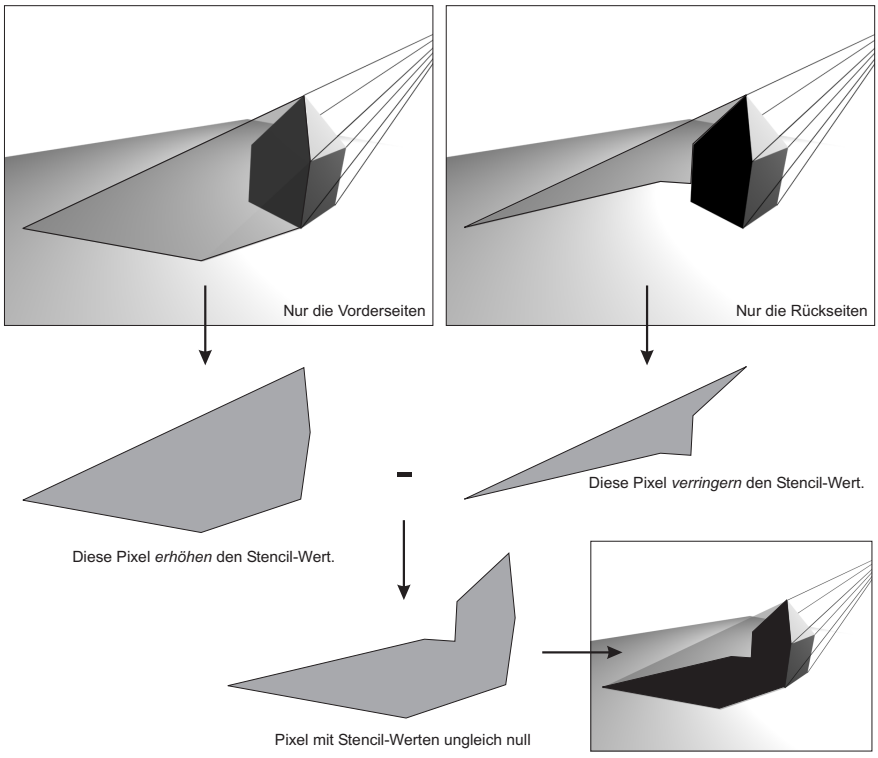


**Abbildung 9.3** Wie man den Stencil-Buffer benutzt, um Schattenbereiche zu finden

Nun sind also alle Pixel, die im Schatten liegen, durch einen Stencil-Wert ungleich null markiert. Jetzt ist es ein Leichtes, diese Pixel zu verdunkeln. Dazu zeichnen wir einfach ein großes Rechteck in der Farbe des Schattens (mit Alpha-Blending) über den ganzen Bildschirm. Den Stencil-Test stellen wir so ein, dass nur dort Pixel gezeichnet werden, wo der Stencil-Wert ungleich null ist.

#### 9.2.3.3 Sichtbarkeit eines Dreiecks

Bevor wir nun im nächsten Schritt besprechen, wie wir die Vertizes für das Schattenvolumen erzeugen, ist es noch wichtig zu wissen, wie man die Sichtbarkeit eines Dreiecks bestimmen kann. Denn wir dürfen ja nur die Seiten von *sichtbaren* Dreiecken zur Liste hinzufügen, die am Ende die Silhouette enthalten soll. Dazu machen wir es Direct3D nach: *Culling* heißt das Zauberwort! Wir müssen nur das Punktprodukt aus dem Normalvektor des Dreiecks und der Richtung vom Licht zum Dreieck hin bestimmen. Wenn das Ergebnis positiv ist, dann ist das Dreieck sichtbar, weil es uns seine Vorderseite zuwendet, und bei einem negativen Punktprodukt ist es unsichtbar.

Wie man nun diesen Richtungsvektor vom Licht zum Dreieck bestimmt, hängt ganz vom Typ der Lichtquelle ab:

- Bei dem Typ `D3DLIGHT_POINT`, wo die Lichtquelle ja eine *Position* hat, nehmen wir einfach den Verbindungsvektor von der Lichtposition zu Mittelpunkt des Dreiecks (den wir hier einfach durch den Mittelwert der Positionsvektoren der drei Vertizes berechnen).
- *Richtungslichter* (`D3DLIGHT_DIRECTIONAL`) kann man sich so weit entfernt vorstellen, dass die Verbindungsvektoren zu allen Dreiecken gleich sind (die Lichtstrahlen kommen praktisch parallel an). Also brauchen wir hier auch keinen Mittelpunkt oder Sonstiges – als Richtungsvektor nehmen wir die Lichtrichtung (`D3DVECTOR D3DLIGHT9::Direction`).
- Spotlichter (`D3DLIGHT_SPOT`) können wir wie Punktlichter behandeln, da diese ja auch eine *Position* besitzen. Allerdings kann der spezielle Charakter des Spotlichts hier nicht so leicht berücksichtigt werden (was mit Light-Mapping jedoch kein Problem wäre).

Eine einfache Punktproduktrechnung verrät uns also, ob die Lichtquelle das Dreieck sehen kann, und damit verrät sie uns auch, ob wir dieses Dreieck bei der Erzeugung des Schattenvolumens beziehungsweise der Silhouette berücksichtigen müssen.

Es geht hier übrigens nicht nur um eine „kleine Optimierung“ (je weniger Dreiecke wir zu verarbeiten haben, desto besser), sondern die Technik des Shadow-Volume-Renderings würde gar nicht funktionieren, wenn man ganz einfach *jedes* Dreieck verarbeitet.

#### 9.2.3.4 Generieren der Vertizes

Vor dem Rendern des Schattenvolumens erzeugen wir aus jeder Seite, die zur Silhouette gehört, genau sechs Vertizes: Wenn man eine Seite extrudiert, wird sie ja zu einem Viereck, welches wiederum aus zwei Dreiecken besteht.

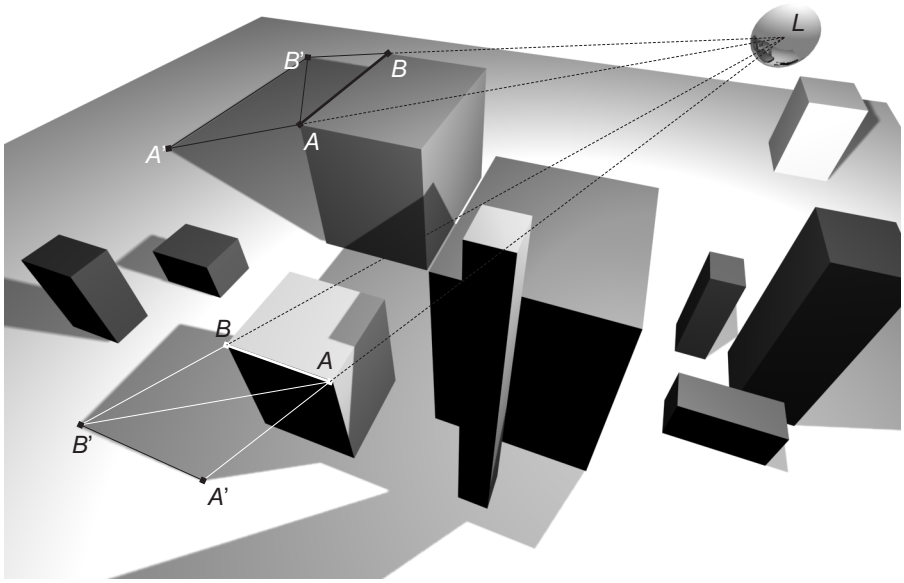


**Abbildung 9.4** Extrusion einiger Seiten, die zur Silhouette eines Objekts gehören. *A*, *A*' und *B*' sowie *A*, *B*' und *B* bilden jeweils ein Dreieck.

Punkt *A* und *B* kennen wir – das sind die Positionen der beiden Vertizes, die zu der Dreiecksseite gehören. *A*' und *B*' sind dann einfach die Verlängerungen von *L* aus gesehen:

$$A' = A + \left\| A - L \right\| \cdot Länge$$
$$B' = B + \left\| B - L \right\| \cdot Länge$$

Bei einem Richtungslicht ist es einfacher – das langsame Normalisieren des Vektors entfällt, da die Richtung zum Licht ja für jeden Vertex gleich ist.

$$A' = A + Lichtrichtung \cdot Länge$$
$$B' = B + Lichtrichtung \cdot Länge$$

Die Abbildung könnte den falschen Eindruck erwecken, dass A' und B' zwangsweise immer auf der Ebene, also auf dem Objekt, das den Schatten empfängt, liegen. In Wirklichkeit achtet man aber gar nicht auf irgendwelche Objekte, sondern man verlängert die Linie zum Beispiel einfach um 1000 Einheiten (das ist die Variable *Länge* in den vorherigen Gleichungen). Alles, was sich noch weiter weg befindet, kann dann keinen Schatten mehr empfangen. Die Länge des Schattenvolumens sollte nicht übertrieben groß gewählt werden, da es sonst zu Ungenauigkeiten kommen könnte, die man später als flimmernde Pixel wahrnimmt.

#### 9.2.3.5 Probleme mit der Kamera

Leider funktioniert das Shadow Volume Rendering in der hier gezeigten Form nicht mehr richtig, wenn die Kamera selbst sich im Schattenvolumen befindet. Allerdings gibt es auch für dieses Problem eine Lösung. Der Stencil-Buffer wird dann zu Beginn der Szene nicht auf *null* gesetzt, sondern auf *eins*.

Damit es dann funktioniert, müssen die Befehle *Stencil-Wert erhöhen* und *Stencil-Wert verringern* vertauscht werden. Also muss der Stencil-Wert dann bei den *Rückseiten* erhöht und bei den *Vorderseiten* verringert werden. Wenn sich die Kamera außerhalb des Schattenvolumens befindet, hat diese Änderung normalerweise keine sichtbaren Folgen. Es reicht also, die Befehle *immer* zu vertauschen und – je nach Lage der Kamera – den Stencil-Buffer zu Beginn auf null oder eins zu setzen.

Ob sich die Kamera im Schattenvolumen befindet, lässt sich sehr leicht prüfen: Wir brauchen dazu nur eine Linie von der Kameraposition bis zur Lichtquelle zu erstellen (oder bei einem Richtungslicht bis zu einem sehr weit entfernten Punkt entgegen der Lichtrichtung). Dann testen wir mit `tbLineHitsModel`, ob diese Linie unterwegs irgendwo das Modell schneidet, das den Schatten wirft. Falls ja, dann befindet sich die Kamera im Schattenvolumen, da sie von den Lichtstrahlen ja nicht erreicht werden kann.

### 9.2.4 Die Klasse *tbShadowVolume*

Wir werden nun die Klasse `tbShadowVolume` implementieren, die uns das Zeichnen von Schatten mit dem Stencil-Buffer abnehmen wird.

#### 9.2.4.1 Die Klassendefinition

**Variablen**

- Ein `tbModel`-Zeiger auf das Modell, dessen Schatten gerendert wird
- Ein dynamisches `tbVector3`-Array, das die Mittelpunkte aller Dreiecke enthält (die brauchen wir ja für punktförmige Lichtquellen)
- Eine Liste mit den Dreiecksseiten (dafür gibt es die Struktur `tbEdge`, die lediglich zwei `DWORD`-Variablen `dwPointA` und `dwPointB` enthält) und die Anzahl der Einträge
- Zwei Effekte (`tbEffect`): einen für das Rendern des Schattenvolumens und einen für das spätere Rendern des eigentlichen Schattens (in Form eines großen Rechtecks)
- Eine weitere Liste, die alle Vertizes des Schattenvolumens enthält. Da Dinge wie Farbe, Texturkoordinaten und Normalvektor für diese Vertizes keine Bedeutung haben (man sieht das Schattenvolumen ja sowieso nicht), reicht hier ein einfacher `tbVector3`-Vektor als einziges Element des Vertexformats aus. Wir speichern auch die Anzahl der Vertizes.

**Methoden**

- Wie immer brauchen wir natürlich eine `Init`-Methode. Sie wird hier nur einen einzigen Parameter haben, und zwar einen Zeiger auf das Modell, dessen Schatten berechnet werden soll.
- Eine Methode `AddEdge` fügt eine durch zwei Vertizes (`DWORD`) angegebene Dreiecksseite zur Liste hinzu. Existiert diese Seite bereits, dann wird sie aus der Liste gelöscht. Vielleicht werden Sie sich fragen, wie man denn einen Vertex als `DWORD`-Wert darstellen kann! Ganz

einfach: In diesem Fall bezieht sich der DWORD-Wert auf die *Nummer* des Vertex in der Vertexliste des Modells (tbVector3* tbModel::m_pvVectors).

- Die Methode tbShadowVolume::ComputeVolume berechnet das Schattenvolumen. Als Parameter brauchen wir hier die Lichtquelle (D3DLIGHT9), die inverse Transformationsmatrix des Objekts, die Länge des Schattenvolumens und einen BOOL-Wert, der bestimmt, ob im Falle eines punktförmigen Lichts eine Normalisierung der Verbindungsvektoren durchgeführt werden soll (was langsamer ist). ComputeVolume wird dann alle Dreiecke des Modells durchgehen und auf Sichtbarkeit überprüfen. Die drei Seiten jedes sichtbaren Dreiecks werden durch AddEdge zur Liste hinzugefügt. Anschließend erzeugen wir hier mit den verbleibenden Dreiecksseiten das eigentliche Schattenvolumen.
- Das Schattenvolumen wird in tbShadowVolume::RenderVolume gerendert. Wir setzen dazu einen Effekt ein, der sich um die richtigen Stencil-Buffer-Einstellungen kümmert. Als Parameter benötigen wir hier lediglich die Transformationsmatrix des Modells.
- Am Ende wird man tbShadowVolume::RenderShadow aufrufen. Hier zeichnen wir das große Rechteck über den gesamten Bildschirm (mit Alpha-Blending), um die zuvor maskierten Pixel abzudunkeln. Die Farbe des Rechtecks, also die Schattenfarbe, soll per Parameter angegeben werden können.

### Code

```
// ******************************************************************
// Klasse für ein Schattenvolumen
class TRIBASE_API tbShadowVolume
{
private:
    // Variablen
    tbModel*   m_pModel;              // Das Modell
    tbVector3* m_pvTriangleCenters;   // Mittelpunkte der Dreiecke
    tbEdge*    m_pEdges;              // Liste der Dreiecksseiten
    DWORD      m_dwNumEdges;          // Anzahl der Dreiecksseiten
    tbEffect*  m_pShadowVolumeEffect; // Effekt zum Rendern des Schattenvolumens
    tbEffect*  m_pShadowEffect;       // Effekt zum Rendern des Schattens
    tbVector3* m_pvVertices;          // Die Vertizes für das Schattenvolumen
    DWORD      m_dwNumVertices;       // Anzahl der Vertizes

public:
    // Konstruktor und Destruktor
    tbShadowVolume();
    ~tbShadowVolume();

    // Methoden
    tbResult Init(tbModel* pModel);                              // Initialisierung
    void     AddEdge(const DWORD dwPointA, const DWORD dwPointB);// Neue Seite
    tbResult RenderVolume(const tbMatrix& mModelMatrix);         // Schattenvolumen rendern
    tbResult RenderShadow(const tbColor& ShadowColor);           // Schatten rendern

    // Berechnung des Schattenvolumens
    tbResult ComputeVolume(const tbMatrix& mInvModelMatrix,
                           const D3DLIGHT9& Light,
                           const float fLength = 1000.0f,
                           const BOOL bNormalize = FALSE);
};
```

**Listing 9.1** Die Definition der tbShadowVolume-Klasse

### 9.2.4.2 Die Initialisierung

In der Initialisierungsmethode `tbShadowVolume::Init` müssen wir folgende Dinge erledigen:

- Kopieren des Zeigers auf das Modell (`m_pModel`)
- Erstellen der beiden Effekte
- Genug Speicherplatz für die Dreiecksseiten, die Dreiecksmittelpunkte und die Vertizes reservieren. Bei den Dreiecksseiten und den Vertizes können wir vorher nicht wissen, wie viele es sein werden (und es hängt ja auch davon ab, wie die Lichtquelle zu dem Modell steht), darum rechnen wir mit der maximalen Anzahl. Das bedeutet: Wenn das Modell aus 300 Indizes besteht (100 Dreiecke), dann reservieren wir Speicherplatz für 300 Dreiecksseiten (3 pro Dreieck) und für 1800 Vertizes (6 pro Dreiecksseite).

Achten Sie besonders auf den Effektquellcode! Wie im zweiten Kapitel angesprochen, gibt es ja in DirectX 9 die Möglichkeit, für Vorder- und Rückseiten *verschiedene* Stencil-Buffer-Befehle einzustellen. Dafür erzeugen wir dann einfach eine eigene Technik. Wenn sie von der Grafikkarte nicht unterstützt wird, wird automatisch die andere verwendet, die mit zwei Durchgängen (*Passes*) arbeitet: Einer zeichnet nur die *Vorderseiten* und *verringert* die Stencil-Werte und einer zeichnet nur die *Rückseiten* und *erhöht* die Stencil-Werte. Für die Stencil-Buffer-Masken setzen wir die Standardwerte `0xFFFFFFFF` ein, so dass *alle* Bits berücksichtigt werden. Der Stencil-Test entfällt beim Effekt für das Schattenvolumen, da ja *alle* Pixel gezeichnet werden sollen – also wird `D3DRS_STENCILFUNC` auf `D3DCMP_ALWAYS` gesetzt. Anders hingegen ist das beim Effekt für den *Schatten* (das große Rechteck), denn hier sollen ja nur solche Pixel gezeichnet werden, deren Stencil-Wert ungleich null ist (`D3DRS_STENCILFUNC` auf `D3DCMP_NOTEQUAL` und `D3DRS_STENCILREF` auf 0 setzen).

```
// ****************************************************************
// Initialisiert das Schattenvolumen
tbResult tbShadowVolume::Init(tbModel* pModel)
{
    // Parameter prüfen
    if(pModel == NULL)       TB_ERROR_NULL_POINTER("pModel", TB_ERROR);
    if(!pModel->ExtraData()) TB_ERROR("Keine Extradaten vorhanden!", TB_ERROR);


    // Modell kopieren
    m_pModel = pModel;

    // Effekt für das Schattenvolumen erstellen
    m_pShadowVolumeEffect = new tbEffect;
    if(m_pShadowVolumeEffect->Init("TECHNIQUE T1\n"
                                   "{\n"
                                   "    PASS P1\n"
                                   "    {\n"
                                   "        Texture[0]          = Null;\n"
                                   "        ZEnable             = True;\n"
                                   "        ZWriteEnable        = False;\n"
                                   "        ShadeMode           = Flat;\n"
                                   "        FogEnable           = False;\n"
                                   "        ColorOp[0]          = SelectArg1;\n"
                                   "        ColorArg1[0]        = Current;\n"
                                   "        ColorOp[1]          = Disable;\n"
                                   "        Lighting            = False;\n"
                                   "        StencilEnable       = True;\n"
                                   "        StencilFunc         = Always;\n"
                                   "        StencilFail         = Keep;\n"
                                   "        StencilZFail        = Keep;\n"
                                   "        StencilPass         = Decr;\n"
                                   "        StencilMask         = 0xFFFFFFFF;\n"
```