

### Zusammenfassung

Binary Space Partitioning (BSP) Trees sind eine in der Computergrafik an vielen Stellen eingesetzte Datenstruktur. Sie erlauben eine rekursive Klassifikation von  $d$ -dimensionalen Objekten durch Partitionieren an  $(d - 1)$ -dimensionalen Hyperebenen. In diesem Artikel werde ich die Entstehung von BSP Trees als Technik zum Hidden Surface Removal (HSR) motivieren und verschiedene weitere Anwendungsmöglichkeiten aufzeigen, darunter auch die Erzeugung von Schatten in Szenen mit statischer und dynamischer Geometrie.

## 1 Geschichte

BSP Trees als Datenstrukturen in der Computergrafik wurden von Fuchs et al. 1980 in [FKN80] zum ersten Mal formal untersucht. Die Bestrebung war, das Sichtbarkeitsproblem (*hidden surface determination*, manchmal auch *hidden surface removal*) für statische Szenen mit wechselndem Betrachterstandpunkt zu lösen und die Bilderzeugung zu beschleunigen. Eine Beschleunigung würde nicht nur komplexere Darstellungen mit denselben Mitteln erlauben, sondern sich auch in den Kosten für die Erzeugung von 3D-Szenen bemerkbar machen – schließlich waren bildgebende Anlagen im Jahre 1980 noch sehr teuer. Fuchs et al. kündigten daher ihr Vorhaben wie folgt an:

[...] A new algorithm for solving the hidden surface (or line) problem to more rapidly generate realistic images [...]

Der in [FKN80] vorgestellte Ansatz nutzt aus, dass viele Anwendungen ausschließlich aus *statischen* Szenen bestehen. Dazu gehören beispielsweise Anwendungen aus der Architektur (virtuelle Gebäuderundgänge), sowie die Visualisierung von Daten, die sich beispielsweise in der Biologie und Chemie ergeben (Molekül- und Gendarstellungen). Wenn dynamische Änderungen ausgeschlossen werden bzw. nicht auftreten können, ist es möglich, durch einen einmaligen Vorverarbeitungsschritt die Berechnungen zur Laufzeit zu minimieren. Fuchs et al. stellten dazu ein klassisches *divide et impera*-Verfahren vor:

---

**Algorithmus 1** „Binary Space Partitioning“

---

- 1: Wähle eine Ebene im Objektraum.
  - 2: Klassifiziere alle Polygone anhand des Halbraumes, in dem sie liegen.
  - 3: Setze diesen Prozess rekursiv in jedem Halbraum fort.
- 

Durch die rekursive Klassifikation von Objekten anhand ihrer relativen Lage zu Schnittebenen wird ein *Binärbaum* erzeugt. Diese Art der binären Aufteilung des Raumes ist namensgebend für das „Binary Space Partitioning“.

Die Ideen von Fuchs et al. wurden an verschiedenen Stellen aufgegriffen: Chin & Feiner beschreiben 1989 in [CF89], wie BSP Trees zur Schattenerzeugung in statischen Szenen eingesetzt werden können. Dabei verwenden sie eine Erweiterung der BSP Trees, Shadow Volume BSP Trees. Diese setzen Sie zur

Beschreibung von sogenannten *Schattenvolumina* ein. Chrysanthou & Slater stellen 1995 in [CS95] ein Verfahren vor, das Shadow Volume BSP Trees auch für dynamische Szenen zulässt. Ich werde auf beide Themen in diesem Artikel noch eingehen.

Die Ergebnisse von Fuchs et al. sind auch in der Computerspieleindustrie aufgenommen worden: John Carmack implementierte 1993 BSP Trees in der Game Engine „id Tech 1“, die unter anderem im Computerspiel „Doom“ verwendet wurde. BSP Trees verhalfen diesem Ego-Shooter zu einer für seine Erscheinungszeit sehr flüssigen Quasi-3D-Darstellung. Der Erfolg von „Doom“, das auch heute noch eine Fangemeinde hat<sup>1</sup>, trug maßgeblich dazu bei, dass BSP Trees bei vielen 3D-Engines<sup>2</sup> für ganz unterschiedliche Zwecke Verwendung finden, beispielsweise zur Schattenberechnung oder zum Speichern der Geometriedaten der Spielwelt.

## 1.1 Relevanz

Alle modernen Grafikkarten verfügen über Hardware  $z$ -Buffer, sodass BSP Trees nicht mehr zur Sichtbarkeitsberechnung eingesetzt werden müssen. Bei integrierten Grafikkarten, wie sie in Mobiltelefonen und PDAs auftreten, stellen sie allerdings noch eine sinnvolle Möglichkeit dar, 3D-Szenen in annehmbarer Geschwindigkeit darzustellen. Ebenso werden BSP Trees immer noch zum schnellen und einfachen Berechnen von Schatten genutzt. In den letzten Jahren zeichnete sich auch eine vermehrte Verwendung im Bereich des Realtime-Raytracings ab – siehe hierzu auch [IWP08].

Je nach Anwendungszweck gibt es alternativen Datenstrukturen zu BSP Trees. Dies sind zum einen *kd*-Trees („axis-aligned BSP Trees“, die an den Koordinatenachsen ausgerichtet sind), die einen Spezialfall von BSP Trees darstellen, sowie Quad- und Octrees. Auf diese werde ich hier *nicht* eingehen. Ich verweise den Leser auf [dBvKOS00]. Dort sind einige dieser Datenstrukturen mit weiterführenden Referenzen aufgeführt.

## 2 Grundlagen

Die Problemstellung von Fuchs et al. ist klar umrissen: *Gegeben* ist eine beliebig komplexe 3D-Szene mit einer Menge  $P$  von planaren Polygonen,  $P = \{p_1, p_2, \dots, p_n\}$ , sowie eine Betrachterposition. Die Positionen der Polygone werden dabei als *statisch* angenommen.

Es soll ein Bild erzeugt werden, das die Szene aus der Betrachterposition zeigt. Dabei müssen die in der Computergrafik üblichen Schritte durchgeführt werden:

1. Koordinatentransformation für alle Objekte, um sie vom Objektraum auf

---

<sup>1</sup>Unter <http://www.newdoom.com> finden beispielsweise immer noch rege Diskussionen über das Spiel statt, obwohl es bereits mehrere Nachfolger gibt. Ebenso gibt es freie Implementierungen der Game Engine, beispielsweise unter <http://prboom.sourceforge.net>.

<sup>2</sup>Eine Nutzung ist belegt für viele der „id Tech“ Engines sowie ältere Versionen der „Unreal Engine“ bzw. „Unreal Technology“. Aktuelle kommerzielle Engines stellen Informationen zu ihrer internen Arbeitsweise meistens nicht zur Verfügung, sodass über eine Verwendung von BSP Trees nur spekuliert werden kann.

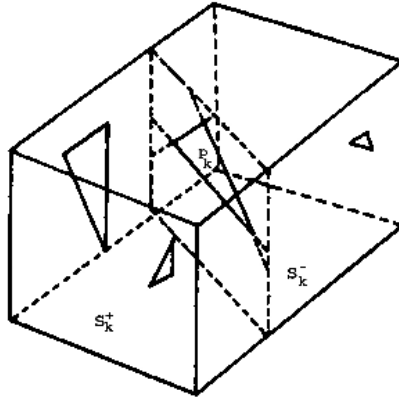


Abbildung 1: Beispiel für partitionierende Ebene; nach [FKN80]

den Bildschirm<sup>3</sup> zu überführen.

2. Das Wegschneiden (Clipping) unnötiger Polygone, die beispielsweise außerhalb des sichtbaren Bereiches liegen.
3. Die eigentliche Bilderzeugung aus allen übrigen Polygonen. Hierzu sind pro Pixel des Bildschirms zwei Schritte nötig:
  - (a) Das Polygon mit der *geringsten* Entfernung zum Betrachter ist zu suchen. Dies ist das Polygon, das an diesem Pixel sichtbar sein wird.
  - (b) Die Farbe des Pixels ist über ein entsprechendes Lichtmodell zu berechnen.

Um das Polygon mit der geringsten Entfernung zum Betrachter möglichst schnell suchen zu können, können wir wie folgt vorgehen: Es liege eine Menge  $d$ -dimensionaler Objekte  $P$  wie oben vor, und  $p_k$  sei beliebig aus  $P$  gewählt. Die Hyperebene, in der  $p_k$  liegt, sei durch  $\langle \vec{n}, \vec{x} \rangle + d = 0$  beschrieben<sup>4</sup>. Sie partitioniert  $\mathbb{R}^d$  in zwei Teilmengen:

$$S_{k+} = \{x \in \mathbb{R}^d \mid \langle \vec{n}, \vec{x} \rangle + d \geq 0\}$$

$$S_{k-} = \{x \in \mathbb{R}^d \mid \langle \vec{n}, \vec{x} \rangle + d < 0\}$$

Das wesentliche Konzept von BSP Trees ergibt sich nun durch folgende *Beobachtung*: Angenommen, die Betrachterposition befindet sich in  $S_{k+}$ . Dann kann ein Polygon in  $S_{k-}$  weder  $p_k$  noch *irgendein* anderes Polygon in  $S_{k+}$  verdecken. Abbildung 1 illustriert diesen Sachverhalt.

Wenn für jeden Teilraum, der durch die obige Aufteilung entsteht, wieder eine partitionierende Ebene gewählt wird, führt dies in kanonischer Weise zu folgendem Verfahren:

<sup>3</sup>Ich werde im Folgenden diesen Begriff verwenden, ohne die Menge der Ausgabegeräte einschränken zu wollen.

<sup>4</sup>Dabei bezeichnet  $\langle \cdot, \cdot \rangle$  das Standardskalarprodukt von  $\mathbb{R}^d$ .

---

```

Tree MakeTree(PolygonList pl)
{
    int k = SelectPolygon(pl);
    PolygonList PosList, NegList;
    Polygon PosParts, NegParts;
    for(int i = 0; i < pl.size(); i++)
    {
        if(i != k)
        {
            SplitPolygon(pl[i], pl[k], PosParts, NegParts);
            PosList.push_back(PosParts);
            NegList.push_back(NegParts);
        }
    }
    return(CombineTree(MakeTree(PosList),
                       pl[k],
                       MakeTree(NegList)));
}

```

---

Listing 1: Algorithmus zur Baumerzeugung in Pseudocode, nach [FKN80]

---

#### Algorithmus 2 Erstellen eines BSP Trees

---

- 1: Partitioniere alle Polygone  $p_i \in P \setminus \{p_k\}$  an der Ebene von  $p_k$ . Falls ein Polygon relativ zur Partitionierungsebene in *beiden* Halbräumen liegt, wird es in zwei Teile geteilt, die jeweils in nur einem Halbraum liegen. Daher bezeichnet man diesen Prozess auch als *Splitting* und  $p_k$  als den *Splitter*.
  - 2: Wähle in  $S_{k+}$  und  $S_{k-}$  je ein neues Polygon, das zum Partitionieren der Mengen verwendet wird.
  - 3: Wiederhole diesen Prozess mit beiden Teilmengen, bis nur noch einelementige Mengen vorliegen.
- 

**Ergebnis** Bei der Durchführung des Algorithmus entsteht sukzessive ein Binärbaum. In jedem Schritt enthalten die Blätter eine Menge von Polygonen; die inneren Knoten enthalten dabei jeweils den Splitter. Wenn der Prozess terminiert, enthält der Baum in jedem Knoten nur noch genau ein Polygon.

In Listing 1 ist der Algorithmus in Pseudocode aufgeführt: Der BSP Tree wird durch sukzessives Einfügen der Polygone und rekursiven Aufruf der Funktion `MakeTree` erzeugt. Bei jedem Aufruf werden alle zu bearbeitenden Polygone an der aktuellen Ebene aufgeteilt und ihre Fragmente in die entsprechende Liste eingefügt. `CombineTree` erstellt dann aus einem Polygon sowie zwei Teilbäumen einen Knoten mit Kindern.

**Beispiel** Abbildung 2 zeigt eine sehr einfache Beispielszene, die einige Polygone enthält. Die Pfeilrichtung gibt die Richtung des Normalenvektors des jeweiligen Polygons an. Wendet man den obigen Algorithmus auf die Szene an, so ergibt sich der in Abbildung 3 dargestellte BSP Tree.

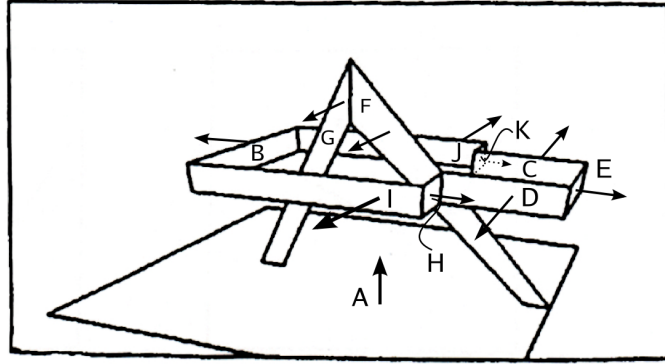


Abbildung 2: Beispielszene mit Polygonen und Normalen; nach [FKN80]

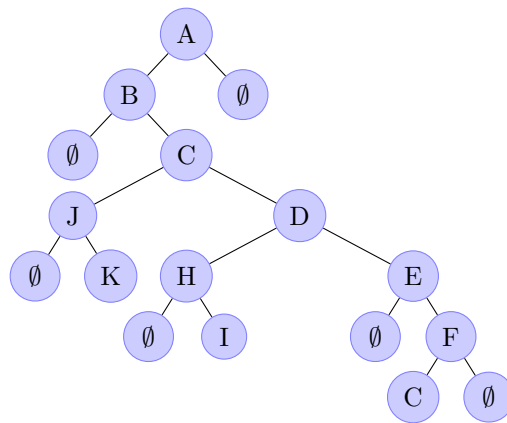


Abbildung 3: BSP Tree zu Abbildung 2

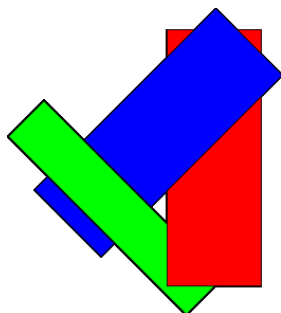


Abbildung 4: Zyklische Überlappung, aus [Mu09]

**Bemerkungen** Der oben beschriebene Algorithmus kann in euklidischen Vektorräumen beliebiger Dimension angewandt werden. Im Folgenden werde ich aber ausschließlich die in der Computergrafik üblichen Dimensionen 2 und 3 betrachten und besonders auf die Lösung von *Sichtbarkeitsproblemen* eingehen, um beispielsweise die korrekte Zeichenreihenfolge von Objekten zu bestimmen. Mit einigen konzeptuellen Änderungen kann ein BSP Tree dann auch zur *Schattenberechnung* benutzt werden, also zur Bestimmung der beleuchteten und unbeleuchteten Teile eines Objektes. Ebenso ist es möglich, einen BSP Tree bei *Klassifikationsproblemen* einzusetzen: Ein Durchlaufen des Baumes bis zu den Blättern erlaubt die Lokalisierung von Punkten oder Objekten relativ zu anderen Objekten. Diesen und anderen Anwendungen, die hier nicht erwähnt werden, widmet sich Bruce F. Naylor in [Nay01].

## 2.1 Der Maleralgorithmus

Die ursprüngliche Problemstellung von Fuchs et al. war es, die korrekte *Zeichenreihenfolge* einer Menge von Polygonen zu bestimmen. Dazu könnte beispielsweise ein naiver Maleralgorithmus<sup>5</sup> eingesetzt werden. Er läuft wie folgt ab:

---

### Algorithmus 3 Naiver Maleralgorithmus

---

- 1: Sortiere alle Polygone nach absteigender Distanz zur aktuellen Betrachterposition.
  - 2: Zeichne die Polygone nacheinander.
- 

Der Maleralgorithmus stellt einen einfachen, aber sehr effektiven Algorithmus zur Lösung des Sichtbarkeitsproblems dar: Objekte, die sich *vor* anderen Objekten befinden, werden später gezeichnet und überlagern diese somit. Es gibt jedoch Situationen, in denen der obige Algorithmus keine korrekte Zeichenreihenfolge bestimmen kann. Eine solche ist in Abbildung 4 dargestellt. Um dieses Problem zu lösen, schlagen Fuchs et al. ein „back-to-front“-Rendering unter Verwendung eines BSP Trees vor. Pathologische Fälle, wie in Abbildung 4 gezeigt, können dann nicht mehr auftreten, da die Polygone der Szene beim Erstellen des Baumes gegebenenfalls an einer Ebene aufgeteilt werden.

---

<sup>5</sup>Auch als „Painter’s Algorithm“ bekannt. Der Name ergibt sich daraus, dass Maler in Bildern die nahen Objekte über die weit entfernten Objekte malen.

---

```

BackToFront(EyePos e, Tree t)
{
    if (CheckPos(t.root, e) == POS)
    {
        BackToFront(e, t.neg);
        Draw(t.root);
        BackToFront(e, t.pos);
    }
    else
    {
        BackToFront(e, t.pos);
        Draw(t.root);
        BackToFront(e, t.neg);
    }
}

```

---

Listing 2: Maleralgorithmus nach [FKN80], in Pseudocode

Listing 2 zeigt den verbesserten Maleralgorithmus im Pseudocode: Zunächst wird überprüft, ob die Betrachterposition *vor* oder *hinter* der Wurzel des Baumes liegt. Falls sie *vor* der Wurzel liegt, also im „positiven“ Teilraum, wird der „negative“ Teilraum zuerst gezeichnet, dann die Wurzel, und zuletzt der Teilraum, in dem sich der Betrachter befindet. Der andere Fall läuft analog ab.

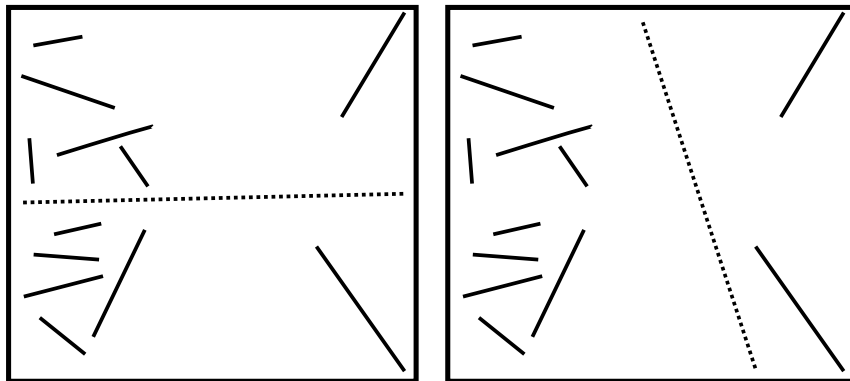
Der Algorithmus arbeitet korrekt, da die Polygone<sup>6</sup> je durch eine Ebene voneinander getrennt werden. Polygone auf der jeweiligen anderen Seite können also, wie wir bereits gesehen haben, *keine* Polygone auf der Seite des Betrachters verdecken.

Neben der Korrektheit offenbart sich auch ein weiterer Vorteil dieses Verfahrens: Da der BSP Tree *unabhängig* von der Betrachterposition ist und vor dem Rendering der Szene berechnet werden kann, muss zu jedem Zeitpunkt nur *ein* BSP Tree für die statischen Objekte der Szene im Speicher behalten werden. Der Maleralgorithmus kann also, ausgehend von einem vorberechneten BSP Tree, für *beliebige* Betrachterpositionen innerhalb der Szene durchgeführt werden. Aufgrund der elementaren Berechnungen, die im Wesentlichen nur aus Skalarprodukten bestehen, ist dies auch mit wenig Aufwand verbunden.

**Bemerkung** Das Sichtbarkeitsproblem lässt sich ebenso mit dem von Edwin Catmull entworfenen *z*-Buffer-Algorithmus lösen<sup>7</sup>. Dieser hat jedoch den Nachteil, dass zusätzlicher Speicher benötigt wird und ein Test der *z*-Koordinate *pro* Pixel erforderlich ist. Für statische Szenen ist die Verwendung von BSP Trees an dieser Stelle weitaus effizienter, da Speicher *und* CPU-Zeit gespart werden – unter Berücksichtigung des Erscheinungsdatum von [FKN80], ist dies ein nicht zu verachtender Aspekt. Wie eigene Messungen in Tabelle 1 auf Seite 25 zeigen, kann eine Verwendung von BSP Trees bei integrierten Grafikkarten jedoch auch heute noch leichte Leistungsvorteile erbringen.

<sup>6</sup>Ich bezeichne damit auch *Polygonfragmente*.

<sup>7</sup>Dies ist auch *heute*, da alle Grafikkarten über entsprechende Hardware verfügen, das Mittel der Wahl.



(a) Uneinheitlich verteilte Objekte, optimiert für einen balancierten Baum (b) Uneinheitlich verteilte Objekte, optimiert für Klassifikationsprobleme

Abbildung 5: Beispiele für heuristische Wahlen der partitionierenden Polygone

## 2.2 Probleme

Zwei Seelen wohnen, ach! in meiner Brust, die eine will sich von der andern trennen

Faust I, Johann Wolfgang von Goethe

Wie Listing 1 auf Seite 4 zeigt, stellt die *Anzahl* der Polygone im BSP Tree ein Problem dar: Sie ist offensichtlich abhängig von der Wahl des Splitters. Ein ungeschickt gewählter Splitter kann die Anzahl der Polygone, die im Baum gespeichert werden, drastisch erhöhen. Dies ist im Falle von Algorithmen wie dem Maleralgorithmus, die *jeden* Knoten besuchen müssen, katastrophal.

Für andere Anwendungen, beispielsweise Klassifikationsprobleme, ist es wiederum wichtig, dass der Baum möglichst *balanciert* ist, und damit ein schnelleres Durchsuchen erlaubt.

*Beide* Forderungen gleichzeitig erfüllen und somit einen „perfekten“ BSP Tree zu erzeugen, ist unmöglich – das Problem ist sogar NP-vollständig. Der BSP Tree muss daher je nach Anwendung entsprechend erzeugt werden. Einfache Heuristiken wählen in jedem Erzeugungsschritt des Baumes das Polygon aus, das entweder die kleinste Anzahl an neuen Splits durchführt oder das den Baum möglichst balanciert hält. Je nachdem, welche Eingabedaten vorliegen, können solche einfachen Heuristiken auch ineffizient arbeiten. Beispielsweise ist für ein Klassifikationsproblem der Splitter in Abbildung 5(a) weitaus schlechter gewählt als in Abbildung 5(b): Die erste Wahl sorgt zwar für einen balancierten Baum, wird aber der Struktur des Problems nicht gerecht. Klassifikationsanfragen für Punkte im rechten Teil der Szene erfordern den Vergleich mit einer unnötig großen Anzahl an Objekten.

Fuchs et al. schlagen für die Wahl eines Splitters eine Gewichtung von Polygonen vor. Ob eine solche Gewichtung bessere Ergebnisse liefert, lassen sie jedoch im Unklaren. Eine Analyse verschiedener Verfahren zur Splitterwahl findet sich in [RE01]. Für viele Anwendungen, insbesondere solche mit *wechselnden* Eingabedaten, ist eine zufällige Splitterwahl vorteilhafter als eine vorzeitige Optimierung, die je nach Eingabedaten viel Rechenzeit in Anspruch nehmen kann.

Es gilt Knuths Grundsatz: „We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.“

## 2.3 Erweiterungen

Bei der Betrachtung der bisher vorgestellten Algorithmen und Konzepte bieten sich einige natürliche Erweiterungen an. Dazu gehört beispielsweise die Einbeziehung dynamischer Szenen beziehungsweise dynamischer Objekte: Offensichtlich bereitet das *Einfügen* neuer Objekte keine Probleme; auch nicht, wenn dies nachträglich geschieht (der BSP Tree wird dann einfach um zusätzliche Knoten erweitert). Es ist allerdings nicht einfach, Objekte direkt aus einem BSP Tree zu entfernen, denn diese können beim Einfügen in Fragmente zerlegt werden, oder selber Splitter eines Teilraumes sein. Daher sind Fallunterscheidungen beim *Löschen* von Objekten nötig. Weiterhin ist darauf zu achten, dass nach dem Entfernen von Knoten die Struktur des BSP Trees erhalten bleibt. Im Allgemeinen sind dazu Neu- beziehungsweise Umsortierungen des Baumes erforderlich. Erst nach Abschluss der Löschoperation kann ein sich bewegendes Objekt an seiner neuen Position wieder in den Baum eingefügt werden.

Zu dynamischen BSP Trees gibt es unter anderem Ergebnisse von Naylor und Chrysanthou & Slater. Letztere widmen sich in [CS95] der dynamischen Berechnung von Schatten. Diesem Thema werden wir später nochmals als Beispiel für Dynamik in BSP Trees begegnen.

## 3 Anwendungen

Ich möchte nun auf zwei größere Anwendungsbereiche eingehen, in denen BSP Trees besonders gut verwendet werden können. Dies ist zum einen die Nutzung als „Grafikbeschleuniger“ im Bereich der Computerspiele – hier am Beispiel „Doom“, einem sogenannten „First-Person-Shooter“. Zum anderen werden wir Erweiterungen zu BSP Trees kennenlernen, die es ermöglichen, in statischen und dynamischen Szenen Schatten zu berechnen.

### 3.1 Computerspiele

BSP Trees erlangten ihre Bedeutung für die Spieleindustrie vor allen Dingen durch ein Spiel: „Doom“. Dieses bot 1993 als erstes Spiel seiner Art vergleichsweise riesige 3D-Spielwelten, die zudem texturiert waren. Obwohl das interne Datenformat keine Räume erlaubte, die direkt übereinander lagen, und Höhenunterschiede somit separat eingefügt werden mussten, vermittelte Doom den Spielern die Illusion einer großen, realistischen Spielwelt. Die grafischen Unterschiede zu anderen Spielen die zu dieser Zeit erhältlich waren, sind beträchtlich. Kein Spiel war in der Lage, Spielwelten in dieser Größe mit handelsüblicher Hardware flüssig darzustellen. Dahinter steckte die Game Engine „id Tech 1“, an deren Entwicklung unter anderem John Carmack und John Romero beteiligt waren. Aufgrund des großen Erfolges dieses Spielkonzepts wurde die Engine in vielen anderen Spielen von id Software verwendet: Doom II, Hexen, Heretic. . . Die grafische Darstellung lieferte id Software einen Vorteil, den Konkurrenten erst langsam aufholen konnten. Bis heute steht id Software daher für Spiele, die einen möglichst hohen Realismusgrad in ihrer Darstellung anstreben.



(a) „Freedoom“ / „PrBoom“, aus [Wik09a]

(b) Doom Cover Art, aus [Wik09b]

Abbildung 6: Artwork von Doom

Abbildung 6(b) zeigt ein Cover der Verkaufsversion von Doom, das den Spielablauf zusammenfasst: Als einfacher Soldat muss sich der Spieler gegenüber anrückenden Monsterhorden behaupten, um zu überleben und die Welt retten zu können. Über die Originalität der Hintergrundgeschichte lässt sich streiten, doch die Überlegenheit der Darstellungsqualität von Doom war 1993 unangefochten.

Doom benutzte BSP Trees, um ein schnelleres Rendering zu erreichen. Nur so war es mit den damaligen Grafikkarten möglich, 3D-Grafik flüssig darzustellen. Abbildung 6(a) zeigt einen Screenshot aus „Freedoom“, einem freien Nachbau der Levels des Originalspiels, zusammen mit „PrBoom“, einem freien Emulator, der unter anderem den mittlerweile freigegebenen Quelltext von Doom verwendet und die Benutzung unter modernen Betriebssystemen erlaubt. Es ist grafisch jedoch mit dem Originalspiel vergleichbar und vermittelt einen guten Eindruck der Darstellungsqualität.

### 3.1.1 Funktionsweise von BSP Trees in Doom

Der Prozess, der das gespeicherte Level in eine Bildschirmausgabe überführt, lässt sich sehr skizzenhaft in etwa wie folgt darstellen:

1. Zunächst werden die Geometriedaten eines Levels angelegt. Der Editor des Spiels gewährt dabei viele Freiheiten, allerdings sind gewisse technische Einschränkungen zu beachten: Eine Plazierung von Räumen direkt übereinander ist beispielsweise verboten.
2. Der Leveleditor erzeugt dann aus den Geometriedaten einen BSP Tree. Dieser ist so angelegt, dass jeder Knoten des Baumes einen *Bereich* innerhalb des Levels beschreibt. Der Wurzelknoten beschreibt dabei das gesamte Level.
3. Die Blätter des Baumes enthalten sogenannte *Subsektoren*. Dies sind konvexe Polygone, die nicht weiter aufgeteilt werden sollen.

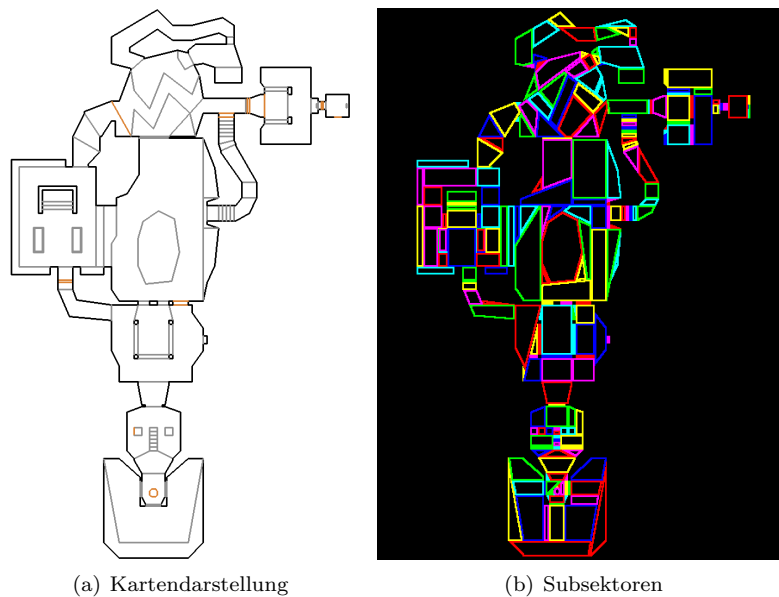


Abbildung 7: Das Level „E1M1: Hangar“ aus Doom; nach [Doo09]

4. Zum Zeichnen werden die Subsektoren zunächst sortiert, um eine korrekte Zeichenreihenfolge zu erhalten (vergleiche das Problem der zirkulären Überlappung aus Abbildung 4 auf Seite 6). In dieser Reihenfolge werden sie schließlich gezeichnet, bis allen Pixeln des Bildschirms ein Farbwert zugewiesen werden konnte.

Abbildung 7 zeigt eine Kartendarstellung des ersten Levels sowie die entsprechenden Subsektoren. Dabei ist gut erkennbar, dass die Engine kleinere Gänge in mehr Subsektoren aufgeteilt hat als größere Flächen im Level. Diese Flexibilität wird erst durch die BSP Trees sichergestellt und gibt den Entwicklern der Levels mehr Freiheiten als in anderen Spielen.

Listing 3 zeigt einen Ausschnitt aus dem Quelltext von Doom. Es handelt sich um den Code, der für das Rendering eines Knotens im BSP Tree zuständig ist: Nach dem Zeichnen des entsprechenden Subsektors wird die Position des Spieles klassifiziert. Dies wird rekursiv fortgesetzt.

Genauer soll an dieser Stelle nicht auf die Interna der Engine eingegangen werden. Für weitere Informationen sei auf den Quelltext von Doom verwiesen. Dieser steht Interessierten mittlerweile frei<sup>8</sup> zur Verfügung.

### 3.2 Schattenberechnung

Ein weiteres Anwendungsgebiet der BSP Trees stellt die Berechnung von Schatten dar. Chin & Feiner prägten dazu in [CF89] den Begriff „Shadow Volume BSP Tree“. Dieser stellt eine Erweiterung der bekannten BSP Trees dar. Bevor

<sup>8</sup>Eine GPL-lizenzierte Version kann beispielsweise unter <http://www.doomworld.com/idgames/?id=14576> bezogen werden.

---

```

void R_RenderBSPNode (int bspnum)
{
    node_t*    bsp;
    int       side;
    if (bspnum & NF_SUBSECTOR)
    {
        if (bspnum == -1)
            R_Subsector (0);
        else
            R_Subsector (bspnum & (~NF_SUBSECTOR));
        return;
    }
    bsp = &nodes[bspnum];

    side = R_PointOnSide (viewx, viewy, bsp);
    R_RenderBSPNode (bsp->children[side]);

    if (R_CheckBBox (bsp->bbox[side ^ 1]))
        R_RenderBSPNode (bsp->children[side ^ 1]);
}

```

---

Listing 3: Doom, Quelltextauszug

wir uns den Details widmen können, sind zunächst einige Begriffsdefinitionen notwendig:

**Definition 1** (Shadow Volume). Das Schattenvolumen, das ausgehend von einer Lichtquelle innerhalb der Szene erzeugt wird. Siehe Abbildung 8 auf Seite 13.

**Definition 2** (Shadow Plane). Die Schattenebene, die von einer Lichtquelle und einer *Kante* des Polygons erzeugt wird. Anhang der Position eines Polygons relativ zu einer Schattenebene kann entschieden werden, ob Teile des Polygons beleuchtet sind.

**Definition 3** (Shadow Node). Ein Blatt im (noch zu definierenden) Shadow Volume BSP Tree wird *in* genannt, wenn es eine Region beschreibt, die im *Schatten* liegt. Andernfalls wird das Blatt mit *out* bezeichnet.

Weiterhin möchte ich an dieser Stelle annehmen beziehungsweise voraussetzen, dass alle Polygone *konvex* und *planar* sind. Dies ist keine wirkliche Einschränkung, da konkave, nicht-planare Polygone in Dreiecke zerlegbar sind. Dafür erleichtert es den Umgang mit Geometriedaten erheblich.

Ausgehend von den obigen Begriffen kann nun der Shadow Volume BSP Tree definiert werden:

**Definition 4** (SVBSPT). Der Shadow Volume BSP Tree ist ein spezieller BSP Tree. Die inneren Knoten enthalten Schattenebenen, die Blätter sind entweder *in* oder *out*. Ein SVBSPT wird für jede Lichtquelle einzeln berechnet.

Abbildung 8 zeigt ein Beispiel für die obige Terminologie: In einer Szene mit einigen Liniensegmenten und einer Lichtquelle stellen die schattierten Bereiche

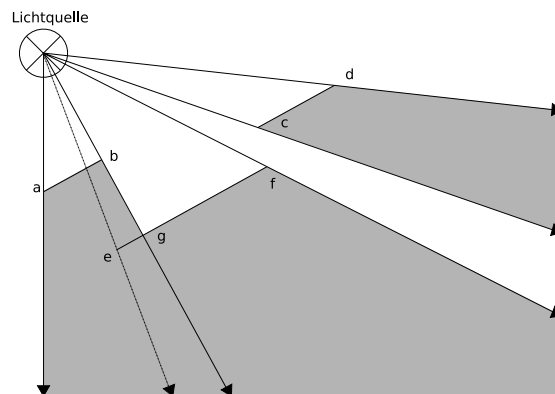


Abbildung 8: Schattenvolumen für eine Lichtquelle; nach [CF89]

das Schattenvolumen dar. Wir werden anhand dieser Abbildung später noch einen SVBSPT aufbauen.

**Bemerkung** Strenggenommen kann für Liniensegmente im  $\mathbb{R}^2$  nicht von einer Orientierung gesprochen werden. Die folgenden Beispiele dienen nur zur Illustration der entsprechenden Sachverhalte und setzen das Wissen um eine „Richtung“ von „Normalen“ für Liniensegmente und Punkte voraus.

### 3.2.1 Der SVBSPT-Algorithmus

Nach den vorangehenden Definitionen können wir nun den klassischen Algorithmus zur Berechnung von Schattenvolumina über BSP Trees nach [CF89] betrachten.

**Voraussetzungen** Dazu nehmen wir an, dass eine Szene mit *genau einer* Punktlichtquelle (diese kann gerichtet oder ungerichtet sein) und *statischer* Geometrie vorliegt. Weiterhin setzen wir einen BSP Tree voraus, der es uns ermöglicht, die Polygone in aufsteigender Distanz zur Lichtquelle zu sortieren<sup>9</sup>. Die letzte Voraussetzung betrifft die Normalenvektoren der Polygone: Normalenvektoren von planaren Polygonen, die Seitenflächen eines konvexen Objektes darstellen, sollen nach außen (vom Objekt *weg*) zeigen. Dies stellt sicher, dass die Erzeugung der Schattenknoten im SVBSPT-Algorithmus korrekt durchgeführt werden kann. In der Praxis kann dies einfach durch eine feste Reihenfolge in der Spezifikation der Eckpunkte eines Polygons angegeben werden (vergleiche hierzu beispielsweise die OpenGL-Funktion `glFrontFace`).

**Idee** Der Algorithmus selbst geht von einer einfachen Idee aus: Um zu überprüfen, ob ein Polygon, das *neu* zum SVBSPT hinzukommt, beleuchtet ist, genügt es, die Vereinigung aller bekannten Schattenvolumina zu bilden, die beleuchteten Teile des Polygons zu suchen, sowie die unbeleuchteten Teile zu igno-

<sup>9</sup>Diese Sortierung kann auch anderweitig vorgenommen werden. Wir haben aber bereits gesehen, dass ein BSP Tree einfach zu erstellen ist und pathologische Fälle wie zyklische Überlappung löst.

rieren. Sofern das Polygon nicht komplett im Schatten liegt, wird das Schattenvolumen erweitert.

Eine formalisierte Version dieser Überlegungen ist in Algorithmus 4 beschrieben. Dabei wird ein anfänglich leerer SVBSP Tree iterativ um Schattenebenen erweitert.

---

**Algorithmus 4** SVBSP-Algorithmus; nach [CF89]

---

- 1: Beginne mit einem leeren Baum, der nur einen „out“-Knoten enthält.
  - 2: **for** Polygon  $p$  aus der Szene **do**
  - 3:   Filtere das Polygon durch den SVBSP. Partitioniere dazu das Polygon an den *Schattenebenen* der inneren Knoten.
  - 4:   Das Polygon erreicht nach endlich vielen Schritten ein Blatt  $k$  des Baumes.
  - 5:   **if**  $k = \text{in}$  **then**
  - 6:      $p$  liegt im Schatten. Füge  $p$  *nicht* in den Baum ein, da das Schattenvolumen durch unbeleuchtete Polygone nicht erweitert wird.
  - 7:   **else**
  - 8:     Ersetze  $k$  durch das Polygonfragment: Erzeuge je einen Knoten aus den *Kanten* des Polygonfragmentes und setze sie zu einem Teilbaum zusammen. Die Knoten enthalten neue Schattenebenen, die das Schattenvolumen erweitern. Füge an die neuen Blätter einen „out“-Knoten in Richtung der Normalen an und einen „in“-Knoten in der entgegengesetzten Richtung.
  - 9:   **end if**
  - 10: **end for**
- 

Wenden wir den Algorithmus auf die Szene an, die in Abbildung 8 dargestellt ist, so erhalten<sup>10</sup> wir sukzessive den in Abbildung 9 beschriebenen SVBSP Tree.

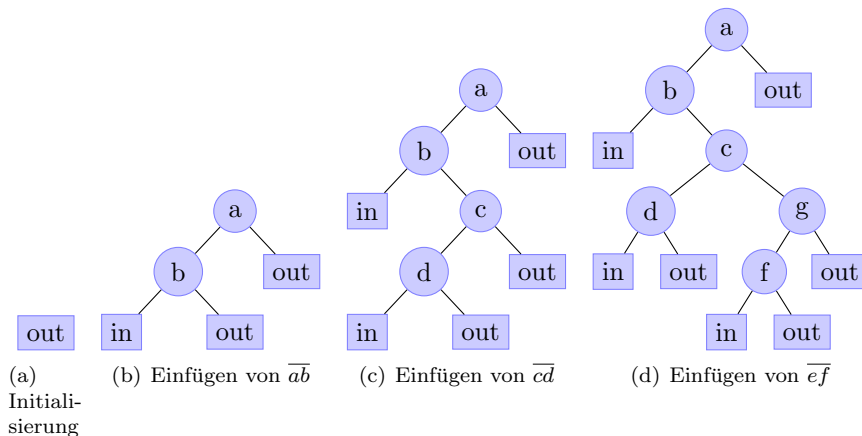


Abbildung 9: Iterativer Aufbau eines Shadow Volume BSP Trees

Beim Einfügen der einzelnen Liniensegmente ist zu beachten, dass die Reihenfolge, in der die Kanten für die Schattenknoten erzeugt werden, *willkürlich*

<sup>10</sup>Wie oben erwähnt, setzen wir die Orientierung dabei stillschweigend als gegeben bzw. bekannt voraus.

---

```

SVBSPT ShadowGenerator(Light PLS, BSPTree B)
{
    PolygonList P = B.FrontToBack();

    SVBSPT S = OUT;
    for(int i = 0; i < P.size(); i++)
    {
        S = DetermineShadow(P[i], S, PLS);
    }

    return(S);
}

```

---

Listing 4: Schattenberechnung mit Shadow Volume BSP Trees, Pseudocode

*gewählt* ist. Beispielsweise ist im zweiten Schritt eine Vertauschung von  $a$  und  $b$  zulässig und ebenso wohldefiniert.

Der letzte Schritt bei der Baumerzeugung muss noch gesondert erklärt werden: Wie in Abbildung 8 bereits angedeutet wurde, wird das Segment  $\overline{ef}$  in  $\overline{eg}$  und  $\overline{gf}$  geteilt. Diese Segmente entstehen durch Partitionieren am Knoten  $b$ .  $\overline{eg}$  liegt nun *entgegen* der Richtung der Normalen am Knoten  $b$  und somit in einem „in“-Knoten. Dagegen kann  $\overline{gf}$  weiter den Baum durchlaufen, bis es als Kind von  $c$  eingefügt wird und ein entsprechender Teilbaum erstellt wird.

Listings 4 und 5 zeigen den Ablauf des oben vorgestellten Algorithmus. Dabei bezeichnet `MakeSVBSPTNode` die Funktion, die aus dem Polygonfragment einen Teilbaum mit Schattenebenen erstellt. Beide Listings setzen eine Punktlichtquelle voraus.

Einige Ergebnisse aus [CF89] sind in Abbildung 10 dargestellt. Unter Berücksichtigung des Entstehungsdatums – 1989 – ist die Qualität der Bilder beeindruckend.

### 3.2.2 Optimierung

Der Algorithmus ist in seiner ursprünglichen Fassung nicht allzu effizient, da unter Umständen sehr viele Polygonfragmente betrachtet werden müssen. Chin & Feiner schlagen daher in [CF89] einige Optimierungsmöglichkeiten vor:

- Ein neues Polygon(fragment) muss immer bis zu den Blättern des Baumes eingefügt werden. Somit bietet eine *Balancierung* des Baumes Geschwindigkeitsvorteile. Dies sollte auch bei der Erzeugung der inneren Knoten berücksichtigt werden.
- Weiterhin ist die Größe des Baums kontrollierbar, indem nur die *Silhouette* von Objekten betrachtet wird anstatt das komplette Objekt. Dies verringert auch die Anzahl der Polygonfragmente, die zum Erstellen des Baumes benötigt werden.
- Zuletzt können bestimmte Objekte als nicht-schattenwerfend markiert werden und somit von jeder Berechnung ausgenommen sein. Obwohl damit

---

```

SVBSPT DetermineShadow(Polygon p, SVBSPT S, PLS)
{
    if(S == IN)
    {
        p.status = UNLIT;
        return(S);
    }
    else if(S == OUT)
    {
        p.status = LIT;
        return(MakeSVBSTNode(S, p, PLS));
    }
    else
    {
        SplitPolygon(p, S, PosPart, NegPart);
        S.NegNode = DetermineShadow(NegPart, S.NegNode, p);
        S.PosNode = DetermineShadow(PosPart, S.PosNode, p);
    }
    return(S);
}

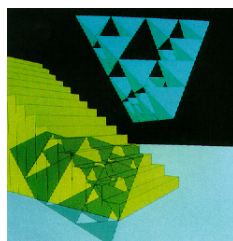
```

---

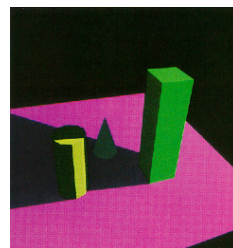
Listing 5: Filtern eines Polygons durch den Shadow Volume BSP Tree



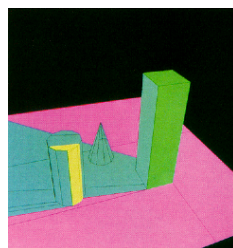
(a) Szene mit zwei Lichtquellen



(b) Rekursiver Tetraeder



(c) Szene mit einer Lichtquelle, gerendert



(d) Szene mit einer Lichtquelle, Schattenvolumina und Fragmente

Abbildung 10: Ergebnisse von Chin & Feiner; nach [CF89]

der Realismus von Bildern eingeschränkt wird, kann es sich für bestimmte Objekte lohnen, keine Schatten zu berechnen, da diese in der Szene nicht auftauchen würden. Falls an diesen Objekten viele Schatten auftreten, wird die Schattenberechnung beschleunigt. Dieses Vorgehen bietet sich beispielsweise *immer* bei begrenzenden Wänden einer Szene an: Zwar zeigen sich an der Wand selbst viele Schatten, die eine feine Unterteilung erfordern, doch die Wand selbst kann als Begrenzung in die weitere Szene keinen Schatten werfen.

### 3.2.3 Erweiterungen

**Mehrere Lichtquellen** Eine natürliche Erweiterung der oben beschriebenen Technik besteht in der Berechnung von Schatten in Szenen mit mehreren Lichtquellen. Hierzu schlagen Chin & Feiner vor:

---

**Algorithmus 5** SVBSPT-Algorithmus mit mehreren Lichtquellen

---

- 1: Berechne einen globalen BSP Tree. Dieser wird benötigt, um die Objekte in der richtigen Reihenfolge durchlaufen zu können sowie um die Schatten zu speichern.
  - 2: **for all** Punktlichtquelle *p* **do**
  - 3:   Führe SVBST-Algorithmus für eine Lichtquelle aus.
  - 4:   Die Polygonfragmente, die von *einem* Durchlauf erzeugt werden, speichern, von welcher Lichtquelle sie beleuchtet werden. Sie werden dann als Kinderknoten des ursprünglichen Polygons in dem *globalen* BSP Tree eingehängt.
  - 5: **end for**
- 

Es ist zu beachten, dass zu jeder Lichtquelle nur *ein* SVBSPT im Speicher gehalten werden muss. Die Ergebnisse der Schattenberechnung können in den BSP Tree der Szene übernommen werden. Um die Helligkeit eines Fragmentes zu bestimmen, kann dann beispielsweise die Anzahl der Lichtquellen, die das Fragment beleuchten, benutzt werden.

Bei der Implementierung dieser Erweiterung ist es ratsam, darauf zu achten, *wie* die Fragmente eines Polygons im BSP Tree gespeichert werden. Es ist *nicht* nötig, sie durch den bekannten Algorithmus in den Baum einzufügen. Dies würde nur die Anzahl der Knoten erhöhen, jedoch keine geometrischen Vorteile bringen: Da der BSP Tree bereits eine korrekte Zeichenreihenfolge der Polygone bestimmt hat, ändert sich nichts, wenn statt des ursprünglichen Polygons nur ein Fragment gezeichnet wird. Stattdessen sollte jedes Objekt im BSP Tree in der Lage sein, eine Liste von seinen Fragmenten zu verwalten. Es bietet sich eine Verwendung von entsprechenden Datenstrukturen wie ein Array aus Zeigern (in C/C++) an<sup>11</sup>.

**Realistischere Schatten** Ein weiteres Problem wird beim Darstellen der Szene offenbar: Die durch den Algorithmus erzeugten Schatten sind *harte* Schatten, die sehr künstlich erscheinen. Um natürlichere Schatten zu modellieren, ist es nötig, die *Ausdehnung* von Lichtquellen zu berücksichtigen. Dies wird auch mit

---

<sup>11</sup>Genauer ist beispielsweise in der Klasse `polygon` aus dem später vorgestellten Beispielprogramm zu finden.

„Area Lights“ bezeichnet. Ebenso ist es sinnvoll, wie bei natürlichen Schatten zwischen einem Halb- (Penumbra) sowie einem Kernschatten (Umbra) zu unterscheiden.

Beiden Problemen haben sich Chin & Feiner in [CF92] angenommen. Dabei verwenden sie für die Halb- und Kernschattenbereiche jeweils einen separaten Baum und führen komplexe Mischoperationen von Teilbäumen durch. Auf diese soll hier nicht weiter eingegangen werden.

**Dynamische Schatten** Alle bisher beschriebenen Algorithmen mit BSP Trees sind nur für Szenen mit *statischer Geometrie* entworfen. Gerade bei Schatten ist aber ein gewisses Maß an Dynamik wünschenswert. Chrysanthou & Slater beschreiben in [CS95] ein Verfahren zur dynamischen Berechnung von Schatten, das auf dem oben beschriebenen Algorithmus aufsetzt. Sie verwenden dabei einen *ungeordneten* SVBSPT (USVBST), bei dem *keine* vorherige Sortierung der Polygonfragmente vorgenommen wird – ein zweiter BSP Tree entfällt also. Um diesen Mangel an Information auszugleichen, sind einige Änderungen nötig:

- Polygone speichern, an welchen Stellen ihre Fragmente im Baum landen. Dies werden wir benötigen, um Polygone aus dem Baum löschen zu können, auch wenn sie in Fragmente zerlegt wurden.
- *Komplette Polygone* werden mit in den Baum eingefügt. Es wird dann zwischen „SP“-Knoten, die *Schattenebenen* enthalten, und „PP“-Knoten, die *Polygone* enthalten, unterschieden. Zum Vergleich: Im SVBSP Tree werden nur Knoten gespeichert, die eine Schattenebene beschreiben.
- „in“-Knoten können Polygone bzw. Polygonfragmente enthalten.
- Es gibt Zeiger („Rückwärtskanten“) im Baum. Diese werden verwendet, wenn ein Knoten tiefer im Baum einen Schatten auf einen *höheren* Knoten wirft. In [CS95] wird dies auch als „Target-Beziehung“ bezeichnet. Diese Beziehung ist nötig, da die Einfügereihenfolge der Polygone zufällig ist und es somit vorkommen kann, dass ein Objekt, das *vor* einem anderen Objekt steht und dieses (teilweise) verdeckt, erst *nach* diesem in den Baum eingefügt wird. Zudem erlauben die Zeiger ein leichteres Entfernen von Polygonen (und damit auch deren Schatten) aus einer Szene.

An der generellen Funktionsweise des USVBST wird nichts geändert: Auch hier findet eine Partitionierung an Schattenebenen statt, wie es bereits aus dem SVBSPT-Algorithmus bekannt ist. Allerdings enthält jedes Polygonfragment das Schattenvolumen des *ursprünglichen* Polygons. Wenn ein Polygon an einer Ebene partitioniert wird, ist also *keine* Neuberechnung der entsprechenden Schattenebenen nötig. Dies wird benötigt, um nach dem Entfernen von Knoten aus dem Baum die Teilbäume entsprechend mischen zu können, macht es aber nötig, zur Berechnung der Schatten Polygone zu „clippen“<sup>12</sup>.

Abbildung 11 zeigt eine einfache Szene, die nur aus einem Polygon besteht, und den entsprechenden USVBST. In den inneren Knoten stehen jeweils die Eckpunkte, welche die entsprechende Schattenebene definieren.

<sup>12</sup>Da ich die dynamische Schattenberechnung hier nur konzeptionell vorstellen möchte, verweise ich für genauere Informationen zur Berechnung des eigentlichen Schattens auf [CS95].

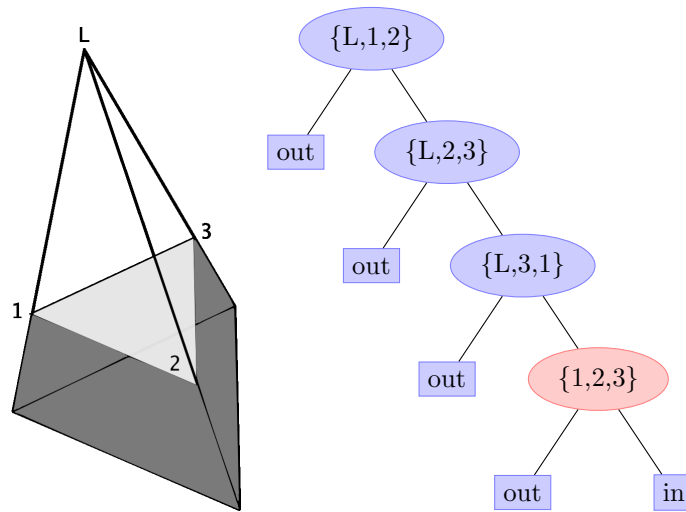


Abbildung 11: Ungeordneter Shadow Volume BSP Tree eines Polygons; nach [CS95]

Die Verwendung eines USVBSPT erlaubt ein einfacheres *Löschen* von Polygonen: Zunächst werden alle Stellen, an denen Fragmente eines Polygons im Baum stehen, markiert – dies ist möglich, da das Polygon eine entsprechende Liste von Zeigern auf Knoten verwaltet. Für jede dieser markierten Stellen ist nun eine Fallunterscheidung zu treffen:

1. Das zu löschende Fragment befindet sich in einem „in“-Knoten. In diesem Fall kann der entsprechende Knoten entfernt werden.
2. Das Fragment ist ein Blatt. Der Knoten wird durch einen „out“-Knoten ersetzt. Über die „Target“-Beziehung werden alle Schatten entfernt, die das Fragment wirft.
3. Das Fragment ist ein innerer Knoten. Dies ist der schwierigste Fall, da bei Entfernen des Fragments in jedem Fall *Teilbäume* entstehen, die wieder vereinigt werden müssen. Algorithmus 6 beschreibt das Vorgehen. Falls das Fragment selbst Schatten warf, so müssen diese durch Schatten von Polygonen ersetzt werden, die in einer „Target“-Beziehung mit dem Fragment stehen. Zuletzt müssen alle Polygone, die in einem „in“-Knoten *hinter* dem Fragment liegen, neu in den Baum eingefügt werden, nachdem die Teilbäume vereinigt wurden.

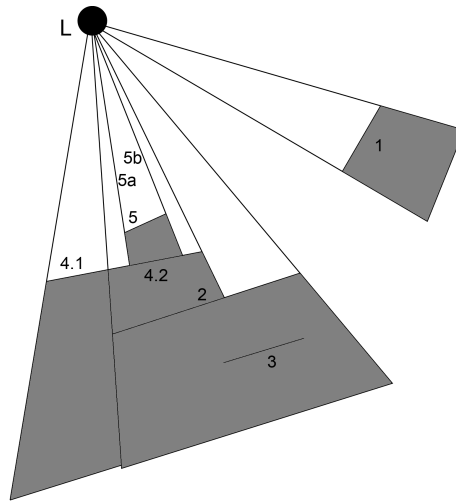


Abbildung 12: Einfache 2D-Szene mit Schattenvolumina; nach [CS95]

Die Teilbäume, die im letzten Fall entstehen, können wie folgt vereinigt werden:

---

**Algorithmus 6** Vereinigung von Teilbäumen; nach [CS95]

---

- 1: Suche den „PP“-Knoten, der zur Wurzel des *kleineren* Teilbaumes gehört.
  - 2: Filtere diesen Knoten dann mit allen Kinderknoten durch den größeren Teilbaum.
  - 3: Füge den „SP“-Knoten *vor* den „PP“-Knoten ein, sobald ein „out“-Knoten erreicht wird.
  - 4: Setze das Verfahren rekursiv mit den Teilbäumen der „SP“-Knoten der positiven Seite fort. Die negative Seite ist bereits durch den „PP“-Knoten bearbeitet worden.
- 

Abbildung 12 zeigt eine 2D-Szene mit verschiedenen Liniensegmenten. Die jeweiligen Schattenvolumina sind eingezeichnet. In Abbildung 13 ist *ein* möglicher USVBSPT für diese Szene dargestellt. Dabei sind „SP“-Knoten in blau gehalten, „PP“-Knoten in rot. Liniensegmente werden von links nach rechts gezeichnet und ihre Endpunkte sind mit Kleinbuchstaben (a,b, ...) versehen. Mit 1a ist dann beispielsweise das Liniensegment gemeint, das entsteht, wenn man die Lichtquelle mit dem linken Endpunkt des Segments 1 verbindet. Analog für die anderen Segmente. Geteilte Segmente werden durch einen Punkt durchnummeriert (4.1, 4.2, ...). Ich möchte an dieser Stelle nicht mehr auf die Erzeugung des Baumes eingehen – diese folgt hier einfach der Numerierung der Liniensegmente –, sondern stattdessen die Eigenschaften eines USVBSPT anhand dieses Beispielbaumes erklären:

- Da das Liniensegment 3 komplett im Schattenvolumen von 2 liegt, wird es in den entsprechenden „in“-Knoten eingefügt. Gäbe es noch weitere Polygone im Schattenvolumen von 2, so wären diese dort ebenfalls enthalten.
- Da 5 später in den Baum eingefügt wird als 4, gibt es eine Rückwärtskante

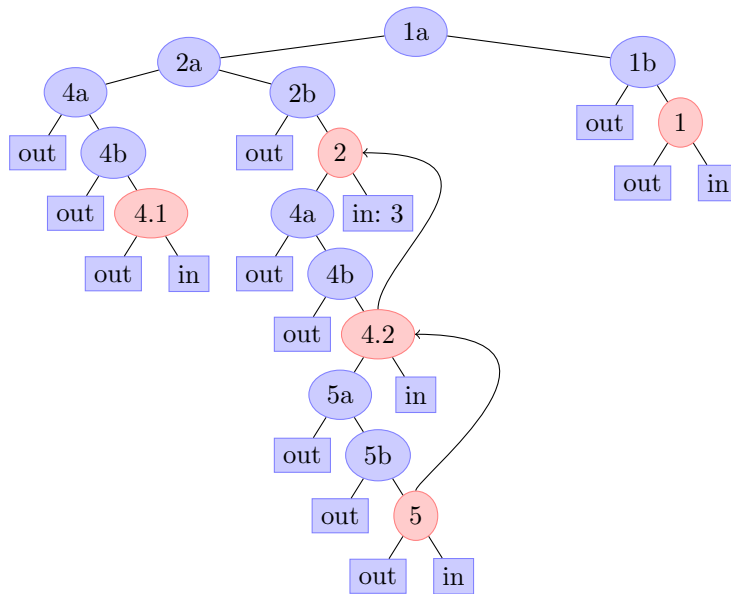


Abbildung 13: Ungeordneter Shadow Volume BSP Tree zu Abbildung 12

auf das Teilsegment 4.2. Dergleichen für die Rückwärtskante von 4.2 auf 2.

- Die Vaterknoten von 4.1 und 4.2 beschreiben jeweils *dasselbe* Schattenvolumen. So werden aufwendige Berechnungen von kleineren Schattenvolumina vermieden. Da das Segment selbst auch eingefügt wird, kann durch Betrachten der Orientierung immer noch entschieden werden, ob ein anderes Segment verdeckt wird oder nicht.

Nachdem nun die Unterschiede zum statischen Verfahren herausgearbeitet wurden, können wir anhand von Abbildung 13 auch den Löschvorgang von Polygonen beschreiben. Wir betrachten dazu alle Fälle, die weiter oben beschrieben wurden:

- Segment 3 kann einfach entfernt werden, da es sich innerhalb eines „in“-Knotens befindet. Weiter ist nichts zu beachten.
- Segment 5 (inklusive seines Schattenvolumens) befindet sich an einem Blatt und kann damit auch einfach entfernt werden. Allerdings muss dann, unter Berücksichtigung der „Target“-Beziehung, der Schatten von 4.2 entfernt werden.
- Segment 4 zu entfernen, erfordert ein Mischen von Teilbäumen nach Algorithmus 6 auf Seite 20. Ebenso muss der Schatten von 4.2 auf 2 durch einen Schatten von 5 auf 2 ersetzt werden. Diese Ersetzung ist aufgrund der Rückwärtskanten möglich.

Das Mischen ist in diesem Beispiel sehr einfach: Der kleinere Teilbaum ist der Teilbaum, der das Segment 5 enthält. Segment 5 erreicht den „out“-Knoten von Segment 2 und wirft einen Schatten auf selbiges. Danach kann

das Schattenvolumen wieder angefügt werden. Weitere Änderungen sind nicht nötig.

**Ergebnisse** Dieses scheinbar aufwendige Verfahren liefert in der Praxis überraschende Ergebnisse: Der Speicherverbrauch ist *geringer* als der einfacher anmutende SVBSPT-Algorithmus, der nur für statische Szenen gedacht ist. Ebenso ist die Rechenzeit reduziert, was insgesamt zur Folge hat, dass der Algorithmus, wie erhofft, *schneller* ist, als den SVBSPT in jedem Schritt neu aufzubauen. In [CS95] wird eine Dauer von 0.1s pro Frame angegeben, um dynamische Transformationen durchzuführen. Dieser Wert ist von 1995 und bezieht sich auf eine Szene mit ca. 60 Polygonen, sodass man davon ausgehen kann, dass heutige Hardware mit Szenen in der Größenordnung von mehreren 1000 Polygonen keine Probleme haben sollte.

Weitere Informationen über die Geschwindigkeit des Algorithmus können dem Artikel von Slater & Chrysanthou[CS95] entnommen werden. Ein früherer Artikel von Slater[Sla92] enthält einen Vergleich von verschiedenen Algorithmen zur Berechnung von Schatten mit BSP Trees. Da das USVBSPT-Verfahren noch *nicht* bekannt war, ist es dort allerdings nicht aufgeführt.

## 4 Demonstrationsprogramm in OpenGL

Um die erhaltenen Kenntnisse über BSP Trees in die Praxis zu übertragen, habe ich ein Demonstrationsprogramm in C++ geschrieben, das die wichtigsten Konzepte der vorliegenden Arbeit illustriert.

Das Programm enthält vollständige Anwendungsbeispiele zum Hidden Surface Removal, wie in [FKN80] beschrieben, sowie zur Berechnung von Schatten in statischen Szenen nach [CF89]. Weiterhin ist eine einfache Beschreibungssprache für simple Szenen, die aus konvexen Polygonen aufgebaut sind, implementiert worden. Alle Quelltexte sind unter einer BSD-Lizenz freigegeben und stehen unter <http://canmore.annwfn.net/uni/bsp.html> zur Verfügung.

### 4.1 Dateiformat

Das Dateiformat ist bewusst einfach gehalten, um die prozedurale Erzeugung von Szenen zu erleichtern. Es ist vorgesehen, dass Benutzer Lichtquellen in einer Szene definieren und beliebige Polygone beschreiben können. Ebenso sind Kommentare zur besseren Lesbarkeit der Datei gestattet. Aufgrund der Trivialität des Formats erscheint eine direkte Beschreibung anhand von Beispielen angemessen.

**Kommentare** Zur Gliederung der Szenenbeschreibung oder zu Dokumentationszwecken können durch Einfügen einer Zeile, die mit `#` beginnt, Kommentare geschrieben werden. Diese werden bei der Verarbeitung der Szenendatei nicht berücksichtigt.

```
#  
# Dies ist ein Kommentar.  
#
```

**Lichtquellen** Lichtquellen werden für die Schattenerzeugung benötigt. Falls eine Szene keine Lichtquelle enthält, wird ein leerer BSP Tree gezeichnet, d.h. es ist *nichts* sichtbar. Um eine Lichtquelle zur Szene hinzuzufügen, genügt die folgende Zeile:

```
L 0 0 0
```

Dies fügt eine Lichtquelle am Ursprung des Koordinatensystems ein. Beliebige Koordinaten sind erlaubt, es wird allerdings *nicht* überprüft, ob die Koordinaten im aktuellen Bezugssystem sinnvoll sind. Die Lichtquelle wird dann bei der Schattenerzeugung an die entsprechende Position gesetzt.

**Polygone** Polygone werden anhand ihrer Eckpunkte beschrieben. Jede Zeile, die 3 durch Leerzeichen oder Tabulatoren getrennte Zahlen enthält, wird als Punkt eines konvexen Polygons aufgefasst. Die Konvexität des Polygons ist obligatorisch, da die Polygondaten von OpenGL als `GL_POLYGON` gezeichnet werden und diese Funktion nur konvexe Polygone unterstützt. Ebenso sind *alle* implementierten Algorithmen nur für konvexe Polygone definiert.

Beim Lesen der Daten wird allerdings *nicht* überprüft, ob das Polygon tatsächlich konvex ist. Um ein Polygon abzuschließen und das Programm zu veranlassen, die Verarbeitung des nächsten Polygons zu beginnen, wird eine Zeile verwendet, die nur `EOP` enthält. Dieser Mechanismus erlaubt auch die Beschreibung von konkaven Objekten, solange diese in konvexe Teilstücke zerlegt werden. Ein Beispiel für eine Polygonbeschreibung:

```
# Beschreibt ein Dreieck
0 0 0
1 0 0
1 1 0
```

**Beispiel** Listing 6 zeigt den vollständigen Quelltext einer sehr einfachen Szene. Sie enthält zwei Lichtquellen, sowie ein Dreieck, das über eine Ebene schwebt.

---

```
# lights
L -5.0 2.6 -3.0
L -5.0 2.6 3.0

# floor
-5      0      +5
-5      0      -5
+5      0      -5
+5      0      +5
EOP

# triangle
-0.25  0.8      +0.25
-0.25  0.8      -0.25
+0.25  0.8      -0.25
EOP
```

---

Listing 6: Quelltext einer Beispielszene

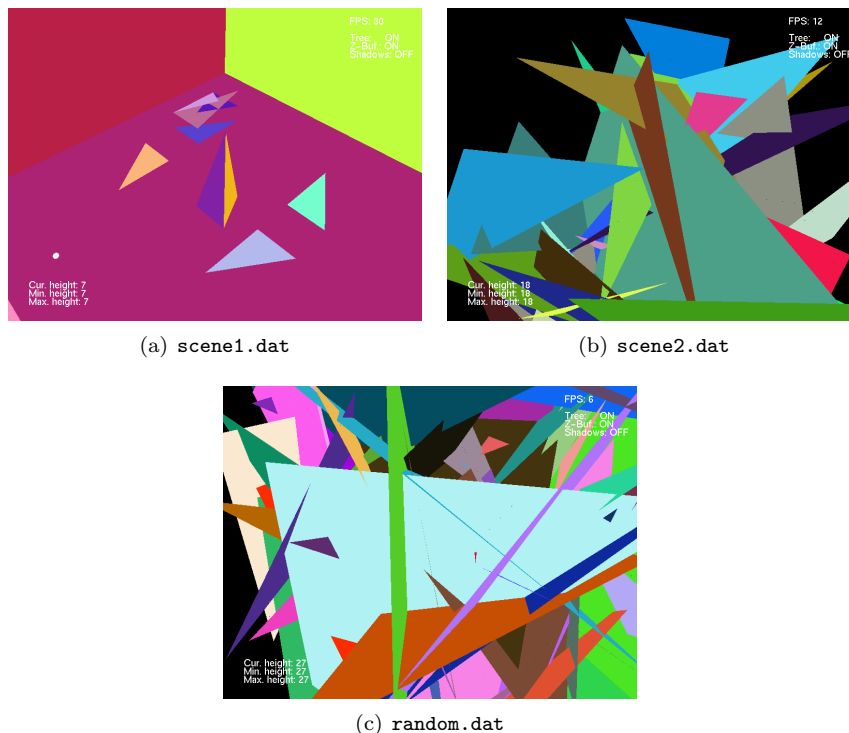


Abbildung 14: Szenen zum Hidden Surface Removal mit BSP Trees

## 4.2 Hidden Surface Removal

Der erste Betriebsmodus des Programms folgt im Wesentlichen der in [FKN80] vorgestellten Ideen: Nach dem Lesen der Szenendaten werden die Polygone in einen BSP Tree eingefügt. Dabei findet eine automatische Partitionierung an zufällig ausgewählten Polygonen statt.

Auf die so gespeicherten Polygone wird dann unter Benutzung des BSP Trees der Maleralgorithmus angewandt, wie er in Listing 2 auf Seite 7 beschrieben ist. Dabei wird der standardmäßig vorhandene Tiefenpuffer von OpenGL ausgeschaltet, da er für die Berechnung der korrekten Zeichenreihenfolge nicht mehr benötigt wird.

In Tabelle 1 sind die Messresultate für diverse Szenen verschiedenster Komplexität festgehalten. Es wurden auf zwei unterschiedlichen Maschinenkonfigurationen je zwei Messungen pro Szene durchgeführt: In der ersten Messung wurde der Hardware  $z$ -Buffer der Grafikkarte zum Rendern verwendet (bzw. die Standardmethoden, die OpenGL bietet), in der zweiten Messung wurde der BSP Tree eingeschaltet.

Zum Vergleichen der Daten wurden ein Laptop mit Intel Celeron M 1.4GHz CPU und langsamer interner Grafikkarte sowie eine Workstation mit Intel Pentium 4 2.4GHz CPU und schneller Nvidia-Grafikkarte verwendet. In beiden Fällen war das Betriebssystem FreeBSD 7.1 mit GLUT 7.3.

In Abbildung 14 sind alle Szenen dargestellt. Um die Resultate besser interpretieren zu können, wird jede einzelne Szene noch kurz im Text beschrieben.

Szene	FPS ( $z$ -Buffer)	FPS (BSP Tree)	Höhe BSP Tree
<code>scene1.dat</code>	26	30	7 – 12
	406	434	
<code>scene2.dat</code>	18	14	17 – 26
	370	320	
<code>random.dat</code>	12	6	26 – 34
	300	160	

Tabelle 1: Messsdaten zur Berechnung eines BSP Trees

An dieser Stelle sei darauf hingewiesen, dass die Menge der Polygone im Vergleich zu modernen Grafikanwendung sehr gering ausfällt und die Messungen durch Verwendung besserer Renderingtechniken sicher noch verbessert werden könnten.

`scene1.dat` Die erste Szene ist von einfacher Komplexität: Innerhalb eines Raumes sind verschiedenste simple Objekte wie Dreiecke gruppiert, teilweise durchdringen sie sich.

Hier bietet die Verwendung von BSP Trees einen leichten Geschwindigkeitsvorteil gegenüber dem  $z$ -Buffer. Dies liegt an der Einfachheit der Szene, für die kaum Partitionierungen nötig sind. Insbesondere auf der Maschine mit langsamer Grafikkarte und langsamem Prozessor ist die Framerate so noch leicht ohne Optimierungen steigerbar.

`scene2.dat` Diese Szene ist von hoher Komplexität: Es wurden zufällige Dreiecke so generiert, dass zwei der drei Eckpunkte dieselbe  $Z$ -Koordinate haben. Die meisten Objekte durchdringen sich.

In diesem Fall sind BSP Trees etwas langsamer als der  $z$ -Buffer, denn es entstehen viele Fragmente aufgrund der beinahe zufälligen Plazierung der Objekte. Diese müssen beim Zeichnen des BSP Trees berücksichtigt werden, wohingegen der  $z$ -Buffer direkt auf den ungeteilten Polygondaten operieren kann.

`random.dat` Die letzte Szene ist geometrisch höchst komplex: Es werden zufällige Dreiecke ohne Muster und Beschränkungen erzeugt. Als Ergebnis erhält man eine chaotische anmutende Menge von Polygonen, die sich gegenseitig durchdringen.

Als weitere Verfeinerung von `scene2.dat` sind hier signifikante Geschwindigkeitsunterschiede zwischen  $z$ -Buffer und BSP Trees sichtbar. Selbst mit dem schnelleren PC wirkt sich die Gesamtanzahl der Fragmente im BSP Trees derart negativ auf die Performance aus, dass die Darstellung durch BSP Trees ca. 50% langsamer ist. Dies liegt daran, dass viele Unterteilungen der Polygone durchgeführt werden müssen, was für einen extrem großen Baum sorgt.

### 4.3 Schattenberechnung

Zur Schattenberechnung werden die in [CF89] eingeführten SVBSP Trees verwendet.

Szene	Lichtquellen	Fragmente vorher	Fragmente nach SVBSPT	Zeit (Aufbau)	Zeit (Berechnung)
<code>simple.dat</code>	4	20	2718	28.704ms	0.818ms
<code>cubes.dat</code>	1	1488	5254	48.87ms	0.997ms
<code>scene1.dat</code>	1	34	145	0.861ms	0.012ms

Tabelle 2: Messdaten zur Berechnung von Schatten mit SVBSP Trees

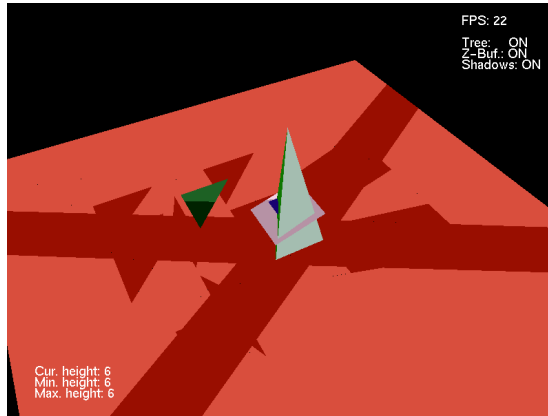
In Tabelle 2 sind die Ergebnisse der Schattenberechnung mit SVBSP Trees aufgeführt. Alle Messungen wurden auf einem Intel Celeron M 1.4GHz unter FreeBSD 7.1 ausgeführt<sup>13</sup>. Es wurden einige Performanceoptimierungen im Quelltext vorgenommen, doch weder werden besondere Fähigkeiten der CPU noch der Grafikkarte ausgenutzt. Abbildung 15 zeigt alle verwendeten Szenen.

`simple.dat` Die Szene ist sehr einfach gehalten: Sie besteht im Wesentlichen aus einem Tetraeder, der auf einer großen Ebene steht und von mehreren Lichtquellen beleuchtet wird. Interessant ist, dass sich die Anzahl der Fragmente beim Berechnen der Schatten um dem Faktor 1000 erhöht – und das, obwohl die Szene bewusst „harmlos“ gehalten wurde. Dies ließe sich vermeiden, wenn nach der Berechnung der Schatten pro Lichtquelle wieder versucht wird, aus benachbarten Fragmenten, die gleiche Beleuchtungen aufweisen, ein größeres Fragment zu machen. Eine solche Vereinigung ist jedoch auch nicht trivial lösbar, sodass es unklar ist, ob sie sich tatsächlich positiv auf die Geschwindigkeit auswirken würde.

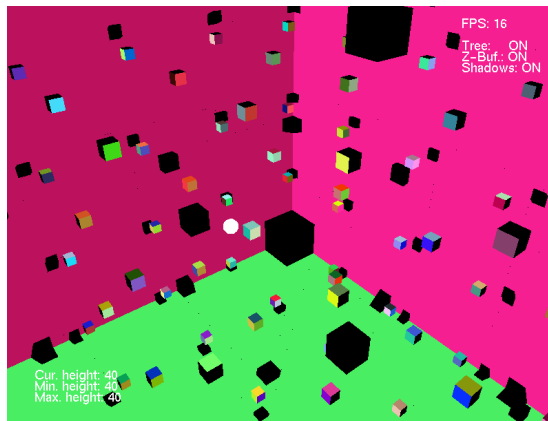
`cubes.dat` Diese Szene besteht aus Würfeln, die von einer zentralen Lichtquelle angestrahlt werden. Aufgrund der Regelmäßigkeit der Geometrie ergeben sich nur ca. 4-mal so viele Polygonfragmente bei der Schattenerzeugung.

`scene1.dat` Die letzte Szene enthält eine Lichtquelle sowie simple Objekte, die sich in einem abgeschlossenen Raum befinden. Die Zeit zur Berechnung der Schatten ist zu vernachlässigen und verdeutlicht, wie Schatten schnell und einfach berechnet werden können, sofern die Szene eine einfache Geometrie hat.

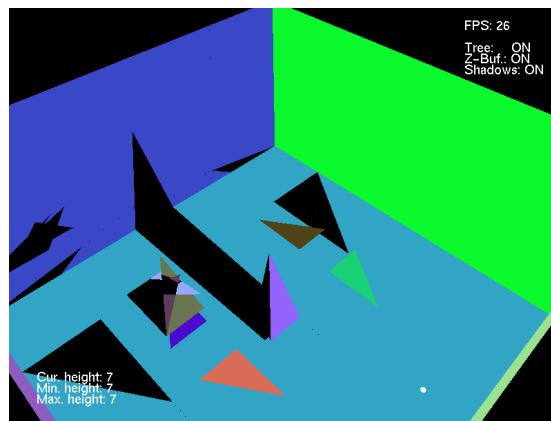
<sup>13</sup>Einen Geschwindigkeitsvorteil unter einem schnelleren System wurde nicht festgestellt. Daher sind die Messungen nicht doppelt aufgeführt.



(a) simple.dat



(b) cubes.dat



(c) scene1.dat

Abbildung 15: Szenen zur Schattenberechnung mit BSP Trees

## Literatur

- [CF89] Norman Chin and Steven Feiner, *Near real-time shadow generation using bsp trees*, SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1989, pp. 99–106.
- [CF92] ———, *Fast object-precision shadow generation for area light sources using bsp trees*, SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics (New York, NY, USA), ACM, 1992, pp. 21–30.
- [CS95] Yiorgos Chrysanthou and Mel Slater, *Shadow volume bsp trees for computation of shadows in dynamic scenes*, SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics (New York, NY, USA), ACM, 1995, pp. 45–50.
- [dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf, *Computational geometry: Algorithms and applications*, Springer, 1997, 2000.
- [Doo09] Doom Wiki, *All ss e1m1.png*, 2009, [Online; Stand 4. Januar 2009].
- [FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor, *On visible surface generation by a priori tree structures*, SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1980, pp. 124–133.
- [IWP08] Thiago Ize, Ingo Wald, and Steven G. Parker, *Ray tracing with the BSP tree*, RT 2008: IEEE Symposium on Interactive Ray Tracing, 2008, pp. 159–166.
- [Muł09] Wojciech Muła, *Painter's problem*, 2009, [Online; Stand 4. Januar 2009].
- [Nay01] Bruce F. Naylor, *A tutorial on binary space partitioning trees*, 2001.
- [RE01] Samuel Ranta-Eskola, *Binary space partitioning trees and polygon removal in real time 3d rendering*, Master's thesis, Uppsala University, 2001.
- [Sla92] Mel Slater, *A comparison of three shadow volume algorithms*, Vis. Comput. **9** (1992), no. 1, 25–38.
- [Wik09a] Wikipedia, *Bild: Playing the freedom iwad in the prboom engine*, 2009, [Online; Stand 4. Januar 2009].
- [Wik09b] ———, *Doom cover art*, 2009, [Online; Stand 4. Januar 2009].

## Abbildungsverzeichnis

1	Beispiel für partitionierende Ebene; nach [FKN80] . . . . .	3
2	Beispielszene mit Polygonen und Normalen; nach [FKN80] . . . . .	5
3	BSP Tree zu Abbildung 2 . . . . .	5
4	Zyklische Überlappung, aus [Mul09] . . . . .	6
5	Beispiele für heuristische Wahlen der partitionierenden Polygone	8
6	Artwork von Doom . . . . .	10
7	Das Level „E1M1: Hangar“ aus Doom; nach [Doo09] . . . . .	11
8	Schattenvolumen für eine Lichtquelle; nach [CF89] . . . . .	13
9	Iterativer Aufbau eines Shadow Volume BSP Trees . . . . .	14
10	Ergebnisse von Chin & Feiner; nach [CF89] . . . . .	16
11	Ungeordneter Shadow Volume BSP Tree eines Polygons . . . . .	19
12	Einfache 2D-Szene mit Schattenvolumina; nach [CS95] . . . . .	20
13	Ungeordneter Shadow Volume BSP Tree zu Abbildung 12 . . . . .	21
14	Szenen zum Hidden Surface Removal mit BSP Trees . . . . .	24
15	Szenen zur Schattenberechnung mit BSP Trees . . . . .	27

## Tabellenverzeichnis

1	Messdaten zur Berechnung eines BSP Trees . . . . .	25
2	Messdaten zur Berechnung von Schatten mit SVBSP Trees . . . . .	26

## Algorithmenverzeichnis

1	„Binary Space Partitioning“ . . . . .	1
2	Erstellen eines BSP Trees . . . . .	4
3	Naiver Maleralgorithmus . . . . .	6
4	SVBSPT-Algorithmus; nach [CF89] . . . . .	14
5	SVBSPT-Algorithmus mit mehreren Lichtquellen . . . . .	17
6	Vereinigung von Teilbäumen; nach [CS95] . . . . .	20