Effizientes Ray-Tracing

Marc Wagner
mcwagner@stud.informatik.uni-erlangen.de
http://wwwcip.informatik.uni-erlangen.de/~mcwagner
Vortrag im Seminar „Graphische Datenverarbeitung“
5. Mai 2000

Zusammenfassung

Ray-Tracing ist eine häufig verwendete Methode zum Erzeugen fotorealistischer Bilder. Es handelt sich um ein relativ rechenintensives Verfahren, das oft sehr viel Zeit benötigt. In diesem Vortrag werden verschiedene Algorithmen und Konzepte präsentiert, die den Vorgang der Bildberechnung deutlich beschleunigen, unter anderem effiziente Schnitt-Tests mit Bounding-Volumes, die richtige Verwendung hierarchischer Bounding-Volumes und der Einsatz von Shadow-Caches.

1 Ray-Tracing

Ray-Tracing ist eine Technik zum Erzeugen fotorealistischer Bilder. Optische Effekte wie Spiegelung an glatten Oberflächen, Transparenz von Objekten und Schattenbildung werden von ihr berücksichtigt. Die dreidimensionale, mathematische Beschreibung dessen, was später auf dem Bild zu sehen sein wird, bezeichnet man als Szene. Eine Szene besteht in der Regel aus einer Kamera, mehreren Objekten (zum Beispiel Dreiecke, Kugeln, Zylinder) und mehreren Lichtquellen. Das Grundprinzip von Ray-Tracing besteht darin, Strahlen von der Kamera auszusenden und festzustellen welches Objekt zuerst getroffen wird. Handelt es sich dabei um ein spiegelndes Objekt, wird vom Treffpunkt ein reflektierter Strahl ausgesendet, ist das Objekt transparent, wird ein gebrochener Strahl berechnet. Die Ergebnisse all dieser Berechnungen werden zu einem Bild kombiniert.

Für jeden Bildpunkt wird von der Kamera ein Strahl ausgesendet (siehe Abbildung 1). Trifft der Strahl kein Objekt, wird dem Bildpunkt die in der Szene definierte Hintergrundfarbe zugewiesen. Trifft der Strahl jedoch ein Objekt, werden dessen Materialeigenschaften (zum Beispiel Farbe, Intensität von Reflektionen, Transparenz) und alle Lichtquellen, die den Treffpunkt beleuchten, ermittelt. Anhand dieser Informationen kann ein Farbwert berechnet werden. Handelt es sich um ein spiegelndes Objekt, wird ein reflektierter Strahl

ausgesendet, der einen zweiten Farbwert liefert. Ist das Objekt transparent, führt ein gebrochener Strahl zu einem dritten Farbwert. Die drei so gewonnenen Farbwerte werden nun in Abhängigkeit von den Materialeigenschaften des getroffenen Objekts zu einem weiteren Farbwert kombiniert, der dem entsprechenden Bildpunkt zugewiesen wird.

Im weiteren Verlauf des Vortrags wird häufig zwischen Intersection-Rays und Shadow-Rays unterschieden.

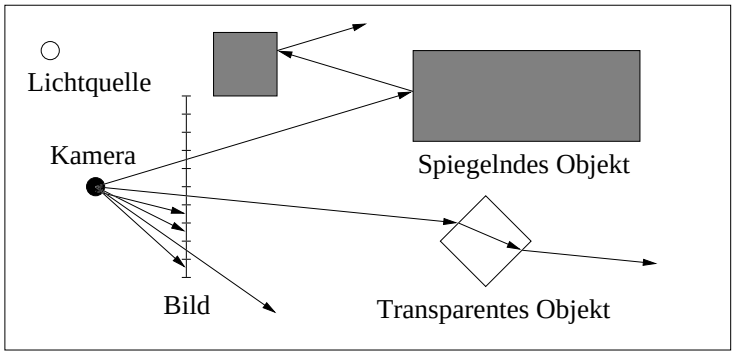

Abbildung 1: Intersection-Rays

Alle von der Kamera ausgehenden, reflektierten und gebrochenen Strahlen werden als Intersection-Rays bezeichnet (siehe Abbildung 1). Bei Intersection-Rays ist es notwendig, das zuerst getroffene Objekt und den genauen Schnittpunkt von Strahl und Objekt festzustellen. Von der Kamera ausgehende Strahlen werden genauso behandelt, wie reflektierte und gebrochene Strahlen. Letztere können also durchaus weitere reflektierte und gebrochene Strahlen zur Folge haben. Um extrem lange, eventuell sogar endlose Berechnungen zu vermeiden, wird das Verfahren in der Regel bei einer bestimmten Rekursionstiefe abgebrochen.

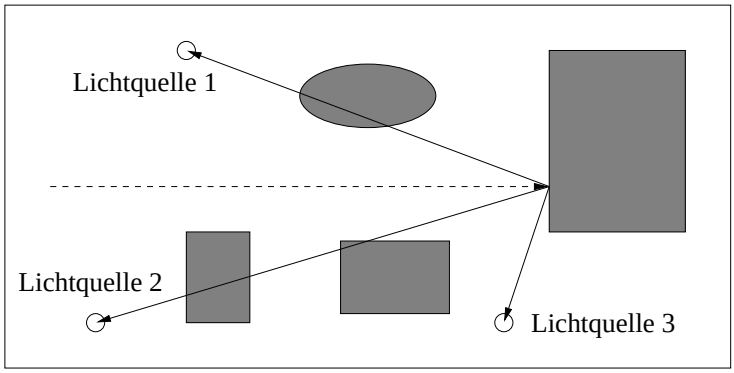

Abbildung 2: Shadow-Rays

Shadow-Rays werden verwendet, um festzustellen, ob eine Lichtquelle einen bestimmten Punkt eines Objekts beleuchtet. Dazu verfolgt man einen Strahl von diesem Punkt zu

dieser Lichtquelle und überprüft, ob der Strahl irgendein Objekt trifft, also ein Objekt die Lichtquelle abschirmt (siehe Abbildung 2).

Da bei den meisten Bildern Millionen von Strahlen ausgesendet werden, ist Ray-Tracing ein sehr rechenintensives Verfahren, das viel Zeit kostet. Das Optimieren von bestehendem Programmcode mag gewisse Zeitersparnisse mit sich bringen, ist jedoch stark hardwareabhängig und durch die verwendeten Ray-Tracing-Algorithmen begrenzt. Wirkliche Verbesserungen wird man nur durch neue, effizientere Algorithmen erzielen. Einige dieser Algorithmen und Konzepte werden in den folgenden Kapiteln vorgestellt.

2 Bounding-Volumes

Beim Ray-Tracing wird ein beträchtlicher Teil der Zeit dazu verwendet, festzustellen, ob ein Strahl ein bestimmtes Objekt trifft oder verfehlt. Der Aufwand eines solchen Schnitt-Tests ist stark abhängig von der Art des Objekts. Er ist zum Beispiel bei Kugeln oder Boxen relativ gering, bei Tensorprodukt-Bezierflächen dagegen sehr hoch.

Als Bounding-Volumes verwendet man einfache Objekte, zum Beispiel Kugeln oder Boxen. Aufwendig zu berechnende Objekte werden möglichst eng durch solche einfachen Bounding-Volumes umgeben (siehe Abbildung 3). An Stelle eines Schnitt-Tests mit dem Objekt wird zuerst ein Schnitt-Test mit dem umgebenden Bounding-Volume durchgeführt. Verfehlt der Strahl das Bounding-Volume, verfehlt er mit Sicherheit auch das Objekt, und man kann sich den aufwendigen Schnitt-Test mit dem Objekt ersparen. Trifft der Strahl jedoch das Bounding-Volume, ist es durchaus möglich, dass er auch das Objekt trifft, und ein Schnitt-Test mit dem Objekt muss durchgeführt werden.

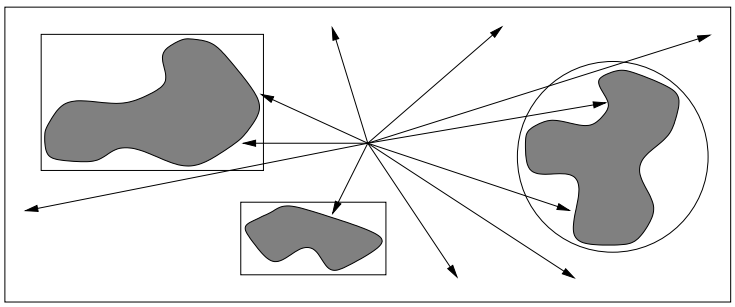

Abbildung 3: Bounding-Volumes

Bounding-Volumes sind vor allem bei sehr komplexen aber kleinen Objekten vorteilhaft. Solche Objekte werden von den meisten Strahlen verfehlt, genau wie die sie umgebenden Bounding-Volumes. Dadurch erspart man sich viele der aufwendigen Schnitt-Tests mit den komplexen Objekten.

3 Schnitt-Tests mit Bounding-Volumes

Da in der Regel sehr viel mehr Schnitt-Tests mit Bounding-Volumes als mit Objekten durchgeführt werden, ist ein effizienter Algorithmus für Schnitt-Tests mit Bounding-Volumes wünschenswert. Im folgenden werden als Bounding-Volumes ausschliesslich axial ausgerichtete Boxen verwendet, also Boxen, deren Seiten parallel zu je zwei Achsen des Koordinatensystems sind.

Ein relativ aufwendiger aber mit Sicherheit korrekter Schnitt-Test ist die Berechnung der Schnittpunkte von Strahl und Bounding-Box. Der Strahl trifft die Box genau dann, wenn Schnittpunkte existieren.

Ein sehr viel effizienteres Verfahren ist das folgende. Eine Bounding-Box ist gegeben durch sechs Werte, $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, $z_{\min}$ und $z_{\max}$, welche die Ausdehnung der Box in x-, y- und z-Richtung beschreiben. Der Strahl

$$\vec{r}(t) = \vec{a} + t\vec{b} \qquad , \qquad t \in [0, \infty) = I$$

liegt in parametrischer Darstellung vor. Man ermittelt nun die drei Intervalle

$$\begin{aligned} I_x &= \{t \mid \vec{r}(t) \text{ liegt zwischen } x_{\min} \text{ und } x_{\max}\} \quad , \\ I_y &= \{t \mid \vec{r}(t) \text{ liegt zwischen } y_{\min} \text{ und } y_{\max}\} \quad , \\ I_z &= \{t \mid \vec{r}(t) \text{ liegt zwischen } z_{\min} \text{ und } z_{\max}\} \quad , \end{aligned}$$

bildet deren Schnitt, und schneidet das Ergebnis mit dem Intervall I, dem Definitionsbereich des Strahls. Ist das resultierende Intervall nicht leer, bedeutet das, dass ein Teilstück des Strahls innerhalb der Bounding-Box verläuft, mit anderen Worten der Strahl die Box trifft. Ist das Intervall dagegen leer, haben Strahl und Bounding-Box keine gemeinsamen Punkte, das heisst der Strahl verfehlt die Box.

Der Pseudocode verdeutlicht die geringe Komplexität des Algorithmus.

```
bool RayBoxIntersection(ray, box)
  Interval inside = ray.range
  for i = x, y, z
    inside = Intersection(inside, (box.range[i]-ray.origin[i])/ray.direction[i])
    if inside is empty
      return false
  return true
```

Die hinter dem Algorithmus stehende Idee versteht man am besten, indem man ein Beispiel betrachtet (siehe Abbildung 4).

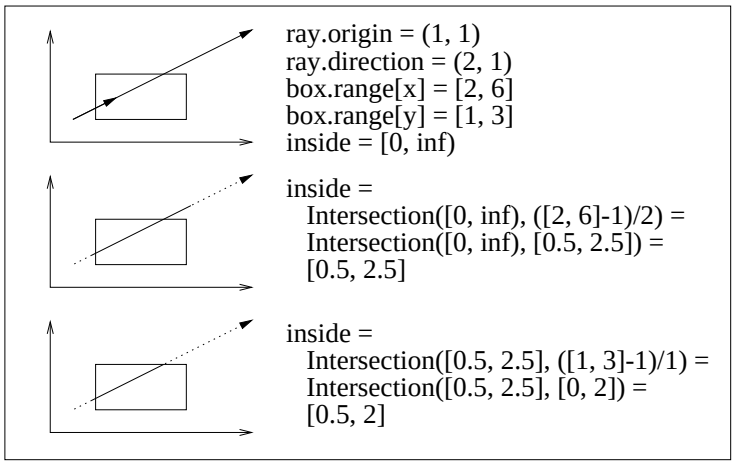

Abbildung 4: Schnitt-Test durch Schneiden von Intervallen

Eine experimentelle Bewertung dieses Verfahrens ist in Kapitel 8 zu finden.

4 Hierarchische Bounding-Volumes

Hierarchische Bounding-Volumes sind Bäume von Bounding-Volumes (siehe Abbildung 5). Bounding-Volumes an inneren Knoten umgeben die Bounding-Volumes ihrer Kinder. Bounding-Volumes an Blättern umgeben jeweils ein Objekt. Trifft ein Strahl ein Bounding-Volume, sind Schnitte mit Objekten im Unterbaum des Bounding-Volumes möglich. Verfehlt ein Strahl ein Bounding-Volume, so verfehlt er mit Sicherheit auch alle Objekte im entsprechenden Unterbaum. Auf diese Weise kann man durch einen einzigen Schnitt-Test eine ganze Reihe von Objekten aus der Liste möglicher Kandidaten für einen Treffer entfernen.

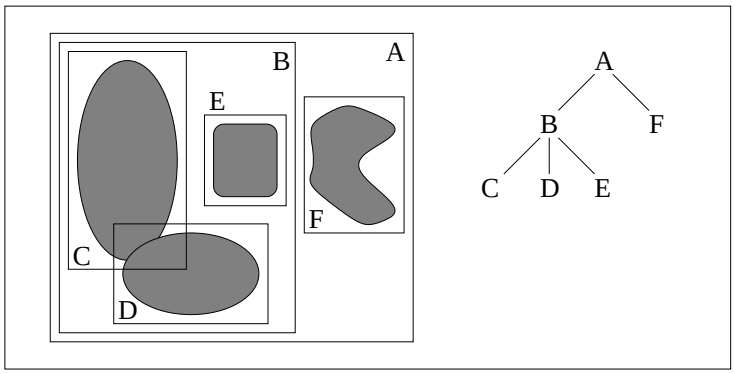

Abbildung 5: Hierarchische Bounding-Volumes

Ein einfacher Algorithmus zum Erzeugen hierarchischer Bounding-Volumes ist der folgen-

de. Man umgibt jedes Objekt durch ein Bounding-Volume, und legt eine Liste all dieser Bounding-Volumes an. Die Liste wird nun entlang einer Achse des Koordinatensystems sortiert und danach halbiert. Jede Hälfte umgibt man durch ein neues Bounding-Volume. Das Verfahren wird nun rekursiv fortgesetzt, wobei die Koordinatenachsen beim Sortieren zyklisch durchgewechselt werden (siehe Abbildung 6).

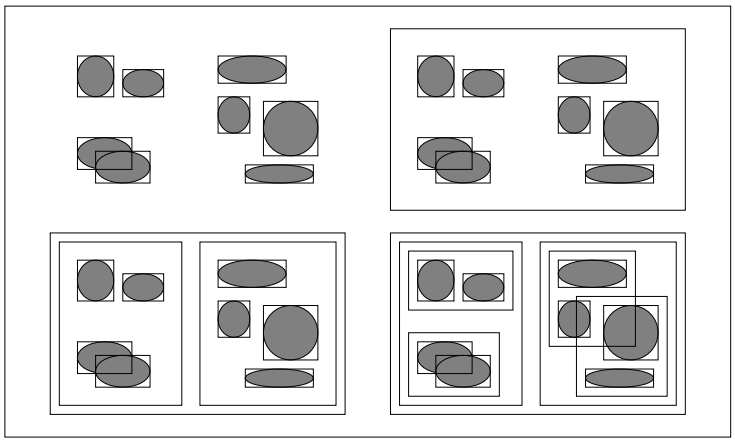

Abbildung 6: Erzeugen hierarchischer Bounding-Volumes

Der Pseudocode präzisiert den oben beschriebenen Algorithmus.

```
BoundingVolume BuildHierarchy(bvList, start, end, axis)
  if end == start
    return bvList[start]
  BoundingVolume parent
  for i = start...end
    expand parent to enclose bvList[i]
  sort bvList[start...end] along axis
  axis = next axis
  AddChild(parent, BuildHierarchy(bvList, start, (start+end)/2, axis))
  AddChild(parent, BuildHierarchy(bvList, (start+end)/2+1, end, axis))
  return parent
```

Hierarchische Bounding-Volumes wirken sich vor allem dann positiv aus, wenn Szenen sehr viele Objekte enthalten, deren Bounding-Volumes sich kaum überschneiden. Viele Strahlen verfehlen dann Bounding-Volumes nahe der Wurzel des Baumes, dass heisst durch wenige Schnitt-Tests können sehr viele Objekte vernachlässigt werden.

5 Durchlaufen hierarchischer Bounding-Volumes

Eine naheliegende Möglichkeit, Schnitt-Tests bei Verwendung hierarchischer Bounding-Volumes durchzuführen, ist das rekursive Durchlaufen des Baumes in Depth-First-Order. Für jeden Knoten ist dabei eine Liste aller seiner Kinder zu speichern (siehe Abbildung 7). Trifft ein Strahl ein Bounding-Volume an einem inneren Knoten, müssen Schnitt-Tests mit allen Kindern des Knotens durchgeführt werden. Befindet sich das Bounding-Volume an einem Blatt, ist ein Schnitt-Test mit dem entsprechenden Objekt durchzuführen. Verfehlt der Strahl jedoch das Bounding-Volume, kann der gesamte Unterbaum übersprungen werden.

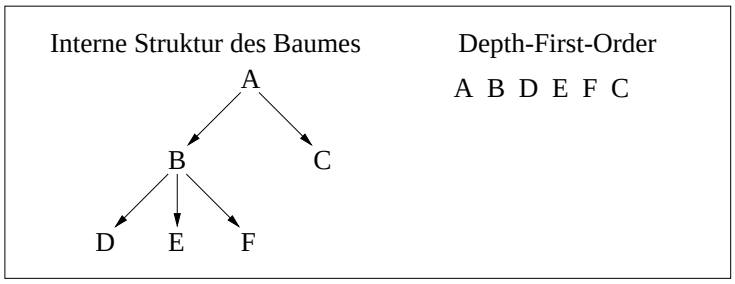

Abbildung 7: Rekursives Durchlaufen des Baumes in Depth-First-Order

Der Nachteil dieser Methode ist der erhebliche Zeitverlust, der durch die vielen rekursiven Funktionsaufrufe entsteht. Es ist allerdings nicht schwierig, einen ähnlichen, iterativen Algorithmus zu konstruieren. Für jeden Knoten muss das ganz links stehende Kind, der rechte Geschwisterknoten und der Elternknoten gespeichert werden (siehe Abbildung 8). Trifft ein Strahl ein Bounding-Volume eines inneren Knotens, geht man zum ganz links stehenden Kind. Verfehlt ein Strahl ein Bounding-Volume, geht man zum rechten Geschwisterknoten, falls ein solcher existiert, andernfalls zum Elternknoten. Trifft ein Strahl ein Bounding-Volume an einem Blatt, führt man einen Schnitt-Test mit dem entsprechenden Objekt durch und geht dann ebenfalls zum rechten Geschwisterknoten beziehungsweise zum Elternknoten.

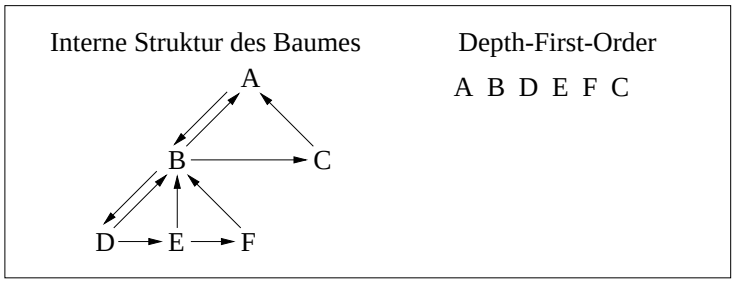

Abbildung 8: Iteratives Durchlaufen des Baumes in Depth-First-Order

Der Pseudocode zeigt, dass es sich um einen rein iterativen Algorithmus handelt.

```
TreeTraversal(ray, bvNode)
  while(true)
    if RayBoxIntersection(ray, bvNode→box)
      if bvNode→object != NULL
        if RayObjectIntersection(ray, bvNode→object)
          handle intersection
      else
        bvNode = bvNode→leftChild
        continue
    while(true)
      if bvNode→rightSibling != NULL
        bvNode = bvNode→rightSibling
        break
      bvNode = bvNode→parent
      if bvNode == NULL
        return
```

Dieses Verfahren kann jedoch noch weiter verbessert werden. Die Berechnung des nächsten Knotens beim Überspringen von Unterbäumen ist überflüssig, da, abhängig von der gegenwärtigen Position im Baum, das Ergebnis bereits feststeht. Man ersetzt also rechten Geschwisterknoten und Elternknoten durch einen einzigen „Überspringknoten“ (siehe Abbildung 9).

Der Pseudocode zeigt, dass man einige zeitraubende Schleifen beziehungsweise Verzweigungen einspart.

```
TreeTraversal(ray, bvNode)
  while(bvNode != NULL)
    if RayBoxIntersection(ray, bvNode→box)
      if bvNode→object != NULL
        if RayObjectIntersection(ray, bvNode→object)
          handle intersection
        bvNode = bvNode→skipNode
      else
        bvNode = bvNode→leftChild
    else
      bvNode = bvNode→skipNode
  return
```

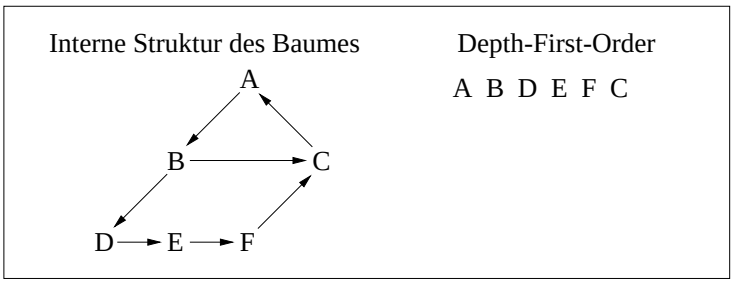

Abbildung 9: Iteratives Durchlaufen mit „Überspringknoten“

Speichert man den Baum entsprechend dem Depth-First-Pfad in einem Array (siehe Abbildung 10), erreicht man eine bessere Zusammenhängigkeit im Speicher. Dies führt zu kürzeren Zugriffszeiten.

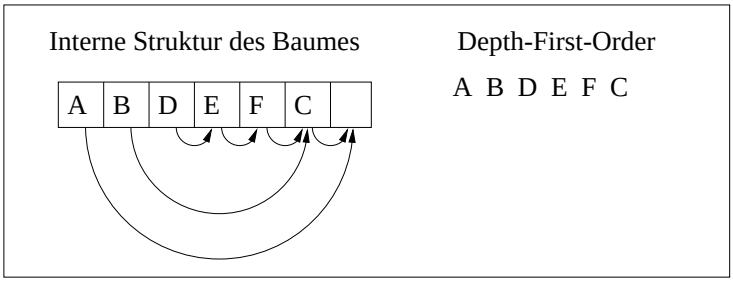

Abbildung 10: Speichern des Baumes in einem Array

Der Pseudocode ändert sich dadurch kaum.

```
TreeTraversal(ray, bvNode)
  stopNode = bvNode→skipNode
  while(bvNode < stopNode)
    if RayBoxIntersection(ray, bvNode→box)
      if bvNode→object != NULL
        if RayObjectIntersection(ray, bvNode→object)
          handle intersection
        bvNode = bvNode→skipNode
      else
        bvNode++
    else
      bvNode = bvNode→skipNode
  return
```

Kapitel 8 enthält eine experimentelle Bewertung der verschiedenen Durchlaufstrategien.

6 Intersection-Rays und Shadow-Rays

Intersection-Rays und Shadow-Rays unterscheiden sich in der Menge der von ihnen geforderten Informationen gewaltig.

Bei Intersection-Rays ist es notwendig, das zuerst getroffene Objekt zu ermitteln. Es muss daher immer die gesamte Hierarchie durchlaufen werden. Für dieses Objekt sind dann im Normalfall der genaue Schnittpunkt, die Normale und die Texturkoordinaten zu berechnen.

Bei Shadow-Rays ist es ausreichend, festzustellen, ob irgendein Objekt getroffen wird. Das Durchlaufen der Hierarchie kann daher bei einem Treffer sofort abgebrochen werden. Weitere Berechnungen, wie zum Beispiel der genaue Schnittpunkt, sind ohnehin überflüssig.

Das Verhältnis von Shadow-Rays zu Intersection-Rays hängt stark von der Anzahl der Lichtquellen in der Szene ab. In der Regel ist jedoch die Anzahl der Shadow-Rays um ein Vielfaches grösser als die Anzahl der Intersection-Rays. Häufig machen Shadow-Rays sogar mehr als 90% aller Strahlen aus. Im Sinne der Effizienz ist es daher sinnvoll, spezielle Funktionen für Shadow-Rays zu verwenden, die Informationen, die nur von Intersection-Rays benötigt werden, gar nicht berechnen.

Je mehr Lichtquellen eine Szene enthält, desto mehr Shadow-Rays müssen ausgesendet werden. Daher ist der Effizienzgewinn bei diesem Vorgehen stark von den verwendeten Szenen abhängig. Eine experimentelle Bewertung ist in Kapitel 8 zu finden.

7 Shadow-Caches

Ein Shadow-Cache ist ein Speicherbereich, in dem ein Verweis auf das zuletzt von einem Shadow-Ray getroffene Objekt abgelegt wird. Beim Aussenden des nächsten Shadow-Rays wird zuerst ein Schnitt-Test mit dem im Cache gespeicherten Objekt durchgeführt, in der Hoffnung, dass dieses erneut getroffen wird.

Es ist klar, dass Shadow-Caches sich nur dann vorteilhaft auswirken, wenn aufeinander folgende Shadow-Rays ähnlich sind, also mit grosser Wahrscheinlichkeit das selbe Objekt treffen. Bei zeilenweiser Berechnung eines Bildes, Verwendung von nur einer Lichtquelle und Verzicht auf reflektierte und gebrochene Strahlen sind die Voraussetzungen dafür günstig.

Befinden sich mehrere Lichtquellen in der Szene, sind aufeinander folgende Shadow-Rays in der Regel nicht mehr ähnlich. Man löst das Problem, indem man für jede Lichtquelle einen eigenen Shadow-Cache verwendet.

In gleicher Weise verhält es sich bei reflektierten und gebrochenen Strahlen. Die Punkte,

von denen aufeinander folgende Shadow-Rays ausgesendet werden, unterscheiden sich, und damit auch die Strahlen selbst. Auch hier behilft man sich durch die Verwendung eines extra Shadow-Caches pro reflektiertem beziehungsweise gebrochenem Strahl.

Eine weitere Voraussetzung für den effizienten Einsatz von Shadow-Caches sind Objekte von ausreichender Grösse. Selbst wenn aufeinander folgende Shadow-Rays ähnlich sind, sind Shadow-Caches wirkungslos, wenn die Objekte in der Szene so klein sind, dass sie immer nur von einem Strahl getroffen werden.

Darüber hinaus ist es erforderlich, dass die meiste Zeit Objekte in den Shadow-Caches gespeichert sind, also der Grossteil der Shadow-Rays Objekte trifft.

Es dürfte klar sein, dass die durch Shadow-Caches erreichte Zeitersparnis stark von den verwendeten Szenen und Modellen abhängt. Kapitel 8 enthält eine experimentelle Bewertung von Shadow-Caches.

8 Experimentelle Überprüfung der Effizienz der vorgestellten Verfahren

Die im folgenden präsentierten Ergebnisse entstammen [Smits98]. Bei allen Experimenten wurden 1.000.000 Strahlen berechnet, zufällig generiert durch zwei Punkte innerhalb einer Bounding-Box, 20% grösser als die Bounding-Box der gesamten Szene. Die ersten beiden Szenen enthalten wirkliche Modelle, ein Theater, bestehend aus 46.502 Polygonen, und ein wissenschaftliches Labor, bestehend aus 4.045 Polygonen. Die restlichen Szenen bestehen aus zufällig angeordneten Einheitsdreiecken, wobei die Zahl die Anzahl der Dreiecke angibt. Klein, mittel und gross beziehen sich auf die Grösse der gesamten Szene. Klein entspricht einem Würfel mit Kantenlänge 20, mittel einem Würfel mit Kantenlänge 100, gross einem Würfel mit Kantenlänge 200. Die Zeiten in der Tabelle sind in Sekunden angegeben. Es wurden ausschliesslich Strahlen berechnet, das heisst weitere beim Ray-Tracing anfallende Berechnungen sind in die angegebenen Zeiten nicht eingegangen.

	1	2	3	4	5	6	7	8
Theater (46.502)	64	36	30	21	22	11	10	6
Labor (4.045)	79	41	32	22	20	12	12	7
10.000 klein	415	223	191	142	110	48	50	27
10.000 mittel	392	185	154	103	81	77	79	65
10.000 gross	381	179	152	104	82	79	77	69
100.000 klein	995	620	550	449	351	62	63	33
100.000 mittel	932	473	424	324	230	146	148	89
100.000 gross	1024	508	442	332	240	210	212	156
300.000 mittel	1093	597	536	421	312	120	121	64

1. Schnitt-Tests mit Bounding-Boxen durch Berechnung der Schnittpunkte, rekursives Durchlaufen der Hierarchie, eine Funktion für Intersection-Rays und Shadow-Rays, keine Shadow-Caches.
2. Schnitt-Tests mit Bounding-Boxen durch Schneiden von Intervallen (siehe Kapitel 3).
3. Iteratives Durchlaufen der Hierarchie (siehe Kapitel 5).
4. Iteratives Durchlaufen der Hierarchie, Verwendung von „Überspringknoten“ (siehe Kapitel 5).
5. Iteratives Durchlaufen der Hierarchie, Speichern des Baumes in einem Array (siehe Kapitel 5).
6. Verwendung einer speziellen Funktion zur Berechnung von Shadow-Rays (siehe Kapitel 6).
7. Verwendung von Shadow-Caches (siehe Kapitel 7).
8. Verwendung von Shadow-Caches, zweifaches Aussenden von 500.000 verschiedenen Strahlen (siehe Kapitel 7).

9 Schlussbemerkungen

Die in diesem Vortrag präsentierten Massnahmen stellen nur einen Teil der zur Verfügung stehenden Möglichkeiten zur Steigerung der Effizienz beim Ray-Tracing dar. Verbesserungen in anderen Bereichen oder auf anderem Wege sind durchaus denkbar. Auch sollte man sich stets vor Augen halten, dass der Zeitgewinn in vielen Fällen stark von den verwendeten Szenen und Modellen abhängt. Insbesondere der Einsatz von Shadow-Caches sollte gut überlegt werden. Dennoch haben sich die hier vorgestellten Konzepte in den meisten Fällen als sehr vorteilhaft erwiesen.

Der Vortrag basiert zu grossen Teilen auf [Smits98]. Vieles lässt sich aber auch in ähnlicher Form in [Foley97] finden.

Literatur

[Smits98] Smits, Brian (1998). *Efficiency Issues for Ray Tracing.* Journal of Graphics Tools.

[Foley97] Foley; van Dam; Feiner; Hughes (1997). *Computer Graphics: Principles and Practice.* Addison Wesley.