İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GRAFİK KARTI ÜZERİNDE PARALEL HIZLANDIRILMIŞ IŞIN İZLEME

YÜKSEK LİSANS TEZİ

Mustafa Alper ÇOLAK

Anabilim Dalı: Bilgisayar Mühendisliği

Programı: Bilgisayar Mühendisliği

Tez Danışmanı: Prof. Dr. Nadia ERDOĞAN

HAZİRAN 2010

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GRAFİK KARTI ÜZERİNDE PARALEL HIZLANDIRILMIŞ IŞIN İZLEME

YÜKSEK LİSANS TEZİ

Mustafa Alper ÇOLAK
(504071526)

Tezin Enstitüye Verildiği Tarih : 07 Mayıs 2010

Tezin Savunulduğu Tarih : 04 Haziran 2010

Tez Danışmanı : Prof. Dr. Nadia ERDOĞAN (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. M. Bülent ÖRENCİK (TÜBİTAK)
Yrd. Doç. Dr. D. Turgay ALTILAR (İTÜ)

HAZİRAN 2010

ÖNSÖZ

Öncelikle, hayatım boyunca benden maddi ve manevi desteklerini esirgemeyen aileme sonsuz şükranlarımı sunarım. Bu onlar için yeterli olmasa da…

Bu tez çalışması süresince büyük bir sabır ve iyi niyetle bana yol gösteren değerli hocam Prof. Dr. Nadia ERDOĞAN'a teşekkür ederim.

Son olarak bünyesinde bulunduğum TÜBİTAK UEKAE Bilişim Teknolojileri Enstitüsü'ne, beraber çalışma fırsatı bulduğum, beni sürekli teşvik eden ve destekleyen Dr. Deniz BÖLÜKBAŞ'a ve UEKAE Müdür Yardımcısı Prof. Dr. Bülent ÖRENCİK'e teşekkür ederim.

Mayıs 2010

Mustafa Alper Çolak

Bilgisayar Mühendisi

İÇİNDEKİLER

Sayfa

KISALTMALAR

CUDA : Compute Unified Device Architecture
NURBS : Non-uniform Rational B-Spline
MİB : Merkezi İşlem Birimi
GİB : Grafik İşlem Birimi
BDT : Bilgisayar Destekli Tasarım

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

GRAFİK KARTI ÜZERİNDE PARALEL HIZLANDIRILMIŞ IŞIN İZLEME

ÖZET

Gerçek dünyada pek çok fiziksel olay meydana gelir. Bilgisayar teknolojisinin gelişimiyle beraber bilgisayarlar, fiziksel olayların sonuçlarının öngörülmesi ve bu olayların birebir benzetiminin yapılması amacıyla da kullanılmaya başlanmıştır.

Bu fiziksel olayların başında ışık etkileri gelmektedir. Bunun dışında yüksek frekans elektromanyetik ve akustik dalgalar da ışığa benzer özellikler taşır. Bu dalgaların ortak özellikleri, basit ortamda doğrusal olarak hareket etmeleri, uzaydaki cisimlere çarpmaları, çarptıkları cismin özelliklerine bağlı olarak yön ve nitelik değiştirip bu cismin özelliklerini ulaştıkları yerlere taşımalarıdır.

Işın izleme, bu tür dalgaların davranışlarının bilgisayarda modellenmesi için kullanılan bir yöntemdir. Işınlar bilgisayarda, ışın merkezinin koordinatlarını belirten bir vektör ile ışının yönünü belirten bir birim vektörle ifade edilir. Bu çalışmada, ışın izleme işlemi, ışık benzetimi için kullanılmış ve 2 boyutlu fotogerçekçi görüntüler elde edilmesi amaçlanmıştır.

Işın izleme işlemi için öncelikle 3 boyutlu cisimler modellenerek bilgisayara aktarılmalıdır. Cisimlerin bilgisayarda modellenmesi için çeşitli yöntemler vardır. Bunlardan bir tanesi, cisimlerin üçgenlerle modellenmesi yöntemidir. Üçgen modelleme, özellikle eğri cisimleri belirli bir yaklaşıklıkta modelleyebilse de, hesaplamayı kolaylaştırdığı, hızlandırdığı ve karmaşık ya da basit tüm cisimler için uygulanabilir hale getirdiğinden, sıklıkla kullanılan bir yöntemdir. Bu çalışmada cisimlerin üçgen modelleri kullanılmıştır.

Cismin üçgen modelinin bilgisayara aktarılmasının ardından, ışınlar oluşturulmalı ve bu ışınların cismi oluşturan üçgenlerin hangilerine hangi noktalarda çarptığı tespit edilmelidir. Bu tespit işlemi, ışın-üçgen kesişim algoritmalarıyla yapılır. Bu işlemin yürütülmesi için her ışın-üçgen ikilisinin kontrol edilmesi gerekmekte, bu da ışın izleme işlemini oldukça maliyetli hale getirmektedir.

Işın izleme işlemini daha verimli hale getirerek hızlandırmak için uzay bölmeleme algoritmaları geliştirilmiştir. Bu algoritmalar kullanılarak her ışın için cismin sadece o ışını ilgilendirebilecek kısımlarının kontrol edilmesi sağlanır. Bu da ışın izleme işleminin verimini ve dolayısıyla hızını artırmaktadır. Bu çalışmada, “Kd-Tree” uzay bölmeleme algoritması kullanılmıştır.

Işın izleme işlemini hızlandırıcı bir diğer yöntem, işlemin paralel olarak yürütülmesidir. Bu amaçla zaman içinde vektör makineler veya bilgisayar kümeleri kullanılmış, ancak bu kullanım maliyeti nedeniyle yaygın hale gelememiştir. NVIDIA firmasına ait CUDA teknolojisi ise, kişisel bilgisayarlarda bulanan grafik kartlarının genel amaçlı kullanılmasına olanak sağlamış ve bunun gibi muazzam derecede paralel işlemlerin kişisel bilgisayarlarda da yürütülebilmesine imkan

vermiştir. Bu çalışmada uzay bölmeleme algoritması kullanılarak verimli hale getirilen ışın izleme işleminin, CUDA kullanılarak grafik kartları üzerinde yürütülebilmesi amaçlanmıştır.

Son olarak bu çalışmada, grafik kartı üzerinde çalışan ışın izleme uygulamasının, MİB üzerinde çalışan sürümüyle ve ticari olarak kullanılan benzer uygulamalarla karşılaştırılarak yapılan bir başarım analizine yer verilmiştir.

PARALEL ACCELERATED RAY TRACING ON GRAPHICS CARDS

SUMMARY

In the real world, many physical events occur. By the development in computer technology, computers have been used for prediction and simulation of these physical events.

Some of these physical events are effects of light, high frequency electromagnetic and acoustic waves. The common properties of these waves are; moving linearly in simple environment, bouncing from objects in space, changing direction and attributes because of the physical properties of objects and carrying these properties along with themselves through their path.

Ray tracing is a method used for modeling the behavior of these waves on computer environment. Rays are symbolized in computer with an origin vector and a unit vector, which represents the direction of the ray. In this work, we use ray tracing in order to simulate light and generate 2D photorealistic images.

In order to execute ray tracing, firstly 3D objects have to be modeled on computer. There are numerous methods of modeling objects on computer. One of the most common methods is triangle meshing. Triangle mesh method can model 3D objects with an approximation. However, it greatly simplifies working on the objects and accelerates the process. In this work, we use triangle mesh modeling.

After the object is modeled on computer, rays are generated and tracked through their path. During this process, ray-triangle intersection algorithms are used. However, the fact that every ray-triangle pair has to be controlled makes this process very costly.

In order to reduce the cost, make the process more efficient and accelerate it, space partitioning algorithms are used. These algorithms makes the rays deal only with the part which they need to be deal, instead of all geometry. This process greatly increases the efficiency and speed. In this work, “Kd-Tree” space partitioning algorithm is used.

Another acceleration method for ray tracing is parallelism. In time, vector machines and clusters have been used for parallel processing. These systems are not very widely used because of their high cost. However, CUDA technology of NVIDIA Corporation allows us to use the parallel processing power of graphics cards, which are parts of personal computers, for general-purpose applications. In this work, we developed a ray tracer which is accelerated using space partitioning algorithms and parallelly executed on graphics cards using CUDA.

Finally in this work, we compared this CUDA ray tracer with the CPU version and other commercial ray tracers in the market and made a performance analysis.

1. GİRİŞ

Bu çalışmada, genel maksatlı hesaplama yapabilme yeteneğine sahip bir grafik kartı işlemcisi kullanılarak 3 boyutlu cisimler üzerinde ışın izleme yazılımı geliştirilmiştir. Işın izleme işleminin amacı, bir kaynaktan yola çıkan bir ışının cisim uzayını terk edene kadar takip edilmesi ve cisim üzerinde çarptığı noktaların bulunmasıdır.

Işın izleme, Fermat prensibine göre basit ortamda doğrusal olarak hareket ettiği varsayılan dalgaların benzetiminde kullanılan bir yöntem olup, en yoğun kullanım alanı ışığın modellenerek 2 boyutlu fotogerçekçi görüntülerin üretilmesidir. Bunun dışında yüksek frekans elektromanyetik ve akustik dalgalar da bu yöntemle modellenebilir.

Işın izleme, yüksek maliyetli bir işlem olduğundan özellikle son 15 yıl içerisinde bu işlemi daha verimli hale getirecek ve hızlandıracak yöntemler üzerinde çalışılmıştır. Bu çalışmalar sonucunda hızlı ışın-üçgen kesişim algoritmaları, ışın-kutu kesişim algoritmaları, uzay bölmeleme algoritmaları ile istatistik yaklaşımlar ortaya çıkmıştır. Bu çalışmada bahsi geçen algoritmalardan örnekler kullanılmıştır.

Işın izlemeyi hızlandırıcı bir diğer yöntem yazılımın işlemin paralel olarak yürütülmesidir. Işın izleme, yapısal olarak muazzam derecede paralelliğe uygun bir işlemdir. Muazzam derecede paralel veri işleme, uzun yıllar sadece süper bilgisayarlar ve bilgisayar kümeleri kullanılarak yapılabilecek bir işlem olarak kalmışken, 2007 yılının sonunda grafik kartı üreticisi NVIDIA firmasının yayınladığı CUDA mimarisi sayesinde kişisel bilgisayarlarda da yürütülebilir hale gelmiştir. CUDA, oldukça fazla sayıda ancak basit yapıda işlemcilerden oluşan NVIDIA üretimi grafik kartlarının genel maksatlı hesaplama amaçlı kullanılmasını sağlayan bir uygulama ara yüzüdür.

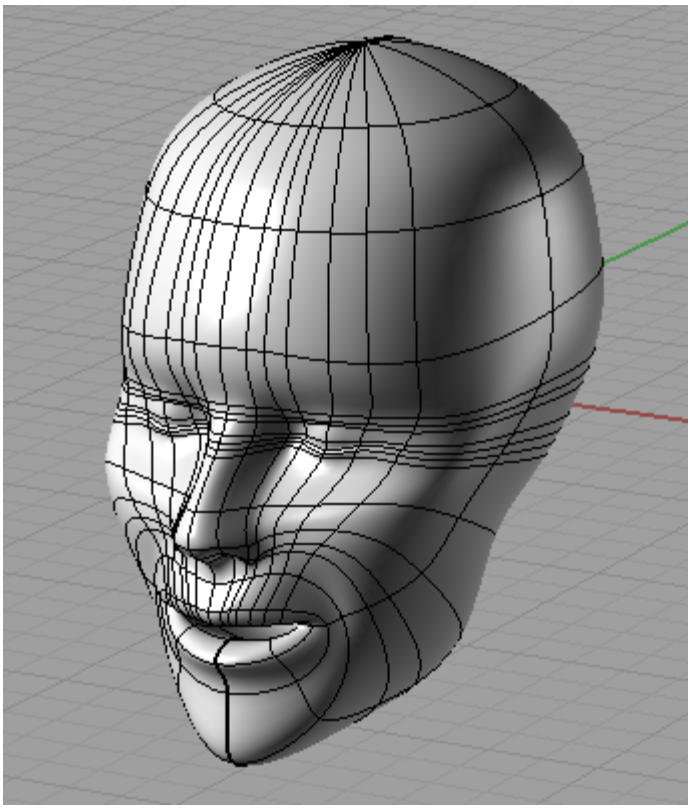
Bu çalışmada, Kd-Tree uzay bölmeleme algoritması kullanılarak verimliliği artırılmış, CUDA kullanılarak grafik kartı üzerinde çalışabilen bir ışın izleme yazılımı geliştirilmiş ve geliştirilen bu yazılımla ilgili temel prensipler sunulmuştur.

2. BİLGİSAYARDA GRAFİK MODELLER

Matematiksel olarak modellenebilen her kavram, bilgisayarda modellenebilir. Gerçek dünyadaki 3 boyutlu cisimlerin de bilgisayarda modellenebilmesi için bu tür matematiksel modellere uyarlanması gerekmektedir. Bilgisayarda 3 boyutlu modelleme yapmak için bir kısmı deneysel olmak üzere pek çok yöntem bulunmakla birlikte, en çok kullanılan yöntemler NURBS ve poligon modelleme yöntemleridir.

2.1 NURBS Modelleme

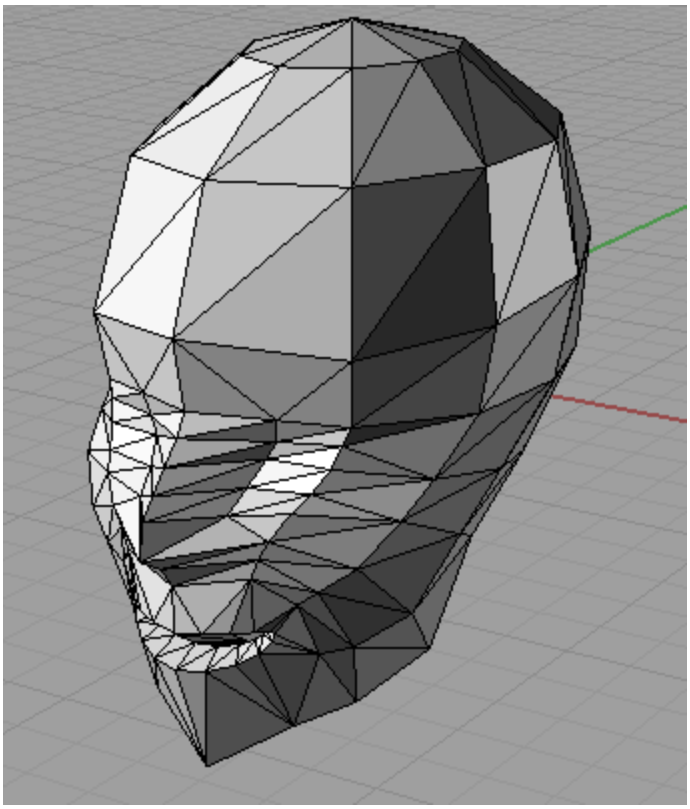
İngilizce "Non-uniform Rational B-spline" ifadesinin kısaltmasıdır. NURBS modellemede yüzeyler, polinom eğrilerinin belirli noktalarda birleştirilmesi ile oluşturulurlar. NURBS kullanılarak özellikle eğri yüzeyler yüksek doğrulukta modellenebilir. Bununla birlikte NURBS, ışın izleme gibi işlemler için ciddi miktarda hesaplama yükü oluşturmaktadır.



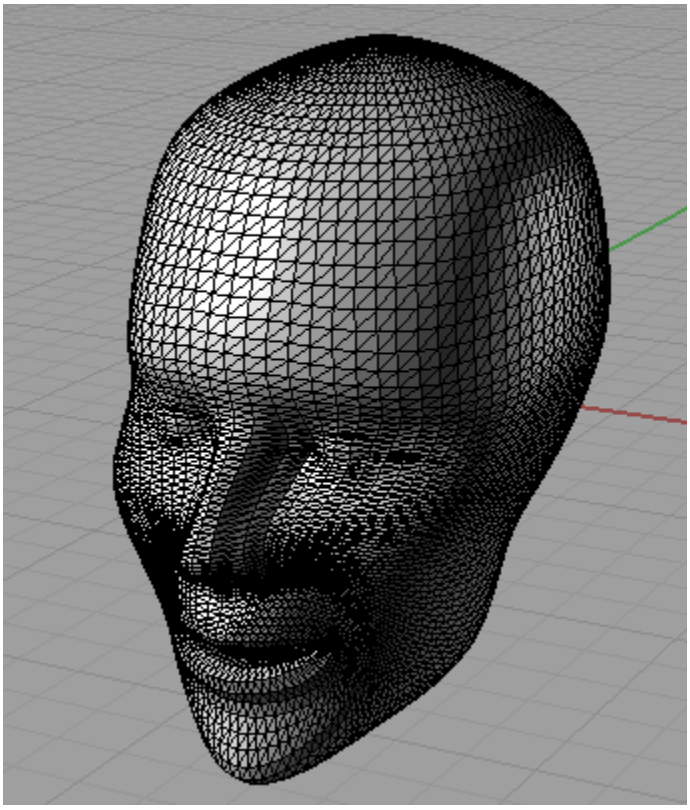
Şekil 2.1: Bir insan başının NURBS modeli

2.2 Çokgen Modelleme

Yüzeylerin çokgenlerle (üçgen, dörtgen vs.) modellenmesidir. Bu yöntemde düzlemler basitçe modellenebilirken, eğri yüzeyler belirli bir yaklaşımla modellenir. Dolayısıyla eğri yüzeylerin yüksek doğrulukta modellenmesi, ne kadar çok çokgen kullanıldığına bağlıdır. Eğri yüzeyler modellenirken, modelin kullanılacağı uygulama için yeterli miktarda çokgen kullanılmalıdır. Çokgen sayısı arttıkça hassasiyet artarken, kullanılacak uygulamadaki hesaplama yükü artmaktadır. Bununla birlikte modelleme yapılırken aynı sayıda köşeye sahip çokgenlerin (örneğin sadece üçgenlerin) kullanılması, model üzerinde yapılacak hesaplama için önemli ölçüde kolaylık sağlamaktadır.



Şekil 2.2: Bir insan başının az sayıda üçgen ile modellenmesi



Şekil 2.3: Bir insan başının çok sayıda üçgen ile modellenmesi

Bu çalışmada 3 boyutlu modelleme için çokgen modelleme yöntemi kullanılmıştır. Hesap yükünü azaltmak adına kullanılan çokgenler üçgenler ile sınırlandırılmıştır. Dosya formatı olarak "RAW" kullanılmıştır. Bu formatta üçgenler, arka arkaya sıralanan üçgen köşelerinin koordinatlarını belirten 9 adet kayan noktalı sayı ile ifade edilir. Bu noktaların dışında ışın izleme işleminde üçgen normali de kullanılmaktadır. Üçgen normali, üçgenin bulunduğu düzleme dik olan bir birim vektörle ifade edilir. Üçgen normalinin dosya formatı içerisinde belirtilmesine gerek olmayıp, köşe koordinatlarından sağ el kuralına göre hesaplanabilir.

```
10.000000 10.000000 0.000000 -10.000000 10.000000 0.000000 -10.000000 -10.000000 0.000000
10.000000 10.000000 0.000000 -10.000000 -10.000000 0.000000 10.000000 -10.000000 0.000000
```

Şekil 2.4: X-Y düzleminin merkezinde oluşturulan 20 metre kenarlı bir karenin RAW formatında modellenmesi

3. IŞIN İZLEME

3.1 Amaç

Işın izleme işlemindeki amaç, belirli bir merkez ve yöne sahip doğrusal bir ışının 3 boyutlu uzayda herhangi bir cisme çarpıp çarpmadığını, çarptıysa çarpma noktasını ve çarptıktan sonra ışının hangi yönde yoluna devam ettiğini ışın cisim uzayından çıkana kadar takip ederek belirlemektir. Işınlar bilgisayarda ışının başladığı merkez noktasının koordinatları (3 adet kayan noktalı sayı) ve ışının yönünü ifade eden bir birim vektörle temsil edilir.

3.2 Kullanım Alanları

Işın izleme yöntemi, düzlemsel dalga yaklaşımına sahip bütün fiziksel olayların bilgisayardaki benzetiminde kullanılabilir. Dalganın düzlemsel olarak temsil edilip edilemeyeceği, dalganın sahip olduğu dalga boyunun cisme göre büyüklüğüne ve dalganın uzayda kat ettiği mesafeye göre belirlenir. Bu yaklaşıma göre başta ışık olmak üzere, yüksek frekans elektromanyetik dalgalar ve yüksek frekans akustik dalgalar da doğrusal dalga olarak temsil edilebilirler. Bütün bu dalga çeşitlerinin aralarındaki fark, cisme çarptıklarında oluşturdukları fiziksel etkiden ibarettir. Elektromanyetik dalgalar çarptıkları yüzey üzerinde akım oluştururken, akustik dalgalar titreşime, ışık ise ışık-gölge etkilerine neden olur. Bu çalışmada gerek yapılacak hesapların daha basit olması, gerekse elde edilen sonuçların herkes tarafından anlaşılabilecek görüntülerden oluşması nedeniyle, ışık benzetimi tercih edilmiştir.

3.3 Işın Başlatma

Işın başlatma, belirli bir merkez ve yöne sahip ışının ilk çarptığı noktayı bulma işlemidir. Bu işlemde her piksel için bir ışın başlatılır. Başlatılan ışın görüntü

penceresinden geçerek cisme çarpar. Işının üçgenlerle modellenmiş cisme çarpıp çarpmadığı, ışın üçgen kesişim algoritmalarıyla belirlenir.

3.4 Işın Üçgen Kesişim Algoritması

Işın üçgen kesişim algoritmaları, 3 boyutlu uzayda bir ışının bir üçgeni kesip kesmediğini, kestiyse ne kadar yol kat edip hangi noktada kestiğini hesaplamak için kullanılır. Bu konudaki en hızlı ve bellek gereksinimi düşük yöntem 1997 yılında Thomas Möller tarafından yayınlanmıştır. Möller'in ışın üçgen kesişim algoritmasına göre koordinat ekseninin merkezi üçgenin bir köşesine ötelenir. Ardından koordinat ekseni, üçgen x-y düzlemine oturacak şekilde döndürülür. Ardından ışının x-y düzlemini hangi noktada kestiği bulunur ve bu noktanın üçgen sınırları içerisinde olup olmadığına bakılır. Son olarak yapılan döndürme ve öteleme işlemleri tersine çevrilerek asıl koordinat eksenindeki kesişim noktası hesaplanır. **(3.1)**

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{3.1}$$

Bu formülde t, u ve v ışının üçgeni kestiği noktanın barisentrik koordinatlarıdır. V değişkenleri üçgenin köşe koordinatları, O ışının merkez noktası, D ise ışının yönü olmak üzere:

$$\begin{aligned} E_1 &= V_1 - V_0 \\ E_2 &= V_2 - V_0 \\ T &= O - V_0 \\ P &= D \times E_2 \\ Q &= T \times E_1 \end{aligned}$$

Olarak ifade edilir [1].

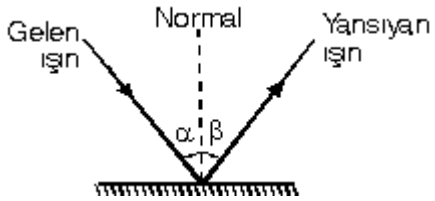
3.5 Gölgeleme

Gölgeleme işlemi, ışının çarptığı yüzeyin renklendirilmesine verilen addır. Cisim üzerinde bir yüzey, ışık kaynağından gelen ışının yüzey normali ile yaptığı açı, ışığın rengi, yüzeyin rengi ve yansıtıcılık özelliğine göre renklendirilir. Basit bir gölgeleme işlemi şu formüle göre yapılabilir: [2]

$$L_{dir}(\lambda) = L_{if}(\lambda)F_{dir}(\lambda)(N \cdot L) \quad \textbf{(3.2)}$$

3.6 Yansıma

Işın bir üçgene çarptıktan sonra, başka üçgenlere çarpıp çarpmadığı ışın cisim uzayından ayrılıncaya kadar ardışıl olarak kontrol edilir. Bunun için ışının çarptığı noktadan hangi yöne doğru yansıdığı hesaplanır. Bunun için fizikteki yansıma kuralları kullanılır. Bu kurallara göre ışın, düzlemsel bir yüzeye çarptıktan sonra yüzey normali ile geliş açısıyla aynı açıyı yaparak yansır.



Şekil 3.1: Işının düzlemden yansıması

Yansıma açısı, lineer cebirde şu şekilde bulunur:

$$D' = 2(D \cdot N)N - D \quad \textbf{(3.3)}$$

Bu formülde D' yansıma açısı, D geliş açısı, N ise yüzey normalini temsil eder.

3.7 Başarım Analizi

Işın izleme işlemi, temel olarak bir ışının yolu üzerinde çarptığı en yakın üçgeni bulmaktan ibarettir. Bu nedenle bu işlem için uygulanacak en basit yöntem her ışın için bütün üçgenleri kontrol etmektir. Her ışının çarpacağı tek bir üçgeni bulmak için bütün üçgenlerin kontrol edilmesi, oldukça yüksek bir hesaplama yükü getirmektedir. Örneğin, 20000 üçgen ile modellenmiş bir küreden 400x400 piksellik bir görüntü elde etmek, bu yöntemle yaklaşık 1 dakika sürmektedir. Bununla birlikte, bu işlemin mümkün olduğunca gerçek zamanlı olarak yapılması istenmektedir. Bu nedenle ışın izleme işlemiyle ilgili çeşitli hızlandırma yöntemleri geliştirilmiştir.

Bunların ilki, her ışın için kontrol edilmesi gereksiz olan üçgenlerin mümkün olduğunca elenerek her işlemin belirli bir alt uzaya odaklanmasıdır. Bir diğer yöntem ise, ışın izleme işleminin paralelleştirmesidir. Gerek bir ışın için ışın-üçgen kesişimi

algoritmasının birçok üçgen için yürütülmesi, gerekse aynı işlemlerin bütün ışınlar için tekrarlanması, bu işlemi paralelleştirmeye son derece uygun hale getirmektedir.

4. IŞIN İZLEMEYİ HIZLANDIRICI YÖNTEMLER

Daha önce de belirtildiği gibi ışın izleme, hesaplama yükü yüksek bir işlemdir. Hesaplama yükünün azaltılması için pek çok metot geliştirilmiştir. Işın-üçgen kesişimi algoritmaları önemli gelişim göstermiş, yeni nümerik yöntemler geliştirilmiş, veri yapıları oluşturulmuş, istatistik yöntemler geliştirilmiştir. Bu işlemlerin tamamı, ışın üçgen kesişimi işlemlerini hızlandırmıştır.

Fakat buradaki asıl sorun, işlemsel yükü azaltılmış da olsa, her ışın için birkaç tanesi hariç yürütülen ışın-üçgen kesişim algoritmalarının gereksiz oluşudur. Her ışının keseceği tek bir üçgen için cismi oluşturan bütün üçgenlerin kontrol edilmesi, verimsiz bir işlemdir. Bu işlemin verimli hale getirilmesi için, kontrol edilen gereksiz üçgen sayısının azaltılması gerekmektedir. Bu amaçla, uzay bölmeleme algoritmaları geliştirilmiştir.

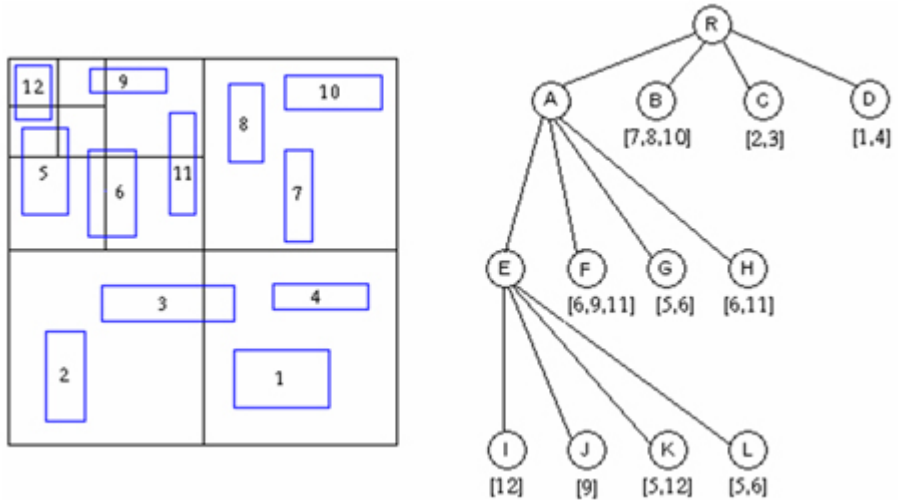
Işın izleme işlemindeki bir diğer husus, aynı işlemin pek çok ışın için tekrarlanmasıdır. Örneğin, 600x600 piksellik bir görüntü elde edebilmek için 120000 ışını takip etmek gerekmektedir. Bu da ışın izleme işlemini muazzam derecede paralelliğe uygun hale getirmektedir.

4.1 Uzay Bölmeleme Algoritmaları

Uzay bölmeleme algoritmaları, esas olarak en yakın komşu problemlerini çözmek üzere geliştirilmişlerdir. Bu algoritmalara göre, cismin içinde bulunduğu uzay farklı derinliklerde alt uzaylara bölünerek, gereksiz işlemlerin yapılması önlenir. Işın izleme işlemi de, ışının merkezine ışının yönü doğrultusundaki en yakın üçgeni bulma işleminden ibaret olduğundan, en yakın komşu problemine indirgenebilir. Dolayısıyla uzay bölmeleme algoritmalarının kullanılması, ışın izleme işleminin verimliliğini önemli ölçüde artırmaktadır. Literatürde en çok kullanılan uzay bölmeleme algoritmaları; “QuadTree”, “Octere”, “BSP-Tree” ve “Kd-Tree” dir.

4.1.1 QuadTree

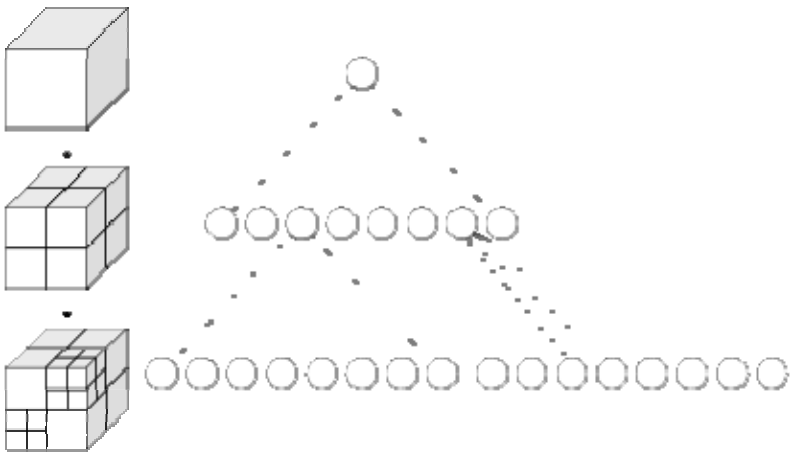
QuadTree yapısı, ilk olarak 1974 yılında R. Finkel ve J.L. Bentley tarafından adlandırılmıştır. QuadTree, Latince "Quad" yani "dört" ile İngilizce "Tree" yani ağaç kelimelerinin birleşiminden oluşup, 4 alt uzaylı yapıyı ifade eder. QuadTree yapısında uzay, her seferinde 4 eşit alt uzaya bölünür. Her alt uzayın sahip olabileceği maksimum sayıda bir üçgen sayısı bulunur. Alt uzayın içerdiği üçgen sayısı, alt uzayın sahip olabileceği maksimum üçgen sayısından büyükse, uzay bir sonraki eksende tekrar dörde bölünür. Her alt uzay, ağaçta bir düğüm tarafından temsil edilir. Her alt uzayın sahip olduğu 4 alt uzay ise ağaçta bu alt uzayın çocukları olarak ifade edilir. Ağacın yapraklarında ise üçgenler bulunur. [3]



Şekil 4.1: QuadTree

4.1.2 Octree

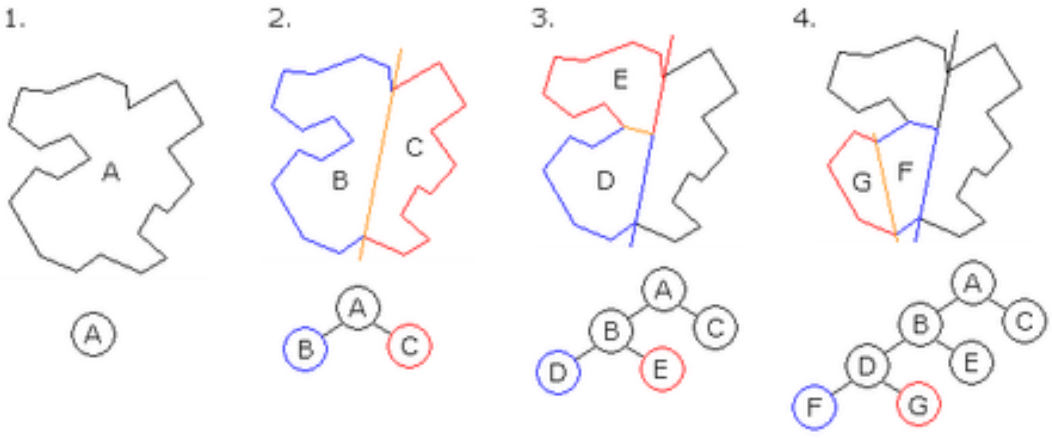
Octree yapısı, 1990 yılında Gervautz ve Purgathofer tarafından öne sürülmüştür. Octree yapısı "Octa" yani "Sekiz" ve "Tree" yani "Ağaç" ifadelerinin birleşmesinden oluşup, sekiz alt uzaylı yapıyı temsil eder. Octree, esasen QuadTree'nin 3 boyutlu uzaya uyarlanmış halidir. Octree yapısında, QuadTree yapısından farklı olarak uzay, her seferinde 8 eşit alt uzaya bölünür. QuadTree yapısında olduğu gibi, her alt uzayın sahip olabileceği maksimum sayıda bir üçgen sayısı bulunur. Alt uzayın içerdiği üçgen sayısı, alt uzayın sahip olabileceği maksimum üçgen sayısından büyükse, uzay bir sonraki eksende tekrar sekize bölünür. Her alt uzay, ağaçta bir düğüm tarafından temsil edilir. Her alt uzayın sahip olduğu 4 alt uzay ise ağaçta bu alt uzayın çocukları olarak ifade edilir. Ağacın yapraklarında ise üçgenler bulunur. [4]



Şekil 4.2: Octree

4.1.3 BSP Tree

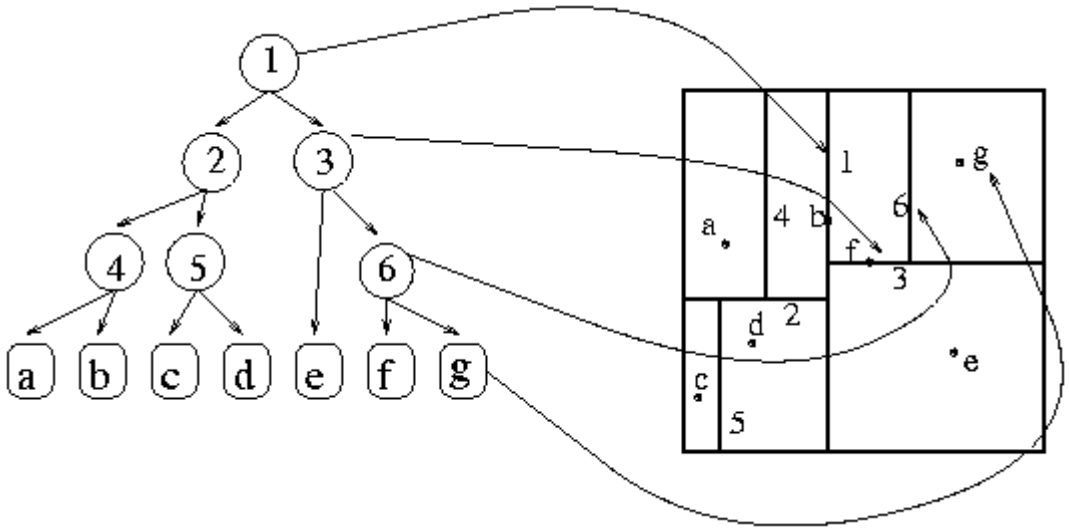
"BSP" ifadesi "Binary Space Partitioning" in kısaltması olup "İkili Uzay Bölmelemesi" anlamına gelir. Adından da anlaşılacağı üzere BSP Tree, uzayı her kademede iki alt uzaya bölen bir veri yapısıdır. BSP Tree yapısında uzay, QuadTree ve Octree'de olduğu gibi dikdörtgen prizma şeklinde temsil edilmez. Uzayı bölen düzlem için de herhangi bir geometrik kısıtlama yoktur. Buna karşın, uzayı bölen düzlem, arama işleminde en yüksek verimliliği sağlayacak şekilde seçilir. Daha önce bahsettiğimiz Octree ve QuadTree'de olduğu gibi üçgenler yapraklarda bulunur.[5] BSP Tree, "Doom" gibi ilk 3 boyutlu bilgisayar oyunlarında kullanılmıştır. Işın-üçgen kesişimi işlemlerini oldukça verimli bir hale getirse de, uzayı alt uzaylara bölen düzlemlerin belirli kısıtlara sahip olmaması, arama esnasında yapraklara gidene kadar yüksek bir maliyete neden olmaktadır.



Şekil 4.3: BSP Tree'nin oluşturulması

4.1.4 Kd-Tree

"Kd-Tree", "K-dimensional tree" ifadesinin kısaltması olup, "K boyutlu ağaç" anlamına gelir. Kd-Tree, BSP Tree'nin özelleştirilmiş bir halidir. Bu yapıda uzay, Octree ve QuadTree'de olduğu gibi koordinat eksenlerine paralel bir dikdörtgen prizma olarak düşünülür. Uzay, belirli bir maksimum üçgen sayısına ya da maksimum derinliğe ulaşana kadar yine dikdörtgen prizma şeklindeki alt uzaylara bölünür. Octree ve QuadTree'deki uygulamaya karşın, Kd-Tree'de uzay, bir kenarının orta noktasından ikiye bölünmek zorunda değildir. Yani bölünen alt uzaylar, geometrik olarak birbirlerine eşit değildir. Bunun yerine Kd-Tree'de, bölünen alt uzayların hesaplama yükü açısından eşit olmaları hedeflenmiş ve bu amaçla sezgisel yöntemler geliştirilmiştir. BSP Tree ile de pek çok ortak noktaya sahip olan Kd-Tree, hesaplama yükünü düşüren eksenlere paralel olarak bölünen alt uzaylara sahip olmasıyla BSP Tree'den ayrılır. Bu nedenle BSP-Tree'ye oranla daha yüksek bir başarım sağlar.[6] Bu çalışmada, uzay bölmeleme algoritması olarak Kd-Tree gerçeklenmiştir.



Şekil 4.4: Kd-Tree

4.2 Kd-Tree'nin Oluşturulması

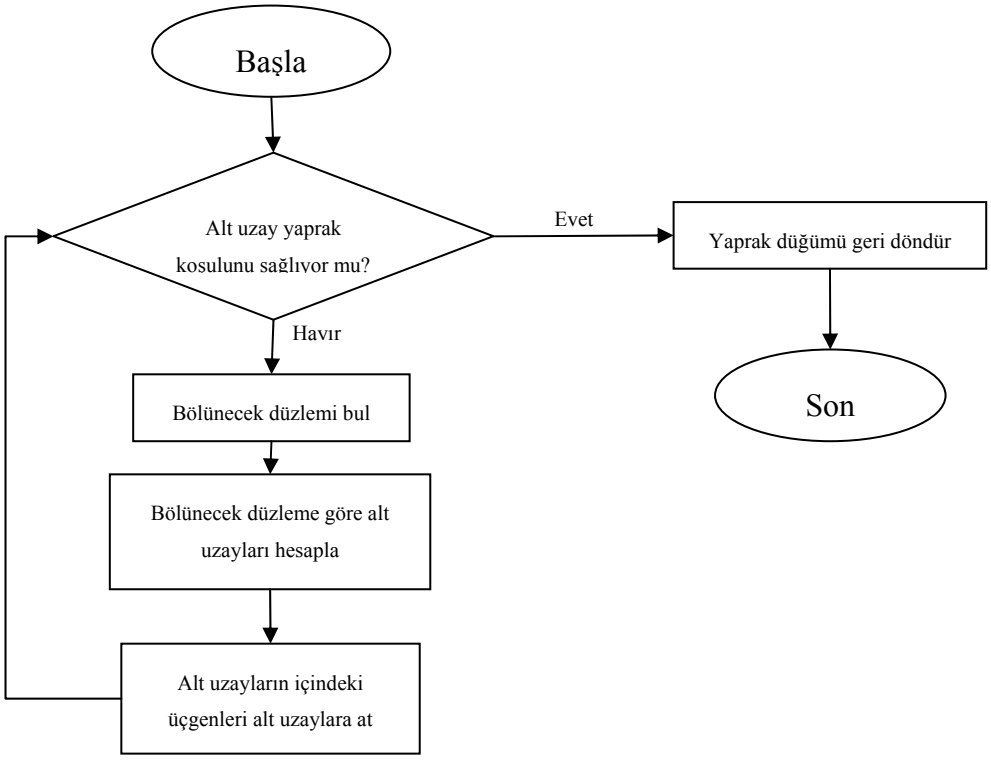
Kd-Tree'nin kullanılması için, öncelikle üçgenlerden oluşan cisim için bir Kd-Tree'nin oluşturulması gerekmektedir. Kd-Tree'nin oluşturulması, ilk bakıldığında ciddi bir hesaplama yükü getirse de, yapı bir kere oluşturulup cismin bütün açılardan

görüntüsünü elde etmekte kullanılabileceği ve bu işlemlerde oldukça yüksek bir başarım sağlayacağı için bu hesaplama yükü, göz ardı edilebilecek düzeydedir.

Daha önce bahsedildiği gibi, Kd-Tree oluşturulurken her aşamada uzay, iki alt uzaya bölünür. Uzayın bölüneceği düzlemi belirlemek için çeşitli sezgisel yöntemler kullanılır. Bu yöntemlerin en çok kullanılanları; "Orta Nokta Yaklaşımı", "Medyan Yaklaşımı" ve "Yüzey Alanı Sezgisel Yaklaşımı"dır.

Kd-Tree oluşturulması konusunda bir diğer önemli nokta, üçgenlerin bölünen her alt uzayın içinde olup olmadığının hesaplanmasıdır. Bunun için de kutu-üçgen kesişim algoritmaları geliştirilmiştir.

Kd-Tree, bu iki ölçüte göre özyinelemeli olarak oluşturulur. Kd-Tree'nin oluşturulma algoritması şu şekildedir:



Şekil 4.5: Kd-Tree oluşturma algoritması

Kd-Tree, şu sözde koda göre oluşturulabilir: [7]

```
function RecBuild (triangles T, voxel V) returns node
    if Terminate(T,V) then
        return new leaf node(T)
    p = FindPlane(T,V)
    (V_L, V_R) = Split V with p
    T_L = {t ∈ T | (t ∩ V_L) ≠ ∅
    T_R = {t ∈ T | (t ∩ V_R) ≠ ∅
    return new node (p, RecBuild(T_L , V_L) , RecBuild(T_R , V_R))
end function

function BuildKDTree (triangles[] T) returns root node
    V = B(T)
    return RecBuild(T,V)
end function
```

4.2.1 Bölme yüzeyinin belirlenmesi

Kd-Tree'nin oluşturulmasında en önemli işlemlerden biri, bölme yüzeyinin hesaplanmasıdır. Bunun için çeşitli metotlar geliştirilmiştir.

4.2.1.1 Uzaysal ortanca yaklaşımı

Bölme yüzeyinin belirlenmesindeki en basit, bununla birlikte en çok kullanılan yöntem, uzaysal ortanca yaklaşımıdır. Orta nokta yaklaşımında bölünecek eksen, sıralı olarak belirlenir. Bölme düzlemi ise, sıradaki ekseni dik olarak ortadan ikiye kesen bir düzlemdir. Bu bölme yöntemi, Kd-Tree'yi Octree ve QuadTree'ye yaklaştırır.

$$p_k = D(V) \bmod 3$$
$$p_\epsilon = \frac{V_{min,p_k} + V_{max,p_k}}{2} \quad \textbf{(4.1)}$$

Bu formülde p_k bölme ekseni, D(V) şu andaki ağaç derinliği, p_ϵ bölme yüzeyi, $V_{min,pk}$ bölme ekseninin en küçük noktası, $V_{max,pk}$ ise bölme ekseninin en büyük noktasıdır. [7]

4.2.1.2 Cisimsel ortanca yaklaşımı

Bölme yüzeyinin belirlenmesindeki bir diğer yöntem, cisimsel ortanca yaklaşımıdır. Bu yaklaşımda da bölünecek eksen, uzaysal ortanca yaklaşımında olduğu gibi sıralı olarak belirlenir. Bölme düzlemi ise, cismi oluşturan üçgenlerin orta noktalarının bölme eksenindeki değerlerinin ortancası olarak belirlenir.

$$p_k = D(V) \bmod 3$$
$$p_\epsilon = \frac{\sum_{i=t_{min}}^{t_{max}} O(t_i, p_k)}{t_{max} - t_{min} + 1} \quad \textbf{(4.2)}$$

Bu formülde p_k bölme ekseni, D(V) şu andaki ağaç derinliği, p_ϵ bölme yüzeyi, t_{min} alt uzayın ilk üçgen indeksi, t_{max} alt uzayın son üçgen indeksi, i sıradaki üçgen indeksi, $O(t_i, p_k)$ ise i numaralı üçgenin orta noktasının bölme eksenindeki elemanıdır.

4.2.1.3 Yüzey alanı sezgisel yaklaşımı

Daha önce belirtilen ortanca yaklaşımlar; klasik, gerçeklenmesi basit, buna karşın başarımı yüksek olmayan yaklaşımlardır. Kd-Tree'nin, köke en yakın ve en yüksek miktarda boşluklara sahip bir ağaç olarak daha yüksek bir başarıma sahip olduğu tespit edilmiştir. Bununla birlikte, her görüntü için yüksek başarımda bir çözüm bulmak oldukça zordur. Bu nedenle, önerilen çözümlerin çoğu, belirli tipte bir görüntüye özgü olarak sınırlı kalmıştır.

Bu sorunu çözmek için, her çeşit görüntünün ortak özelliklerine odaklanılmalıdır. Yapılan çalışmaların sonucunda, "Yüzey Alanı Sezgisel Yaklaşımı" (Surface Area Huristic) ön plana çıkmıştır. YAS yaklaşımında; bir V alt uzayının p yüzeyi tarafından bölünen iki alt uzayı olan V_L ve V_R ile, bu alt uzayların sahip olduğu

üçgen sayıları olan N_L ve N_R için beklenen arama maliyetleri hesaplanır. Bu yaklaşım, şu kabullerle hareket eder:

- Işınlar, düzgün olarak dağılmışlardır; cisim uzayının dışında başlar ve dışında sonlanırlar.
- Her alt uzay için kutu-ışın kesişimi ile her üçgen için ışın-üçgen kesişimi algoritmalarının maliyetleri bilinmektedir. Bunlar K_T ve K_I olarak temsil edilirler.
- Toplam ışın üçgen kesişimi maliyeti, üçgen sayısı (N) ile lineer olarak artar ve "N K_I" ile temsil edilir.

Bu varsayımlara göre, bir V uzayını kestiği bilinen bir ışının V uzayının bir alt uzayı olan V_{sub} uzayını kesme olasılığı P koşullu olasılığı şu şekilde ifade edilir:

$$P_{[V_{sub}|V]} = \frac{SA(V_{sub})}{SA(V)} \tag{4.3}$$

Bu formülde SA(V), V uzayının yüzey alanıdır.

Bu durumda p düzleminin beklenen maliyeti $C_V(p)$, ağaç üzerinde bir alt uzaya inmenin maliyeti ile bu alt uzayın sahip olduğu iki çocuk düğümün beklenen kesişim maliyetlerinin toplamıdır.

$$C_V(p) = K_T + P_{[V_l|V]}C(V_l) + P_{[V_r|V]}C(V_r) \tag{4.4}$$

(4.4) numaralı denklemi bütün ağaç (T) için genişletirsek:

$$C(T) = \sum_{\text{nedüğümler}} \frac{SA(V_n)}{SA(V_S)} K_T + \sum_{\text{leyapraklar}} \frac{SA(V_i)}{SA(V_S)} K_I \tag{4.5}$$

Denklemi bulunur. Burada V_S, bütün sahneyi içeren S uzayının sınırlarını kapsayan hacimdir. Bu durumda S sahnesi için en iyi T ağacı, C(T) değerinin en düşük olduğu ağaçtır. İşte YAS yaklaşımı, C(T) değerini en küçüğe indirgemeyi amaçlar. Bununla birlikte, sahne büyüdükçe C(T) maliyetinin en düşük olabileceği muhtemel ağaçların

sayısı da hızla artar. Dolayısıyla çok basit sahneler haricinde uygulanabilir bir ağaç bulmak imkansız hale gelir.

Bu nedenle genel bir ideal çözüm yerine, yerel bir yaklaşım uygulanabilir. Bu yaklaşıma göre, her V uzayının sahip olduğu alt V_L ve V_R alt uzaylarının yaprak olduğu varsayılır. Dolayısıyla bu varsayım, açgözlü bir çözüm olarak nitelendirilebilir. Bu durumda p düzleminin beklenen maliyeti $C_V(p)$ şu şekilde hesaplanır:

$$C_V(p) = K_T + K_I\left(\frac{SA(V_L)}{SA(V)}|T_L| + \frac{SA(V_R)}{SA(V)}|T_R|\right) \quad \textbf{(4.6)}$$

YAS yaklaşımına göre yazılım içerisinde beklenen maliyet şu sözde koda göre hesaplanır: [7]

```
function C(P_L, P_R,N_L,N_R) returns (C_V (p))
        return λ(p)(K_T + K_I (P_L N_L + P_R N_R))
end function

function SAH(p,V ,N_L,N_R,N_P ) returns (C_p, p_side)
        (V_L, V_R) = SplitBox(V, p)
        P_L = SA(V_L)/SA(V )
        P_R = SA(V_R)/SA(V )
        C_p->L = C(P_L, P_R,N_L + N_p,N_R)
        C_p->R = C(P_L, P_R,N_L,N_R + N_P )
        if cp->l < cp->r  then
            return (cp->L,LEFT)
        else
            return (cp->R,RIGHT)
end function
```

4.2.2 Kutu-üçgen kesişim algoritması

Kd-Tree oluşturulurken, her adımda o alt uzayın içerdiği üçgenler belirlenir. Alt uzaylar birer dikdörtgen prizma (kutu) olarak temsil edildiğinden, bu işlem için bir kutu-üçgen kesişim algoritması kullanılmalıdır.

Literatürde en bilinen kutu-üçgen kesişim algoritmalarından biri, 2001 yılında Tomas Möller tarafından yayınlanmıştır. Bu algoritmada, öncelikle koordinat ekseni kutunun merkezine gelecek şekilde kaydırılır. Ardından üç aşama test yapılır.

Birinci aşamada, kutu ile üçgeni çevreleyen kutunun kesişip kesişmediği test edilir Eğer bu kutular kesişmiyorsa, üçgen kutuyla kesişmiyordur.

İkinci aşamada, kutunun üçgenin bulunduğu düzlemi kesip kesmediği kontrol edilir. Eğer kutu, düzlemi kesmiyorsa, üçgeni de kesmiyordur.

Üçüncü aşamada ise üçüncü olarak ise üçgen kenarlarının, kutuyla kesişip kesişmediği kontrol edilir.

Bu testler olumsuz sonuç veriyorsa, üçgen, kutuyu kesmemiş demektir.[8]

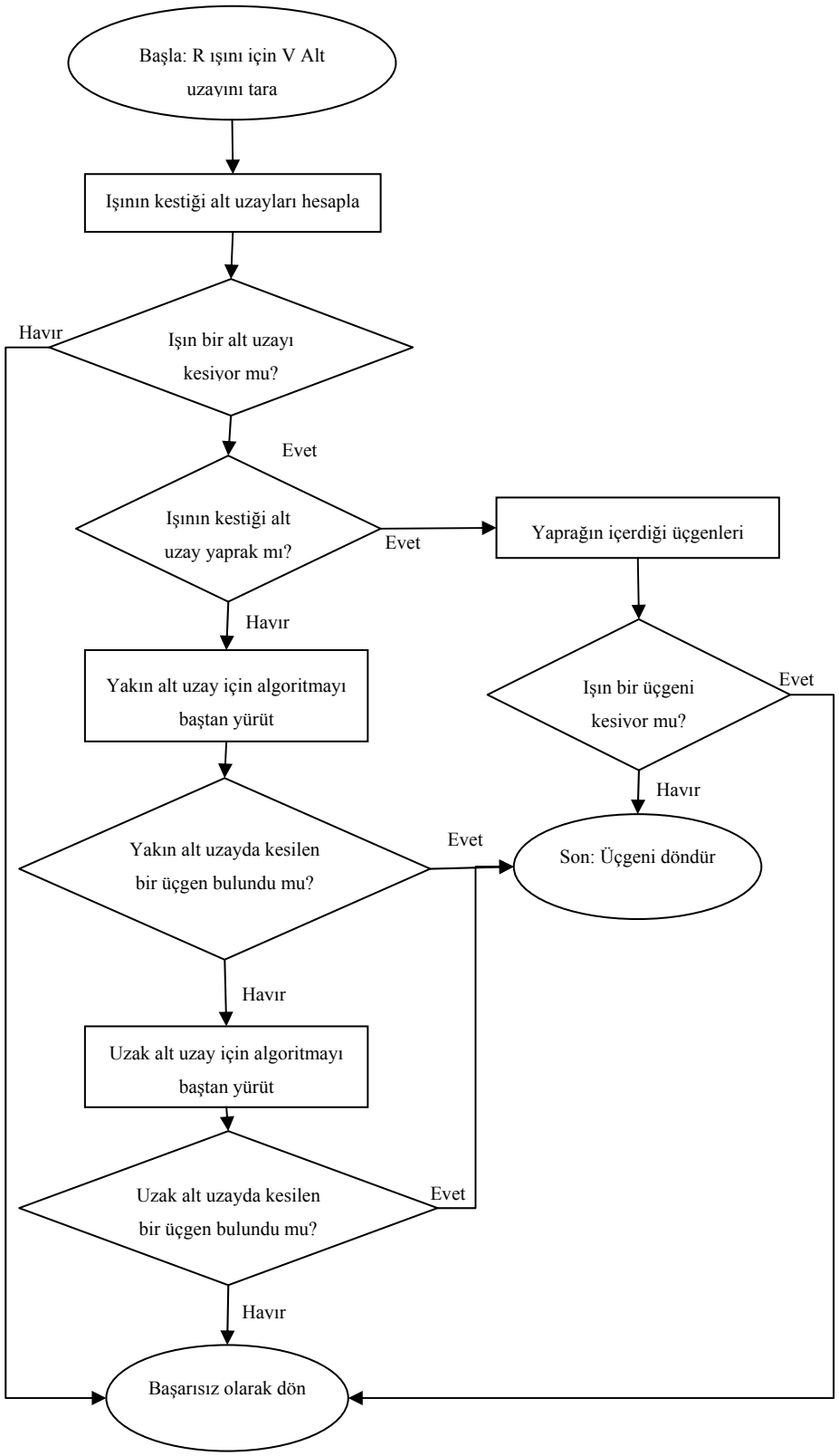
4.3 Kd-Tree Üzerinde Arama

Daha önce de belirtildiği gibi, Kd-Tree gibi uzay bölmeleme algoritmalarının asıl amacı, ışın izleme gibi işlemlerin daha verimli şekilde yapılmasını sağlamaktır. Bunun için ilgili yapının hazırlanmasının yanı sıra, yapıyı kullanan arama algoritmalarının da verimli olması gerekir. Bu amaçla özyinelemeli bir arama algoritması geliştirilmiştir.

4.3.1 Özyinelemeli Kd-Tree arama algoritması

Kd-Tree için özyinelemeli arama algoritması, esasen genel BSP-Tree'ler için Jansen tarafından 1986 yılında geliştirilmiş ve yayınlanmıştır. Bu algoritmada her derinlikte ışının kestiği alt uzay ya da uzaylar belirlenir. Işın, herhangi bir alt uzayı kesmiyorsa, alt uzayların içerdiği üçgenleri de kesmiyor demektir. Işın, alt uzaylardan birini kesiyorsa, arama alt uzayın sahip olduğu çocuk alt uzaylar için tekrarlanır. Eğer ışın bir derinlikteki iki alt uzayı birden kesiyorsa, aramaya ışının kaynağına en yakın alt uzaydan başlanır. Bu şekilde özyinelemeli olarak bir yaprak düğüme ulaşıldığında, o yaprak düğümün içerdiği bütün üçgenler için ışın-üçgen kesişim algoritması

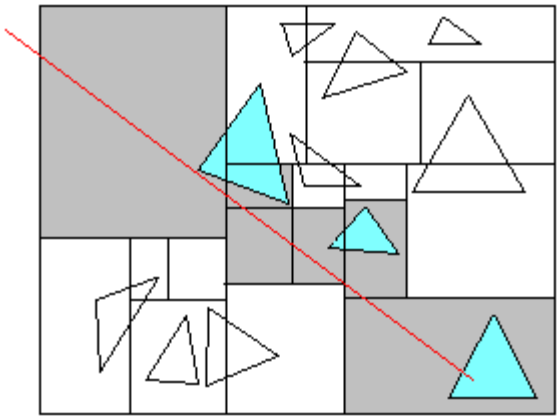
yürütülerek ışının kestiği en yakın üçgen bulunur. [9] Özyinelemeli Kd-Tree arama algoritması şu şekilde yürütülebilir:



Şekil 4.6: Özyinelemeli Kd-Tree arama algoritması

Özyinelemeli Kd-Tree arama agoritması, şu sözde kodla gerçeklenebilir: [10]

```
function Kd_Intersect(ray, node)
    if ray.interval is empty of node is nil then
        return false;
    if node is a leaf then
        for each triangle ∈ node
            ray_triangle_intersect(ray, triangle)
        if ray intersects a triangle then
            return true
        else
            return false
    else
        near = box_inters (ray,node.child[1])> ray_box_intersect(ray,node.child[0])
        far = 1- near
        if not Kd_Intersect(ray, true) then
            return Kd_Intersect(ray, far)
        else
            return true
end function
```



Şekil 4.7: Kd-Tree üzerinde ışın takibi

Kd-Tree üzerinde örnek bir ışının takibi şekil 4.7'de gösterilmiştir. İlgili şekilde ışın kırmızı renkli bir çizgi ile, aranan alt uzaylar gri renk ile, kontrol edilen üçgenler mavi renk ile gösterilmiştir. Şekilde gösterilen senaryo, Kd-Tree üzerinde bir üçgene çarpan bir ışın için en kötü olasılıklı senaryodur. Bu durumda bile, uzayın içerdiği 11 üçgenden yalnızca 3 tanesi kontrol edilmiş ve görüldüğü gibi gereksiz kontrollerden kurtulunmuştur. Görüldüğü gibi Kd-Tree, örnekteki temsili senaryoda bile yüksek bir verime sahiptir.

4.3.2 Işın-kutu kesişim algoritması

Bölüm 4.3.1'de belirtildiği gibi, Kd-Tree üzerinde arama yapılırken öncelikle ışının alt uzayları kesip kesmediği belirlenir. Kd-Tree yapısında alt uzaylar birer kutu (dikdörtgen prizma) ile temsil edilmiştir. Dolayısıyla ışının herhangi bir alt uzayı kesip kesmediğinin belirlenmesi için bir ışın-kutu kesişim algoritması kullanılmalıdır.

Bu amaçla Smits'in önerdiği algoritma şu şekilde gerçeklenebilir: [11]

```
bool Box::intersect(const Ray &r, float t0, float t1) const {

	float tmin, tmax, tymin, tymax, tzmin, tzmax;
	if (r.direction.x() >= 0) {
		tmin = (bounds[0].x() - r.origin.x()) / r.direction.x();
		tmax = (bounds[1].x() - r.origin.x()) / r.direction.x();
	}
	else {
		tmin = (bounds[1].x() - r.origin.x()) / r.direction.x();
		tmax = (bounds[0].x() - r.origin.x()) / r.direction.x();
	}
	if (r.direction.y() >= 0) {
		tymin = (bounds[0].y() - r.origin.y()) / r.direction.y();
		tymax = (bounds[1].y() - r.origin.y()) / r.direction.y();
	}
	else {
		tymin = (bounds[1].y() - r.origin.y()) / r.direction.y();
		tymax = (bounds[0].y() - r.origin.y()) / r.direction.y();
	}
	if ( (tmin > tymax) || (tymin > tmax) )
		return false;
	if (tymin > tmin)
		tmin = tymin;
	if (tymax < tmax)
		tmax = tymax;
	if (r.direction.z() >= 0) {
		tzmin = (bounds[0].z() - r.origin.z()) / r.direction.z();
		tzmax = (bounds[1].z() - r.origin.z()) / r.direction.z();
	}
	else {
		tzmin = (bounds[1].z() - r.origin.z()) / r.direction.z();
		tzmax = (bounds[0].z() - r.origin.z()) / r.direction.z();
	}
	if ( (tmin > tzmax) || (tzmin > tmax) )
		return false;
	if (tzmin > tmin)
		tmin = tzmin;
	if (tzmax < tmax)
		tmax = tzmax;
	return ( (tmin < t1) && (tmax > t0) );
}
```

Bu algoritma, neredeyse bütün durumlar için doğru çalışmaktadır. Fakat eğer ışın yönünün x bileşeninin "-0.0"a eşit olması durumunda ilk if belirteci doğru döndürmekte ve (-∞, +∞) yerine yanlış olarak (+∞, -∞) aralığını üretmektedir. Aynı problem y veya z bileşenleri için de geçerlidir. Bu durumlarda bu algoritma yanlış çalışmaktadır. Bu durumlar için, yön bileşenlerinin işaretlerini yakalayan ayrı bir değişken kullanılabilir. Aşağıdaki kod parçasında, bu değişken "divx" olarak tanımlanmıştır.

```
divx = 1 / r.direction.x();
if (divx >= 0) {
        tmin = (bounds[0].x() - r.origin.x()) * divx;
        tmax = (bounds[1].x() - r.origin.x()) * divx;
}
else {
        tmin = (bounds[1].x() - r.origin.x()) * divx;
        tmax = (bounds[0].x() - r.origin.x()) * divx;
}
```

Bu kod, sadece x bileşeni için geçerli olup, benzer bir işlem y ve z bileşenleri için de yapılabilir. Bu işlem fazladan bir maliyete neden olmakla birlikte, algoritmanın tüm durumlarda hatasız çalışması için gereklidir.

Bununla birlikte, işaret bilgisini taşıyan değişkene ait değerin her ışın-kutu kesişimi için tekrarlanmasına gerek yoktur. Bu değişken, ışın yapısına eklenerek algoritma, aynı ışının birden fazla kutuyla kesişiminde kullanılabilir. Böylece algoritma, birden fazla kutu kesişimi için optimize edilmiş olur. Optimize algoritma için ışın sınıfı şu şekilde gerçeklenebilir: [12]

```
class Ray {
        public:
        Ray(Vector3 &o, Vector3 &d) {
        origin = o;
        direction = d;
        inv_direction = Vector3(1/d.x(), 1/d.y(), 1/d.z());
        sign[0] = (in v_direction.x() < 0);
        sign[1] = (inv_direction.y() < 0);
        sign[2] = (inv_direction.z() < 0);
        }
        Vector3 origin;
        Vector3 direction;
        Vector3 inv_direction;
        int sign[3];
};
```

4.4 Paralelleştirme

Işın izlemeyi daha verimli olarak yapmayı sağlayan çeşitli yöntemler geliştirilmişse de, özellikle son yıllarda yeniden canlanan paralel sistemler kullanılarak bu işlem çok daha yüksek başarımlı olarak yürütülebilir. Işın izleme işleminin nasıl paralelleştirilebileceği düşünüldüğünde, akla iki yöntem gelmektedir:

- Tek bir ışın için paralel arama
- Birden fazla ışını paralel olarak yürütme

4.4.1 Ağaç üzerinde paralel arama

Işın izlemeyi paralelleştirme işlemindeki ilk seçenek, tek ışın için ağaç üzerinde paralel arama yapmaktır. Bu işlemde tek ışın için ağaç aramasının her bir adımı paralel gerçekleştirilir. Yapraklara ulaşıldığında ise ışın-üçgen kesişimi işlemleri, paralel işlemciler arasında paylaştırılır. Bu yöntem tek bir ışın izleme işlemini

hızlandırsa da, özellikle yüksek derinliğe sahip olan ağaçlarda verimli bir çalışma gösteremez. Bu yöntem, özellikle az miktarda paralel işlemciye sahip sistemler için nispeten verimli olarak çalışabilir.

4.4.2 Işınların paralelleştirilmesi

Işın izleme kullanan işlemlerin pek çoğunda binlerce (bazılarında milyonlarca) ışın kullanılır. Bu ışınların tamamı, merkezleri ve yönleri birbirinden farklı olan, bununla birlikte aynı veri seti (ağaç) üzerinde çalışan yapılardır. Kd-Tree gibi ağaçlar bir kere oluşturulup ışın izleme işlemi boyunca değiştirilmediğinden, ışınların tamamen birbirlerinden bağımsız olarak çalıştığı düşünülebilir. Bu nedenle ışın izleme işleminin ışın düzeyinde paralelleştirilmesi, muazzam derecede paralel olarak nitelendirilen sistemler üzerinde kolaylıkla çalıştırılabilir.

5. GRAFİK KARTI ÜZERİNDE VERİ İŞLEME

Işın izleme işlemi, birbirinden bağımsız çalışan ışınlar göz önüne alındığında, muazzam derecede paralelleştirilebilir bir işlemdir. "Muazzam derecede paralel bilgisayar" kavramı, bu güne kadar vektör makineler ya da çok sayıda yüksek performanslı bilgisayarın bir araya getirilmesiyle oluşmuş bilgisayar kümeleri için kullanılmıştır. "Süper bilgisayar" olarak adlandırılan bu bilgisayarlar, maliyetleri yüz binlerce dolardan başlayıp milyonlarca dolara çıkan sistemlerdir. Dolayısıyla bu bilgisayarlar, dünyada az sayıda bulunmakta ve çoğunlukla üniversiteler, enstitüler, savunma sanayisindeki büyük şirketler tarafından kullanılmaktadır.

Süper bilgisayarlar ile birlikte, özellikle son 10 yılda kişisel bilgisayarlar üzerinde de muazzam derecede paralel sistemler gelişmeye başlamıştır. Fakat bu sistemler, kişisel bilgisayarın merkezi işlem birimlerini değil de grafik işlem birimlerini oluşturmaktadır. Muazzam derecede paralel veri işleme kapasitesine sahip bu sistemler, kullanıcıların doğrudan grafik kart üzerlerinde program yürütmelerini sağlayan bir yapı olmadığından, uzun yıllar sadece grafik kartı üreticilerinin yayınladığı sürücüler üzerinden grafik üretmek için kullanılmıştır.

5.1 CUDA Teknolojisi

Özellikle son yıllarda, bilgisayar oyunları ve grafik tasarım programlarının gelişmesi, "Grafik İşlem Birimi (GİB)" adı verilen ve grafik kartlarının yerel merkezi işlem birimleri olarak nitelendirebileceğimiz bileşenlerin oldukça yüksek bir ivmeyle evrimleşmesine neden olmuştur. Zaman içerisinde GİB'ler, yüksek işlem kapasitesine sahip, muazzam derecede paralel, aynı anda birden fazla iplik işleyebilen, çok çekirdekli bir işlemciye dönüşmüştür.

CUDA™ (Compute Unified Device Architecture), grafik kartı üreticisi NVIDIA® tarafından 2006 yılının sonunda yayınlanmış genel maksatlı bir paralel veri işlemem mimarisidir. CUDA, yeni bir programlama modeli ve komut seti mimarisi ile NVIDIA üretimi GİB'lere sahip grafik kartlar üzerinde paralel hesaplama

yapılabilmesine olanak sağlar. CUDA; C, Fortran, OpenCL gibi farklı dilleri, belirli ek komut ve kısıtlamalarla desteklemektedir. [13]

CUDA, grafik kartları (NVIDIA GeForce ve Quadro) üzerinde hesaplama yapabilmek amaçlı geliştirilse de, NVIDIA, CUDA ile veri işleme yeteneğine sahip ve sadece veri işleme amaçlı NVIDIA Tesla hesaplama kartlarını da üretmiştir.

Şekil 5.1: NVIDIA GTX285, bilgisayar oyunları için üretilmiş, CUDA ile hesplama amacıyla kullanılabilen grafik kartı.

Şekil 5.2: NVIDIA Tesla, GIB içeren hesaplama kartı

Şekil 5.3: NVIDIA Tesla kartlar içeren bir masaüstü süper bilgisayar

5.2 CUDA C Dili, Eklenti ve Kısıtlamaları

CUDA, sistemdeki GİB'lerin kullanılabilmesini sağlayan bir takım komut eklentileriyle birlikte, çeşitli kısıtlamalar da getirmektedir. CUDA'nın ANSI C standardına getirdiği belli başlı eklentiler şunlardır:

- __device__ belirteci GİB üzerinde çalışan ve yalnızca GİB üzerinde çalışan diğer fonksiyonlar tarafından çağrılabilen fonksiyonları tanımlamak için kullanılır.
- __global__ belirteci GİB üzerinde çalışan ve yalnızca MİB üzerinde çalışan fonksiyonlar tarafından çağrılabilen fonksiyonları tanımlamak için kullanılır.
- __host__ belirteci MİB üzerinde çalışan ve yalnızca MİB üzerinde çalışan fonksiyonlar tarafından çağrılabilen fonksiyonları tanımlamak için kullanılır. Fonksiyon, bu ifadelerin herhangi biri tarafından tanımlanmazsa, bu fonksiyonun bir __host__ fonksiyonu olduğu varsayılır.
- __device__ belirteci, aynı zamanda genel bellekte yer alacak olan değişkenleri tanımlamak için kullanılır.
- __shared__ belirteci, paylaşılan bellekte yer alacak olan değişkenleri tanımlamak için kullanılır.
- __constant__ belirteci, değişmez bellekte yer alacak olan değişkenleri tanımlamak için kullanılır.

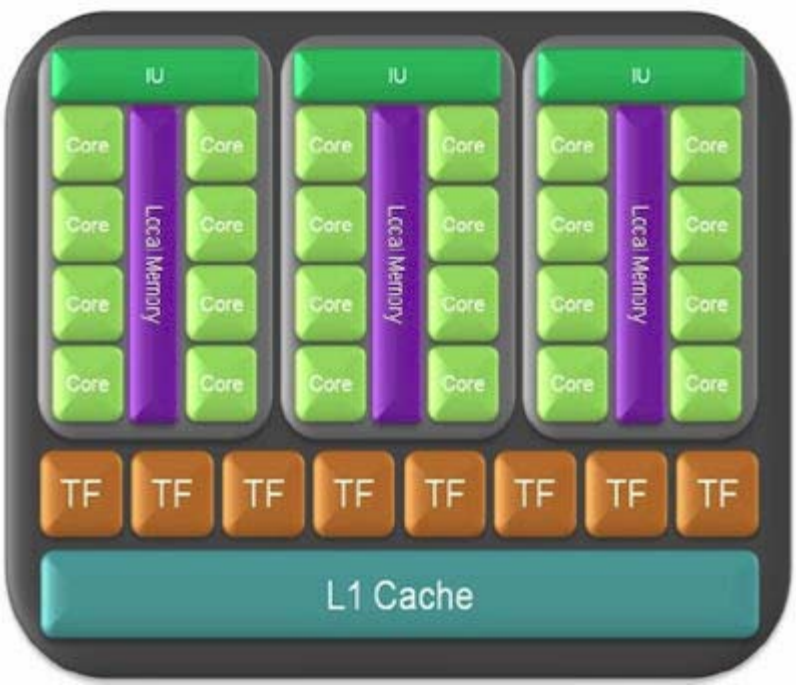
CUDA'nın getirdiği bir takım kısıtlamalar ise şunlardır:

- CUDA, donanımsal olarak C++ dili desteklememektedir.
- CUDA, çift hassasiyetli kayan noktalı sayıları donanımsal olarak desteklememektedir.
- __device__ ve __global__ fonksiyonlarda özyineleme kullanılamaz.
- __device__ ve __global__ fonksiyonlarda bellek alanı dinamik olarak ayrılamaz.

- __device__ ve __global__ fonksiyonlarının gövdelerinde statik değişken kullanılamaz.
- __device__ ve __global__ fonksiyonlar parametre dizileri kabul etmezler.
- __device__ fonksiyonlardan __global__ fonksiyonlara fonksiyon işaretçisi tanımlanamaz.
- __global__ ve __host__ belirteçleri bir arada kullanılamaz.
- __global__ fonksiyonlar "void" döndürmek zorundadır.
- __global__ fonksiyonu asenkron çalışır, yani GİB'deki çalışmasını bitirmeden önce döner.
- __global__ fonksiyonlara geçilen parametreler, değişmez bellekte yer alır ve 4KB ile sınırlıdır.
- __device__, __shared__ ve __constant__ değişlenler, "extern" belirteci kullanılarak harici olarak tanımlanamaz.
- __device__ ve __constant__ değişkenlere sadece tek dosya kapsamında izin verilir.
- __constant__ değişkenler GİB üzerinde çalışan fonksiyonlarda atama yapılamaz. Bu değişkenlere sadece MİB üzerinde çalışma anında atama yapılabilir. Bu tip değişkenler için dinamik olarak yer ayrılamaz.
- __shared__ değişkenlere ilk değer verilemez.
- __device__, __shared__ ve __constant__ değişkenlerin adresleri, sadece GİB üzerinde çalışan fonksiyonlarda kullanılabilir. [13]

5.3 Grafik İşlem Biriminin Yapısı

CUDA destekleyen GİB'lerin mimarileri, nesillerine göre birbirinden farklı olmakla birlikte, temel yapı şu şekildedir: GİB, pek çok çekirdek kümesinin ızgara yapısında birleştirilmesinden oluşur. Her çekirdek kümesinde, çoklu iplik işleme yeteneğine sahip 8 çekirdek bulunur. [13]



Şekil 5.4: GT200 GİB mimarisi

5.4 Grafik Kartlarının Bellek Yapısı

CUDA iplikleri, çalışırken birden fazla bellek uzayına erişebilirler. Bu bellek uzayları; "Genel Bellek", "Yerel Bellek", "Değişmez Bellek", "Doku Belleği" ve "Paylaşılan Bellek"tir.

5.4.1 Genel bellek

Genel bellek, çalışan bütün ipliklerin erişebileceği yüksek kapasiteye sahip bir oku/yaz bellektir. Aygıt belleği olarak adlandırılan kısımda bulunur. Bu kısmın gecikme değeri yüksek, bant genişliği düşüktür. Genel belleğe 32, 64 veya 128 baytlık segmanlar halinde erişilebilir. Segman boyutu, genel belleğe erişen bir komut yürütüldüğünde, komutun içerdiği değişkenin boyuna ve kaç ipliğin belleğe eriştiğine bağlı olarak değişebilir. Genel bellek, bir önbelleğe sahip değildir fakat sıralı erişimler için optimize edilmiştir. [13]

5.4.2 Yerel bellek

Yerel bellek, her ipliğin sadece kendisinin erişebileceği bir oku/yaz bellektir. Donanımsal olarak her genel bellek gibi aygıt belleğinde bulunur, ancak her iplik, yalnızca kendisi için ayrılmış yerel bellek alanına erişebilir. Aygıt belleğinde olması

nedeniyle, genel bellek gibi yüksek gecikme ve düşük bant genişliği değerlerine sahiptir. Yerel bellek kullanımı, iplik tarafından kullanılma oranına göre, MİB üzerindeki donanımsal saklayıcıların kullanılamayacağı durumlarda derleyici tarafından otomatik olarak ayarlanır. [13]

5.4.3 Değişmez bellek

Aygıt belleğinde bulunan, GİB için salt oku olarak çalışan bir bellektir. Sadece fonksiyon dışında global ve statik olarak tanımlanabilir. Değişmez bellekten dinamik olarak yer ayrılmasına izin verilmez ve bu bellek alanına sadece MİB üzerinde çalışan fonksiyonlar tarafından veri atanabilir. Önbelleğe sahiptir ve bu nedenle genel belleğe göre daha yüksek bir hızda çalışmaktadır. [13]

5.4.4 Doku belleği

Doku belleği, aygıt belleğinde bulunan bir bellektir. Adından da anlaşılacağı gibi, doku bilgisinin saklanması için optimize edilmiştir. Kendisine ait bir doku önbelleği vardır. Doku işlemlerine göre, farklı adresleme kiplerine sahiptir. [13]

5.4.5 Paylaşılan bellek

Paylaşılan bellek, diğer bellek türlerinin aksine donanımsal olarak GİB'in sahip olduğu her çekirdek kümesinin içinde bulunur. Dolayısıyla sadece aynı küme içerisinde bulunan iplikler aynı bellek alanına erişebilir. Yonga üzerinde olduğundan, özellikle ipliklerin eriştiği bellek alanları arasında çakışma olmadığında oldukça yüksek bir hızda çalışır. Buna karşın genel belleğe nazaran çok daha düşük kapasiteli bir bellek türüdür. CUDA yazılımlarını hızlandırmak için çoğunlukla kullanılan yöntem, genel bellekte bulunan veriler, bloklar halinde paylaşılan belleğe aktarılarak işlenir. [13]

5.5 Kd-Tree'nin Grafik Kartı Belleği Üzerinde Gerçeklenmesi

MİB ve GİB'in kullandığı bellekler, birbirinden tamamen farklı olduğundan ve bu bellekler arasında doğrudan bir iletişim kurulamadığından, oluşturulan Kd-Tree'nin CUDA kullanılarak GİB üzerinde işlenebilmesi için öncelikle kullanılan ağaç yapısının grafik kartı belleğine aktarılması gerekir.

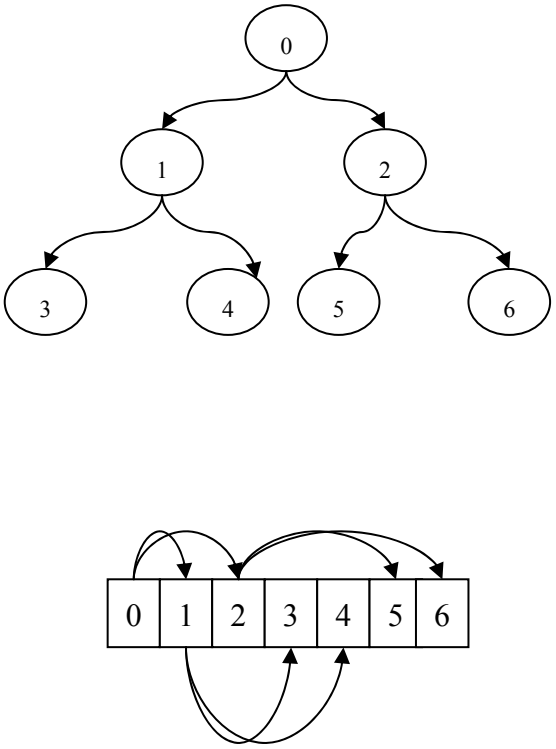
5.5.1 Ağacın dizi üzerinde gerçeklenmesi

Işın izleme için oluşturulan Kd-Tree, temelde bir ikili ağaçtır. Standart bir Kd-Tree, şu yapıdadır:

```
typedef struct kdTreeType
{
        kdNode * root;
}kdTree;
typedef struct kdNodeType
{
        struct kdNodeType * leftChild;
        struct kdNodeType * rightChild;
        triangle * triangleList
        //other node data
}kdNode;
```

Görüldüğü gibi her düğüm, çocuk düğümlerine işaret eden iki adet işaretçi içermektedir. Bununla birlikte MİB ve GİB'in kullandığı bellekler ve bellek uzayları birbirinden farklı olduğundan, GİB üzerinde yer ayrılan her düğüm için adres bilgisi alınmalı ve GİB üzerinde ilgili işaretçilere atama yapılmalıdır. Bu da, değişken yapılı ve karmaşık ağaçlarda oldukça zahmetli bir işlemdir.

Buna karşın herhangi bir tipten bir dizi, grafik kartı belleğine rahatlıkla aktarılabilmektedir. Bu nedenle ağacın dizi üzerinde gerçeklenmesi, hem yapının grafik kartı belleğine aktarılmasını oldukça kolaylaştırmakta, hem de erişim maliyetlerini düşürdüğü için işlemlerin daha hızlı yapılmasını sağlamaktadır.



Şekil 5.5: Kd-Tree'nin dizi üzerinde gerçeklenmesi

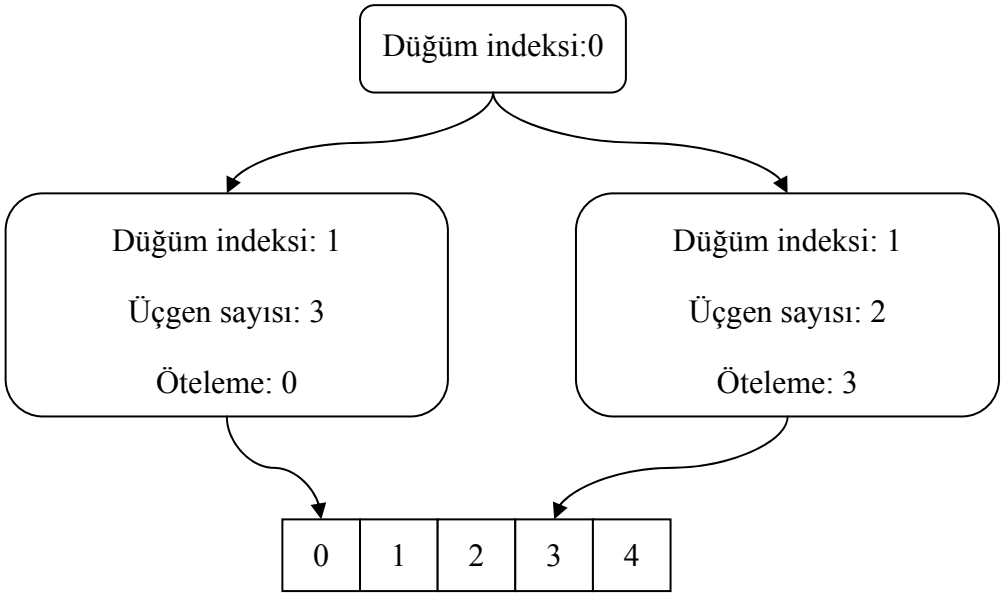
Şekil 5.5'te görüldüğü gibi, ikili ağacın her düğümü diziye atıldığında, her n numaralı indeks değerine sahip düğümün sağ çocuğunun indeksi 2*n+2, sol çocuğunun indeksi 2*n+1'dir. Bu da, kendi indeksini bilen her düğümün çocuk düğümlerine erişmesi için herhangi bir işaretçiye ihtiyaç duymamasını sağlar. Bu durumda yeni Kd-Tree yapısı şu şekilde gerçeklenebilir:

```
typedef struct cudaKdNodeType
{
        cudaTriangleList triangleList;
        //other node data
}cudaKdNode;


typedef struct cudaKdTreeType
{
        int size;
        cudaKdNode * nodeList;
}cudaKdTree;
```

5.5.2 Üçgenler için bellek havuzu oluşturulması

Ağacın sahip olduğu düğüm yapısında olduğu gibi, yaprak düğümlerinin sahip olduğu üçgen listesi için de benzer problemler geçerlidir. Her yaprağın sahip olduğu üçgen sayısı değişken olduğundan, dinamik bellek ayrılmasına gerek duyulur ve bu da her yaprak için ayrı ayrı işlem yapılmasını gerektirir. Buna karşın, yaprakların sahip olduğu üçgen listeleri ardışıl olarak tek bir diziye atılıp, yaprakların üçgen listelerini gösteren işaretçilerinin değerleri, bu dizi üzerinden öteleme değerleri ile belirlenir.



Şekil 5.6: Üçgenler için bellek havuzu oluşturulması

5.5.3 Ağaç ve üçgenlerin grafik kartı belleğine aktarılması

Ağaç ve üçgenler dizi üzerinde gerçeklendikten sonra, bu dizilerin grafik kartı belleğine aktarılması kolaylaşır. Grafik kartı üzerinde dinamik bellek ayırma işlemi "cudaMalloc" fonksiyonuyla, verilerin grafik kartı belleğine aktarılması işlemi "cudaMemcpy" fonksiyonuyla yapılır. Bununla birlikte, yapıda halen var olan işaretçiler nedeniyle yapının grafik kartı belleğine atılması işlemi iki aşamada yapılır. İlk aşamada ilgili işaretçiler hesaplanıp MİB belleğinde geçici değişkenlere atılır. Böylece bu değişkenler grafik kartı belleğine aktarıldığında, işaretçiler geçerli adres değerlerine sahip olur. Bu işlem, şu kod parçasıyla yapılabilir:

```
cudaKdTree * h_ckd = NULL, * d_ckd = NULL;
cudaKdNode * h_nodeList = NULL, * d_nodeList = NULL;
triangle * h_elements = NULL, * d_elements = NULL;

h_ckd = (cudaKdTree *)malloc(sizeof(cudaKdTree));
h_nodeList = (cudaKdNode *)malloc(totalNodes * sizeof(cudaKdNode));
h_elements = (triangle *)malloc(totalElements * sizeof(triangle)) ;
* h_ckd = * scn->cKd;
 int offset = 0;
 offset = 0;
 for (int i = 0; i<totalNodes; i++)
 {
            h_nodeList[i] = scn->cKd->nodeList[i];
            if (h_nodeList[i].triangleList.size>0)
            {
                        h_nodeList[i].triangleList.elements = h_elements + offset;
                        offset = offset + h_nodeList[i].triangleList.size;
            }
            else
                       h_nodeList[i].triangleList.elements = NULL;
 }
```

```
cudaMalloc( (void**)&d_ckd, sizeof(cudaKdTree) );
cudaMalloc( (void**)&d_nodeList, totalNodes * sizeof(cudaKdNode)  );
cudaMalloc( (void**)&d_elements, totalElements * sizeof(triangle) );
offset = 0;
for(int i = 0; i < totalNodes; i++ ) {
        h_nodeList[i].triangleList.elements = d_elements + offset;
        offset +=  h_nodeList[i].triangleList.size;
}
cudaMemcpy( d_elements, h_elements, totalElements * sizeof(triangle), cudaMemcpyHostToDevice);
cudaMemcpy( d_nodeList, h_nodeList, totalNodes * sizeof(cudaKdNode), cudaMemcpyHostToDevice) ;
h_ckd->nodeList = d_nodeList;
cudaMemcpy( d_ckd, h_ckd, sizeof(cudaKdTree), cudaMemcpyHostToDevice );
```

5.6 Görüntünün Paralel Olarak Oluşturulması

Ağaç yapısı ve üçgenlerin grafik kartı belleğine aktarılmasıyla beraber, ışın izleme işleminin ön hazırlıkları tamamlanmıştır. Bir sonraki adım, ışın izleme işleminin GİB üzerinde yürütülmesidir.

5.6.1 İpliklerin yaratılması

Bir CUDA programında yaratılacak iplik sayısı için iki parametre önemlidir. Bunlardan birincisi GİB üzerindeki çekirdek bloklarından kaç tanesinin kullanılacağını, ikincisi ise bu blokların her birinde kaçar iplik kullanılacağını belirtir. Çoğunlukla, bloklarda kullanılacak iplik sayısı (blok boyutu) sabit bir sayı seçilerek, kullanılacak blok sayısı hesaplanacak toplam eleman sayısına göre değişken olarak hesaplanır. Daha sonra, "__global__" etiketli CUDA çekirdek fonksiyonu çağrılırken bu değerler fonksiyona özel parametreler şeklinde aktarılır. Aktarılan bu parametrelere göre iplikler, CUDA tarafından otomatik olarak oluşturulur. Örnek bir CUDA çekirdek fonksiyonu şu şekilde çağrılır:

```
cudaTraceRayOnDevice <<< blok_sayisi, blok_boyutu>>> (…parametreler…)
```

5.6.2 Işınların ipliklere paylaştırılması

CUDA programlarında, işlenecek verinin her bir birimi bir iplik tarafından işlenir. Burada önemli olan, işlenecek bu veri birimini doğru olarak belirlemektir.

Bu çalışmadaki veri birimi, oluşturulacak görüntünün her bir pikseli olarak seçilmiştir. Bu durumda toplam iplik sayısı, toplam piksel sayısına, yani toplam ışın sayısına eşittir. Blok boyutu sabit bir sayı seçildiğinde; blok sayısı toplam sütun sayısının blok boyutuna bölünmesiyle bulunur.

```
block_size = 100;
n_blocks = RAYCOUNT/block_size + ((RAYCOUNT)%block_size == 0 ? 0:1);
```

5.6.3 Işınların ipliklerde işlenmesi

Işınların ipliklerde işlenebilmesi için, öncelikle her ipliğin sorumlu olduğu ışınları bilmesi gerekir. Bunun için, her ipliğe ayrı ayrı bilgi gönderilmesine gerek yoktur.

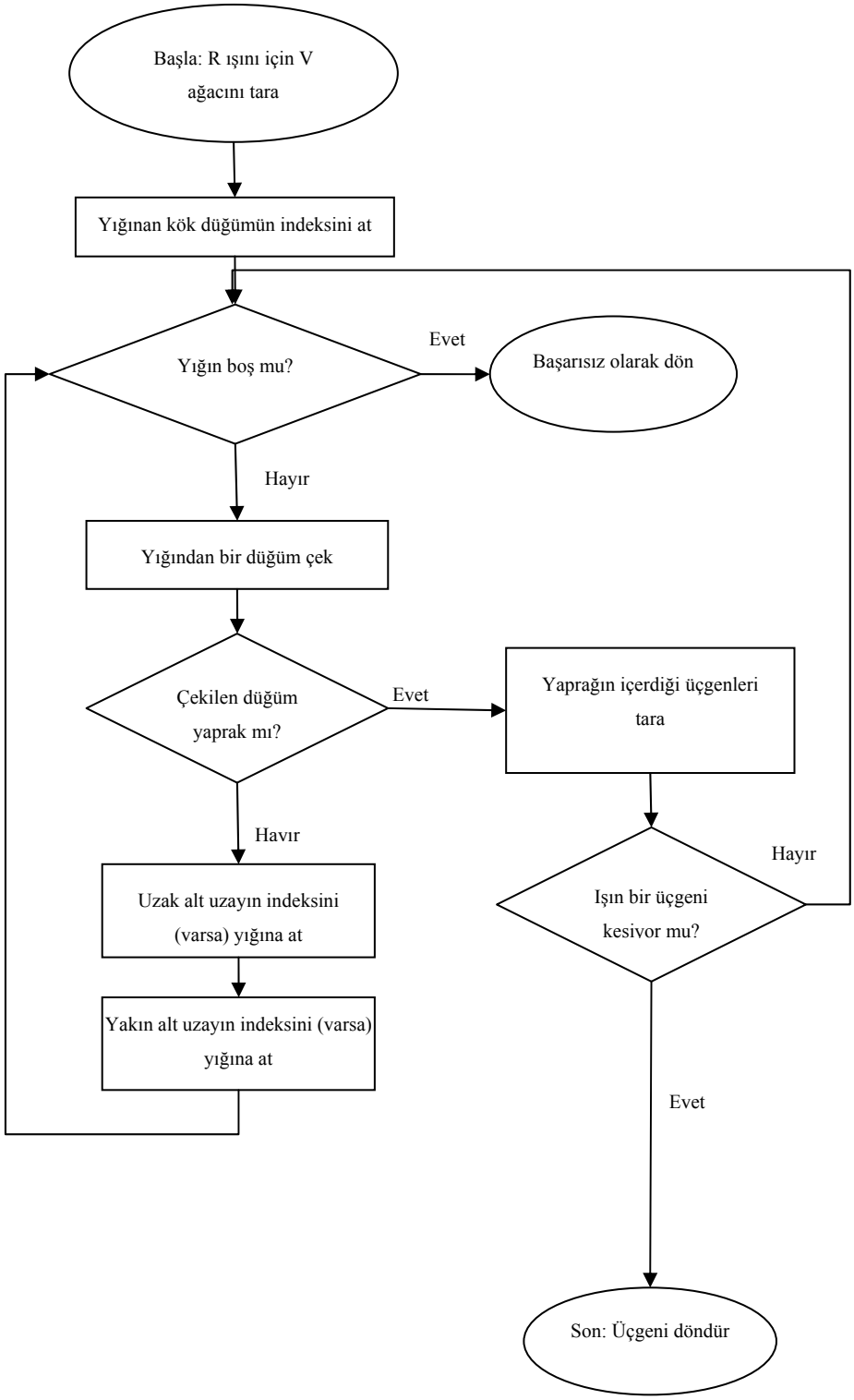
CUDA'da çalışan her iplikte, ipliğin kaç numaralı bloğun kaçıncı ipliği olduğunu saklayan değişkenler bulunur. "blokckIdx" değişkeni blok numarasını, "blockDim" değişkeni toplam blok sayısını, "threadIdx" değişkeni ise blok içerisindeki iplik numarasını saklar.

Bu durumda, her ipliğin işleyeceği sütun numarası şu şekilde hesaplanabilir:

```
int rowId = blockIdx.x * blockDim.x + threadIdx.x;
```

İşlenecek sütun numarası belirlendiğine göre, bu sütunun her satırı için bölüm 4.3.1 de belirtilen Kd-Tree arama algoritması yürütülür. Fakat bölüm 5.2'de belirtilen kısıtlamalar nedeniyle özyinelemeli olan bu fonksiyon, GİB üzerinde doğrudan yürütülemez.

Kd-Tree üzerinde arama fonksiyonunun GİB üzerinde yürütülebilmesi için, öncelikle özyinelemeden kurtarılması gerekmektedir. Bunun için bir yığın benzetimi yapısı kullanılabilir. Bu yapıda, her düğüm işlendiğinde çocuk düğümler için aynı fonksiyonun özyinelemeli olarak çağrılması yerine, her adımda işlenilmesi istenen düğüm indeksleri bir yığına atılır. Bir sonraki adımda bu indeks yığından çekilerek, işlem yığından çekilen indekse sahip düğüm için tekrarlanır. Bu işlem yığın boşalıncaya kadar devam eder.



Şekil 5.7: Ardışıl Kd-Tree arama algoritması

5.7 Sonuçların Bilgisayar Belleğine Aktarılması

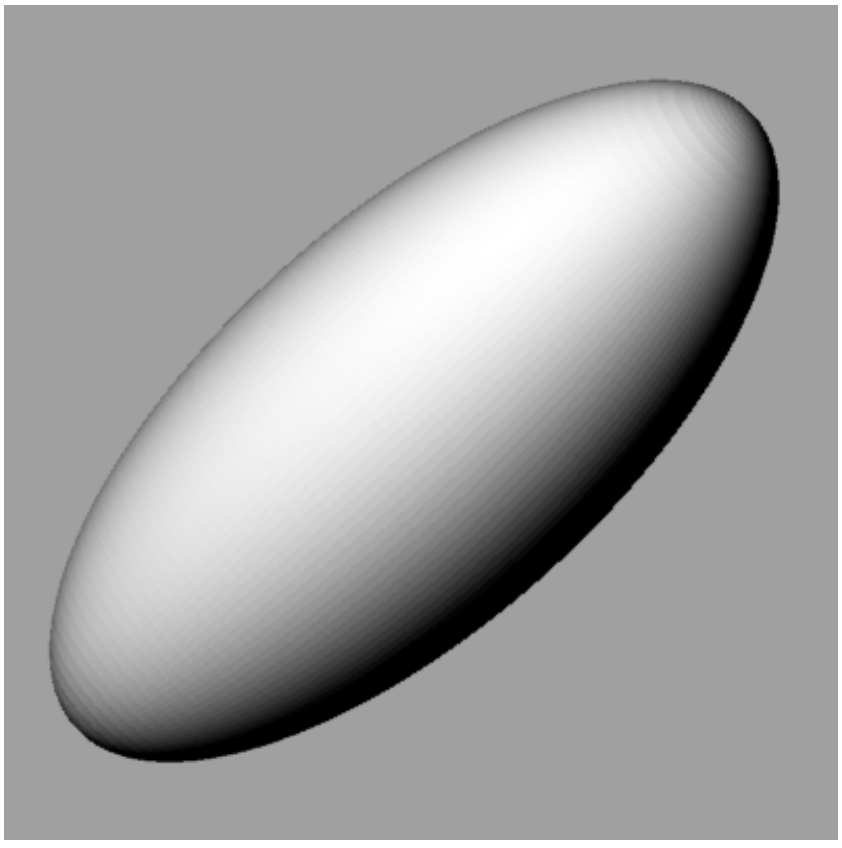
CUDA ipliklerinin her piksel için hesapladığı renk değerleri, kullanılmak için bilgisayar belleğine aktarılmalıdır. Yapılan ışın izleme işlemi sonucunda elde edilen görüntü, esasen renk değerlerinden oluşan iki boyutlu bir matristir. Bunun için öncelikle bu matris için grafik kartının genel belleğinde yer ayrılmalı, CUDA iplikleri hesapladıkları renk değerlerini bu bölüme yazmalı ve ardından bu bölüm, bilgisayar belleğine geri aktarılmalıdır. Bu işlemin kolaylaştırılması için öncelikle iki boyutlu matris, tek boyutlu bir dizi olarak gerçeklenir. Bu dizi için grafik kartı belleğinden yer ayrıldıktan sonra, diziye ait işaretçi GİB üzerinde çalışacak çekirdek fonksiyona parametre olarak gönderilir. Hesaplama bittikten sonra bu dizi, bilgisayar belleğine geri aktarılır. Bu işlem şu kod parçası ile yapılabilir:

```
cudaMemcpy(hostPictBuffer,devicePictBuffer,
sizeof(color3)*RAYCOUNTY*RAYCOUNTX, cudaMemcpyDeviceToHost);
```

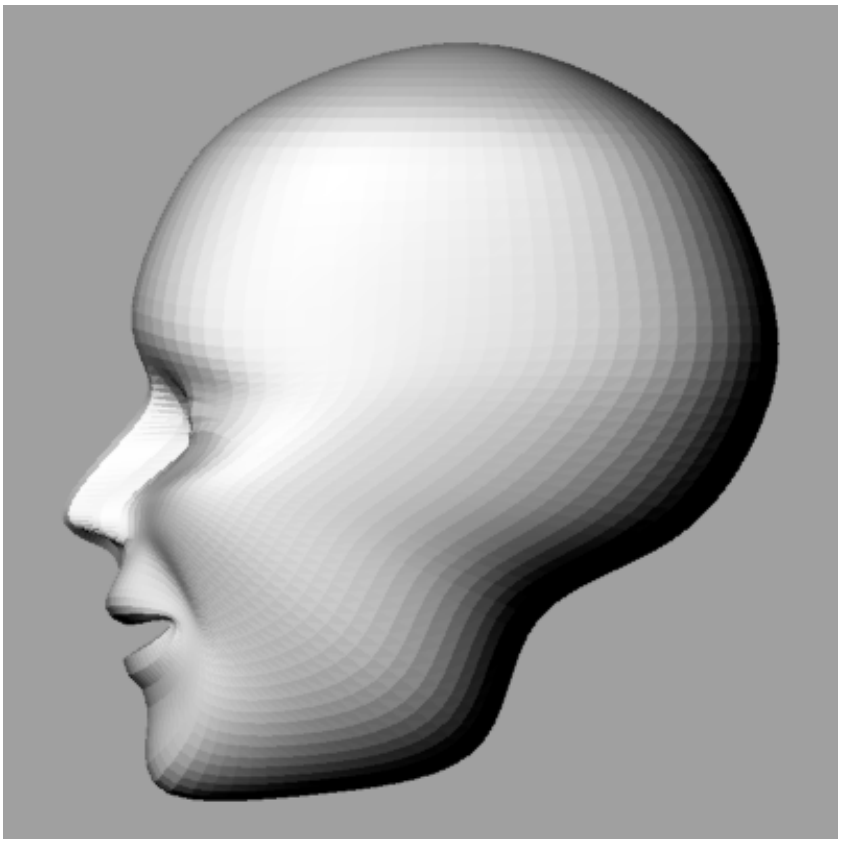
Bilgisayar belleğine aktarılan görüntü, bir bit eşlem resmi olarak sabit diske kaydedilebilir ya da herhangi bir görüntü kütüphanesi kullanılarak görüntülenebilir.

6. UYGULAMA SONUÇLARI

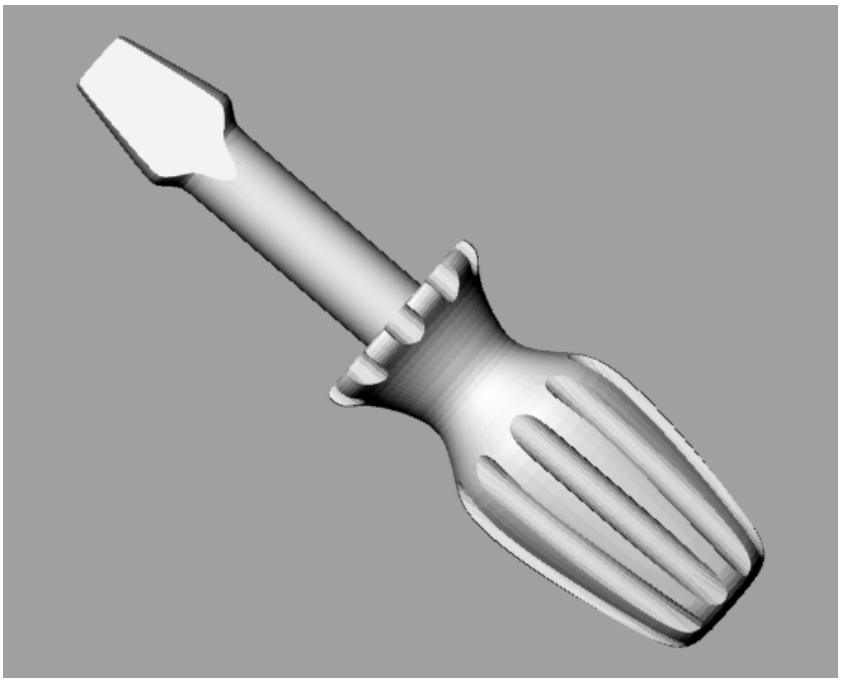
Bu bölümde, GİB üzerinde çalışan ışın izleme uygulaması, aynı uygulamanın MİB üzerinde çalışan sürümü ile karşılaştırılmış ve hızlanma oranları hesaplanmıştır. Ayrıca GİB üzerinde çalışan bu uygulamanın başarımı, ticari bir BDT yazılımı olan ve bir ışın izleme aracına sahip "Rhinoceros"un başarımı ile karşılaştırılmıştır.[14] Karşılaştırmalar, üç ayrı geometri ve beş ayrı çözünürlük değeri kullanılarak yapılmıştır. Kullanılan örnek geometriler, 21182 üçgene sahip bir elipsoid modeli, 14360 üçgene sahip bir insan kafası modeli ve 16400 üçgene sahip bir tornavida modelidir. Her sonuç, 20 koşumun ortalama değerini belirtir.



Şekil 6.1: Örnek elipsoid modeli



Şekil 6.2: Örnek insan kafası modeli



Şekil 6.3: Örnek tornavida modeli

6.1 MİB ve GİB üzerindeki Çalışma Başarımlarının Karşılaştırılması

Bu bölümde, GİB üzerinde çalışan ışın izleme uygulamasının MİB üzerinde çalışan sürümüyle yapılan başarım karşılaştırmasına ve hızlanma oranlarına yer verilmiştir. Söz konusu karşılaştırma değerleri, Çizelge 6.1, Çizelge 6.2 ve Çizelge 6.3'te belirtilmiştir. Kullanılan MİB, Intel Q9550; grafik kartı ise NVIDIA Geforce GTX285'tir. GİB üzerinde çalışan uygulamada, blok başına 128 iplik kullanılmıştır.

Çizelge 6.1: Elipsoid modeli için GİB ve MİB başarımlarının karşılaştırılması

Çözünürlük	**200x200**	**400x400**	**800x800**	**1600x1600**	**2400x2400**
MİB süresi (saniye)	3,203	12,672	50,875	210,531	469,467
GİB süresi (saniye)	0,054	0,203	0,750	2,922	6,469
Hızlanma oranı	59,31	62,42	67,45	72,05	72,57

Çizelge 6.2: İnsan kafası modeli için GİB ve MİB başarımlarının karşılaştırılması

Çözünürlük	**200x200**	**400x400**	**800x800**	**1600x1600**	**2400x2400**
MİB süresi (saniye)	2,109	8,422	34,016	140,063	313,844
GİB süresi (saniye)	0,062	0,140	0,485	2,002	4,226
Hızlanma oranı	34,02	60,16	70,14	69,96	74,26

Çizelge 6.3: Tornavida modeli için GİB ve MİB başarımlarının karşılaştırılması

Çözünürlük	**200x200**	**400x400**	**800x800**	**1600x1600**	**2400x2400**
MİB süresi (saniye)	2,219	8,812	35,422	146,781	329,765
GİB süresi (saniye)	0,035	0,140	0,453	1,859	3,875
Hızlanma oranı	63,4	62,94	78,19	78,96	85,10

Yapılan karşılaştırmalar sonucunda en düşük hızlanma değeri 34,02 ile 200x200 çözünürlüğe sahip insan kafası görüntüsünün oluşturulmasında, en yüksek hızlanma değeri 85,10 ile 2400x2400 çözünürlüklü tornavida görüntüsünün oluşturulmasında gözlemlenmiştir. Ortalama hızlanma değeri 67,40 olup, hızlanma değerleri ışın sayısıyla birlikte artan bir eğilim göstermiştir. Ayrıca, koşum süresinin cismin modellendiği üçgen sayısıyla beraber arttığı gözlemlenmiştir.

6.2 GİB Üzerindeki Başarım ile Ticari Bir Işın İzleyicinin Başarımlarının Karşılaştırılması

Bu bölümde GİB üzerinde çalışan ışın izleme uygulamasının ticari bir ışın izleme uygulamasıyla karşılaştırılmasına yer verilmiştir. Söz konusu ticari uygulamada kullanılan görüntü oluşturma yöntemi tam olarak bilinemediğinden sonuçlar çok kesin bir anlam taşımasa da, geliştirilen uygulamanın niteliğinin anlaşılması açısından önemlidir.

Çizelge 6.4: Elipsoid modeli için GİB ve ticari uygulama başarımlarının karşılaştırılması

Çözünürlük	**200x200**	**400x400**	**800x800**	**1600x1600**	**2400x2400**
Rhinoceros süresi (saniye)	0,263	0,708	2,323	11,551	22,952
GİB süresi (saniye)	0,054	0,203	0,750	2,922	6,469

Çizelge 6.5: İnsan kafası modeli için GİB ve ticari uygulama başarımlarının karşılaştırılması

Çözünürlük	**200x200**	**400x400**	**800x800**	**1600x1600**	**2400x2400**
Rhinoceros süresi (saniye)	0,201	0,572	1,928	7,635	17,659
GİB süresi (saniye)	0,062	0,140	0,485	2,002	4,226

Çizelge 6.6: İnsan kafası modeli için GİB ve ticari uygulama başarımlarının karşılaştırılması

Çözünürlük	**200x200**	**400x400**	**800x800**	**1600x1600**	**2400x2400**
Rhinoceros süresi (saniye)	0,21	0,536	1,679	6,559	15,277
GİB süresi (saniye)	0,035	0,140	0,453	1,859	3,875

7. SONUÇ VE ÖNERİLER

Bu tez çalışmasında, grafik kartı üzerinde çalışan bir ışın izleme uygulaması geliştirilmiştir. Uygulamanın amacı, 3 boyutlu cisimlerden mümkün olduğunca hızlı ve verimli bir şekilde fotogerçekçi görüntüler üretmektir.

Bu çalışmada ilk adım olarak 3 boyutlu cisimler üçgenlerle modellenerek bilgisayara aktarılmıştır. Ardından Kd-Tree uzay bölmeleme algoritmasıyla ışın izleme işleminin verimli olarak yürütülebilmesi için altyapı hazırlanmıştır. Hazırlanan bu yapı, CUDA kullanılarak grafik kartı belleğine aktarılmıştır. Ardından GİB üzerinde ışınları temsil eden iplikler oluşturulmuştur. Her ipliğin, sorumlu olduğu ışını Kd-Tree yapısı üzerinde izleyerek ilgili piksellere ait renk değerlerini bulmaları sağlanmıştır. Son olarak hesaplanan renk değerleri, bit eşlem resmi olarak görüntülenmiştir.

Çalışmanın sonucunda elde edilen uygulamanın başarımı MİB üzerinde ardışıl olarak çalışan sürümüyle ve ticari bir ışın izleme uygulamasıyla karşılaştırılmış ve oldukça yüksek hızlanma değerleri elde edilmiştir.

7.1 Gelecek Çalışmalar

Bu çalışmada geliştirilen ışın izleme uygulaması, grafik kartı üzerinde paralel olarak yürütülebilecek benzer uygulamalar için bir örnek teşkil eder. Bu uygulama, daha gelişmiş ışık benzetimi yöntemleri kullanılarak ve GİB ile grafik kartı belleğinin daha verimli bir şekilde kullanılması sağlanarak mükemmelleştirilebilir.

Bunun dışında bu uygulama, gerekli değişiklikler yapılarak ışın izleme yönteminden yararlanan yüksek frekans elektromanyetik ve akustik dalgaların benzetiminde kullanılabilir.

KAYNAKLAR

[1] **Möller, T. and Trumbore, B.**, 2005: Fast, Minimum Storage Ray/Triangle Intersection, *International Conference on Computer Graphics and Interactive Technique,* Los Angeles, California, USA, July 2005.

[2] **Glassner, A. S.**, 1989: Diffuse Reflection, in *An Introduction To Ray Tracing,* p.144, Morgan Kauffman Publishers, San Francisco, California, USA.

[3] **Finkel, R. A. and Bentle,y J. L.**, 1974: Quad Trees a Data Structure For Retrieval on Composite Keys, *Acta Informatica,* Vol. **4**, no. 1,pp. 1-9.

[4] **Gervautz, M., and Purgathofer, W.**, 1990: A Simple Method For Color Quantization: Octree Quantization, in *Graphics Gems,* pp.287-293, Academic Press Professional, San Diego, California, USA.

[5] **Fuchs, H., Kedem, Z. M. and Naylor, B. F.**, 1988: On Visible Surface Generation by A Priori Tree Structures, in *Tutorial: Computer Graphics; Image Synthesis,* pp. 315-322, Computer Science Press, New York, NY, USA.

[6] **Sherrod, A.**, 2007: *Data Structures and Algorithms for Game Developers,* pp. 39-48, Charles River Media, Rockland, MA, USA.

[7] **Wald, I., and Havran, V.**, 2006: On Building Fast Kd-Trees for Ray Tracing, and On Doing That In O(N log N), *RT'06: IEEE Symposium on Interactive Ray Tracing,* Salt Lake City, UT, USA, September 2006.

[8] **Möller, T.**, 2005: Fast 3D Triangle-Box Overlap Testing, *International Conference on Computer Graphics and Interactive Techniques,* Los Angeles, California, USA, July 2005.

[9] **Jansen, F. W.**, 1986: Data Structures for Ray Tracing, *Data structures for Raster Graphics, Proceedings Workshop,* Berlin, Germany, June 1986.

[10] **Glassner, A. S.**, 1989: Nonuniform Spatial Subdivision, in *An Introduction To Ray Tracing,* pp. 218-223, Morgan Kauffman Publishers, San Francisco, California, USA.

[11] **Smits, B.**, 2002: Efficient Bounding Box Intersection, *Ray Tracing News,* Vol. **15**, no. 1,pp. 7-9.

[12] **Williams, A., Barrus, S., Morley, R. K. and Shirley, P.**, 2005: An Efficient and Robust Ray-Box Intersection Algorithm, *International Conference on Computer Graphics and Interactive Technique,* Los Angeles, California, USA, July 2005.

[13] CUDA C Programming Guide, <*http://developer.nvidia.com/object /cuda_3_0_downloads.html*>, alındığı tarih 03.03.2010.

[14] <*http://www.rhino3d.com/*>, alındığı tarih 01.03.2010.

ÖZGEÇMİŞ

Ad Soyad: Mustafa Alper Çolak

Doğum Yeri ve Tarihi: Gümüşhacıköy/AMASYA, 23.12.1984

Adres: TÜBİTAK Gebze Yerleşkesi Bilişim Teknolojileri Enstitüsü

41470 Gebze/KOCAELİ

Lisans Üniversite: İstanbul Teknik Üniversitesi