YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

# GÖRÜNTÜLEMEDE İLERİ TEKNİKLER

**Bilg. Bil. Müh. Alp-er YURDAKUL**

**F.B.E. Bilgisayar Bilimleri Mühendisliği Anabilim Dalında hazırlanan**

**YÜKSEK LİSANS TEZİ**

**Tez Danışmanı : Y. Doç. Dr. Selim AKYOKUŞ**

Y.Doç.Dr. Tülay Yıldız

Doç. Dr. F. Ayhan Sakarya

**İSTANBUL, 1997**

**İÇİNDEKİLER**

# ŞEKİL LİSTESİ

## TABLO LİSTESİ

## ÖZET

Günümüzde üç-boyutlu bilgisayar grafik dalı, hayali film ve televizyon dünyasından makina mühendisliği parça tasarımlarında kullanılan bilgisayar destekli tasarım yazılımlarına kadar birçok uygulama alanını kucaklamaktadır. Bu bağlamda üç-boyutlu grafik, bilgisayar grafik dalının en önemli koludur. Diğer kollar belirli alanlar üzerinde odaklanmış olmasına rağmen, mimarlar, moleküler bilimadamları ve televizyon animatörleri gibi çok değişik alanlardaki kullanıcılar üç-boyutlu modelleme tekniklerini kullanmaktadırlar.

Üç-boyutlu grafik kapsamında kullanılan metodların birçoğu 10 yıldan daha kısa süre önce geliştirilmişlerdir. Ancak bu konuda birçok mükemmel kitap yazılmış olmasına rağmen, bu kitapların büyük bir kısmı son yıllarda geliştirilmiş teknikler yerine daha çok belli başlı temel konuları içermektedirler. Üç-boyutlu grafik alanında ihtiyaç duyulan bilgilerin büyük bir kısmı araştırma makalelerinden elde edilebilmektedir. Bu gerçek özellikle son on yıl içerisinde geliştirilmiş teknikler (ışın-izleme ve yeni yansıma modelleri) için geçerlidir. Araştırma yazılarından bilgisayar grafik metodları geliştirmek bazen zor ve usandırıcı bir iş olmaktadır. Metod geliştiren kişi için önemli olan detaylar (haklı olarak) bu tip yayınlardan çıkartılmıştır, ve bu alanda çalışma yapan kişiler bir süre sonra kendilerini gerekli pratik hileleri yeniden keşfetmeye çalışırken bulurlar.

Bu çalışmada, varolan ışın-izleme teknikleri incelenmiş ve bazıları daha iyi görüntüleri daha kısa zamanda oluşturacak biçimde geliştirilmiştir. Ayrıca bazı modeller için yeni metodlar geliştirilerek varolan algoritmalardaki eksik noktalar doldurulmaya çalışılmıştır. Bu çalışmada geliştirilen bütün metodlar grafik gösterimleriyle ve (gereken yerlerde) diğer yaklaşımlarla birlikte detaylı olarak açıklanmıştır.

Bu kitabın okuyucusunun temel vektör teorisi (skalar ve vektörel çarpım, normalizasyon) hakkında bilgisi olduğu varsayılmıştır. Ayrıca okuyucunun üç-boyutlu uzayda doğrusal transformasyonlar konusuna yakın olduğu kabul edilmiştir.

## ABSTRACT

Three-dimensional computer graphics now embraces a larger number of application areas, from the fantasy world of film and television to more practical areas such as computer aided design of mechanical engineering parts. In this sense three-dimensional graphics is possibly the most important aspect of computer graphics. While certain aspects are locked into particular areas, users as diverse as architects, molecular scientists and television animators use three-dimensional modelling techniques.

Many of the methods used in three-dimensional graphics are less than 10 years old and, while many excellent textbooks exist, these have mainly been general texts that have dealt with most of the mainstream topic areas in computer graphics. A good deal of the information needed in three-dimensional graphics is to be found only in research papers. This is particularly true of the techniques that have emerged in the last decade; for example, ray tracing and new reflection models. Implementing computer graphics methods from research papers is sometimes a difficult and tedious business. Details that are important to the implementer are quite rightly omitted from such publications, and would-be image synthesizers sometimes find themselves spending an inordinate time rediscovering the important practical tricks that are necessary to produce a rendered image.

In this work, most of the existing ray-tracing methods are studied and some of them are improved to create better images as well as employing a shorter processing time. Also, for some models, some new methods are developed so as to fill in some of the gaps in existing algorithms. All methods developed with this work are explained in detail with graphical diagrams, comparing with different approaches (where necessary).

The reader of this book is expected to have elementary knowledge of vector theory such as the concept of dot and vector products and normalization. The reader is also expected to be familiar with linear transformation in three-dimensional space.

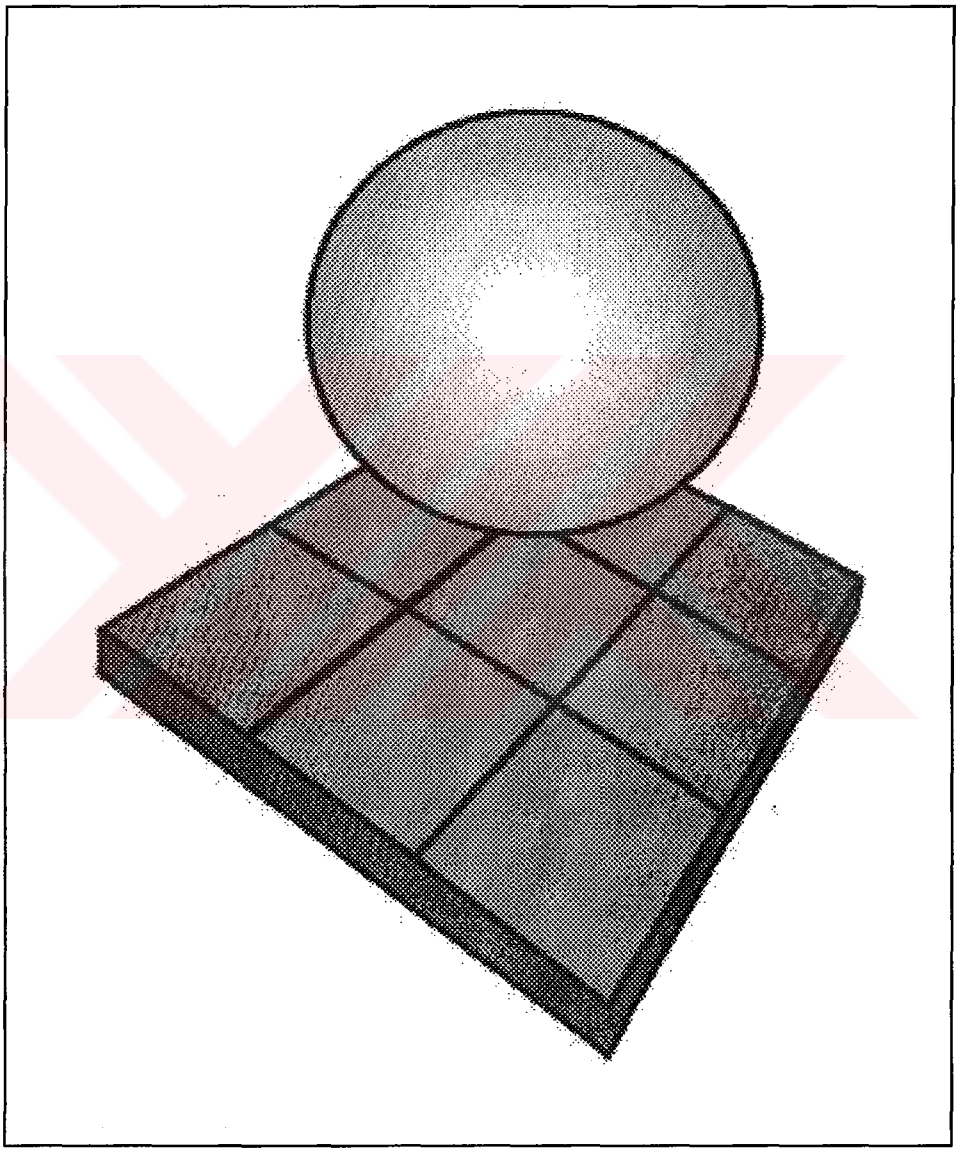
# 1. GİRİŞ

Eğer son on yıl içerisinde sinemaya gittiyseniz veya televizyonda yeni bir film izlediyseniz, mutlaka gelişmiş bilgisayar grafik yöntemlerini iş başında görmüşsünüzdür. Ancak eğer adınızın kısaltması NBC veya MGM değilse, muhtemelen bu tip görüntüler oluşturacak bir bilgisayar sistemi donanımı için, her ne kadar zevkli olursa olsun, ekstra 100.000 $ 'ınız yok demektir. Ancak bu alanda geliştirilen yeni yöntemler sayesinde artık böyle görüntüler elde etmek için bu kadar yüksek miktarlarda paraya ihtiyaç duyulmamaktadır. Günümüzde satılan en ucuz kişisel bilgisayar üzerinde bile birçok amaca yönelik olarak kullanılabilecek bir grafik kartı bulunmaktadır. Ayrıca grafiksel programlamanın yaygınlaşması sayesinde yazılım geliştirmede kullanılan birçok derleyici beraberinde ucuz, hızlı ve kolay kullanılabilen garfik programlama ek yazılımlarıyla gelmektedir. Bilgisayar grafiğinin bu derece ucuzlayıp evlerimize kadar girebilmesi, grafik alanında elde edilen gelişmeler sayesinde olmuştur. Bu gelişmelerin kişisel bilgisayar kullanıcılarına sağladığı özgürlükler beş temel sınıfta toplanabilir:

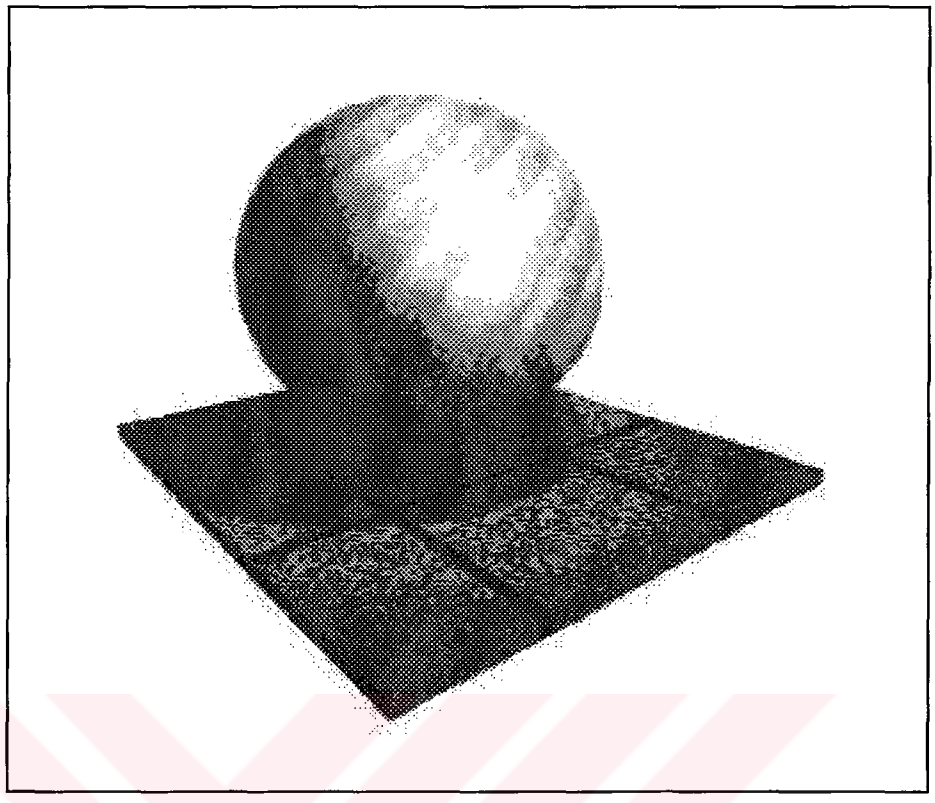
1. Gerçek 3-boyut: Eski-tip bilgisayar grafikleri bilgisayar ekranının 2-boyutlu yapısı ile sınırlanmıştı. Ancak, ileri programlama teknikleri, geliştirilmiş arayüz tasarımları ve hızlı yöntemler sayesinde kişisel bilgisayarımızın ekranı gerçek 3-boyutlu dünyalara açılan bir pencere haline gelebilir. 3-boyutlu nesne oluşturma ve 3-boyutlu uzay içinde dolaşma özgürlüğü kişisel bilgisayar grafiğini tamamen yeni bir boyuta taşımaktadır. 2-boyutlu grafik yöntemleri ile modellenen bir sistem görüntüsü ile aynı sistemin 3-boyutlu metodlarla modellenmesi arasında oldukça büyük fark vardır. Bu fark basit bir sistem üzerinde bilekolaylıkla görülebilir (bakınız Şekil.1.1 ve Şekil.1.2).

2. Foto-gerçekçilik: 1990 öncesinde kullanılan kişisel bilgisayarların büyük bir çoğunluğu 16 renk duvarını aşamamaktaydı. Ancak karmaşık 3-boyutlu nesnelerin modellenmesi ile oluşturulan görüntülerin foto-gerçekçi olabilmesi için gözün

görebildiği bütün renkleri bilgisayar ekranında görmek gereklidir. Grafiksel programlama alanındaki gelişmeler sayesinde artık çok düşük rakamlarla alınabilecek ekran kartlarını kullanarak, bilgisayarınızın ekranında doğadaki gerçek renkleri görebilirsiniz. Böylece ileri tarama teknikleri kullanarak elde edebileceğiniz görüntüler ekran üzerinde de foto-gerçekçi olarak görülebilecektir.



Şekil 1.1. İki-boyutlu perspektif ve gölgelendirme teknikleri kullanılsa bile, ortaya çıkan resim "düz"lükten kurtulamamaktadır.



Şekil 1.2. Aynı sahnenin 3-boyutlu metodlarla modellenmesi, resmin çok daha gerçekçi ve görsel olarak etkileyici olmasını sağlamaktadır.

3. Gerçek-zamanlı hareketli görüntüler: Gerçek dünyada bulunan nesnelerin büyük bır kısmı hareket etmektedir. Bu yüzden hareketli nesneler ne kadar gerçekçi modellenirse modellensinler, eğer ekran üzerinde hareketsiz olarak görünürlerse gerçekçiliklerinin büyük bir kısmını yitirirler. Bu ihtiyaç sayesinde ortaya çıkan yeni donanım ve yazılım atılımları sonucunda istenilen görüntüler ekran üzerinde gerçek hayatta olduğu kadar hızlı gösterilebilirler. Bu sayede elimizdeki foto-gerçekçi görüntüler canlılıklarını yeniden kazanırlar.

4. Gerçek dünya modellenmesi: Bilgisayarlar üzerinde yaratılan siber-uzayı keşfetmek oldukça zevkli olmasına rağmen, gerçek dünyadan bilgisayara bilgi aktarımı, ve bu bilgilerin grafik yöntemleri ile işlenmesi sonucunda, gözümüzün tek başına asla

göremeyeceği sırların ortaya çıkarılması daha da cezbedici olmaktadır. İleri görüntü işleme tekniklerinden 3-boyutlu tıbbi görüntüleme yöntemlerine kadar birçok değişik metod kullanılarak, gerçek dünyadan elde edilen veriler arasında gizli kalmış gerçekler ortaya çıkarılabilir.

5. Gerçek etkileşimlilik: Bilgisayar grafiğinin sağladığı en temel özgürlük aynı zamanda uzun yıllardan beri beklenen ve teknik olarak gerçekleştirilmesi en zor olanıydı. Ekran üzerinde hareket eden görüntülerin gerçek anlamda kontrol edilebilmesi ancak bu modellere gerçek-zamanda müdahele edilebilmesiyle mümkün olmuştur. Her ne kadar bazı teknikler doğrudan etkileşim konusunda biraz yavaş kalsalar da, sürekli olarak geliştirilen yeni metodlar sayesinde kullanıcıların yarattıkları modelleri tam kontrol altında tutmaları sağlanmaktadır.

# 2. GÖRÜNTÜ SENTEZİNE GİRİŞ

Görüntü sentezi, bilgisayar bilimlerinin grafik kolunun bir alt sınıfıdır. Bu sınıfta yapılan çalışmaların esas amacı gerçekçi resimler oluşturabilmektir. Görüntü sentezi metodları geliştirilmeden önce değişik birçok modelleme tekniği kullanılmaktaydı. Bu yöntemlerden en basiti ve en yaygın olanı, modellenecek nesnenin kalem-kağıt yardımıyla çizilmesidir. Ancak bu yöntemin büyük bir dezavantajı vardır: Eğer resim çizme yeteneğiniz çok iyi değilse, çizim anında, çizmeye çalıştığınız nesnenin ne olduğunu sözlü olarak belirtmeniz gerekmektedir. Bu yöntemin bir adım ilerisinde 2-boyutlu bir modelleme programının kullanılması gelmektedir. Bu yöntem sayesinde modellenen nesnenin daha düzgün bir şekilde ifade edilmesi sağlanmış olur. Ancak, her ne kadar bu teknik kağıt-kalem yönteminden daha gelişmiş olmasına rağmen, yine de birçok insan 2-boyutlu bir taslağa bakarak 3-boyutlu bir nesneyi hayal etmekte oldukça zorlanabilir. Ayrıca her insanda bulunan hayalgücü değişik seviyelerde olduğu için insanların kafalarında şekillenen nesneler anlatılmak istenilen nesneden çok farklı olabilir. Bu yüzden, modelleme çalışmalarında üç boyutlu bir modelleme sistemine ihtiyaç doğmaktadır. Bu şekilde ifade edilen nesneler diğer yöntemlere göre daha kolay anlaşılır bir yapıda modellenirler. Eğer modellenilen nesneler küre, küp, prizma gibi üç-boyutlu basit şekiller ise, bu yöntem yeterli gelebilir. Ancak eğer modellenmeye çalışılan nesne yeni bir otomobil tasarımı ise, sadece 3-bayutlu gösterim yeterli olmamaktadır. Oluşturulan modelin değişik parçalarının değişik renklerde boyanması da çok fazla yardımcı olmadığından yeni bir modelleme tekniğine, yani "foto-gerçekçi" modelleme tekniğine, ihtiyaç duyulmuştur.

Foto-gerçekçi terimi, modellenen nesnenin, bu amaçla oluşturulan görüntüsünün gerçek fotoğrafıyla aynı kalitede bir görünüşe sahip olması için kullanılır. Bu niteliğe sahip olan resimler birçok amaç altında değişik kıstaslarla oluşturulabilirler. Reklamcılık, mimari çalışmalar, sanat, bilgisayar destekli tasarım, savunma, eğitim, eğlence, imalat, tıp, bilimsel görüntüleme ve simülasyon gibi birçok alanda bir nesnenin fiziksel olarak varolmamasına rağmen görüntülenmesi gerekebilir. Foto-gerçekçi görüntü sentezi yöntemleri yardımıyla böyle bir nesnenin şekli sentetik olarak elde edilebilir. Bu

yöntemler yardımıyla, tasarlanılan modelin birçok karakteristik yapısı üzerinde kolayca değişiklikler yapılırak elde edilecek nesnenin son hali oldukça kolay ve de masrafsız bir biçmde görülebilir. Böylece hem zaman hem de para kaybının engellenmesinin yanısıra tasarım yapan kişilerin hayal gücü hiçbir şekilde kısıtlanmamış olur.

Görüntü sentezi, bilimsel bir yaklaşımla, gerçek dünyada varolan optik oluşumların simülasyonu olarak da nitelendirilebilir. Işığın dağılımı ve gerçek nesneler üzerinde yarattığı efektlerin fiziksel olarak modellenmesi oldukça karışık olduğundan, hemen hemen bütün uygulamalarda daha basite indirgenmiş matematiksel ve/veya fiziksel modeller kullanılmaktadır. Bu amaçla oluşturulan modellerde ağırlık, olayın fiziksel olarak değil görüntüsel olarak gerçeklik içermesi doğrultusunda olmuştur. Ancak buna rağmen, yine de gerçekçilik ve model karmaşıklığı arasında bir ters orantı mevcuttur: Görüntülerin daha gerçekçi olması için daha karmaşık model ve algoritmalar kullanılması gerekmektedir ki, bu da daha fazla hesap zamanı ve bellek kullanımını doğurmaktadır.

## 2.1. Görüntü Sentezi Problemleri

Görüntü sentezinde temel olarak karşılaşılan problemler, gerçek dünyada ışığın dağılımının simülasyonuna dayanarak üç sınıfta incelenebilir:

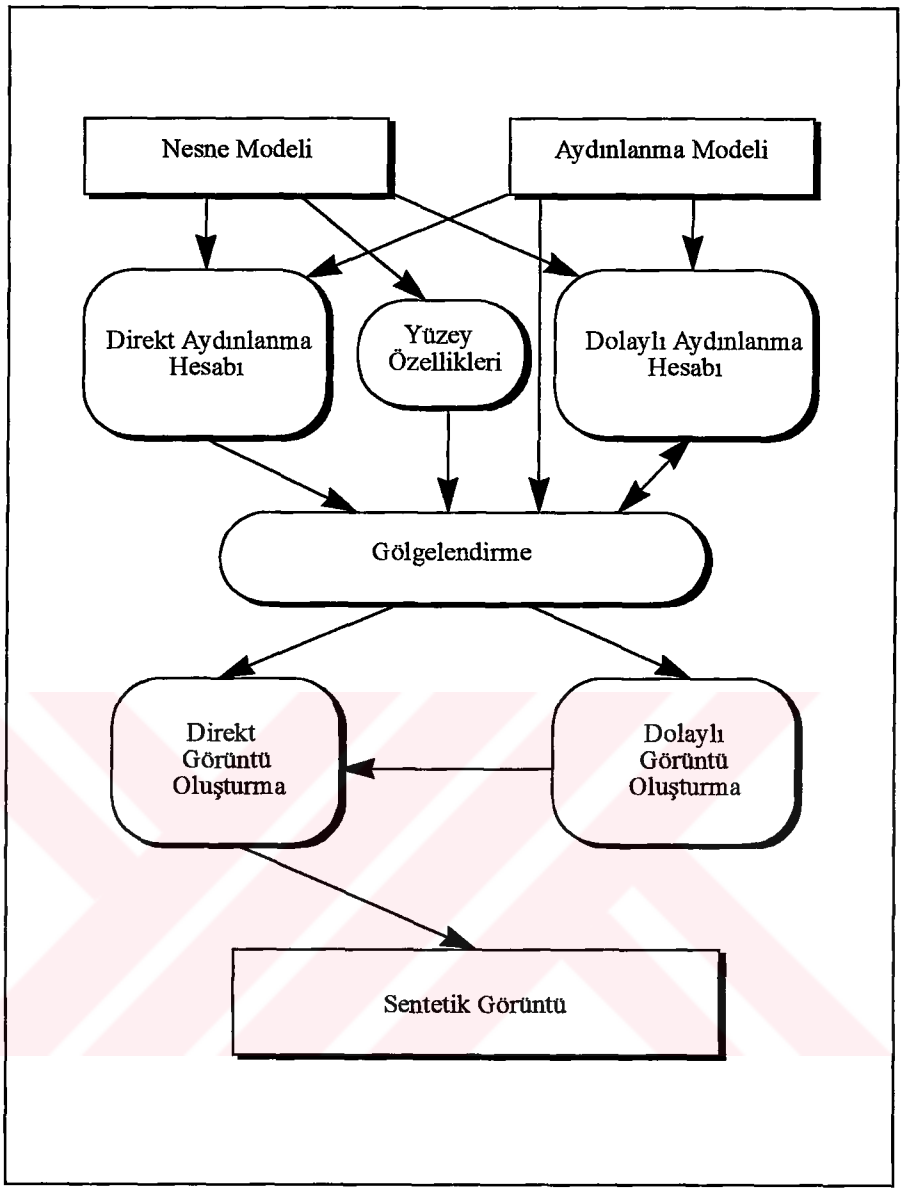
1. Görüntü oluşturma: Görüntüsü oluşturulacak sahnenin, bakılan yere bağlı olarak belirlenen, elemanlarının modellenmesi. Bu problemde amaç, ışığın görünen nesneler üzerinde direkt olarak yaptığı optik etkinin (direkt görüntü oluşturma - direct image formation) yanısıra, bakılan noktadan görünmeyen, ancak yine de görünen diğer nesneler üzerindeki ışık dağılımını etkileyen nesneleri analiz etmektir. Direkt görüntü oluşturma, görüntü işlemede, saklı yüzey problemi olarak da karşımıza çıkar. Dolaylı yollardan aydınlanma içeren görüntülerin oluşturulması ise aynalı yüzeylerden

kaynaklanan yansımalar, ve saydam nesnelerin oluşturduğu kırılım hesaplarını içerir.

2. Aydınlanma: Görüntüsü oluşturulacak sahnenin, bakılan yere bağlı olmadan belirlenen, ışığın dağılımının oluşturduğu efektler. Bu problemde amaç, bir nesne üzerine ışık kaynağından direkt olarak veya diğer nesneler üzerinden yansıyarak gelen ışığın oluşturduğu aydınlanmayı hesaplamaktır. Direkt aydınlanma genellikle gölge problemi olarak karşımıza çıkar ve ışık kaynağı tarafından görülebilen nesnelerin belirlenmesi olarak nitelendirilebilir. Dolaylı aydınlanma ise diğer nesneler üzerinden gelen yansımanın, ve saydam ve/veya yarı-saydam nesneler içinden geçerek gelen ışığın oluşturduğu aydınlanmanın bulunmasıdır. Bu problemde ayrıca, her nesnenin diğer nesneleri görüp göremediği hesaplanmak zorundadır.

3. Gölgelendirme: Bir yüzeyi terkeden ışık yayılımının, o yüzeyin optik özellikleri ve gelen ışık miktarına bağlı olarak bulunması. Bu problem en genel haliyle, belirli bir iki-yönlü dağılım fonksiyonu ile ifade edilebilir. Bu fonksiyonda, nesne üzerindeki herhangi bir noktaya gelen ışığın geliş ve yüzeyi terkediş açısı kullanılarak o noktadaki aydınlanma ifade edilebilir.

Görüntü sentezinde karşılaşılan bu temel işlemler Şekil.2.1 üzerinde gösterilmiştir. Bu işlemlerin çoğunda karşımıza görünen noktaların bulunması problemi çikar. Hemen hemen bütün görüntü sentez sistemlerinde en fazla zaman ve emek bu işlem sırasında harcanır.

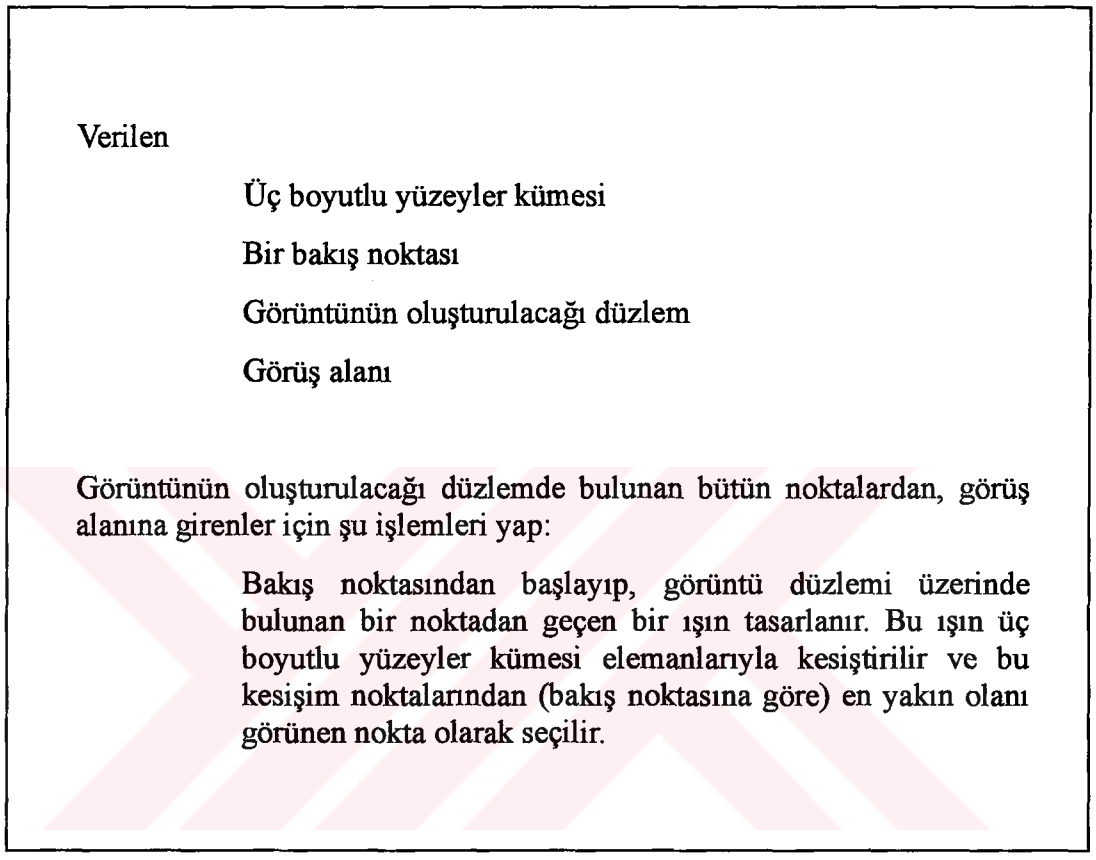



Şekil 2.1. Görüntü sentezinde karşılaşılan alt-problemler

## 2.2. Görünen Noktaların Bulunması

Görünen nokta tespiti, görüntü sentez işlemlerinde birçok şekilde ve yerde karşımıza çıkar. Bu problem en basit şekliyle, belirli bir anda sabit bir noktadan görülebilen noktaların bulunması olarak ifade edilebilir. Bu ifade Şekil.2.2 üzerinde basit olarak

gösterilmiştir. Kullanılan modeller karmaşıklaştıkça, bu problem de daha yüksek seviyeli görünebilirlik terimleri (bulanık ortamlar, odaklanmış nesneler, dolaylı yoldan aydınlanmış yüzeyler, vb.) içermeye başlayacaktır.




Şekil 2.2. En basit haliyle Görünen Nokta Problemi

En basit haliyle Şekil.2.2 de görüldüğü biçimde ifade edilen bu problemin çözümünde kullanılabilecek algoritmalar iki temel sınıfta incelenebilir: Sürekli Algoritmalar ve Nokta-örneklemeli Algoritmalar. Her iki sınıfta kullanılan yöntemler bu probleme çok farklı yönlerden yaklaşırlar. Sürekli algoritmalar, bütün görüntü düzlemini kapsayan sürekli alanlar üzerinde çözüm kümesini ararlar. Yerine ve büyüklüğüne bakılmaksızın, bütün yüzeylerin görünen her parçası bu algoritmaların oluşturduğu çözüm kümesine dahil edilir. Doğru sonucun elde edilmesi için sürekli filtreleme ve gölgelendirme

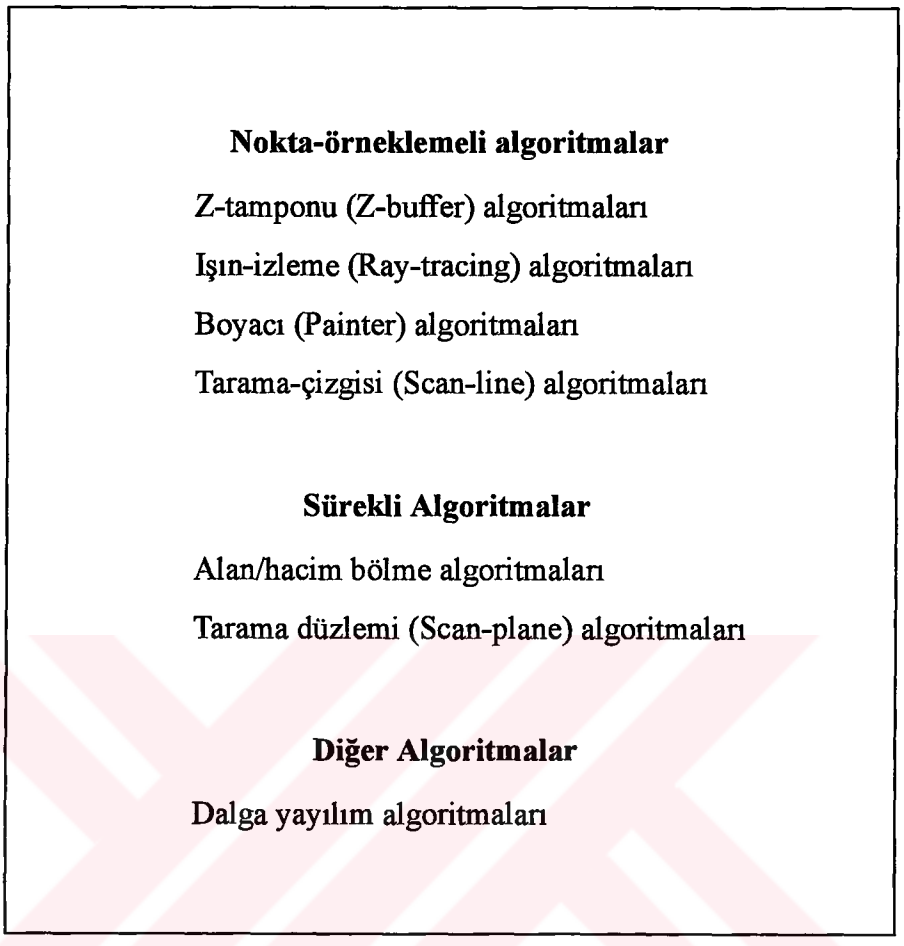
algoritmaları bu işlemler sırasında kullanılabilir. Nokta-örneklemeli algoritmalar ise görünen nokta bulma problemine yaklaşık bir sonuç getirmek amacıyla kullanılır. Bu sınıfa ait algoritmalar, üç boyutlu yüzeyler kümesinde belirlenen sonlu sayıda örnekler üzerinde çalışırlar ve bu örneklere dayanarak nesnelerin hangi bölümlerinin görünüp görünmediğine karar verirler.

Herhangi bir sürekli algoritma, sürekli çıkışın örneklenmesi suretiyle nokta-örneklemeli algoritma haline dönüştürülebilir. Belki dönüşüm için daha efektif bir çözüm de bulunabilir, ama çıkışın örneklenmesi her zaman işe yarar. Ancak bu işlemin tersi, yani nokta-örneklemeli bir algoritmadan sürekli çıkış almak, her zaman bu kadar kolay değildir.

Pratik kullanımda, çok karmaşık olduklarından ve sınırlı sayıda görüntü efekti destekleyebildiklerinden dolayı, sürekli algoritmalar çok fazla kullanılmazlar. Bu tip algoritmalarda genellikle dolaylı aydınlanma içermeyen poligonlar kullanılır. Ancak nokta-örneklemeli metodlar, hem çözümü çok karmaşık olmayan hem de oldukça ileri seviyede görüntü efekti içeren modeller (kavisli yüzeylerde oluşan çoklu yansıma ve kırılım efekti gibi) kullanılmasına imkan tanırlar. Bu tip efektlerin çoğu sürekli algoritmalar tarafından gerçeklenememektedir.

Görünür yüzey algoritmalarının temel olarak ayrılabileceği sınıflar Şekil.2.3 üzerinde gösterilmiştir.

Nokta örneklemeli ve sürekli algoritmalar, ışık dağılımını çok basit bir şekilde modelleyerek çalışırlar. Bu modele göre ışık, homojen bir ortamda düz ışınlar halinde yayılır ve cisimlerin yüzeylerinde yarattığı etki geometrik optik kuralları sınırları içinde tanımlanır. Kırınım, faz farkı, polarizasyon, kuvantizasyon, ışığın dalga boyu ve nesne boyu arasındaki ilişki gibi konular bu modelde gözönüne alınmaz. Bu unsurlar dalga yayılım algoritmaları tarafından kullanılır.




Şekil 2.3. Görünür yüzey algoritma sınıfları

## 2.3. Nokta-örneklemeli Algoritmalar

Günümüzde kullanılan birçok görüntü sentez sisteminde genel olarak dört tip metod kullanılmaktadır. Bu dört algoritmayı birbirinden ayıran temel unsur sıralama biçimlerinde ortaya çıkar. Sıralama biçimlerinin daha açık ifadesi Şekil.2.4 üzerinde gösterilmiştir.

**Z-tampon Algoritması**

Bütün nesneler için

    Bütün (x,y) ler için

        En küçük z değerli noktayı seç

**Işın-izleme Algoritması**

Bütün (x,y) pikselleri için

    Bütün nesneler için

        En küçük z değerli noktayı seç

**Boyacı Algoritması**

Bütün nesneleri z değerlerine göre sırala

Bütün nesneler için

    Bütün (x,y) leri seç

**Tarama-çizgisi Algoritması**

Bütün nesneleri y değerlerine göre sırala

Bütün y değerleri için

    Bütün nesneleri x değerine göre sırala

    Bütün x ler için

        En küçük z değerli noktayı seç

Şekil 2.4. Nokta-örneklemeli algoritmalar

Z-tampon ve ışın-izleme algoritmaları, kullandıkları metod bakımından benzer olmalarına rağmen, aralarındaki temel fark, döngüler içinde yapılan işlemlerin sırasıdır. Z-tampon algoritması, bir seferde bir nesnenin kapsadığı bütün (x,y) çiftleri üzerinde çalışırken, ışın-izleme yönteminde ise her (x,y) çifti için bütün nesnelerin bakış noktasına olan uzaklıkları karşılaştırılır.

Z-tampon ve boyacı algoritmaları, derinlik karşılaştırmaları bakımından birbirlerinden farklılık gösterirler. Z-tampon algoritması derinlik karşılaştırmalarını iç döngüde yaparken, boyacı metodunda ise bu karşılaştırmalar önceden yapılan sıralama işlemi sırasında gerçekleştirilir.

Tarama-çizgisi ve Z-tampon algoritmaları, görüntü düzlemi üzerinde y ve x doğrultuları üzerinde yaptıkları işlemlerin sırası bakımından farklılık gösterirler.

Yukarıdaki paragraflarda da görülebileceği gibi nokta-örneklemeli algoritmalar temel bazı farklılıklar dışında birbirleri ile aynı özellikleri taşımaktadırlar. Bu çalışmada nokta-örneklemeli algoritmalardan, ışın-izleme yolu ile görüntü oluşturma metodu incelenecektir.

# 3. GÖRÜNTÜ VE IŞIK

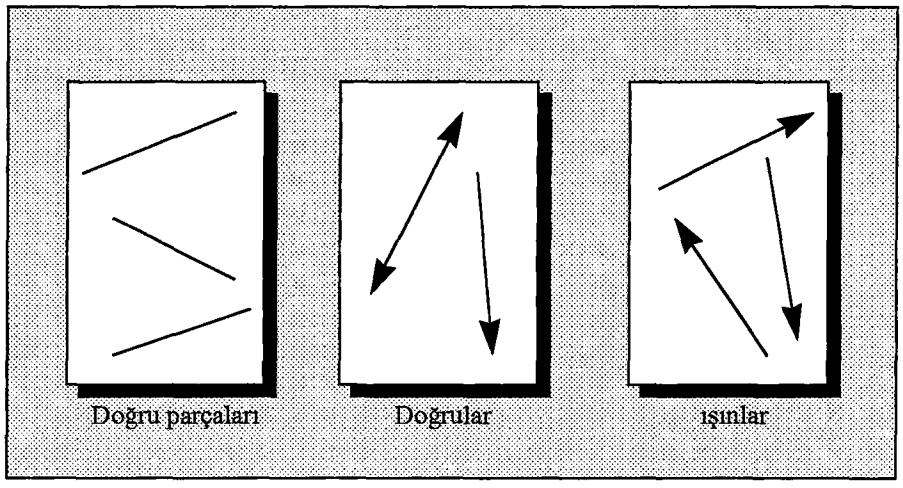
Fiziksel bir olgu olan ışık ile bu olgunun gözlemlenmesinden elde edilen görüntü arasında güçlü bir ilişki vardır. Gerçeğe yakın bir görüntü oluşturmak için, ışığın görme sistemimizde yarattığı uyarmaların bilgisayar üzerinde simülasyonu gerekmektedir. Ancak gerçek anlamda insanda bulunan türden bir görme sistemi oluşturmadan, beynin gelen uyarmaları anlayıp görüntü olarak yorumladığı yapıya benzer bir yapı oluşturmak yeterlidir.

Ressamlar ışık, renk, gölge, yansıma ve perspektif gibi olaylarla genellikle sadece belirli bir çerçeve dahilinde ilgilenirler. Ancak ışın-izleme metodu bu çerçeveden biraz daha kapsamlı olarak çalışmak zorundadır, çünkü işlemler sırasında ışık gerçek haliyle, yani fizik kurallarına uyan bir olgu şeklinde, simüle edilir.

Modern fizik, ışığı, foton adı verilen küçük enerji parçaları şeklinde ifade eder. Ayrıca bu parçaların dalga şeklinde bir yapıda olduğu kabul edilmiştir. Işığın bu ikili yapısı (parça ve dalga), bütünüyle fiziksel yapısıyla modellenmesini çok zor (hatta imkansız) hale getirmektedir. Işık modelimizi foton yönünü kapsayacak şekilde tanımlasak bile çözülmesi gerekli birçok problem ortaya çıkar. Işık kaynağı belirli bir anda her yöne sayısız miktarda foton yaymaktadır. Her fotonun, çarptığı yüzeyden belirli bir oranda yansıyarak, tekrar nesne uzayına geri döndüğünü varsayarsak, bu şekilde karşımıza gelen ve sayısal olarak ifade edilemeyecek kadar büyük miktardaki fotonları gerçek şekliyle simüle etmenin imkansız olduğu görülecektir. Problemin bir şekilde basite indirgenmesi gerekmektedir.

Işığın foton ve dalga şeklinde tanımlanmasından önce ışınlar halinde yayıldığı kabul ediliyordu. Bu model ile foton yapısı birleştirilirse, her fotonun herhangi bir yüzeye çarpana kadar belirli bir ışın üzerinde dağıldığı bilgisi elde edilir. Bu model kullanılarak, ışık kaynaklarından çıkan her bir foton ile ilgilenmek yerine bu fotonların oluşturduğu ışınlar ile ilgilenilmesi sağlanmış olur.

Işın kavramının daha iyi algılanabilmesi için, doğru parçası, doğru ve ışın gösterilimi Şekil.3.1 üzerinde ifade edilmiştir.




Şekil 3.1. Doğru parçası, doğru ve ışın karşılaştırılması

Işık kaynağından yayılan ışık ışınları bir yüzeye çarptığı anda birçok olay meydana gelebilir:

1. Işık yansıyabilir: Yüzeye çarpan ışınlar birçok yönde yansıyabilirler. Bazı ışınlar gözümüz doğrultusunda yansır ve bizim onları (yansıdıkları yüzeyi) görmemizi sağlar. Aslında gerçek anlamda gördüğümüz yüzey değil, yansıyan ışıktır. İleriki konularda bu ayrımın önemi daha açık olarak ifade edilecektir.

2. Işık emilebilir: Çarpan ışık ışınının enerjisi ısıya dönüşür ve ışının ömrü sona erer. Siyah görünen nesneler bu olaydan dolayı bu rengi almıştır. Siyah yüzeye çarpan ışık ışınları bu yüzey tarafından emilirler ve gözümüze çok az veya hiç yansıyan ışık

gelmediğinden yüzeyi renksiz (siyah) görürüz.

3. Işık iletilebilir: Saydam/yarısaydam yüzeylere çarpan ışık ışınları bu yüzey içinden belirli bir oranda kırılarak geçer ve yollarına devam ederler.

4. Işık yeniden yayılabilir: Floresan özellikli bir nesneye çarpan ışık ışını yüzey tarafından emilir ve tekrar yayılabilir.

Sözkonusu herhangi bir ışık ışını için yukarıda bahsedilen olaylardan biri veya bir kısmı bir anda belirli oranlarda gerçekleşebilir. Bunun sebebi ışık ışınının fiziksel olarak fotonlardan oluşmasıdır. Yüzeye çarpan fotonlardan bir kısmı yansıyabilir, emilebilir, iletilebilir, kırılabilir veya yeniden yayılabilir.

Bütün bu olaylar matematiksel olarak ifade edilebilir. Örneğin ışınlar doğru denklemine benzer bir yapıda tanımlanabilir. Küre, düzlem ve geometrik yüzey gibi nesneler için matematiksel ifadeler bulunabilir. Yansıma ve kırılma gibi olaylar fiziksel kurallarla modellenebilir. Bu şekilde matematik ve fizik kuralları kullanılarak, ışık kaynakları, nesneler ve gözümüzün nerede olduğu bilgisayara aktarılabilir, ve bundan sonra ışık kaynağından başlayıp nesneler üzerinden yansıyarak gözümüze kadar gelen ışık ışınlarının zaman alıcı hesabı bilgisayara bırakılabilir.

Işın izleme metodunun bu şekilde ifadesi oldukça basit görünmesine rağmen uygulaması bu kadar kolay değildir. Işık kaynağından çıkan herhangi bir ışın gözönüne alınırsa, bu ışın yukarıda bahsedilen olayların birini veya hepsini sayısız defalar gerçekleyeceğinden hesaplamalar imkansız denecek kadar zorlaşmaktadır.

Bu haliyle çözülemeyecek problem biraz daha basite indirgenebilir. Görüntünün oluşmasına katkıda bulunan ışık ışınları sadece gözümüze kadar gelebilen ışınlardır. Diğer ışınların elde edilen resme katkısı görebildiğimiz ışınların katkısının yanında ihmal edilebilecek kadar azdır. Bu şekle indirgenen problem ışığın fiziksel yayılımının ters

yönünden gidilerek çözülebilir. Yani sadece gözümüze gelen ışınları hesaba katarak çözümü bulabiliriz. Bu yöntemle ışınlar, bakış noktasından başlayıp nesneler üzerinden yansıyarak ışık kaynağında son bulur. Böylece sadece gözümüze kadar ulaşabilen ışınlar hesaplanarak işlem bilgisayar üzerinde gerçekleştirilebilir hale getirilir.

Bundan sonraki bölümlerde ışın-izleme yöntemi bu haliyle gözönüne alınarak çözüme gidilecektir.

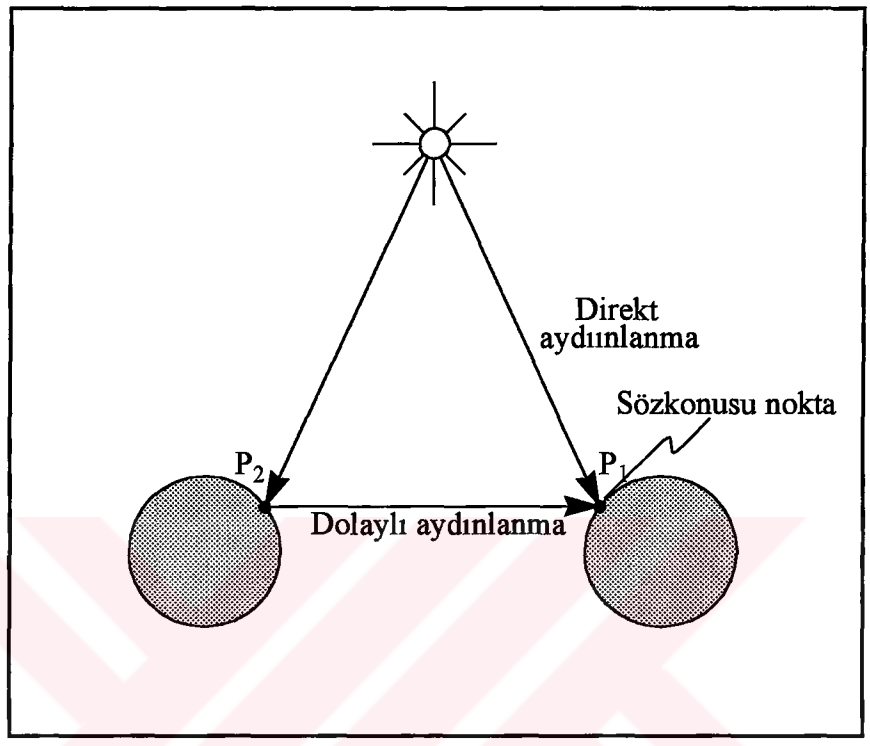
Bu bölümün geri kalan kısmında yansıma modellerinin teorik yanları ve bu modellerin bilgisayar grafik sistemlerine uyarlanması incelenecektir. Bir yansıma modeli, ışık ve yüzeyin etkileşimini yüzeyin özellikleri ve ışığın doğası cinsinden ifade eder. Üç boyutlu ifadesi oluşturulan bir nesnenin dış görüntüsünü kullanılan yansıma modeli belirler. Bu modeller yardımıyla, iki boyutlu ekran düzlemi üzerinde oluşturulan üç boyutlu nesnelerin fotoğrafik gerçekçi olarak görüntülenmesi sağlanır.

Aydınlanma modeli ise yansıma modelinin tersine, bir kaynak sayesinde oluşan ışık ışınlarının doğasını ve yayılma geometrisini tanımlar. Geliştirilen grafik sistemlerinin büyük bir çoğunluğu basit yansıma modelleri ve çok ilkele indirgenmiş aydınlanma modelleri (örneğin nokta kaynaklar) kullanırlar. Bu tip indirgeme günümüzde kullanılan donanım ve görüntü sistemlerinin kapasiteleri yüzünden yapılmaktadır, ve kullanıcılar tarafından kabul edilebilecek nitelikte gerçeğe yakın görüntüler oluşturabilecek kapasitededirler.

Burada bahsedilen "gerçeğe yakın" terimi kullanılan sistemin amacına göre değişebilir. Ne kadar fazla gerçeklik istenilirse, o kadar karmaşık yansıma modeli kullanılmalıdır. Tabii bu da çok daha fazla işlem zamanı anlamına gelmektedir.

Bu sistemde kullanılan yansıma modelleri bölgesel model olarak tanımlanabilir. Bu tip modellerde, sözkonusu olan nesne yüzeyindeki bir nokta ile ışık kaynakları hesaba katılmaktadır. Burada aydınlanma değeri bulunacak noktalara dolaylı yollardan (diğer nesneler üzerinden) gelen aydınlanma etkileri gözönüne alınmaz. Hesaplarda sadece ışık

kaynaklarından direkt olarak gelen ışık ışınları kullanılır. Bu tip aydınlanmalar arasındaki fark Şekil.3.2 üzerinde gösterilmiştir.



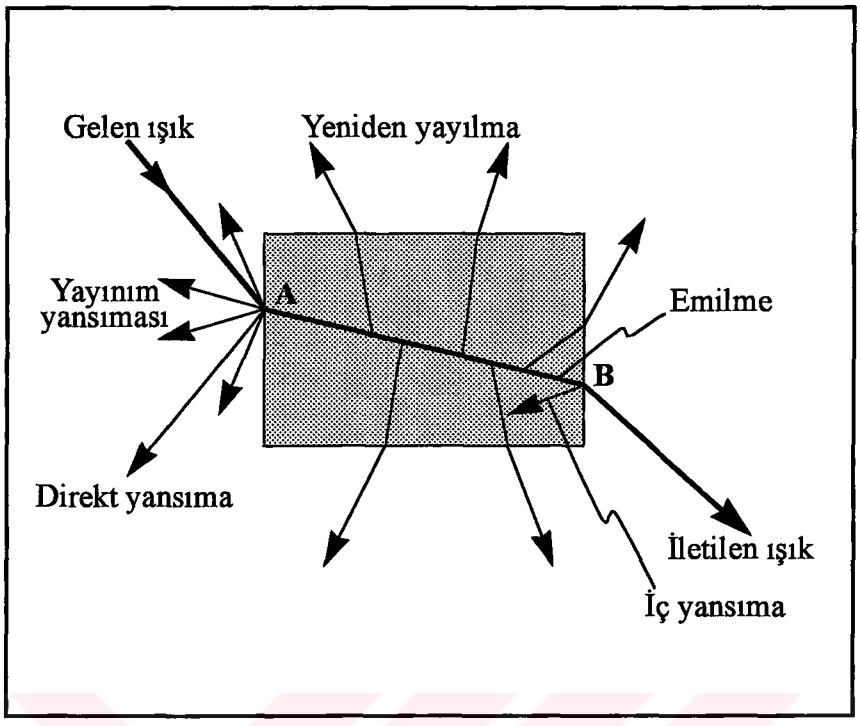

Şekil 3.2. Direkt ve dolaylı aydınlanma

### 3.1. Işığın yansımasında kullanılan teorik kavramlar

Işık ile ilgili sistemlerde kullanılan en temel denklem şu şekilde ifade edilebilir:

Yüzeye geneln ışık miktarı = yansıyan ışık
+ yeniden yayılan ışık
+ emilen ışık
+ iletilen ışık

Bu denklem Şekil.3.3 üzerinde şematik olarak gösterilmiştir.




Şekil 3.3. Işığın katı bir cisim ile etkileşimi

Bilgisayar grafik sistemleri genellikle, bu şekil üzerinde gösterilen kavramlardan sadece yansıyan ve iletilen ışık ışınları üzerinde durmaktadır. Yüzeyden yansıyan ışığın şiddeti ve dalga boyu, gelen ışığın dalga boyuna, geliş açısına, yüzeyin dokusuna ve elektriksel özelliklerine (geçirgenlik, iletkenlik, vs) bağlıdır. Yansıyan ışığın değerini kesin olarak bulmak oldukça karmaşıktır. Örneğin, bir yüzey bazı dalga boylarına göre sert bazı dalga boylarına göre ise düzgün bir aydınlanma sergileyebilir. Veya aynı yüzey aynı dalga boyunda olan ışık ışınlarının farklı geliş açılarına göre farklı aydınlanma değerleri gösterebilir.

# 4. IŞIN İZLEME (RAY TRACING) YÖNTEMİ

Işın izleme metodları, geometrik optik kurallarını kullanırlar. Ekran üzerindeki her piksel için, belirli bir bakış noktasından çıkan sonsuz küçük genişlikteki bir ışın, bu piksel içinden geçerek nesne uzayına gönderilir. Tanımlı bütün nesneler için bu ışın test edilir, ve nesneler ile ışının kesişim noktaları (eğer varsa) bulunur. Bu kesişim noktaları arasında bakış noktasına en yakın olanı seçilerek ekran üzerindeki pikselin renk değerinin hesaplanmasında kullanılır. Bu kesişim noktasından ışık kaynağı doğrultusunda ikinci bir ışın başlatılır ve yol boyunca başka bir nesne ile kesişip kesişmediğine bakılır. Eğer arada başka bir nesne varsa, ilgili noktanın ışık kaynağını direkt olarak görmediği (gölgede kaldığı) anlaşılır ve renk değeri kullanılan aydınlatma modeline göre gerçek değerinden biraz düşürülerek bulunur. Ayrıca nesnenin yüzey özelliklerine göre, yansıma doğrultusunda da bir ışın oluşturulup, bu ışının kestiği en yakın yüzeye ait renk değeri de pikselin toplam renk değerini bulmada kullanılabilir. Ancak, bu şekilde yansıma doğrultusunda kurulacak ışınlar da kendilerine ait gölge ve yansıma ışınları doğuracağından, kullanılan algoritma iteratif bir yapıya bürünür. Benzer yöntemler saydam nesne yüzeylerinde oluşan kırılım efektlerinin hesaplanmasında da kullanılabilir.

Gölge, yansıma ve kırılım gibi olayları gözönüne alan algoritmalar her piksel için bir ışın ağacı kurar. Bu ağacın kolları, ilgili piksel üzerinde elde edilen bütün ışınlar ile ilgili bilgileri, düğüm noktaları ise bu ışınlara ait aydınlanma bilgisini içerir. Piksele ait toplam aydınlanma şiddetini bulmak için ağacın bütün kolları ve düğüm noktaları taranır ve ilgili değerler toplanarak ilgili pikselin aydınlanma değeri elde edilir.

Önceki paragraflarda ifade edilen ışın-izleme algoritmasının gerçeklenmesi pek zor değildir. Şekil.4.1 üzerinde bu yöntemi gerçekleyen bir akış ana hatlaryla verilmiştir. Bu algoritma geometrik optiğin kurallarına dayanarak inşa edilmiştir ve istenildiği taktirde birçok görüntüsel efekt ile bütünleştirilerek kullanılabilir. Ancak bu yöntem, şu ana kadar belirtildiği şekliyle, iki ana probleme çözüm getirememektedir. Bu problemlerden ilki, algoritmanın temelinden gelen örnekleme mantığı sebebiyle bazı noktalardaki süreklilik

kaybı; diğeri ise oldukça büyük işlem zamanı gerektiren ışın-yüzey kesişim noktası hesabıdır.

Örneklemeden kaynaklanan görüntüsel bozukluklar bir piksel için birden fazla ışın üretmekle çözülebilir. Bir piksel için üretilmiş alt-ışınlardan kaynaklanan alt-aydınlanma değerlerinin ağırlıklı toplamı bulunarak, ilgili pikselin aydınlanma değeri gerçek değerine daha yakın olacak şekilde bulunabilir. Böylece bir piksele karşılık düşen nesne uzayındaki gerçek aydınlanma dağılımı daha gerçekçi bir şekilde görüntü düzlemi üzerine aktarılmış olur. Ancak piksel başına birden fazla ışın üretmek bazı durumlarda gereğinden fazla zaman kaybına yolaçabilir. Bu şekilde ek ışın üretme işlemi, sadece piksel ile komşuları arasında çok fazla bir aydınlanma fark olduğu zaman yapılmalıdır. Aksi taktirde sonucu değiştirmeyecek, gereksiz fakat zaman kaybettirici işlemler oldukça fazla sayıda gerçekleştirilmiş olacaktır.

Işın-izleme yönteminde ek ışın üretme sadece piksel bazında yapılan bir işlem değildir. Değişik görüntüsel efektler için değişik sayıda ve değişik koşullarda ek ışın üretilebilir. Örneğin farklı zaman aralıklarında ek ışınlar üretilerek hareket eden nesneler modellenebilir; bakış noktası etrafında ek ışınlar üretilerek derinlik efektleri, veya, ışık kaynağı etrafında ek ışınlar üretilerek yumuşak gölge geçişleri elde edilebilir.

Işın-yüzey kesişim noktası hesabı, ışın-izleme algortmasının zaman yönünden karmaşıklığını direkt olarak etkileyen en büyük (hatta tek) faktördür. Işın-küre ve ışın-silindir testleri diğer yüzeylere göre daha hızlı olarak hesaplanabilir. Ancak başka geometrik yapıda olan nesneler ile yapılan kesişim noktası testleri oldukça fazla zaman almaktadır. Bu yüzden bu testleri hızlandracak, hatta gerekmedikçe hiç yapmayacak yöntemler geliştirilmişitir.

```
Işın-izleme Algoritması
    Başla
        Tanımlı bütün P pikselleri için {
            Bakış noktasından başlayıp P'den geçerek nesne uzayına
            doğrultulmuş bir R ışını tanımla
            Aydınlanma= İzle ( R )
            P'nin rengi Aydınlanma değeri kullanılarak bulunur.
        }
    Son

Prosedür İzle ( R : Işin ) : Dönüş değeri Aydınlanma
    Başla
        Görüntüde tanımlı bütün E elemanları için {
            E ve R arasındaki kesişim noktalarını (eğer varsa) bul
            }
        Eğer R hiçbir E ile kesişmiyorsa {
            Fon rengini (aydınlanma değerini) döndür.
            }
        Eğer E ile R arasında en az bir kesişim noktası varsa {
            En yakın kesişim noktasını oluşturan E elemanını bul
            I1= Kesişim noktasındaki I aydınlanma değeri
            I2= İzle ( YansımaDoğrultusundakiIşın )
            I3= İzle ( KırılımDoğrultusundakiIşın )
            Aydınlanma= AydınlanmaModeli (I1, I2, I3)
            Aydınlanma değerini döndür
            }
    Son
```

Şekil 4.1. Işın-izleme algoritması (Gölgelendirme ihmal edilmiştir)

Birçok ışın-izleme uygulaması sadece üretilen ışının yakınında bulunan nesnelerle ilgilenmiştir. Örneğin her nesne için, o nesneyi çevreleyen bir kutu olduğu varsayılmış, ve üretilen ışın ile bu kutu kesiştirilmiştir. Eğer ışın çevreleyen kutuyu kesmiyorsa, kutu içndeki nesne ile de kesişmez demek olduğundan bu nesne kolaylıkla ihmal edilerek bir sonraki nesne üzerinde işlem yapılmıştır.

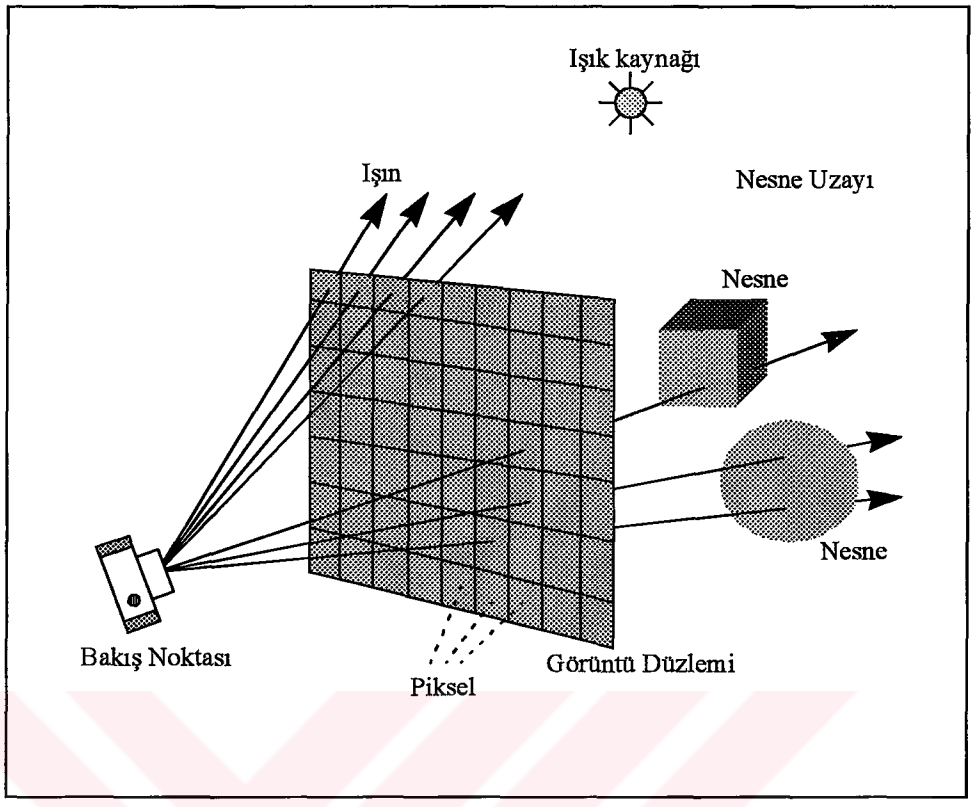
Bazı ışın-izleme uygulamaları ise nesne uzayını birbiriyle kesişmeyen üç boyutlu hücrelere bölmüştür, ve her hücre için bu hücreyi kesen yüzeylerin listesini tutmuştur. Böylece, üretilen her ışın için, bu ışının yolu üzerindeki hücreleri kesen yüzeyler ile kesişim hesabı yapılmıştır. Bu şekilde ışın-yüzey kesişim noktası bulunuşu sadece gereken hücrelerde yapılarak toplam işlem zamanı oldukça azaltılmıştır.

Işın-izleme metodunun daha iyi algılanabilmesi için Şekil.4.2 üzerinde en basit haliyle bir ışın-izleme senaryosu kurulmuştur. Bu şekil üzerinde görülen kavramlar şu şekilde tanımlanabilir:

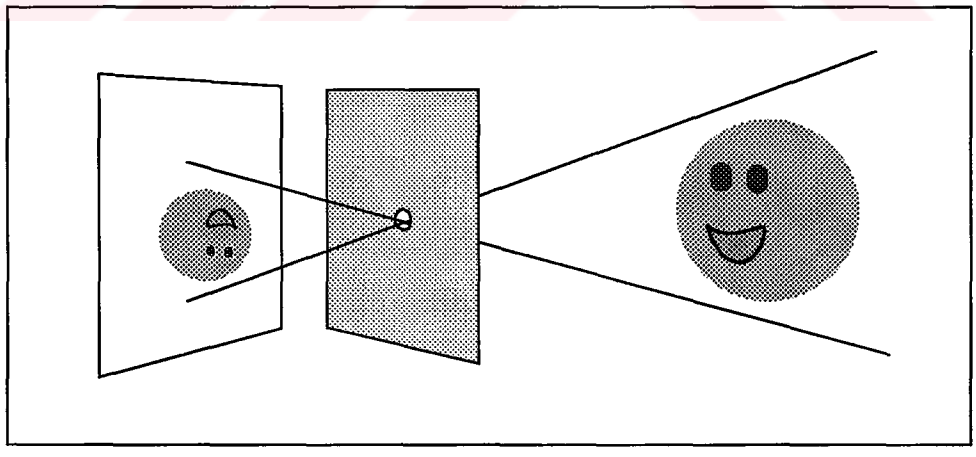
1. Bakış Noktası:

Görüntüsü istenilen üç boyutlu nesne uzayına bakılan nokta. Görüntü düzlemine doğrultulan bütün ışınlar bu noktadan başlayacak şekilde tanımlanır. Göz veya kamera olarak adlandırıldığı da olur.

Bir ışın-izleme sistemi, sanal bir ışık kaynağı tarafından aydınlatılan sanal nesnelerin yine sanal bir kamera tarafından fotoğraflanması olarak tanımlanabilir. Kullanılabilecek en basit kamera cinsi iğne-deliği kameradır. Bu yapıda bir kamera Şekil.4.3 üzerinde gösterilmiştir.



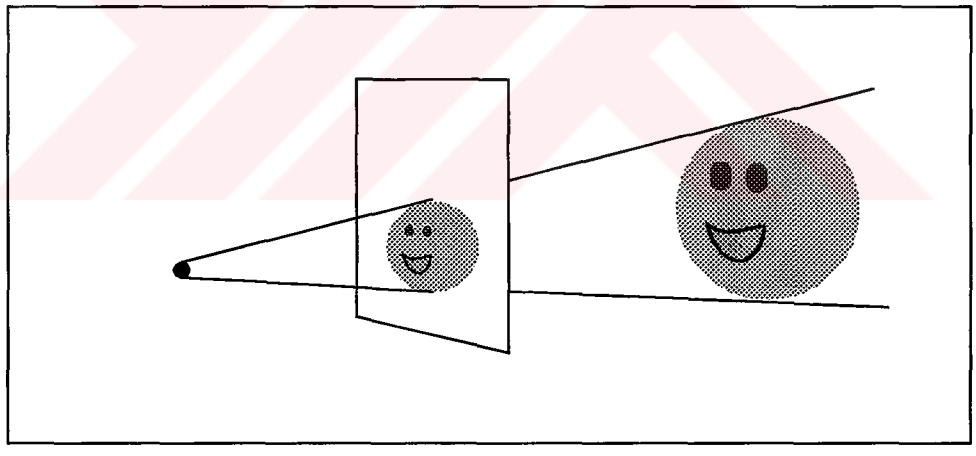

Şekil 4.2. Işın izleme metodunda kullanılan temel kavramlar



Şekil 4.3. İğne-deliği kamera modeli

Bu modelde birbirine paralel iki yüzey kullanılır. Bu yüzeylerden, nesne tarafında olanın üzerinde küçük bir iğne deliği mevcuttur. Bu delikten geçen ışık-ışınları öbür yüzey üzerinde sonlanır, ve bu yüzey üzerinde, Şekil.4.3 de de görülebileceği gibi, nesne uzayının ters görüntüsü oluşur.

Birçok ışın-izleme sistemi bu kamera modelinin fonksiyonel olarak bir eşdeğerini kullanır. Tipik bir ışın-izleme kamera modeli Şekil.4.4 üzerinde gösterilmiştir. Bu modelde, ışık ışınlarının öbür yüzey üzerine düşmesini sağlayan iğne-deliği yerine, nesne uzayı üzerine bakış ışınları gönderen bir bakış noktası kullanılmıştır. Bu ışınlar sanal bir bakış penceresi içinden geçerek nesne uzayına yönlenirler. Bu bakış penceresi genellikle görüntü düzlemi olarak adlandırılır. Bu kamera modelinde görüntü, bu görüntü düzlemi üzerinde oluşturularak ekrana getirilir.



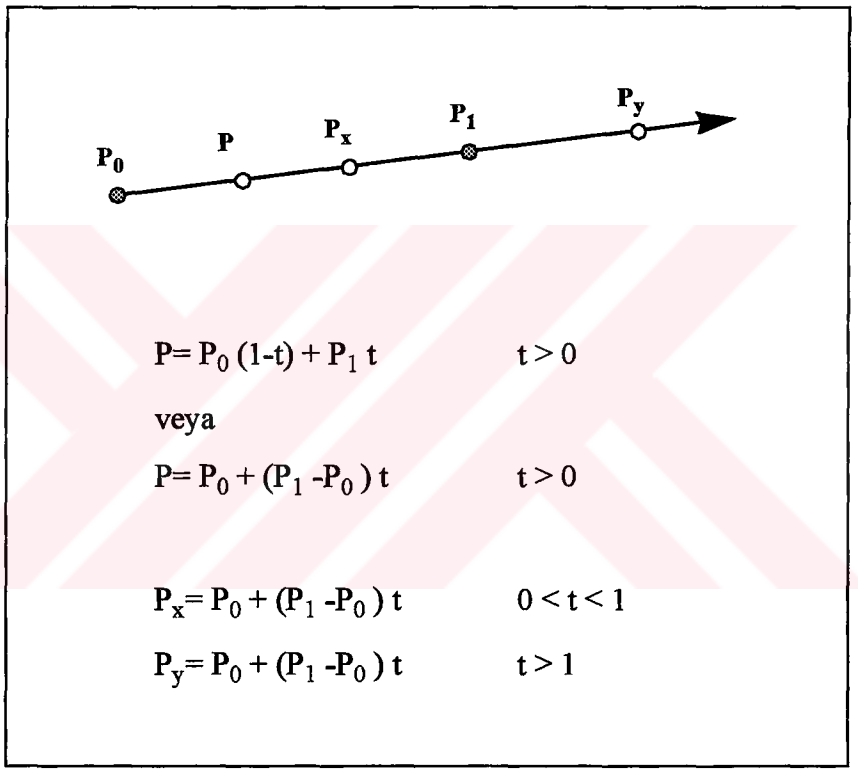
Şekil 4.4. Tipik bir ışın-izleme kamera modeli

2. Işın:

Bir noktadan başlayıp başka bir noktadan geçecek doğrultuda yol alan yarı-doğru.

Genelde parametrik denklemlerle ifade edilir. Işının başladığı nokta $P_0$, ilerlediği doğrultunun $P_1$ noktası olduğu kabul edilirse, bu ışın üzerindeki herhangi bir noktanın (t parametresine bağlı olarak) ifadesi Şekil.4.5 üzerinde gösterilmiştir.

Herhangi bir ışın-izleme sisteminde kullanılabilecek ışın tipleri genel olarak dört sınıfta incelenebilir: Bakış ışınları, gölge ışınları, iletilen ışınlar ve yansıyan ışınlar. Bu ışınlar ile ilgili detaylı bilgi bir sonraki konuda anlatılacaktır.




Şekil 4.5. Parametrik formda ışın ifadesi

3. Görüntü Düzlemi

Üç boyutlu nesne uzayının görüntüsünün oluşturulduğu iki boyutlu düzlem. Bu düzlem üzerine, bakılan noktanın düzleme göre ters tarafındaki, nesneler topluluğunun perspektif

olarak görüntüsü düşürülür. Genellikle birbirine eşit büyüklükteki küçük görüntü elemanlarına (piksel) ayrılarak işlem yapılır. Işın-izleme algoritmasının uygulanması sonucunda ekrana getirilen çıkış bu görüntü düzlemi üzerinde oluşturulan resimdir. Bu düzlem üzerinde işlem, yukarıdan aşağıya her satır için, soldan sağa olacak şekilde gerçekleştirilir. Düzlem üzerindeki her parça ekran üzerinde gösterilecek bir piksel değerine karşılık gelir.

4. Piksel

Görüntü düzleminin, üzerinde işlem yapılan, en küçük elemanı. Boyutu küçüldükçe ekran üzerinde elde edilen resim daha gerçekçi olur. Genellikle resmin gösterildiği ekranın fiziksel özelliklerine (çözünürlük) bağlı olarak şekli ve büyüklüğü değişir.

5. Nesne Uzayı

Resmi istenilen sahnenin bütünü. Nesnelerden oluşmuş üç boyutlu bir alt-uzay olarak da ifade edilebilir.

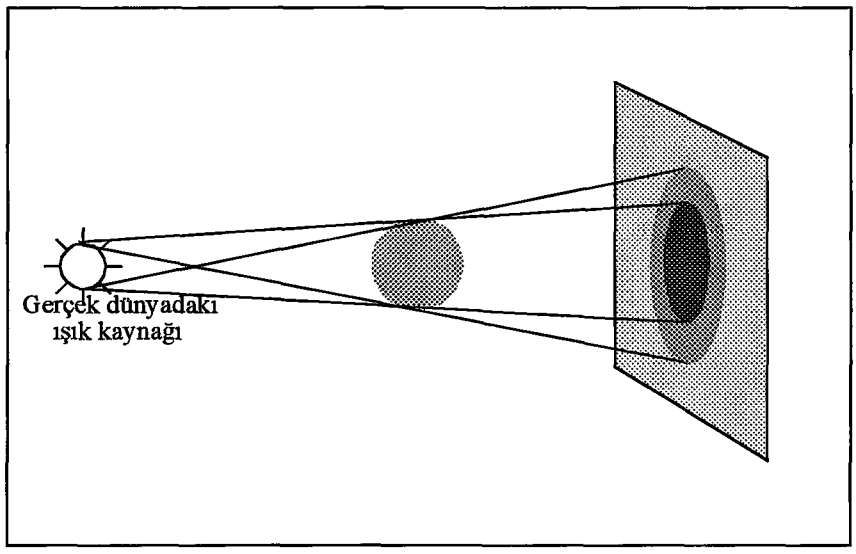
6. Nesne

Görüntüsü oluşturulacak sahneyi oluşturan elemanlardan herbiri. Her nesne için yer bilgisinin yanısıra, yüzey dokusu, madde ve optik özellikler gibi görüntüyü etkileyecek diğer bilgiler de hesaplamalarda kullanılır. Bütün bu özelliklerin bilgisayar tarafından kullanılabilmesi için bellekte belirli bir matematiksel ve/veya geometrik formda saklanması gerekmektedir. Bu da görüntüsü oluşturulabilecek nesnelerin sayısının çok fazla olmasını engellemektedir. Ancak modellenmesi zor olan nesnelerin, daha basit nesne elemanlarının birleşimi olarak ifadesi de mümkündür.

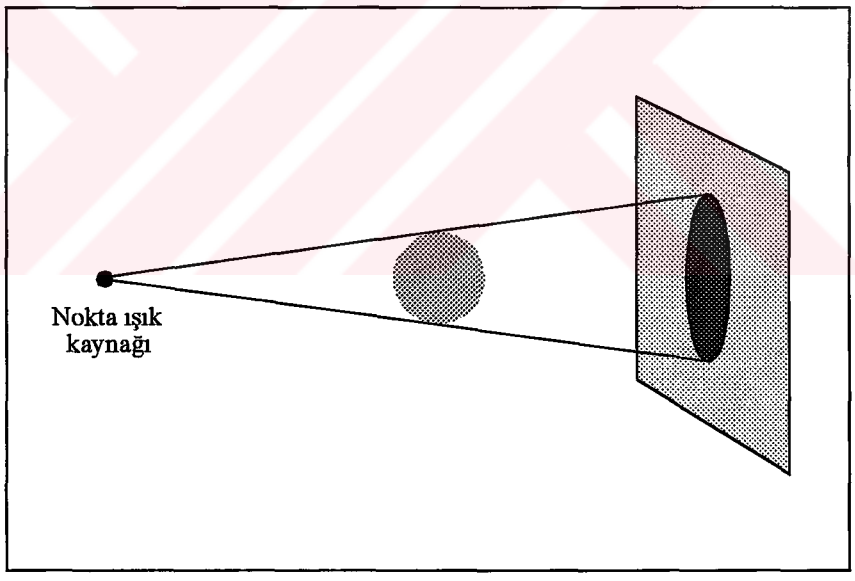
## 7. Işık Kaynağı

Nesne uzayında tanımlı nesneleri aydınlatan eleman. Bu aydınlatma işlemi tanımlandığı nesnenin fiziksel ve optik özelliklerine göre farklılık gösterebilir. Gerçek dünyada varolan ışığın dağılım prensiplerinin oldukça karmaşık olduğuna önceki bölümlerde değinilmişti. Işın-izleme yönteminde kullanılan ışık kaynakları bu karmaşıklıklardan ötürü bazı varsayımlar ile biraz daha basite indirgenerek modellenmiştir. Bu varsayımlardan bir tanesi ışık kaynağından yayılan ışığın tek renkli olduğudur. Modern fizik kurallarına göre ışık ışınlarını oluşturan fotonların herbiri farklı renkte olabilirler. Ancak ışın-izleme metodunda kullanılacak ışığın sarı, yeşil, eflatun, pembe, vs. gibi tek bir renkten oluşmuş olduğu kabul edilir.

Işık kaynakları modellenirken yapılan ikinci büyük varsayım ise kaynağın nokta ışık kaynağı olduğudur. Yani gerçek hayatta olduğu gibi değişik büyüklükte ışık kaynakları kullanılmayacaktır. Bunun sayesinde nesnelerin kenarlarından geçerken oluşan açı farklılığından dolayı ortaya çıkan ikili gölgeler ışın-izleme yönteminde elde edilemeyecektir. Her iki durumda oluşan gölge yapıları Şekil.4.6 ve Şekil.4.7 üzerinde gösterilmiştir.




Şekil 4.6. Gerçek hayatta oluşan yumuşak gölgeler




Şekil 4.7. Işın-izleme yönteminde elde edilen keskin gölgeler

# 5. IŞIN TIPLERI

Işın-izleme sisteminde kullanılan ışınlar temel olarak dört sınıfta incelenebilir:

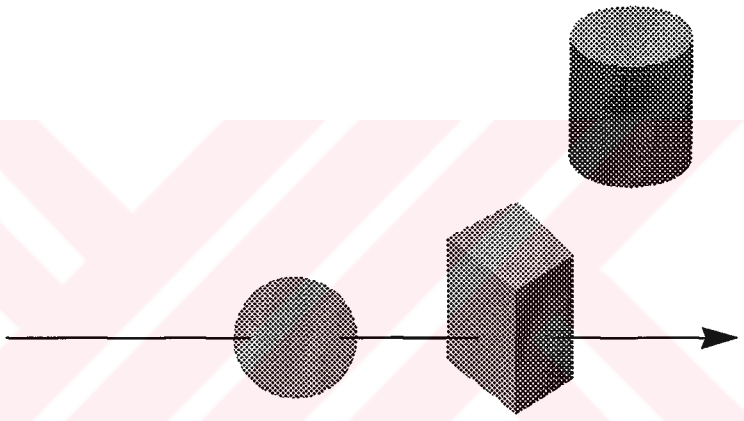
1. Bakış ışınları:

Bu ışınlar kamera tarafından, görüntü düzlemini oluşturan her görüntü elemanı (piksel) için en az bir tane olmak üzere gönderilir ve o piksele karşılık gelen renk değerinin bulunmasında kullanılır. Sistem içerisinde tanımlı olan bütün nesnelerin bir listesi tutulur. Oluşturulan her bakış ışını bu listenin elemanlari ile karşılaştırılarak kesişip kesişmedikleri bulunur. Bu işlem bir denklem sistemi çözmekten ibaret olup genellikle ışın-nesne kesişim testi olarak adlandırılır. Yapılan işlemler büyük hassasiyet gerektiren ve çok sayıda kesirli sayı hesabı içeren çözümler gerektirmektedir.

Nesne uzayına doğru yönlendirilen bakış ışını aynı anda birden fazla nesne ile kesişebilir. Ancak sadece kameraya yakın olan nesne görülebileceğinden ışının ilk olarak kestiği yüzeyin bulunması gereklidir. Dolayısıyla bulunan ilk kesişim noktası her zaman doğru olmayabilir. Tanımlı bütün nesnelerin birer birer ışın-nesne testinden geçirilmesi gereklidir. Elde edilen bütün kesişim noktaları bellekte saklanır ve işlem sonunda kameraya en yakın kesişim noktası görülen nokta olarak seçilir. Bu işlemin bir örneği Şekil.5.1 üzerinde görülebilir.

| KESİŞİM NOKTALARI | |
|---|---|
| NESNE | UZAKLIK |
| Kutu | 5.5 |
| Kutu | 5.8 |
| Küre | 3.2 |
| Küre | 3.9 |
| Silindir | - |

*En yakın nokta*



Şekil 5.1. Işın-nesne kesişim testleri örneği

Her nesne için tutulan bilgi, o nesnenin yapısı ile ilgili her veriyi içermelidir. Bu bilgilerin en önemlisi, nesnenin kesişim noktasındaki renk değeridir. Ancak nesnenin bu noktadaki renk değerini bilmek pikselin alacağı renk değerini bulmakta yeterli değildir. Çünkü noktanın görünen renk değeri o noktaya gelen ışık miktarı ile de direkt olarak bağlantılıdır.

Işık kaynağından gelen bir ışık ışını herhangi bir yüzeye çarptığı zaman (nesnenin aynalı

yüzeyi olmaması koşuluyla) her yöne dağılır. (Aynalı yüzeyler daha sonraki konularda incelenecektir). Işık her yönde dağıldığı için bu tür aydınlanmaya yayınık aydınlanma (diffuse illumination) adı verilir. Doğal olarak, bir ışık kaynağı her nesnenin bütün noktalarını aydınlatmaz. Bazı noktalara gelen ışık ışınları diğer nesneler tarafından engellendiği durumlarda bu noktalar gölgeli alanlarda kalırlar. Hatta, eğer nokta nesnenin ışığa göre ters tarafındaysa, nokta nesnenin kendi gölgesinde kalır.

Kesişim noktalarındaki yayınık aydınlanmanın bulunabilmesi için, ışık kaynaklarından bu noktalara, diğer nesneler tarafından engellenmeden gelebilen ışık ışınlarının tespit edilmesi gerekir. Bunun için de başka bir tür ışın kümesine, gölge ışınlarına ihtiyacımız vardır.

## 2. Gölge ışınları

Bakış ışınları kullanılarak nesne uzayında görüntülenmek üzere bir kesişim noktası bulunduktan sonra, sıra ışık kaynaklarından bu noktaya gelen aydınlatma değerlerinin tespit edilmesine gelir. Bunun için kesişim noktasından başlayıp ışık kaynağı doğrultusunda ilerleyen gölge ışınları tasarlanır. Bu şekilde her ışık kaynağı için tasarlanan gölge ışınları nesne uzayında tanımlı her nesne ile kesişim testinden geçirilir. Eğer bu ışın herhangi bir nesne ile kesişirse, bu noktaya ilgili ışık kaynağından çıkan ışık ışınları gelmiyor demektir. Bu durumda diğer nesneleri test etmeye gerek kalmadan bir sonraki ışık kaynağına yöneltilen gölge ışınlarına ait kesişim testlerine geçilebilir. Eğer herhangi bir gölge ışını hiçbir nesne tarafından engellenmiyorsa, ilgili ışık kaynağının bu noktaya yaptığı aydınlatma katkısının bulunması gereklidir.

Kesişim noktasının görünen renginin bulunabilmesi için ışık kaynağının renginin de bilinmesi gereklidir. Beyaz bir yüzey üzerine kırmızı ışık düşürülürse yüzey kırmızı gözükür. Aynı şekilde kırmızı bir yüzey üzerine yeşil ışık düşürülürse yüzey siyah görünür. Çünkü yüzey üzerindeki kırmızı pigmentler, kırmızı ışık dışındaki bütün renkleri emer. Bu gibi örneklerden dolayı yüzeyi aydınlatan ışık kaynağının renginin de

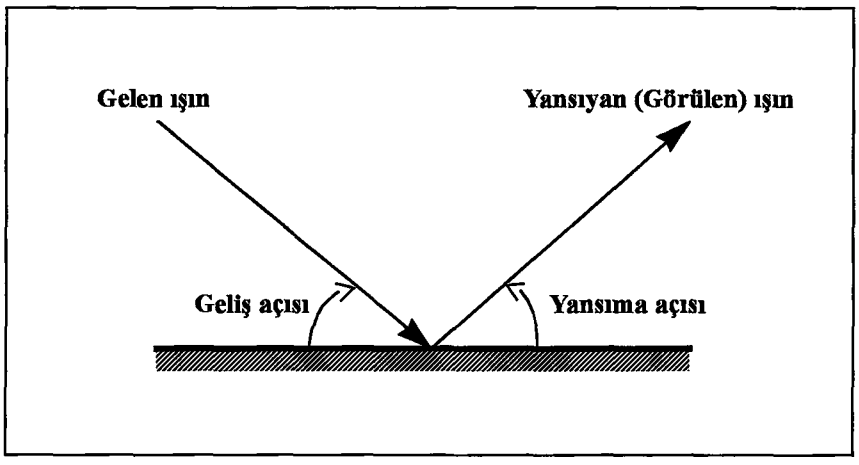
bilinmesi gerekir.

Işık kaynakları tarafından yapılan katkı ve miktarları tespit edildikten sonra noktanın görünen renginin hesaplanmasına geçilebilir. Bu hesapta değerlendirilmesi gereken bazı faktörler vardir: Işık ışınının yüzeye çarpma açısı (dik açılar daha fazla aydınlanma sağlar), parlak nesneler üzerinde bulunan mikroskobik tümsekler üzerinden yansıyan parlak ışınlar, floresan yüzeylerde gerçekleşen yansıma. Bu tip efektlerin simülasyonu için bir çok metod kullanılabilir. Bu metodların büyük bir çoğunluğu yansıyan ve iletilen ışınları kullanmaktadır.

## 3. Yansıyan ışınlar

Optik geometride iki tip yansıma olayı vardir: Mat bir yüzey üzerinde gerçekleşen ve her yönde dağılan yayınık yansıma, ve düzgün, parlak yüzeylerde gerçekleşen düzgün yansıma. Yayınık yansıma günlük hayatta gördüğümüz renklerin çoğunun gerçekleşme sebebidir. Bu tip yansımanın piksel rengine yaptığı katkı, gölge ışınları yardımıyla hesaplanabilir. Düzgün yansıma ise ayna, nikelaj kaplamalı yüzeyler, çok temiz ve parlak porselen gibi yüzeyler üzerinde oluşur. Bu şekilde yansıma sadece ışık kaynağından gelen ışınları değil, çevrede bulunan diğer nesneler üzerinden gelen yansımış ışınları da içerir.

Işık, ayna veya benzer düzgün bir yüzeyden yansıdığı zaman her yönde dağılmaz. Şekil.5.2 üzerinde bu tip bir yansıma gösterilmiştir.




Şekil 5.2. Işınların düzgün yüzeylerden yansıması

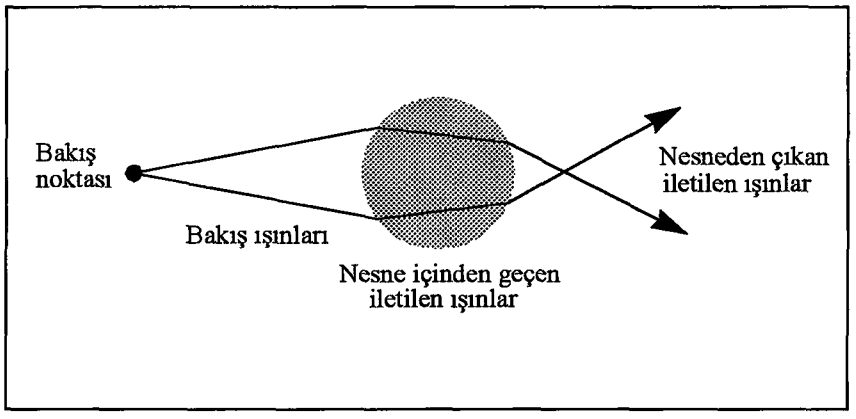
Burada dikkat edilmesi gereken husus, yüzeye gelen ışının geliş açısı, yansıyan ışının yansıma açısına eşittir. Gelen ışının yüzeye çarpma açısı hesaplanabileceğinden, yüzeyden yansıyan açının yansıma yönü de buradan kolaylıkla bulunabilir. Bu şekilde yansıyan ışınlar defalarca yansımaya devam edebilir. Yansıyan ışınları hesaplayan algoritmalar bu amaçla iteratif olarak kullanılabilir. Ancak birçok ışın-izleme sistemi algoritmanın kendi kendini çağırma sayısını belli bir limitin altında tutmayı tercih eder. Bu yolla hesap zamanı belirli bir seviyeye kadar düşürülümüş olur. Düzgün yansımadan elde edilen aydınlanma değeri, diğer hesaplarda elde edilen renk değerleriyle birleştirilerek pikselin toplam renk değeri bulunmuş olur.

Kesişim noktasının renk değerini etkileyen son ışın tipi de iletilen ışınlardır.

4. İletilen ışınlar

Saydam veya yarı-saydam nesneler ışığın içlerinden geçmelerine imkan tanırlar. Bu şekilde nesne içinden geçerek yoluna devam eden ışınlara iletilen ışınlar adı verilir. İletilen ışınlar genellikle tek ışın halinde incelenmezler. Işın-izleme sistemlerinin büyük bir çoğunluğu bu ışınları, kırılım sırasında ve sonrasında olmak üzere iki ayri ışın olarak

inceler. Şekil.5.3 üzerinde iletilen ışınların bu amaçla iki parçaya ayrılması gösterilmiştir.




Şekil 5.3. İletilen ışınlar ve kırılım olayı

Kırılan ışığın fiziksel incelenmesi bu çalışmanın dışında tutulmuştur. Hesaplamalarda sadece gelen ışının yüzeye varış açısı, ve nesne ile ortam arasındaki yoğunluk farklılıkları kullanılmıştır. Işının yüzeye ne kadar küçük bir açıyla gelirse, o kadar çok kırılmaya maruz kalır. Yüzeye dik olarak gelen ışınlar ise doğrultularını değiştirmeden karşı tarafa geçerler. Işının geçtiği ortamlar arasındaki yoğunluk farkı arttıkça kırılması da aynı oranda artar.

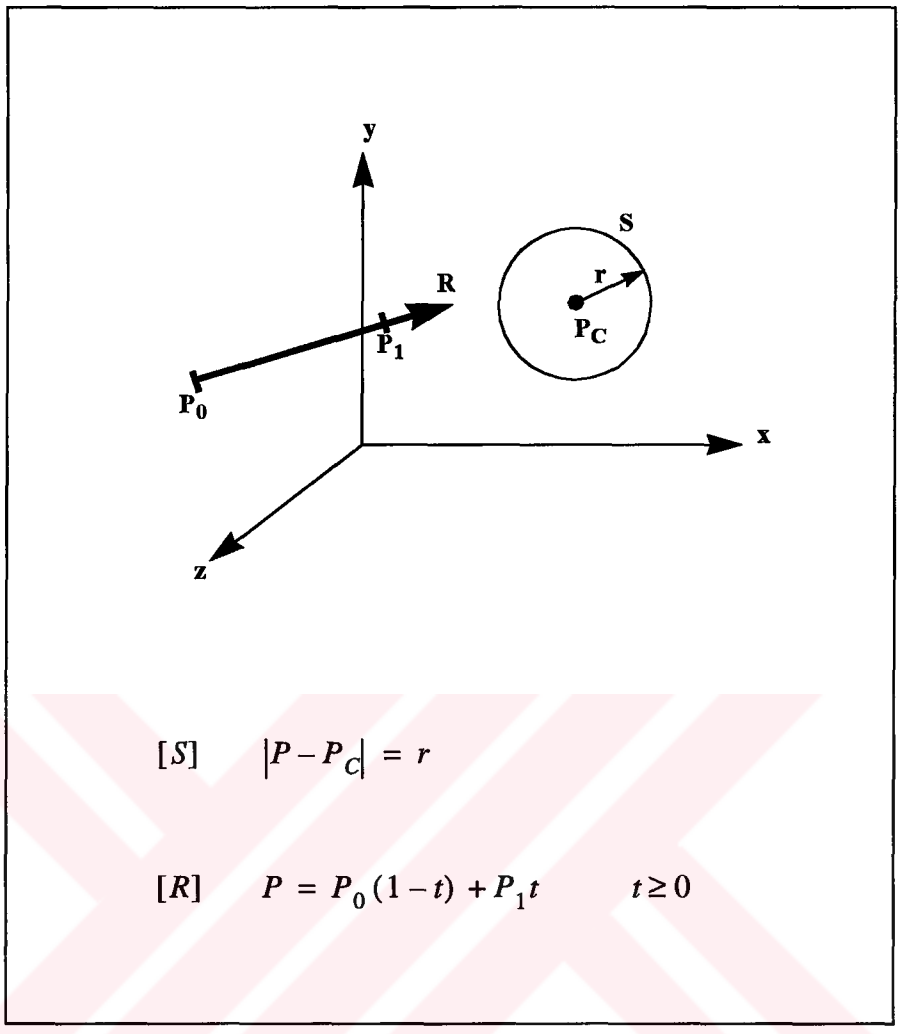
# 6. IŞIN-KÜRE KESİŞİMİ

Işın-izleme metoduyla elde edilen görüntülerde genellikle küresel cisimler kullanıldığından, üzerinde en çok araştırma yapılan nesne tipi küre olmuştur. Bu araştırmalar daha çok ışın-küre kesişim testlerini hızlandırmak doğrultusunda gerçekleşmiştir.

Yapılan hızlandırma çalışmalarına girmeden önce, en basit haliyle ışın ve küre denklemlerini Şekil.6.1 üzerinde görebiliriz.

Işın ve kürenin kesişim noktasının bulunması için, Şekil.6.1 üzerinde gösterilen iki denklemin ortak çözüm kümelerinin bulunması gereklidir. Bu şekilde S ile ifade edilen denklem S küresi üzerinde, R ile ifade edilen denklem ise R ışını üzerinde tanımlı herhangi bir P noktasının t parametresine göre konumunu vermektedir. Küre için t parametresine ihtiyaç yoktur. Denklemden de anlaşıldığı üzere P noktası, $P_C$ noktasına uzaklığı r olan noktalar kümesini sembolize etmektedir. Yani P noktası küre yüzeyi üzerinde tanımlı noktalar kümesinin bir elemanıdır.

Bu iki denklemin ortak çözüm kümesinin bulunması ana hatlarıyla Şekil.6.2 üzerinde gösterilmiştir.




Şekil 6.1. Parametrik formda ışın ve küre denklemleri

$$|P_0(1-t) + P_1 t - P_C| = r$$

$$|(P_0 - P_C) + (P_1 - P_0)t| = r$$

$$|P_{0C} + P_{10}t| = r$$

$$(x_{0C} + x_{10}t)^2 + (y_{0C} + y_{10C}t)^2 + (z_{0C} + z_{10}t)^2 = r^2$$

Denklemde değeri bilinmeyen tek değişkenin t olduğu kıstasından yola çıkılarak yeniden bir düzenleme yapılır ve aşağıdaki türden bir ifade elde edilir:

$$at^2 + bt + c = 0, \text{ ve } \Delta = \sqrt{b^2 - 4ac}$$

Bu denklem tek bilinmeyenli ikinci dereceden bir denklem olup, t değerleri şu şekilde bulunabilir:

$\Delta < 0$ için denklem kümesinin çözümü yoktur. Yani ışın küreyi kesmez.

$\Delta = 0$ için denklem kümesinin bir tek çözümü vardır. Yani işin küreye teğettir. Buna uygun çözüm kümesi şu şekilde yazilabilir: $t_{1,2} = \frac{-b}{2a}$

$\Delta > 0$ için denklem kümesinin iki çözümü vardir. Yani ışın kümeye bir noktadan girip diğer noktadan çıkmaktadır. Buna uygun çözüm kümesi şu şekilde yazılabilir: $t_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$

Şekil 6.2. lşın-küre kesişim noktası bulunuşu

Bu şekilde elde edilen t değerlerinin yorumlanması ise Şekil.6.3 üzerinde ifade edilmiştir.

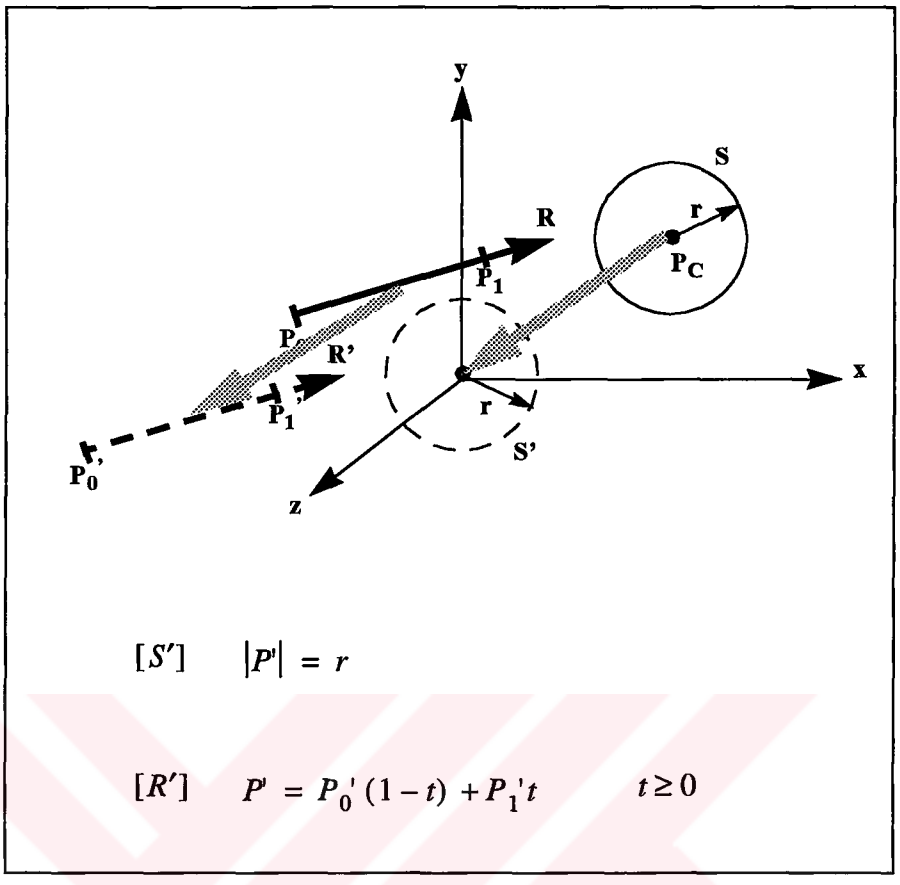
| | |
|---|---|
| $t < 0$ | Elde edilen t değerine karşılık gelen nokta, $P_0P_1$ doğrusu üzerinde yer alır ve $P_0$ noktasının $P_1$'e göre ters tarafında kalır. Eğer bakış noktası $P_0$ ve bakış doğrultusu $P_1$ ise, bu durumda t değerine karşılık gelen nokta kamera tarafından görülmez. |
| $t = 0$ | Elde edilen t değerine karşılık gelen nokta, $P_0$ noktasına eşittir. Eğer bakış noktası $P_0$ ve bakış doğrultusu $P_1$ ise, bu durumda t değerine karşılık gelen nokta kamera tarafından görülmez. |
| $0 < t < 1$ | Elde edilen t değerine karşılık gelen nokta, $P_0P_1$ doğrusu üzerinde yer alır ve $P_0$ ile $P_1$ noktası arasında bulunur. Kurulan bakış modeline göre, bu noktanın kamera tarafından görülüp görülmeyeceğine karar verilir. |
| $t = 1$ | Elde edilen t değerine karşılık gelen nokta, $P_1$ noktasına eşittir. Kurulan bakış modeline göre, bu noktanın kamera tarafından görülüp görülmeyeceğine karar verilir. |
| $t > 1$ | Elde edilen t değerine karşılık gelen nokta, $P_0P_1$ doğrusu üzerinde yer alır ve $P_1$ noktasının $P_0$'a göre ters tarafında kalır. Eğer bakış noktası $P_0$ ve bakış doğrultusu $P_1$ ise, bu durumda t değerine karşılık gelen nokta kamera tarafından görülebilir. |

Şekil 6.3. Çözüm kümesinde elde edilen t değerlerinin yorumlanması

Seçilen bakış modeline göre görülebilen noktalar arasındaki en küçük t değerli nokta ilgili pikselin renk değerinin belirlenmesinde kullanılır. Ancak Şekil.6.2 üzerinde gösterilen ışın-küre kesişimi bulunuşu bu problemin en genel hatlarıyla çözüm ifadesidir. Önceleri bu haliyle uygulanan yöntemin, kullanılan nesne ve efekt sayısı arttıkça, oldukça yavaş kalmakta olduğu görülecektir. Özellikle, kullanılan kök alma işleminin gerçeklenmesi için oldukça fazla zaman harcanmaktadır. Bu yüzden bazi hızlandırıcı yöntemler kullanılmaya başlanmıştır.

Bu amaçla kullanılacak bir yöntem, küre merkezinin eksen merkezinde olduğu şekilde işlemleri gerçekleştirmektir. Böylece çözüm kümesinde bulunan $P_c$ teriminden kurtulunmuş olacaktır. Küre üzerinde doğrultulan her ışın önce $-P_c$ kadar ötelenip bundan sonra kesişim işleminin sonucu bulunacaktır. Eğer uygun t değeri elde edilirse, bu değere karşılık gelen P noktasının $P_c$ kadar ötelenmiş hali bize gerçek kesişim noktası değerini verecektir. Bu yöntemde eğer kamera noktasının da $-P_c$ kadar ötelenmiş hali bellekte saklanırsa, her işin kesişim testi için yeniden öteleme yapmak zorunda kalınmayacaktır. Bu metodun ana hatlarıyla ifadesi Şekil.6.4 üzerinde gösterilmiştir.

Işın-küre kesişim testinin bu şekildeki çözümü Şekil.6.2 üzerinde gösterilen halinden yöntem olarak çok farklı değildir. Şekil.6.4'da görülen yapı için çözüm Şekil.6.5 üzerinde ana hatlarıyla ifade edilmiştir. Burada da görülebileceği gibi çözüm adımları aynı olmakla beraber, sadece gereken hesap süresi bakımından farklılık belirmektedir.




Şekil 6.4. Işın-küre kesişiminde optimizasyon örneği-1

$$|P_0'(1-t) + P_1't| = r$$

$$|P_0' - P_0't + P_1't| = r$$

$$|P_0' + (P_1' - P_0')t| = r$$

$$|P_0' + P_{10}'t| = r$$

$$(x_0' + x_{10}'t)^2 + (y_0' + y_{10}'t)^2 + (z_0' + z_{10}'t)^2 = r^2$$

Denklemde değeri bilinmeyen tek değişkenin t olduğu kıstasından yola çıkılarak yeniden bir düzenleme yapılır ve aşağıdaki türden bir ifade elde edilir:

$$at^2 + bt + c = 0$$

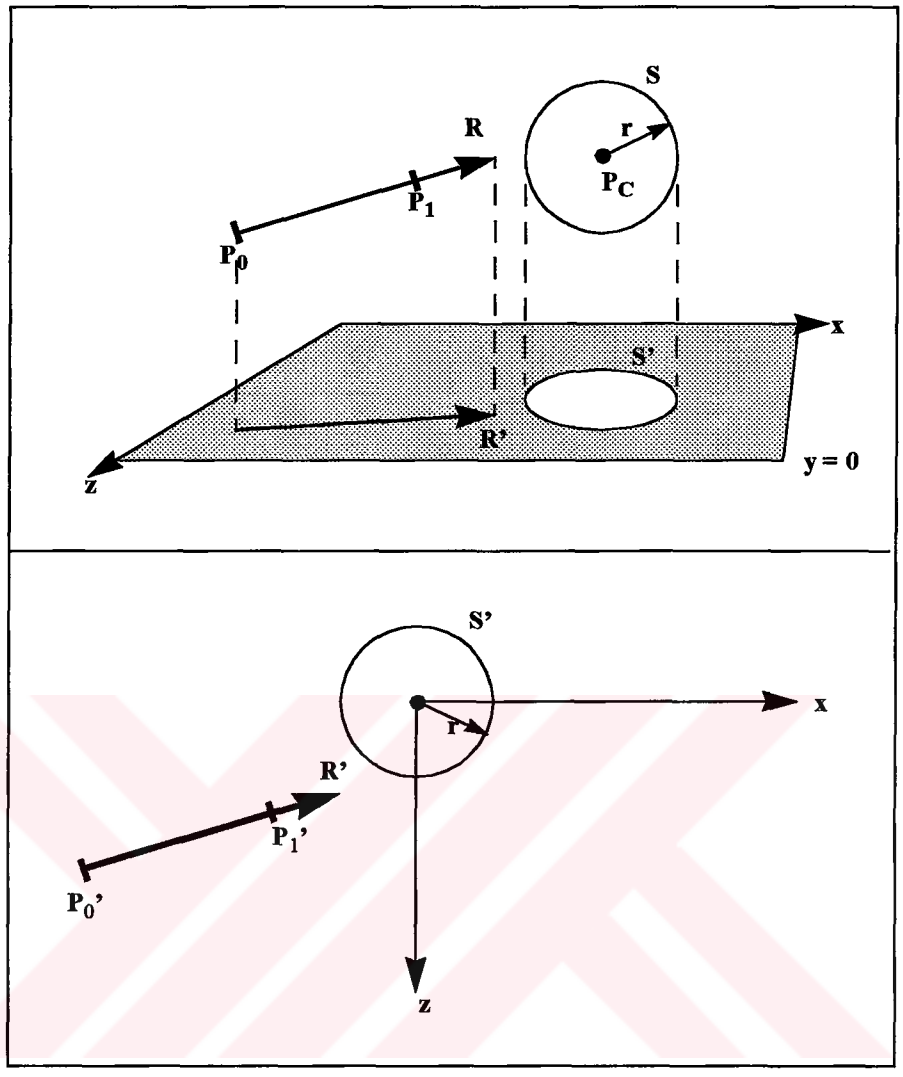
Çözümün bundan sonraki adımları genel olarak Şekil.6.2 üzerinde gösterildiği gibidir. Tek fark, uygun t değeri bulunduktan sonra elde edilen P noktasının gerçek değeri için, bu noktanın $P_c$ kadar ötelenmiş hali kullanılmalıdır.

Şekil 6.5. Optimize edilmiş haliyle ışın-küre kesişim noktası bulunuşu-1

Şekil.6.4 üzerinde gösterilen optimizasyon örneği, sisteme biraz zaman kazandırsa da yeteri kadar hızlılık sağlayamamaktadır. Bu amaçla kullanılacak bir diğer yöntem ise şekli üç boyuttan iki boyuta indirmektir. Bu şekilde kesişim problemi üç boyutlu ışın ile

kürenin kesişim noktasını bulmak yerine, iki boyutlu işin ile çemberin kesişimini bulmak haline indirgenmektedir. Eğer izdüşümü alınan çember ile ışın kesişmiyorsa zaten üç boyutlu hallerinde de kesişmiyorlar demektir. Ancak eğer kesişim noktaları varsa, o zaman bu nokta için üçüncü boyut testi de yapılması gereklidir. Ancak bu ekstra test sadece kesişim noktaları için yapılacağından işlem oldukça hızlandırılmış olacaktır.

Bu amaca yönelik olarak geliştirilen yöntem ana hatlarıyla Şekil.6.6 üzerinde ifade edilmiştir.




Şekil 6.6. Işın-küre kesişiminde optimizasyon örneği-2

Bu yapıda kurulan sistemin çözüm adımları ise Şekil.6.7 üzerinde kısaca gösterilmiştir.

$$|P_0'(1-t) + P_1't| = r$$

$$|P_0' - P_0't + P_1't| = r$$

$$|P_0' + (P_1' - P_0')t| = r$$

$$|P_0' + P_{10}'t| = r$$

$$(x_0' + x_{10}'t)^2 + (y_0' + y_{10}'t)^2 = r^2$$

Denklemde değeri bilinmeyen tek değişkenin t olduğu kıstasından yola çıkılarak yeniden bir düzenleme yapılır ve aşağıdaki türden bir ifade elde edilir:

$$at^2 + bt + c = 0$$

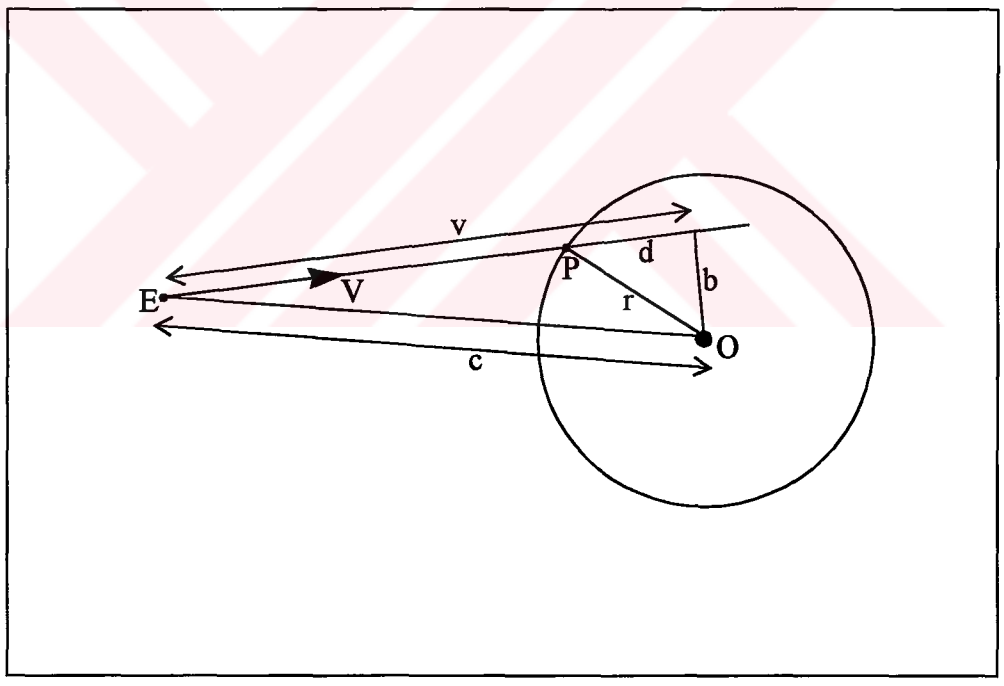
Bu ifadeden elde edilecek $\Delta$ değerine göre, eğer ışın ile çember kesişiyorsa, o zaman çözüm kümesi ifadesine z değerleri de katılarak gerçek çözüm kümesi bulunur. Aksi taktirde z değerleriyle ilgili işlem yapılmadığından Şekil.6.5 üzerinde gösterilen çözüm biraz daha hızlandırılmış olur.

Şekil 6.7. Optimize edilmiş haliyle ışın-küre kesişim noktası bulunuşu-2

Şekil.6.6 üzerinde gösterilen optimizasyon örneği, sistemi ilk haline göre oldukça

hızlandırmasına karşın yine de bu konuda yapılacak ivmelendirici çalışmalar bunlarla sınırlı kalmamaktadır. Bu amaçla kullanılacak bir diğer yöntem ise çevreleyen kutu yöntemi olarak adlandırılabilir. Bu yöntemde ışınlar küre ile kesiştirilmek yerine, küreyi çevreleyen kutu ile kesiştirilmektedir. Eğer herhangi bir ışın kutuyu kesmiyorsa içindeki küreyi de kesmiyor demek olduğundan, bu ışın atlanarak işleme bir sonraki ışın ile devam etmek yeterli olacaktır. Eğer ışın kutuyu kesiyorsa o zaman küre ile gerekli kesişim testleri yapılabilir. Ancak ışın-kutu kesişim testi ışın-küre testine göre çok daha hızlı bir şekilde gerçekleştirildiğinden toplam işlem zamanı diğer yöntemlere göre oldukça azaltılmış olacaktır.

Işın-küre kesişim hesabında geometrik yöntemler kullanarak çözüme başka bir şekilde de ulaşılabilir. Kullanılacak yöntemin geometrik yapısı Şekil.6.8 üzerinde gösterilmiştir.




Şekil 6.8. Işın-küre kesişim hesabında kullanılan geometrik model

Burada bulmak istediğimiz, ışının küreyi kestiği P noktasının değeridir. Geometri

kuralları yardımıyla, şekilden aşağıdaki denklemleri elde edebiliriz:

$$v^2 + b^2 = c^2$$

$$d^2 + b^2 = r^2$$

$$d = \sqrt{r^2 - (c^2 - v^2)}$$

Eğer V vektörünü, sözkonusu ışın doğrultusunda olan bir birim vektör olarak kabul edersek, ışın-küre kesişim noktasını (P) şu şekilde bulabiliriz:

$v = \overrightarrow{EO} \cdot V$

$disc = r^2 - ( (\overrightarrow{EO} \cdot \overrightarrow{EO}) - v^2)$

if $(disc < 0)$

then no intersection

else begin

$d = \sqrt{disc}$

$P = E + (v - d)\, V$

end

Şekil 6.9. Geometri kuralları kullanılarak ışın-küre kesişim hesabı

## 7. IŞIN-DÜZLEM KESİŞİMİ

Işın-izleme sistemlerinde, ışın-düzlem kesişim hesabı oldukça fazla kullanılmaktadır. Dolayısıyla bu hesap için kullanılan işlemler optimize edilmiş olmalıdır. Bu bölümde bu amaç için başvurulan hızlı bir yöntem anlatılacaktır. Bir düzleme ait $N$ normal vektörü ve bu düzlem üzerinde tanımlı her $P$ noktası için $P \cdot N$ çarpımı değeri sabittir. Bu değer $d = -V_0 \cdot N$ çarpmı ile elde edilebilir. Bu durumda, ilgili düzlemin ifadesi,

$$N \cdot P + d = 0$$

şeklinde yazılabilir. Bu yolla elde edilen denklem, düzlem üzerinde bulunan her nokta için aynı değere sahip olduğundan, ışın-izleme kesişim hesaplarına başlamadan önce sadece bir defaya mahsus olarak hesaplanır ve düzlem tanımı içine yerleştirilir.

Düzlem üzerine gönderilen ışınların ifadesinin

$$r(t) = O + Dt$$

şeklinde olduğunu kabul edersek, $t$ parametresinin, ışın ile düzlemin kesişim notkasındaki değeri

$$t = -\frac{d + N \cdot O}{N \cdot D}$$

olarak hesaplanabilir. Bu hesap, 12 ondalıklı sayı işlemi ve üç karşılaştırma içerir:

- Eğer düzlem ve ışın paralel ise (yani $N \cdot D = 0$ ise) kesişim noktası yoktur.

- Eğer kesişim notası, ışının başlangıç noktasından daha gerideyse (yani $t \leq 0$ ise), kesişim noktası reddedilir.

- Eğer daha yakın bir kesişim noktası önceden bulunduysa (yani $t > t_{eski}$ ise) kesişim noktası reddedilir.

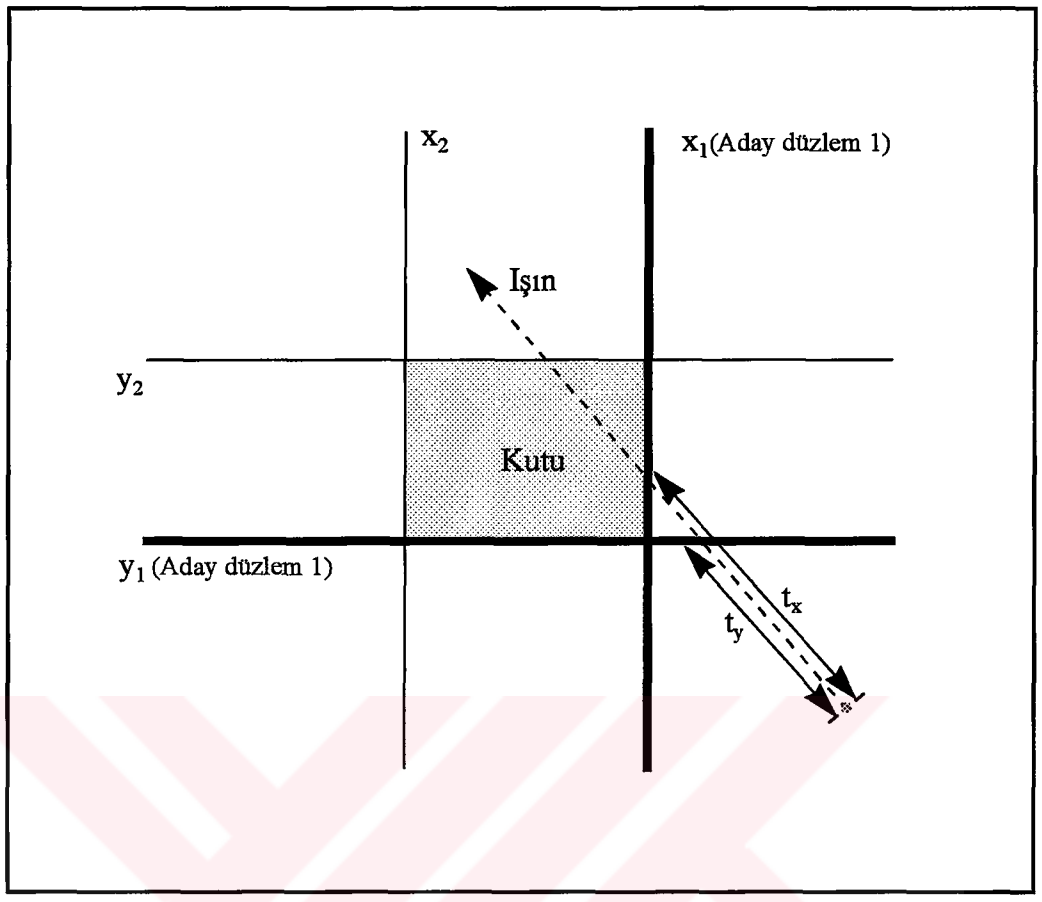
# 8. IŞIN-KUTU KESİŞİMİ

Bu bölümde anlatılacak algoritma oldukça hızlı olmak zorundadır. Çünkü diğer birçok algoritma ile orta olarak kullanılan bu yöntem sayesinde birçok ondalıklı sayı hesabından kurtulunmak istenmektedir.

Bu algoritmada, sözkonusu olan kutunun (koordinat sisteminin temel düzlemlerine (xy, yz, xz) paralel olan) kesişen düzlemler tarafından oluşturulduğu varsayılmıştır.

Üç boyutlu bir sistemde çalıştığımızı gözönüne alırsak, işe önce sözkonusu olan ışının kesebileceği üç aday düzlem aramakla başlarız. Yani, kutuyu oluşturan düzlemler içerisinde arkası bize dönük olanları eleriz. Daha sonra bu üç aday düzlem ile ışının başlangıç noktası arasındaki mesafeler bulunur. En uzak mesafeli kesişim noktasına sahip olan düzlem, ışına en yakın olan düzlem olarak seçilir. Tabii ışın-düzlem kesişiminin bu hesaba dahil edilebilmesi için, kesişim noktasının kutunun içinde olması gereklidir. Aksi taktirde kesişim noktası dikkate alınmaz.

İki boyutta örnek vermek gerekirse (Şekil.8.1), $x_1$, $x_2$, $y_1$ ve $y_2$'nin kutuyu çevreleyen yüzeyler olduğunu ele alalalım. Bu durumda, ışının başlangıç noktasının sağ alt köşe olduğu gözönüne alınırsa, aday düzlemler $x_2$ ve $y_1$ olur. Daha sonra $t_x$ ve $t_y$ mesafeleri, ışının başlangıç noktasından $x_2$ ve $y_1$'e olacak şekilde hesaplanırlar. $t_x > t_y$ olduğu için, $x_2$, ışın tarafından çarpılan en yakın düzlem olarak seçilir.




Şekil 8.1. İki boyutta ışın-kutu kesişimi hesabı

# 9. IŞIN-POLİGON KESİŞİMİ

Işın-izleme işlemlerinde çoklukla kullanılan poligonların taranması sırasında, genellikle poligon veritabanlarından yararlanılır. Özellikle karmaşık nesneleri ifade ederken kullanılan çok sayıda poligon içeren bu veritabanlarının çözüm kümelerini bulabilmek için oldukça hızlı algotirmalara ihtiyaç vardır. Takip eden bölümde, bu amaca yönelik oldukça hızlı bir algoritma tanıtılmıştır.

## 9.1. Hızlı bir ışın-poligon kesişim algoritması

Algoritmanın ilk adımı, ilgilenilen ışının poligon içinden geçip geçmediğini tespit etmektir. Eğer ışın bu testten olumlu sonuç alırsa, yani poligon içinden geçiyorsa, ikinci adımda bu ışının poligonu hangi noktasından delip geçtiği bulunur. Bu kesişim noktası bulunduktan sonra, ilgili noktanın koordinatları, poligonun üzerinde bulunduğu düzleme ve köşe noktalarına göre lokalize edilir. Bu şekilde parametrik olarak ifadesi bulunan noktaya ait değerler kullanılarak, sözkonusu noktadaki yüzeye ait normal değeri yaklaşık olarak elde edilir, ve bu normal vektörü noktayla ilgili aydınlanma değeri hesabında kullanılır.

### 9.1.1. 1. Adım: Taban düzlemi kesim hesabı

Bu adım, diğer ışın-düzlem kesişim hesaplarına oldukça benzemekte olmasına rağmen bu bölümde yine kısaca anlatılacaktır. Bir poligonun köşe noktalarını $V_i (i \in \{0, \ldots, n-1\}, n \geq 3)$ olacak şekilde düşünelim, ve $V_i$ köşe noktasına ait koordinat değerlerinin $x_i$, $y_i$ ve $z_i$ olduğunu varsayalım. Bir poligona taban oluşturan düzleme ait $N$ normal vektörü şu şekilde ifade edilebilir:

$$\vec{N} = \overrightarrow{V_0V_1} \times \overrightarrow{V_0V_2}$$

Düzlemlere ait teoremlerden de bilindiği üzere, bu düzlem üzerinde tanımlı her $P$ noktası için $P \cdot N$ çarpımı değeri sabittir. Bu değer $d = -V_0 \cdot N$ çarpmı ile elde edilebilir. Bu durumda, ilgili düzlemin ifadesi,

$$N \cdot P + d = 0$$

şeklinde yazılabilir. Bu yolla elde edilen denklem, poligon üzerinde bulunan her nokta için aynı değere sahip olduğundan, ışın-izleme kesişim hesaplarına başlamadan önce sadece bir defaya mahsus olarak hesaplanır ve poligon tanımı içine yerleştirilir.

Poligon üzerine gönderilen ışınların ifadesinin

$$r(t) = O + Dt$$

şeklinde olduğunu kabul edersek, $t$ parametresinin, ışın ile düzlemin kesişim notkasındaki değeri

$$t = -\frac{d + N \cdot O}{N \cdot D}$$

olarak hesaplanabilir. Bu hesap, 12 ondalıklı sayı işlemi ve üç karşılaştırma içerir:

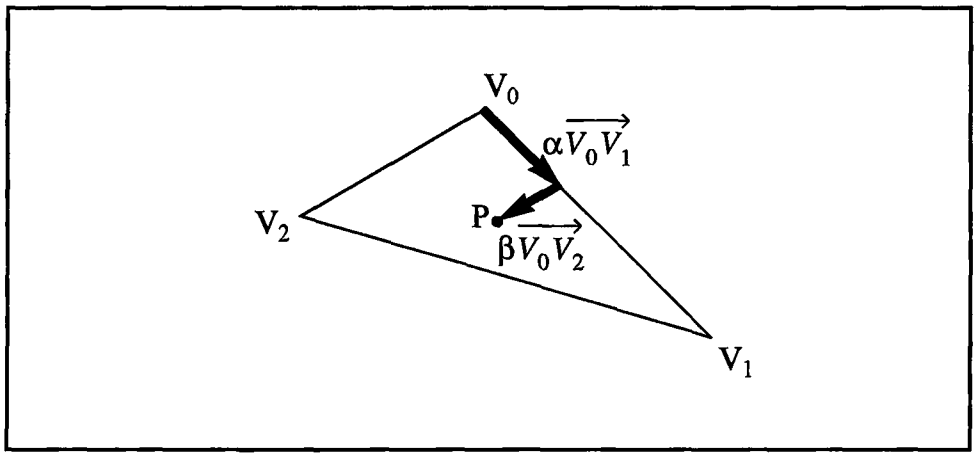
- Eğer poligon ve ışın paralel ise (yani $N \cdot D = 0$ ise) kesişim noktası yoktur.

- Eğer kesişim notası, ışının başlangıç noktasından daha gerideyse (yani $t \leq 0$ ise), kesişim noktası reddedilir.

- Eğer daha yakın bir kesişim noktası önceden bulunduysa (yani $t > t_{eski}$ ise) kesişim noktası reddedilir.

### 9.1.2. İkinci adım: Poligon kesim hesabı

Bu adımda, birinci adımda elde ettiğimiz parametrik ifadeyi ($\vec{N} = \overrightarrow{V_0V_1} \times \overrightarrow{V_0V_2}$) kullanacağız. Ancak dikkat edilirse, bu ifadede, poligona ait sadece üç noktanın bulunduğu görülecektir. Dolayısıyla diğer noktalar hesaba dahil edilmemiştir. Ancak, daha gerçekçi görüntüler elde etmek için poligona ait bütün noktaların işlenmesi gerekmektedir. Bu yüzden, bu adımda poligonları üçgenlere ayırarak işleyeceğiz. Yani $n$ köşesi ( $n \geq 3$ ) bulunan bir poligon, $n-2$ üçgenden oluşan bir üçgenler kümesi olarak ele alınacaktır. Bu sebeple bu yöntem sadece dışbükey poligonlar üzerinde uygulanabilir.




Şekil 9.1. P noktasının parametrik gösterimi

Şekil.9.1 üzerinde görülen $P$ noktası

$$\overrightarrow{V_0P} = \alpha\overrightarrow{V_0V_1} + \beta\overrightarrow{V_0V_2}$$

şeklinde ifade edilebilir. Buradaki $P$ noktası

$$\alpha \geq 0, \beta \geq 0, \alpha + \beta \leq 1$$

koşulları sağlandığı taktirde $\Delta V_0V_1V_2$ üçgeninin içinde yer alır. $P$ noktasının yukarıda gösterilen ifadesi üç bileşenden oluşmuştur:

$$x_p - x_0 = \alpha(x_1 - x_0) + \beta(x_2 - x_0)$$

$$y_p - y_0 = \alpha(y_1 - y_0) + \beta(y_2 - y_0)$$

$$z_p - z_0 = \alpha(z_1 - z_0) + \beta(z_2 - z_0)$$

Bu şekilde ifade edilen sistemin çözüm kümesi vardır ve tektir. Sistemi daha basite indirgemek için, poligonun temel düzlemlerden $(xy, xz, yz)$ herhangi birinin üzerine olan izdüşümü bulunur. Ancak, eğer poligon bu düzlemlerden herhangi birisine dik ise, bu düzlem üzerindeki izdüşümü sadece bir doğru parçasından ibaret olur. Bu problemden kaçınmak ve elde edilecek izdüşümün olabildiğince büyük olmasını sağlamak için, normal vektörüne ait en büyük bileşene dik olan düzlem kullanılır. $yz, xz, xy$ düzlemlerini $i_0, i_1, i_2$ ile ifade edersek, $i_0$ değerinin bulunuşu Şekil.9.2 üzerinde gösterilmiştir. $i_0$ bu yolla bulunduktan sonra, $i_1$ ve $i_2$ değerleri birbirlerinden ve $i_0$'dan farklı olacak şekilde (0,1,2) kümesinin elemanları arasından seçilerek elde edilebilirler. Bulunan $i_1$ ve $i_2$ değerleri, izdüşürülecek temel düzlemi ifade ederler.

$$i_0 = \begin{cases} 0 & \text{eğer } |N_x| = max(|N_x|, |N_y|, |N_z|) \\ 1 & \text{eğer } |N_y| = max(|N_x|, |N_y|, |N_z|) \\ 2 & \text{eğer } |N_z| = max(|N_x|, |N_y|, |N_z|) \end{cases}$$

Şekil 9.2. En büyük izdüşüme sahip düzlemin bulunuşu

İzdüşürülecek bu düzlem üzerinde herhangi bir vektörün iki boyutlu koordinatlarının $(u, v)$ olduğunu varsayarsak, $\overrightarrow{V_0P}$, $\overrightarrow{V_0V_1}$ ve $\overrightarrow{V_0V_2}$ vektörlerinin koordinatları

:

$$u_0 = P_{i_1} - V_{0_{i_1}} \qquad u_1 = V_{1_{i_1}} - V_{0_{i_1}} \qquad u_2 = V_{2_{i_1}} - V_{0_{i_1}}$$

$$v_0 = P_{i_2} - V_{0_{i_2}} \qquad v_1 = V_{1_{i_2}} - V_{0_{i_2}} \qquad v_2 = V_{2_{i_2}} - V_{0_{i_2}}$$

şeklinde ifade edilebilir. Dolayısıyla $P$ noktasının bileşenlerine ait denklem şu şekle indirgenmiş olur:

$$u_0 = \alpha u_1 + \beta u_2$$

$$v_0 = \alpha v_1 + \beta v_2$$

İki elemanlı bir denklem kümesine indirgenmiş bu problemin çözüm kümesi de kolaylıkla:

$$\alpha = \frac{det\begin{bmatrix} u_0 & u_2 \\ v_0 & v_2 \end{bmatrix}}{det\begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix}} \quad \text{ve} \quad \beta = \frac{det\begin{bmatrix} u_1 & u_0 \\ v_1 & v_0 \end{bmatrix}}{det\begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix}}$$

olarak bulunabilir. Daha sonra da bu değerler kullanılarak, $P$ kesişim noktasına ait olan $N_P$ normal vektörü

$$N_p = (1 - (\alpha + \beta))\, N_0 + \alpha N_1 + \beta N_2$$

şeklinde elde edilir. Şekil.9.3 üzerinde, bu bölümde anlatılan ışın-üçgen kesişimi

hesabını gerçekleştiren örnek bir algoritma gösterilmiştir.

O: nokta; Işının başlangıç noktası

D: Vektör Işının yönü

P: Nokta Kesişim noktası

V: [0..2] vektör dizisi Poligonun köşe noktaları

$P = O + Dt;$

$i_1$ ve $i_2$ poligon tanımından elde edilmiştir.

$u_0 = P[i_1] - V[0][i_1];$

$v_0 = P[i_2] - V[0][i_2];$

$u_1 = V[1][i_1] - V[0][i_1];$

$v_1 = V[1][i_2] - V[0][i_2];$

$u_2 = V[2][i_1] - V[0][i_1];$

$v_2 = V[2][i_2] - V[0][i_2];$

Eğer $u_1 = 0$ ise

$\beta = u_0/u_2;$

Eğer $0 \le \beta \le 1$ ise $\alpha = (v_0 - \beta * v_2)/v_1;$

değilse

$\beta = (v_0 * u_1 - u_0 * v_1)/(v_2 * u_1 - u_2 * v_1);$

Eğer $0 \le \beta \le 1$ ise $\alpha = (u_0 - \beta * u_2)/u_1;$

Döndür ( $(\alpha \ge 0)$ *ve* $(\beta \ge 0)$ *ve* ( $(\alpha + \beta) \le 1$ ) )

Şekil 9.3. Işın-üçgen kesişim algoritması

## 9.2. Hızlı ışın-poligon kesişim hesabının optimizasyonu

Birçok ışı- izleme sisteminde olduğu gibi, bu proje sırasında geliştirilen sistemde de, poligonlar karmaşık cisimlerin modellenmesinde kullanılan en küçük birimlerdir. Dolayısıyla ışın-poligon hesabının süresi, sistemin bütün olarak hızını oldukça fazla derecede etkilemektedir. Bu amaçla olabildiğince fazla seviyede optimizasyona ihtiyaç duyulmaktadır.

Birçok ışın-izleme sisteminde, ışın-poligon kesişimi iki adımda hesaplanır: Işın ile poligonun temel düzleminin kesişim kontrolü, ve düzlemle ışının kesişim noktasının poligon içinde olup olmadığı kontrolü. Eğer her poligon etrafında eksenlere paralel bir çevreleyen kutu olduğu düşünülürse, bu kutu varolan algoritmaları hızlandırmak amacıyla kullanılabilir: Işın ile poligonun temel düzlemi kesişim kontrolünden sonra, elde edilen kesişim noktası ile bu kutu karşılaştırılarak, poligonun içinde olup olmadığı kontrolünden kurtulunmuş olur. Bu kontrol, yaklaşık olarak iç-dış testi para ve en kötü ihtimalle fazladan altı karşılaştırma yapılmasına sebep olur. Ama normal koşullarda, bizi birçok ondalıklı sayı hesabından kurtarır.

Bu hızlı kontrol istenilen birçok algoritmayla ortak olarak kullanılabilir. Hızlı çevreleyen kutu hesabı ise 50. sayfadaki "IŞIN-KUTU KESİŞİMİ" bölümünde anlatılmıştır.

# 10. YÜZEY DOKUSUNA BAĞLI EFEKTLER

Bu bölümde, sistem tarafınndan simüle edilen yüzey dokusu örnekleri anlatılacaktır. Sistem, maddelerin kullanıcı tarafından belirtilen yüzey yapısına göre bir renk düzeni oluşturur. Bu renk düzeni oluşturulurken, Bölüm 10.1. üzerinde gösterilen, özel olarak tanımlanmış gürültü fonksiyonları kullanılır. Ayrıca tanımlanan madde cinsine göre bu gürültü fonksiyonlarının kullanılışı da Bölüm 10.2. üzerinde gösterilmiştir.

## 10.1. Yüzey oluşturmada kullanılan gürültü fonksiyonları

```
#define NUMPTS          512
#define P1              173
#define P2              263
#define P3              337
#define PHI             0.6180339
#define MAXINT          32767.0                         // 2**15 - 1
#define MININT          -32768.0                        // -2**15
#define MAXUINT         65536.0                         // 2**16
#define MAXUINT_2       65534.0                         // 2**16 - 2
#define RANDOM()        2.0*(real)rand()/RAND_MAX-1.0   // [-1,1]
static real Pts[NUMPTS];

// Real numbers might have values that are too large to be stored
// in an int. If this is the case we "fold" the value about MAXINT
// or MININT until it is of an acceptable size.
// NOTE: 16 bit integers are assumed.
static real intAdjust (real num)
    {
      while (1)
       {
         if (num > MAXINT)
            num= MAXUINT_2-num;
         else if (num < MININT)
                num= -(MAXUINT+num);
            else
                return num;
      }
   }

// Initialize the array of random numbers.
void NoiseInit (void)
```

```
    {
        randomize();
        srand(rand());
        for (int i=0; i<NUMPTS; i++)
            Pts[i]= RANDOM();
    }

real Noise (Vector * V)
    {
        Vector p;
        int    xi, yi, zi;
        int    xa, xb, xc, ya, yb, yc, za, zb, zc;
        real   xf, yf, zf;
        real   x2, x1, x0, y2, y1, y0, z2, z1, z0;
        real   p000, p100, p200, p010, p110, p210, p020, p120, p220;
        real   p001, p101, p201, p011, p111, p211, p021, p121, p221;
        real   p002, p102, p202, p012, p112, p212, p022, p122, p222;
        real   tmp;

        p.set(*V);
        p.set (intAdjust(p.x()), intAdjust(p.y()), intAdjust(p.z()));
        xi= floor (p.x());
        xa= floor (P1 * (xi * PHI - floor(xi * PHI)));
        xb= floor (P1 * ( (xi+1) * PHI - floor((xi+1) * PHI)));
        xc= floor (P1 * ( (xi+2) * PHI - floor((xi+2) * PHI)));
        yi= floor (p.y());
        ya= floor (P2 * (yi * PHI - floor(yi * PHI)));
        yb= floor (P2 * ( (yi+1) * PHI - floor((yi+1) * PHI)));
        yc= floor (P2 * ( (yi+2) * PHI - floor((yi+2) * PHI)));
        zi= floor (p.z());
        za= floor (P3 * (zi * PHI - floor(zi * PHI)));
        zb= floor (P3 * ( (zi+1) * PHI - floor((zi+1) * PHI)));
        zc= floor (P3 * ( (zi+2) * PHI - floor((zi+2) * PHI)));
        p000= Pts [ (xa + ya + za) % NUMPTS ];
        p100= Pts [ (xb + ya + za) % NUMPTS ];
        p200= Pts [ (xc + ya + za) % NUMPTS ];
        p010= Pts [ (xa + yb + za) % NUMPTS ];
        p110= Pts [ (xb + yb + za) % NUMPTS ];
        p210= Pts [ (xc + yb + za) % NUMPTS ];
        p020= Pts [ (xa + yc + za) % NUMPTS ];
        p120= Pts [ (xb + yc + za) % NUMPTS ];
        p220= Pts [ (xc + yc + za) % NUMPTS ];
        p001= Pts [ (xa + ya + zb) % NUMPTS ];
        p101= Pts [ (xb + ya + zb) % NUMPTS ];
        p201= Pts [ (xc + ya + zb) % NUMPTS ];
        p011= Pts [ (xa + yb + zb) % NUMPTS ];
```

```
        p111= Pts [ (xb + yb + zb) % NUMPTS ];
        p211= Pts [ (xc + yb + zb) % NUMPTS ];
        p021= Pts [ (xa + yc + zb) % NUMPTS ];
        p121= Pts [ (xb + yc + zb) % NUMPTS ];
        p221= Pts [ (xc + yc + zb) % NUMPTS ];
        p002= Pts [ (xa + ya + zc) % NUMPTS ];
        p102= Pts [ (xb + ya + zc) % NUMPTS ];
        p202= Pts [ (xc + ya + zc) % NUMPTS ];
        p012= Pts [ (xa + yb + zc) % NUMPTS ];
        p112= Pts [ (xb + yb + zc) % NUMPTS ];
        p212= Pts [ (xc + yb + zc) % NUMPTS ];
        p022= Pts [ (xa + yc + zc) % NUMPTS ];
        p122= Pts [ (xb + yc + zc) % NUMPTS ];
        p222= Pts [ (xc + yc + zc) % NUMPTS ];
        xf= p.x() - xi;
        x1= xf * xf;
        x2= 0.5 * x1;
        x1= 0.5 + xf - x1;
        x0= 0.5 - xf + x2;
        yf= p.y() - yi;
        y1= yf * yf;
        y2= 0.5 * y1;
        y1= 0.5 + yf - y1;
        y0= 0.5 - yf + y2;
        zf= p.z() - zi;
        z1= zf * zf;
        z2= 0.5 * z1;
        z1= 0.5 + zf - z1;
        z0= 0.5 - zf + z2;
        tmp = z0 * (y0 * (x0 * p000 + x1 * p100 + x2 * p200) +
                          y1 * (x0 * p010 + x1 * p110 + x2 * p210) +
                          y2 * (x0 * p020 + x1 * p120 + x2 * p220));
        tmp += z1 * (y0 * (x0 * p001 + x1 * p101 + x2 * p201) +
                          y1 * (x0 * p011 + x1 * p111 + x2 * p211) +
                          y2 * (x0 * p021 + x1 * p121 + x2 * p221));
        tmp += z2 * (y0 * (x0 * p002 + x1 * p102 + x2 * p202) +
                          y1 * (x0 * p012 + x1 * p112 + x2 * p212) +
                          y2 * (x0 * p022 + x1 * p122 + x2 * p222));
        return tmp;
    }

Vector VectNoise (Vector * P)
    {
        int     xi, yi, zi;
        int     xa, xb, xc, ya, yb, yc, za, zb, zc;
        real    xf, yf, zf;
```

```
real   x2, x1, x0, y2, y1, y0, z2, z1, z0;
real   xd2, xd1, xd0, yd2, yd1, yd0, zd2, zd1, zd0;
real   p000, p100, p200, p010, p110, p210, p020, p120, p220;
real   p001, p101, p201, p011, p111, p211, p021, p121, p221;
real   p002, p102, p202, p012, p112, p212, p022, p122, p222;
real   tmpx, tmpy, tmpz;

xi= floor (P->x());
xa= floor (P1 * (xi * PHI - floor(xi * PHI)));
xb= floor (P1 * ( (xi+1) * PHI - floor((xi+1) * PHI)));
xc= floor (P1 * ( (xi+2) * PHI - floor((xi+2) * PHI)));
yi= floor (P->y());
ya= floor (P2 * (yi * PHI - floor(yi * PHI)));
yb= floor (P2 * ( (yi+1) * PHI - floor((yi+1) * PHI)));
yc= floor (P2 * ( (yi+2) * PHI - floor((yi+2) * PHI)));
zi= floor (P->z());
za= floor (P3 * (zi * PHI - floor(zi * PHI)));
zb= floor (P3 * ( (zi+1) * PHI - floor((zi+1) * PHI)));
zc= floor (P3 * ( (zi+2) * PHI - floor((zi+2) * PHI)));
p000= Pts [ (xa + ya + za) % NUMPTS ];
p100= Pts [ (xb + ya + za) % NUMPTS ];
p200= Pts [ (xc + ya + za) % NUMPTS ];
p010= Pts [ (xa + yb + za) % NUMPTS ];
p110= Pts [ (xb + yb + za) % NUMPTS ];
p210= Pts [ (xc + yb + za) % NUMPTS ];
p020= Pts [ (xa + yc + za) % NUMPTS ];
p120= Pts [ (xb + yc + za) % NUMPTS ];
p220= Pts [ (xc + yc + za) % NUMPTS ];
p001= Pts [ (xa + ya + zb) % NUMPTS ];
p101= Pts [ (xb + ya + zb) % NUMPTS ];
p201= Pts [ (xc + ya + zb) % NUMPTS ];
p011= Pts [ (xa + yb + zb) % NUMPTS ];
p111= Pts [ (xb + yb + zb) % NUMPTS ];
p211= Pts [ (xc + yb + zb) % NUMPTS ];
p021= Pts [ (xa + yc + zb) % NUMPTS ];
p121= Pts [ (xb + yc + zb) % NUMPTS ];
p221= Pts [ (xc + yc + zb) % NUMPTS ];
p002= Pts [ (xa + ya + zc) % NUMPTS ];
p102= Pts [ (xb + ya + zc) % NUMPTS ];
p202= Pts [ (xc + ya + zc) % NUMPTS ];
p012= Pts [ (xa + yb + zc) % NUMPTS ];
p112= Pts [ (xb + yb + zc) % NUMPTS ];
p212= Pts [ (xc + yb + zc) % NUMPTS ];
p022= Pts [ (xa + yc + zc) % NUMPTS ];
p122= Pts [ (xb + yc + zc) % NUMPTS ];
p222= Pts [ (xc + yc + zc) % NUMPTS ];
```

```
xf= P->x() - xi;
x1= xf * xf;
x2= 0.5 * x1;
x1= 0.5 + xf - x1;
x0= 0.5 - xf + x2;
xd2= xf;
xd1= 1.0 - xf - xf;
xd0= xf - 1.0;
yf= P->y() - yi;
y1= yf * yf;
y2= 0.5 * y1;
y1= 0.5 + yf - y1;
y0= 0.5 - yf + y2;
yd2= yf;
yd1= 1.0 - yf - yf;
yd0= yf - 1.0;
zf= P->z() - zi;
z1= zf * zf;
z2= 0.5 * z1;
z1= 0.5 + zf - z1;
z0= 0.5 - zf + z2;
zd2= zf;
zd1= 1.0 - zf - zf;
zd0= zf - 1.0;
tmpx= z0 * (y0 * (xd0 * p000 + xd1 * p100 + xd2 * p200) +
            y1 * (xd0 * p010 + xd1 * p110 + xd2 * p210) +
            y2 * (xd0 * p020 + xd1 * p120 + xd2 * p220));
tmpx += z1 * (y0 * (xd0 * p001 + xd1 * p101 + xd2 * p201) +
              y1 * (xd0 * p011 + xd1 * p111 + xd2 * p211) +
              y2 * (xd0 * p021 + xd1 * p121 + xd2 * p221));
tmpx += z2 * (y0 * (xd0 * p002 + xd1 * p102 + xd2 * p202) +
              y1 * (xd0 * p012 + xd1 * p112 + xd2 * p212) +
              y2 * (xd0 * p022 + xd1 * p122 + xd2 * p222));
tmpy= z0 * (yd0 * (x0 * p000 + x1 * p100 + x2 * p200) +
              yd1 * (x0 * p010 + x1 * p110 + x2 * p210) +
              yd2 * (x0 * p020 + x1 * p120 + x2 * p220));
tmpy += z1 * (yd0 * (x0 * p001 + x1 * p101 + x2 * p201) +
              yd1 * (x0 * p011 + x1 * p111 + x2 * p211) +
              yd2 * (x0 * p021 + x1 * p121 + x2 * p221));
tmpy += z2 * (yd0 * (x0 * p002 + x1 * p102 + x2 * p202) +
              yd1 * (x0 * p012 + x1 * p112 + x2 * p212) +
              yd2 * (x0 * p022 + x1 * p122 + x2 * p222));
tmpz= zd0 * (y0 * (x0 * p000 + x1 * p100 + x2 * p200) +
                  y1 * (x0 * p010 + x1 * p110 + x2 * p210) +
                  y2 * (x0 * p020 + x1 * p120 + x2 * p220));
tmpz += zd1 * (y0 * (x0 * p001 + x1 * p101 + x2 * p201) +
```

```
                        y1 * (x0 * p011 + x1 * p111 + x2 * p211) +
                        y2 * (x0 * p021 + x1 * p121 + x2 * p221));
    tmpz += zd2 * (y0 * (x0 * p002 + x1 * p102 + x2 * p202) +
                        y1 * (x0 * p012 + x1 * p112 + x2 * p212) +
                        y2 * (x0 * p022 + x1 * p122 + x2 * p222));
    return Vector(tmpx, tmpy, tmpz);
  }

real Turbulence (Vector v, int octaves)
  {
    Vector tmp;
    real scale= 1.0,
         t    = 0.0;

    while (octaves--)
     {
       tmp.set(v*scale);
       t += (Noise(&tmp) / scale);
       scale= scale+scale;
     }
    return t;
  }
```

## 10.2. Madde cinsine göre gürültü fonksiyonu kullanımı

### 10.2.1. Ağaç Yüzeyler

```
#define BOARDSIZE 45.0
Color getWoodColor (Point p, real intensity)
  {
    Vector v(p);
    real   chaos,
           tmpx, tmpy, tmpz,
           y, z;

    v.set(v.x()+0.08, v.y(), v.z());
    v.set(v*2);
    chaos= Turbulence (v, 5) * 0.5;

    // Make the pattern "semi"-periodic so it looks as if
    // a new board is used at regular intervals.
```

```
y= v.y();
z= v.z();
if (z > 0)
   {
      tmpz= floor ( (z+BOARDSIZE*0.5) / BOARDSIZE);
      z -= tmpz * BOARDSIZE;
   }
else
   {
      tmpz= floor ( (BOARDSIZE * 0.5 - z) / BOARDSIZE);
      z += tmpz * BOARDSIZE;
   }
if (y > 0)
   {
      tmpy= floor ( (y+BOARDSIZE*0.5) / BOARDSIZE);
      y -= tmpy * BOARDSIZE;
   }
else
   {
      tmpy= floor ( (BOARDSIZE * 0.5 - y) / BOARDSIZE);
      y += tmpy * BOARDSIZE;
   }
// Skew the "stem" a bit so the "cylinders" are not perfectly symmetric
// about the x-axis. Skew the different "logs" slightly differently.
tmpx= 0;
Vector tmp(tmpx, tmpy, tmpz);
real skewoff= Noise(&tmp);
real dummy= (0.05 + 0.03 * skewoff) * (p.x() - 2.0);
z -= dummy;
y -= dummy;

// Calculate the distance from the middle of the "stem" and
// distort this distance with the turbulence value.
real rad= sqrt(y*y + z*z) + chaos,
      val= rad - floor(rad);

// Choose a color dependent on the distorted distance. */
Color C;
if (val < 0.9)
   {
      if (val < 0.1)
            C.set(Base);
      else
         {
            real k= val / 0.8 - 0.1,
               k3= k*k*k,
```

```
                k6= k3*k3;
              real t= 1.0 - k6;
              C.set(Ring+(Base_Ring*t));
            }
        }
    else
        C.set(Ring);

    return C*intensity;
}
```

### 10.2.2. Granit Yüzeyler

```
Color getGraniteColor (Point p, real intensity)
  {
    Vector v(p);
    real   n    = 0.5,
           freq = 1.0;

    for (byte i=0; i<6; i++)
      {
        freq= freq+freq;              // freq = 2*freq
        v.set(v*(freq+freq));         // v= v*4*freq
        real temp= 0.5 * Noise(&v);
        n += temp/freq;
      }
    return ((Base*n + Strip*(1-n))*intensity);
  }
```

### 10.2.3. Mermer yüzeyler

```
Color getMarbleColor (Point p, real intensity)
  {
    real x, t;

    x= p.x() + Turbulence(p,7)*5;
    x= sin(x);
    if (x>-0.1 && x<0.1)
          return (Strip*intensity);
    else if ( x > -0.9 && x < 0.9 )
```

```
        {
            t= 7.0/6.0-1.0/(6.25*fabs(x)+0.375);
            return ((Strip + Base_Strip*t)*intensity);
        }
    else
            return (Base*intensity);
}
```

# 11. MATEMATİKSEL İŞLEMLERİ HIZLANDIRICI YÖNTEMLER

Bu çalışma içerisinde kullanılan metodların mantık olarak optimize edilmesi, işlem hızı açısından her zaman yeterli olmamaktadır. Bazı yöntemler içerisinde ek hızlandırma çabalarına ihtiyaç duyulmuştur. Bu amaçla kullanılan hızlandırıcı yöntemler bu bölüm içerisinde kısaca anlatılmaya çalışılmıştır.

## 11.1. Kök Bulma Yöntemleri

Bir denklemin köklerinin bulunması, bilgisayar grafik çalışmalarında çok karşılaşılan bir problemdir. Özellikle 3-boyutlu ortamda yapılan ışın-izleme işlemleri sırasında, ışın-nesne kesişimi hesabı, ışın denklemi ile nesne denkleminin ortak çözüm kümesinin köklerinin bulunması sayesinde elde edilir. Bu bölüm altındaki alt başlıklar içerisinde çalışna içinde kullanılan sayısal kök bulma yöntemleri kısaca tanıtılacaktır.

### 11.1.1. Kübik ve Kuartik Kökler

Işın-izleme çalışmaları sırasında kullanılan nesneler karmaşıklaştıkça, çözülmesi gereken ışın-nesne kesişim noktaları hesap denklemleri de o kadar zorlaşmaktadır. İteratif yöntemler çoğunlukla yavaş ve sayısal olarak kararsız olmaktadır. Bu tip yöntemler tarafından ihtiyaç duyulan başlama değerleri tahmini ve gerçel kök sayısının bulunması oldukça zor olmaktadır.

Bu tip köklerin bulunmasında kullanılan analitik bir yöntem aşağıda gösterilmiştir. Bu çözüm içinde karmaşık sayı hesabı kullanılmamıştır.

#### 11.1.1.1. Kübik Denklem Kökleri Bulunuşu

Önce standart kübik denklem gösterilimini ele alarak işe başlayalım.

$$c_3x^3 + c_2x^2 + c_1x + c_0 = 0$$

şeklinde olan denklem önce $c_3$'e bölünür ve

$$x^3 + Ax^2 + Bx + C = 0$$

normal formu elde edilir. Bu denklem içerisinde $x = y - \frac{A}{3}$ değiştirilmesi yapılırsa, elde edilen denklem

$$y^3 + 3py + 2q = 0$$

şeklinde olur. Bu denklemin determinantı, Cardano formülüne dayanarak

$$D = q^2 + p^3$$

biçiminde elde edilir. Denklemin kökleri ise $(u, v = \sqrt[3]{(-q \pm \sqrt{D})}$ olduğu kabul

edilirse):

$$y_1 = u + v$$

$$y_{2,3} = -\frac{u+v}{2} \pm \frac{\sqrt{3}}{2}(u-v)\,i$$

olarak bulunur. Elde edilen bu kök değerleri için üç değişik durum sözkonusu olabilir:

| | |
|---|---|
| $D > 0$ | bir gerçel ($y_1$), iki eşlenik karmaşık ($y_2, y_3$) değer |
| $D = 0$ | iki gerçel değer, $y_2 = y_3$ |
| $D < 0$ | üç değişik gerçel değer |

Determinantın sıfırdan küçük olması durumunda ($D < 0$), trigonometrik çözüm yardımıyla karmaşık aritmetik kullanılmadan üç kök değerinin de bulunması aşağıdaki gibi sağlanmış olur:

$$\cos\varphi = -\frac{q}{\sqrt{-p^3}}$$

$$y_1 = 2\sqrt{-p}\cos\frac{\varphi}{3}$$

$$y_{2,3} = -2\sqrt{-p}\cos\frac{\varphi \pm \pi}{3}$$

Bu değerlerin ilk denklemde yerlerine konulması sonucu x değişkeninin gerçek değerleri bulunabilir.

### 11.1.1.2. Kuartik Denklem Köklerinin Bulunuşu

Önce standart kuartik denklem gösterilimini ele alarak işe başlayalım.

$$c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 = 0$$

şeklinde olan denklem önce $c_4$'e bölünür ve

$$x^4 + Ax^3 + Bx^2 + Cx + D = 0$$

normal formu elde edilir. Bu denklem içerisinde $x = y - \frac{A}{4}$ değiştirilmesi yapılırsa, elde edilen denklem

$$y^4 + py^2 + qy + r = 0$$

şeklinde olur. Bunun sonucunda denklemin çözüm kübik denklemi

$$z^3 - \frac{p}{2}z^2 - rz + \frac{rp}{2} - \frac{q^2}{8} = 0$$

olur. Yukarıda gösterilen denklemin bir kökünün z olduğu gözönüne alınarak, kuartik

denklemin kökleri

$$y^2 \pm y\sqrt{2z-p} + z \mp \sqrt{z^2 - r} = 0$$

denklemlerinin çözülmesi ile elde edilebilir. Buradan elde edilecek değerler ilk denklemde yerine konulursa x değişkeninin gerçek kök değerleri elde edilmiş olur.

### 11.1.2. Bézier Eğrilerine Dayanarak Kök Bulma

Matematik, özellikle geometri ve doğrusal cebir, bilgisayar grafik çalışmalarında çok ihtiyaç duyulan bir bilim dalıdır. Çalışmalar sırasında karşılaşılan en önemli problem tipinin fonksiyonların köklerinin bulunması olduğuna daha önce değinilmişti. Ancak kökü bulunacak fonksiyon her zaman iki eğrinin kesişimi kadar basit olmamaktadır. Özellikle çok yüksek dereceli poligonlar sözkonusu olduğu zaman, kapalı bir çözüm kümesi bulunamaması sık karşılaşılan bir problemdir.

Bu bölümde, yüksek dereceli denklemlerin bütün köklerinin bulunmasında kullanılan bir algoritma anlatılacaktır. Ele alınan algoritma ikiye-bölme (bisection) metodunun değiştirilmiş bir şeklidir. Bu yöntem, Bézier eğrilerinin belirli özelliklerinden yararlanarak kapalı bir formu bulunmayan fonksiyonların bütün köklerinin bulunmasını sağlar. Bilgisayar çevrelerinde bu tip denklemlere genellikle Newton-Raphson iterasyon yöntemiyle yaklaşılır. Bu yöntemin temel amacı tahmini bir başlangıç çözüm değeri kullanılarak gerçek kök değerine yaklaşmaktan ibarettir. Ancak, Newton yöntemini anlatan kitapların hemen hemen hepsi "elimizde bir u başlangıç değerinin olduğunu varsayarsak, ..." şeklinde bir giriş yaparlar. Fakat uygulamalarda eğer bu ilk değer doğru olarak seçilemezse iterasyon adımları genellikle ıraksar (çözüm bulunamaz). Burada ele alınan metod için ise böyle bir başlangıç değerine ihtiyaç yoktur.

Metod en basit haliyle iki adımda ifade edilebilir: Fonksiyonu Bernstein-Bézier formuna

çevir, ve denklemin köklerini elde edilen Bézier eğrisinin yatay sıfır ekseni ile kesişimlerini bularak elde et.

### 11.1.2.1. Bernstein-Bézier Formuna Dönüştürme

Fonksiyonel eğriler genellikle $y = f(x)$ şeklinde olur. Burada $f$ herhangi bir polinom olarak karşımıza çıkar. Bu denklem parametrik bir polinom olarak yeniden düzenlenirse şu yapıda bir denklem elde edilir:

$$Graph(t) = (x(t), y(t)) = (X, Y)$$

Şimdi cevap bulunması gereken soru, $Graph$ fonksiyonu için Bézier kontrol poligonunun bulunmasıdır. Başka bir deyişle, öyle $V_i = (x_i, y_i)$ kontrol noktaları seçelim ki bu noktalar

$$(X, Y) = \sum_{i=0}^{n} V_i B_i^n(t)$$

düzlemini sağlasın. Buradaki $x$ ve $y$ bileşenleri birbirlerinden bağımsızdır. Dolayısıyla yukarıdaki denklemin esasında iki ayrı skalar denklemden oluştuğu söylenilebilir:

$$X = \sum_{i=0}^{n} x_i B_i^n(t)$$

$$Y = \sum_{i=0}^{n} y_i B_i^n(t)$$

Bézier eğrilerinin lineer kesinlik özelliğine dayanarak, parametrik polinom gösterilimi aşağıdaki şekilde yeniden yazılabilir:

$$Graph\,(t) \;=\; \sum_{i=0}^{n} (\frac{i}{n}, y_i) B_i^n\,(t)$$

Bu ifade, *Graph* fonksiyonunun Bernstein-Bézier formuna ait kontrol noktalarının [0,1] aralığında eşit olark dağıldığını gösterir.

Böylece problem Y'nin katsayılarını bulmaya indirgenmiş olur. Bu katsayılar *Graph* fonksiyonun Bernstein-Bézier formuna ait kontrol noktalarının y bileşenlerine karşılık gelir. Bundan sonraki adım, üslü gösterimden Bézier gösterilimine geçiştir. Elimizde

$$P\,(t) \;=\; \sum_{i=0}^{n} A_i t^i$$

$$t \in [0,1]$$

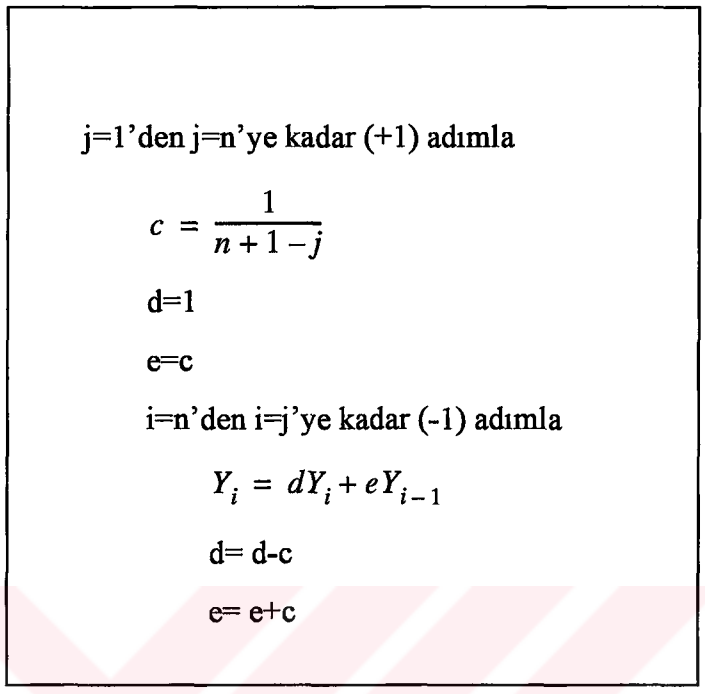
olduğu bilinerek öyle $\{Y_i\}_0^n$ değerleri arıyoruz ki

$$P\,(t) \;=\; \sum_{i=0}^{n} \binom{n}{0} t^i\,(1-t)^{n-i}$$

denklemi gerçeklensin.

Bu tip bir dönüşüm için kullanılabilecek basit bir algoritma Şekil.11.1 üzerinde

gösterilmiştir.




Şekil 11.1. Bernstein-Bézier katsayılarının bulunması

Elde edilen $\{Y_i\}_0^n$ değerleri bulunmak istenen katsayılardır. Bunun sonucunda ortaya çıkan denklemin en son hali şu şekilde yazılabilir:

$$Graph\,(t) = \sum_{i=0}^{n} \left(\frac{i}{n}, Y_i\right) B_i^n\,(t)$$

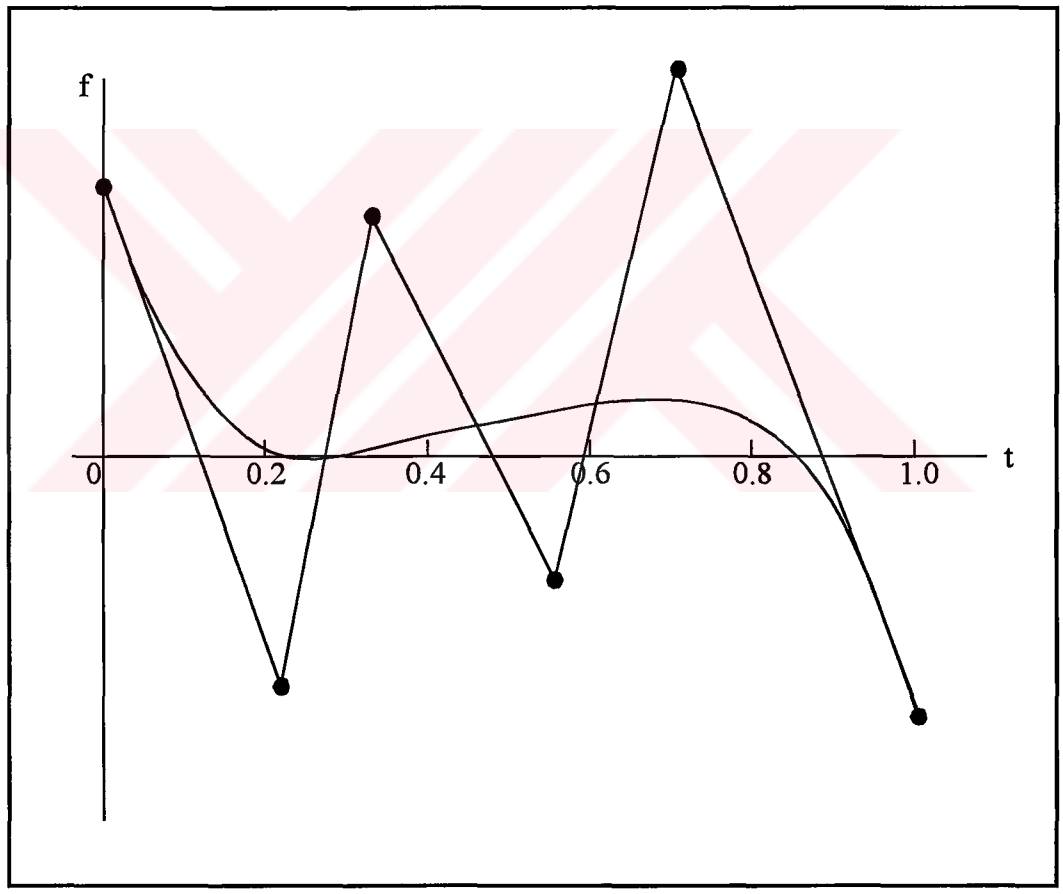
$$t \in [0, 1]$$

Yukarıdaki denklemlerden de görülebileceği gibi, ortaya çıkan sonuç polinomu sadece

[0,1] aralığı için geçerlidir. Ancak bu aralığa düşmeyen polinomlar için de Cauchy kriteri kullanılarak bu polinoma ait kök sınırlayıcı aralık bulunur, ve daha sonra bu aralık [0,1] üzerine eşlenerek gerekli dönüşüm yapılabilir.

### 11.1.2.2. Köklerin Bulunması

Kökleri bulunması istenlen bir eğri örneği Şekil.11.2 üzerinde gösterilmiştir. Örnek olarak alınan fonksiyon beşinci dereceden bir polinom olarak seçilmiştir. Bu polinomun kök değerleri, Bernstein-Bézier eğrisinin yatay ekseni (t-ekseni) kestiği noktalardır.




Şekil 11.2. Beşinci dereceden bir polinomun Bézier formu

Genellikle üslü formda verilen polinomların katsayıları, fonksiyonun oluşturduğu eğri hakkında hiçbir bilgi vermezler; dolayısıyla kök bulunuşu işlemine doğrudan katkıda bulunmazlar. Ancak, Bézier eğrilerinin katsayıları (yani kontrol noktaları değerleri) eğrinin şekliyle doğrudan ilişkilidir. Dolasıyıla, sözkonusu eğrinin köklerinin bulunumasında Bézier eğrilerinin birçok karakteristik özelliklerinden faydalanılır. Bu karakteristik özellikler şunlardır:

1. Dışbükey gövde (convex hull) özelliği, yani, Bézier eğrisi her zaman kendisine ait kontrol noktalarının oluşturduğu gövdenin içinde kalır,

2. Değişim azaltıcı (variation-diminishing) özelliği, yani, Bézier eğrisi kontrol noktalarının oluşturduğu poligonun içinde bu poligonun daha "yumuşatılmış" bir hali olarak yer alır. Daha açık ifade etmek gerekirse, herhangi bir doğru ile Bézier eğrisinin kesişim noktası sayısı, aynı doğrunun Bézier kontrol poligonu ile oluşturduğu kesişim noktası sayısından daha fazla olamaz.

3. Bézier eğrisinin iteratif olarak bölünmesi sonucunda limite ulaşıldığında, eğrinin üzerine yakınsayan kontrol noktaları elde edilir.

Bu gözlemler sonucunda ortaya kendi kendini çağırarak çözüme ulaşan bir algoritma çıkar. Bu algoritma Şekil.11.3 üzerinde gösterilmiştir. Algoritma, işe, Bézier kontrol noktaları listesindeki (katsayı listesi) işaret değişimlerini sayarak başlamaktadır. Elde edilen sayı bize, kontrol poligonun t-eksenini kesme sayısını vermektedir. Descartes'ın işaret kanununa ve Bézier eğrilerinin değişim azaltıcı özelliğine dayanarak, belirli bir aralıkta bulunan kök sayısının bu kesim sayısına eşit veya daha küçük olacağı sonucuna ulaşabiliriz. Eğer hiç işaret değişimi yoksa, Bézier eğrisi t-eksenin kesemez ve dolayısıyla bu aralık içinde eğrinin kökü olmadığına karar veririz. Eğer işaret değişimi sayısı bir ise, kontrol poligonunun "yeterince düz" olup olmadığını veya kendini kendini

çağırma sayısının limite ulaşıp ulaşmadığını kontrol ederiz. "Yeterince düz" terimi, kontrol poligonunun düz bir çizgiye çok yakın olması durumunu ifade etmektedir, ki bu durumda bu aralıktaki kök değeri, kontrol poligonunun ilk ve son noktalarını birleştiren kirişin t-eksenini kestiği nokta olarak kabul edilebilir. Bu kararı, Bézier eğrilerinin dışbükey gövde ve değişim azaltıcı özelliklerine dayanarak kolaylıkla verebiliriz.

```
KökBul (V)
        V: Bézier gösteriliminin kontrol noktaları
        Eğer kontrol poligonu t-eksenini kesmiyorsa
                KökDeğeri= KökYok
                Dön
        Eğer (kontrol poligonu t-eksenini bir kere kesiyorsa) VE (kontrol
        poligonu "yeterince düz" VEYA kendi kendini çağırma limitine
        ulaşıldıysa)
                KökDeğeri= V[0] - V[n] arasında çizilen kirişin
                t-eksenini kestiği nokta
                Dön
        Bézier eğrisini (V) deCasteljau algoritmasını kullanarak ortadan
        ikiye böl.
        KökBul (sol yarı)
        KökBul (sağ yarı)
        KökDeğeri= Elde edilen bütün kök değerleri
        Dön
```

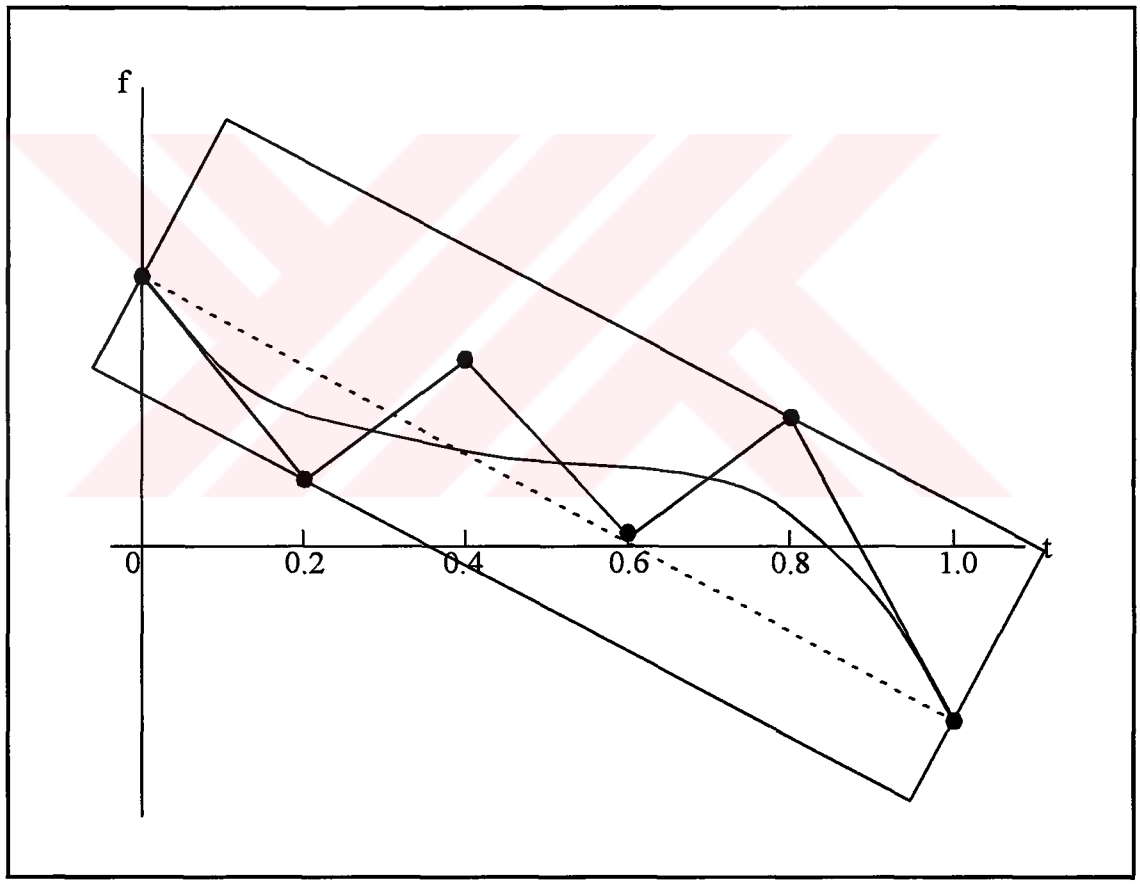
Şekil 11.3. Kök bulma algoritması

Birden fazla işaret değişimi olması durumunda, ilgili aralık içerisinde birden fazla kök değeri olabilir. Bu durumda kontrol poligonunu, deCasteljau algoritmasını kullanarak aralığın ortasından ikiye böleriz. Bu işlem bize ilk Bézier eğrisinin yarısını temsil eden iki yeni Bézier eğrisi verir. Daha sonra KökBul algoritmasını her iki yarı eğri parçası için yeniden çağırınız. Her yeniden çağırma işlemi sonucunda elde edilen kök değerleri bir araya getirilerek, esas Bézier eğrisinin kök kümesi oluşturulur.

Kontrol noktasının yeterince düz olmamasından veya t-ekseninin birden daha fazla sayıda kesilmesinden dolayı Bézier eğrisini tekrarlayan bir şekilde bölmemiz sonucunda ortaya giderek küçülen alt-eğriler çıkmaya başlar. deCateljau algoritmasının uygulanması sonucu ortaya çıkan her eğrinin orjinal eğriden daha küçük olmasından, ve yeni kontrol noktalarının eğriye doğru yakınsamasından dolayı, KökBul algoritmasının belirli bir süre sonra eğrinin t-eksenini kestiği noktaya ulaşacağını rahatlıkla söyleyebiliriz.

Önceki paragraflarda da belirtildiği gibi, ilgili polinoma ait kontrol poligonu yeterince düz olduğu zaman (kontrol poligonu doğru parçası şekline yeterince yaklaştığı ve t-eksenini sadece bir kere kestiği zaman) yöntemin tekrarlanması durdurulabilir. Bu durumda, ilk ve son kontrol noktaları arasında çizilen doğru parçasının t-eksenini kestiği nokta, aranılan kök değerini yaklaşık olarak verir. Bundan sonraki problem, düzlük miktarının yeterli olduğuna nasıl karar verileceğidir. Çünkü bu karar kabul edilebilecek hata oranını göstermektedir. t-eksenini sadece bir kere kesen bir kontrol poligonu için (örneğin Şekil.11.4) bir çevreleyen kutu tanımlarsak, bu kutu, üst ve alt kenarları poligonun ilk ve son noktalarını birleştiren doğru parçasına paralel olan bir dikdörtgen olur. Bu durumda, çözüm içindeki hata oranı, çevreleyen kutunun t-eksenini kesttiği noktalar arasındaki veya (kesişim noktasının/noktalarının [0,1] aralığı dışında kalması durumunda) bu noktalar ile [0,1] aralığının uç noktaları arasındaki mesafenin yarısından daha az olacaktır. Bu şekilde elde edilen hata oranı kabul edilebilir bir seviyeye indiği zaman, o andaki ilk ve son kontrol noktası arasındaki doğru parçasının t-eksenini kestiği noktayı denklemin kökü olarak kabul edebiliriz. Çözüm işlemi sırasında eğriyi ikiye

bölerek ilerlediğimiz ve elimizdeki t değerlerinin [0,1] arasında buluduğunu bildiğimiz için, bölünme sayısı ile hata oranı arasında bir ilişki kurabiliriz. Eğriyi bölmeye başlamadan önce t aralığı [0,1] olduğundan maksimum hata oranı 0.5'i aşamaz. İlk bölme sonucunda aralık ikiye bölüneceğinden, hata oranı da 0.25'e iner. Takip eden her bölme işlemi hata oranını da ikiye bölerek ilerler, ve kabul edilebilecek hata oranının 2 tabanına göre logaritması alındığında elde edilen sayı kadar bölünme sonucunda, istediğimiz doğruluktaki kök değerini bulmuş oluruz. Örneğin, $\frac{1}{128}\left(\frac{1}{2^7}\right)$ oranında bir hatayı kabul edebiliyorsak, 7 adım sonra arzu ettiğimiz kök değerine ulaşabiliriz.




Şekil 11.4. Beşinci-dereceden bir polinom için "çevreleyen kutu"

Bu algoritma, Newton-Raphson metodunun ihtiyaç duyduğu başlangıç değeri ve tanım

aralığını bulmak için de kullanılabilir. Önceki paragraflarda da ifade edildiği gibi bu algoritma, t-ekseni bir defa kesildiği durumlarda içinde kök değeri bulunan bir aralık belirler. Bu aralığın orta noktası, Newton-Raphson yöntemi için başlangıç değeri olarak seçilebilir. Bu şekilde kullanımdan, çok yüksek dereceli polinomlarının köklerinin bulunmasında yararlanılabilinir. Çünkü Bézier eğrisinin çok fazla sayıda ikiye bölünmesi, ondalıklı sayı hataları oluşturmaya başlar ki, bu da elde edilen kök değerlerinin hatalı olmasına yol açabilir.

## 11.2. Uzaklık Bulma Yöntemleri

Uzaklık hesabı bilgisayar grafik işlemlerinde çok karşılaşılan önemli bir problemdir. Genellikle böyle bir hesap sonucunda çok kesin değer istenilmez. Örneğin, üç boyutlu bir çizim programı yazdığımızı varsayalım, ve kullanıcının şekil seçmesi problemi ile uğraştığımızı düşünelim. Kullanıcı işaret edici aygıt yardımıyla ekrandaki bir noktayı seçer ve en yakında bulunan nesne seçilir. Burada yapılan işlem kullanıcı ile etkileşimli olduğundan, hesapların çok kısa sürede sonuçlanması gereklidir. Dolayısıyla seçilen nokta ile diğer nesneler arasındaki mesafelerin çok kesin değerlerinin hesaplanmasına gerek yoktur. Nokta ile nesneler arasındaki mesafelerin tahmini değerleri kullanılarak da aynı sonuca kolaylıkla varılabilir. Eğer yanlış nesne seçilirse, kullanıcı genellikle seçmek istediği şekle biraz daha yaklaşarak seçme işlemini tekrarlar.

Takip eden alt-bölümlerde iki ve üç boyutlu ortamlarda uzaklık hesaplarında kullanılabilecek hızlandırma yöntemleri anlatılmıştır. Burada dikkat edilecek nokta, $d = f(d_1, d_2)$ şeklinde olan iki boyutlu bir uzaklık denklemi, üç boyutlu ortama ($d_3 = f(d_1, f(d_2, d_3))$ şeklinde yazılarak ve $d_3$ için çözülerek) kolaylıkla geçirilebilir. Aynı çözüm yöntemi istenildiği taktirde daha yüksek boyutlar için de kullanılabilir.

### 11.2.1. Hızlı ve Az Duyarlı Bir Karekök Hesabı

Geleneksel karekök bulma yöntemleri bilindiği üzere iteratif yöntemler kullanırlar.

Dolayısıyla bu yöntemler genellikle etkileşimli sistemler için oldukça yavaş kalmaktadırlar. Ayrıca yapılacak karekök hesabı sayısı arttıkça, bu tip metodların kullanım cazibesi oldukça azalmaktadır. Bu tip yöntemlerde elde edilen karekök değerleri, genellikle ihtiyaç duyulandan daha hassastırlar.

Dolayısıya, sadece birkaç basamaklık hassasiyet istenildiği durumlarda çok daha hızlı bir yöntem kullanılabilir. Teknik, karekökü bulunması istenen sayının ondalıklı bölümünün en anlamlı bitlerini anahtar olarak kullanarak, önceden hazırlanmış bir karekök tablosu içinden uygun değerleri bulmak olarak özetlenebilir. Bu şekilde kurulan metod, tablodan elde edilen değeri kullanarak, geleneksel yöntemlerden çok daha hızlı olacak biçimde ihtiyaç duyulan kök değerini vermektedir.

## 12. TARGA GÖRÜNTÜ FORMATI

Birçok ışın-izleme sistemi, oluşturulan sonuç görüntü için değişik formatlar kullanmaktadır. Bu çalışmada ise Targa görüntü formatı kullanılmıştır. Targa'nın seçilmesindeki sebep, bu formatın 16777216 ($2^{24}$) rengi desteklemesidir. Bu şekilde oluşturulan görüntü birçok çözünürlük seviyelerinde olabildiğince çok sayıda rengi destekler. Değişik çözünürlükler için kullanılabilecek maksimum renk sayısı Şekil.12.1 üzerinde gösterilmiştir.

| Çözünürlük | Kullanılabilecek maksimum renk sayısı (*) |
|---|---|
| 320 x 200 | 64.000 |
| 640 x 480 | 307.200 |
| 800 x 600 | 480.000 |
| 1024 x 768 | 786.432 |
| 1280 x 1024 | 1.310.720 |
| 1680 x 1280 | 2.150.400 |

(*) Her piksel için farklı renk kullanıldığı varsayılmıştır)

Şekil 12.1. Çeşitli çözünürlükler ve kullanılabilecek renk sayıları

Şekil.12.1'de de görüleceği gibi, şu anda yaygın olarak kullanılan ekran çözünürlüklerinde, her pikselin ayrı bir renk olması durumunda bile gerekli olan 2.150.400 sayısı 16.777.216 ($2^{24}$) sayısının oldukça altındadır. Bu yüzden bu çalışmada görüntü formatı olarak Targa seçilmiştir.

Bu formata göre her pikselin renk değeri 24-bit ile ifade edilir. Bu 24-bit içerisinde 8-bit kırmızı, 8-bit yeşil, 8-bit ise mavi bileşenler için kullanılmıştır.

Targa formatında saklanan bir görüntünün birden fazla sayıda olasılıkla ifade edilme imkanı vardir. Saklama sırasında hangi yöntemin kullanıldığı, Targa dosyasının başında bulunan bir etiket içinde belirtilmiştir. Bu çalışma sonucu oluşturulan Targa dosyasının başına standart bir etiket konulmuştur. Bu etikette bulunan bilgiler Tablo.12.1 üzerinde gösterilmiştir. Bu tabloda kullanılan bütün değerler onaltılı (hexadecimal) düzende yazılmıştır. Ayrıca "&" işlemi onaltılı düzendeki "VE", "|" işlemi yine onaltılı düzendeki "VEYA" işlemine karşılık gelmektedir. Sistemin çalışması sonucunda elde edilen görüntünün x ekseni yönündeki boyutu (genişliği-width) w, y ekseni yönündeki boyutu (boyu-height) ise h ile gösterilmiştir.

Tabloda ayrıca kullanılmayan bazı alanlar vardır. Bu alanlar tanımlama bilgisi, renk paleti uzunluğu ve bilgisi, sıkıştırma bilgisi, gibi bu çalışmanın ilgilenmediği konularla bağlantılıdır. Sistemin çıkış olarak verdiği resimde gerçek renk değerleri (true color) kullanıldığı için ekstra bir renk paleti bilgisine ihtiyaç duyulmamaktadır.

Ancak Targa formatı $2^{24}$ rengi desteklemesine rağmen, günümüzde kullanılan kişisel bilgisayarların büyük bir kısmı, bu formatta kodlanmış bir resmi bütün renkleriyle ekrana getirememektedir. Böyle bir durumda sistemin çıkışı olarak elde edilen Targa dosyası herhangi bir görüntülenebilir formata (örneğin GIF) çevrilerek ekranda görülebilir. Doğal olarak bu şekilde elde edilen dönüştürülmüş resimler, orjinal resmin içerdiği bütün renkleri kapsamadığından görüntüde bir miktar bozukluklar olabilir. Bu bozulma kullanılan dönüştürücünün kalitesine ve yöntemine göre değişebilir.

Tablo 12.1. Bu çalışmada kullanılan Targa etiketi

| Bayt Adresi (hex) | Değer (hex) | | Bilgi |
|---|---|---|---|
| 01 00 | 00 | 00 | Dosya içinde bulunan tanımlama bilgisi uzunluğu (Bu uzunluğun 0 olması, tanımlama bilgisi bulunmadığını gösterir). |
| 03 02 | 00 | 02 | İkinci tip (sıkıştırılmamış) Targa tipi. |
| 05 04 | 00 | 00 | Kullanılmıyor. |
| 07 06 | 00 | 00 | Kullanılmıyor. |
| 09 08 | 00 | 00 | Kullanılmıyor. |
| 0B 0A | 00 | 00 | Kullanılmıyor. |
| 0D 0C | w & FF00 | w \| 00FF | Görüntünün genişliği (w) ile ilgili bilgiler. |
| 0F 0E | h & FF00 | h \| FF00 | Görüntünün boyu (h) ile ilgili bilgiler. |
| 11 10 | 20 | 18 | Her piksel için 3 bayt (24-bit) kullanılmıştır ve resim yukarıdan aşağıya doğru kodlanmıştır. |
| 14 13 12 | Mavi1 Yeşil1 Kırmızı1 | | 1. pikselin renk bileşenleri. |
| 17 16 15 | Mavi2 Yeşil2 Kırmızı2 | | 2. pikselin renk bileşenleri |
| 1A 19 18 | Mavi3 Yeşil3 Kırmızı3 | | 3. pikselin renk bileşenleri |
| ... ... ... | ... | | Diğer piksellerin renk bileşenleri |

# 13. İŞLENECEK SAHNELERİN TANIMLANMASI

Bu sistem yardımıyla görüntü oluşturabilmek, ancak aşağıda anlatılan dil kullanılarak yaratılan sahne tanıtım dosyaları aracılığıyla olabilir. Bu tip dosyalar herhangi bir ASCII kelime işlemci tarafından oluşturulabilir. Sistem, bu yolla girilen verileri işleyerek kendi içinde bir model oluşturur. Daha sonra bu modeli kullanarak kullanıcı tarafından tasarlanan görüntüyü oluşturur. Aşağıda bu dosyalarda kullanılabilecek komutların alfabetik sırayla tanımı yapılmıştır. Ancak daha önce genel bazı tanımlar gösterilecektir.

## 13.1. Sahne tanıtım dosyalarıyla ilgili genel bilgi

### 13.1.1. Açıklama tanımı

Sahne tanıtım dosyalarını başka kullanıcılar tarafından daha anlaşılır hale getirebilmek için açıklama satırları kullanılabilir. Birbirini takip eden iki bölme işaretinden (//), satırın sonuna kadar olan bütün bölüm açıklama satırı olarak algılanır ve sistem tarafından dikkate alınmaz. Örneğin,

*// iki bölme işaretinde sonra gelen bu kısım bir açıklama satırıdır*

### 13.1.2. Sayı Kullanımı

Kullanılacak sayılar tamsayı veya ondalıklı sayı olabilir. 1.0, 123.456, .5, 100, -123, +321, -123.4e-2, 456.3e2, 567e+2 sayıları bu tip dosyalarda kullanılabilecek sayılara örnek olarak gösterilebilirler.

### 13.1.3. Karakter kullanımı

Tanımlı komutlar kullanılırken küçük veya büyük harf ayrımı yoktur. Yani "Object", "object", "OBJECT", "ObJeCt" komutlarının hepsi aynı görevi görecektir.

## 13.2. Satır kavramı

Sahne tanım dosyalarında satır kavramı yoktur. Yani, bir komut bir satırda başlayıp diğer bir satırda bitebilir. Ancak burada dikkat edilecek nokta, komut kelimelerinin bölünmemesidir. Örneğin, "Object" kelimesinin "Obj" kısmı bir satırda, "ect" kısmı da takip eden satırda yazılamaz.

*object { sphere { <0 0 0> 1 } color <1 0 0> }*

tanımı ile

*object {*

*sphere { <0 0 0> 1 }*

*color <1 0 0> }*

tanımı tamamen aynıdır.

## 13.3. Komut tanımı

### 13.3.1. Ambient

**Tanım**

Bu komut, cisim üzerinde ışık kaynağından direkt olarak etkilenmeyen bölgelerde, çevre etkisiyle oluşan ışığın miktarını belirlemek için kullanılır. Bu yolla oluşturulan ışık değeri gölgede kalan alanların aydınlatılmasında kullanılır.

**Yazım**

*object { <nesne_tipi> { <nesne_tanımı> } ambient <aydınlanma_miktarı> }*

**Açıklama**

Bu komut bir yüzeyin gölgede kalan bölümlerindeki aydınlanma miktarını belirtmek için kullanılır. Bu tip aydınlanma, genellikle çevrede bulunan diğer yüzeyler üzerinden her yönden gelen ışık ışınları sayesinde olur. Bu, ışık kaynağından gelen ışınlarla direkt olarak aydınlanmadan oldukça farklı bir aydınlanma şeklidir. Gerçek dünyada, bu ışık, diğer yüzeyler üzerinde ışık kaynağı tarafından direkt olarak oluşturulan aydınlanmaya ait ışık ışınlarının, bu yüzeylerden yansıması sonucunda oluşmaktadır. Sistemin, her yüzeye ait her noktadan gelen yansıma değerini bulması imkansızdır. Dolayısıyla bu tip aydınlanmayı simüle etmek için bu komut kullanılmaktadır. Komuta parametre olarak aydınlanma değeri girilmelidir. Aydınlanma değeri 0.0 (hiç aydınlanma değeri yok, çok koyu gölgeler) ile 1.0 (ışık kaynağını görecek kadar aydınlanmış) arasında herhangi bir değer olabilir. Eğer bu komut kullanılmazsa, sistem otomatik olarak 0.05 değerini kabul edecektir. Bu değer, gölgeler üzerinde çok hafif bir aydınlanma sağlayacaktır. Gerçekçi bir resim elde edebilmek için, ambient, diffuse ve reflection terimlerine ait değerlerin toplamı 1.0'ı geçmemelidir.

Bu komuta aydınlanma değeri verilirken dikkat edilecek bir nokta da, komut daha çok gölgeli alanlardaki aydınlanmayı etkilemesine rağmen, esasında tanımlı olduğu nesneye ait bütün bölgelerde etkili olacaktır. Dolayısıyla abartılı değerler resmin gerçekçiliğini etkileyecektir.

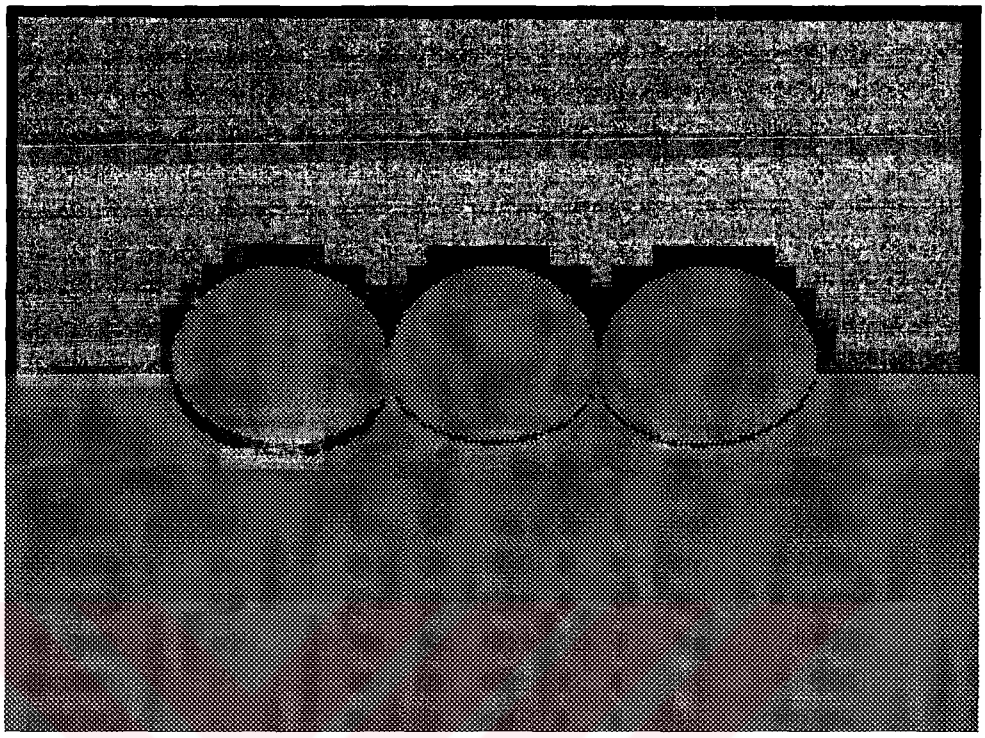
**İlgili diğer komutlar**

diffuse, reflection

**Örnek**

```
// Ambient.TRC

// İlk küre, sistem tarafından tanımlı değerleri kullanmaktadır

object { sphere { <-2 0 0> 1 }

        color < 1.0 0.0 0.0> diffuse 0.5 }

// İkinci kürenin ambient değeri biraz daha fazla olarak tanımlandı

object { sphere { < 0 0 0 > 1 }

        color < 1.0 0.0 0.0 > ambient 0.5 diffuse 0.5 }

// Daha yüksek ambient değerli ve düşük diffuse değerli bir nesne

object { sphere { < 2 0 0 > 1 }

        color < 1.0 0.0 0.0 > ambient 0.5 diffuse 0.2 }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı
```

*screen { width 320 height 240 }*



Şekil 13.1. Ambient komutuna örnek

### 13.3.2. Box

**Tanım**

Bu komut, sisteme, işlenecek cismin şeklinin dikdörtgenler prizması (kutu) olduğunu anlatmak için kullanılır.

**Yazım**

*object { box { <köşe1> <köşe2> } ... }*

**Açıklama**

Bu komut sisteme bir dikdörtgenler prizması tanımlar. Bu cisimde, karşılıklı yüzler birbirlerine paralel olup, komşu yüzler birbirlerine dik olacak şekilde tanımlanmıştır. Kutu tanımlandığında kenar yüzeyler koordinat sitemi eksenine paralel olacak şekilde belirlenir. Ancak daha sonra sistemde tanımlı diğer bazı komutlar yardımıyla gerekli dönüşümler yapılarak kutu istenilen açıya oturtulabilir.

<köşe1> ve <köşe2> vektörleri kutuyu tanımlayan en uç köşe noktalarının üç boyutlu koordinatlarını belirtir. Dolayısıyla <köşe1> noktasının X, Y, Z değerleri <köşe2> noktasının X, Y, Z değerlerinden daha küçük olmak zorundadır. Yani

*box { <1 0 0> <0 1 1> }*

şeklindeki bir tanımlama doğru sonuçlar oluşturmayabilir. Aynı şeklin doğru olarak tanımı

*box { <0 0 0> <1 1 1> }*

olarak yapılabilir.

**İlgili diğer komutlar**

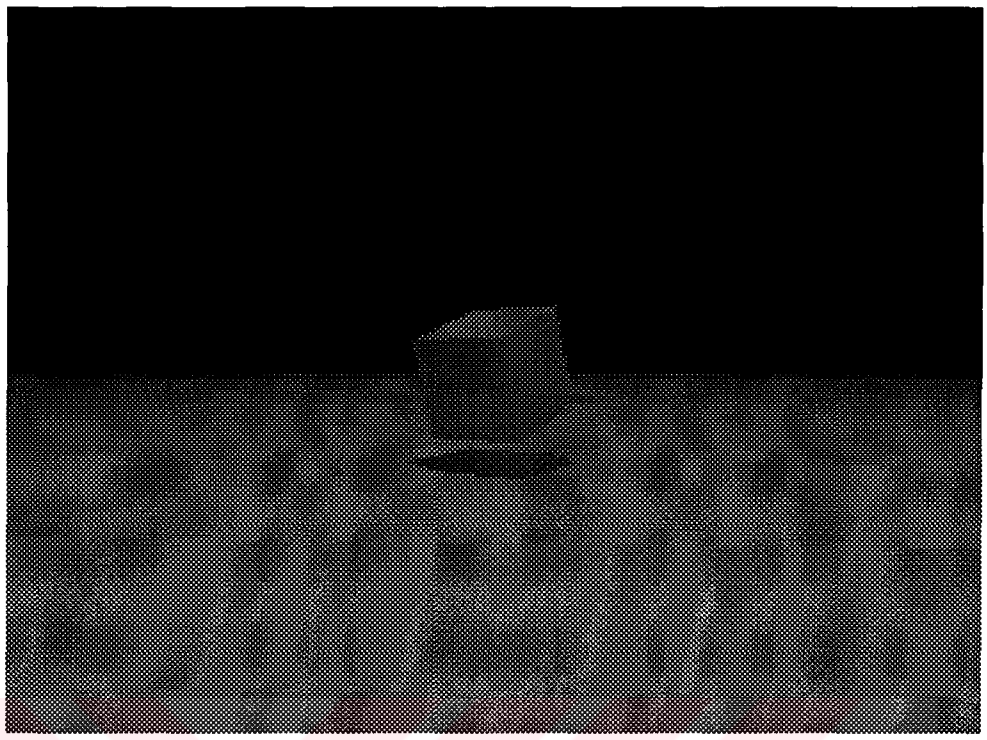
object

**Örnek**

*// Box.TRC*

*object { box { <-0.5 -0.5 -0.5> <0.5 0.5 0.5> }*

```
color < 1.0 0.0 0.0> rotate <-20 35 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.2. Box komutuna örnek

### 13.3.3. Camera

**Tanım**

Bu komut, sahneye bakılan noktanın (kameranın) yerini ve özelliklerini tanımlar.

**Yazım**

*camera { location <kamernın_yeri> lookAt <bakılan_nokta> }*

**Açıklama**

Bu projede geliştirilen sistemin esasında kendisi bir kamera görevi görmektedir. Sistem kullanıcı tarafından tanımlanan sahneye bakan bir kamera olarak gördüğü görüntüyü diske aktarır. Bu komut, kullanıcının bu kamera üzerinde istediği değişikliği uygulamasına imkan tanır.

Kamera, tanımlanan sahneyi tarayan ışınların başladığı görünmez bir nokta olarak da ele alınabilir. Komut içinde kullanılan "location" terimi kameranın bulunduğu yeri belirtmek için kullanılır. Sistemden görüntü elde etmek için bu değerin mutlaka belirtilmesi gereklidir. Aksi taktirde program uygun hata mesajı ile sonlanır.

Kameradan çıkan sanal tarama ışınları, "screen" terimi ile tanımlanmış sanal bir sahneden geçerek nesne uzayına yönlenirler. Sahne tanımı ile daha ayrıntılı bilgi "screen" komutunun anlatıldığı bölümde bulunabilir.

Kamerayı daha iyi tanımlayabilmek için istenilirse lookAt terimi de kullanılabilir. Bu terim, kameranın baktığı noktanın üç boyutlu koordinatlarını belirtmek amacıyla kullanılır. Sistem tanımlanan bu bakış noktasını, "screen" terimi ile belirtilen sahne tanımıyla birleştirerek sonuç görüntüyü elde etmeye çalışır.

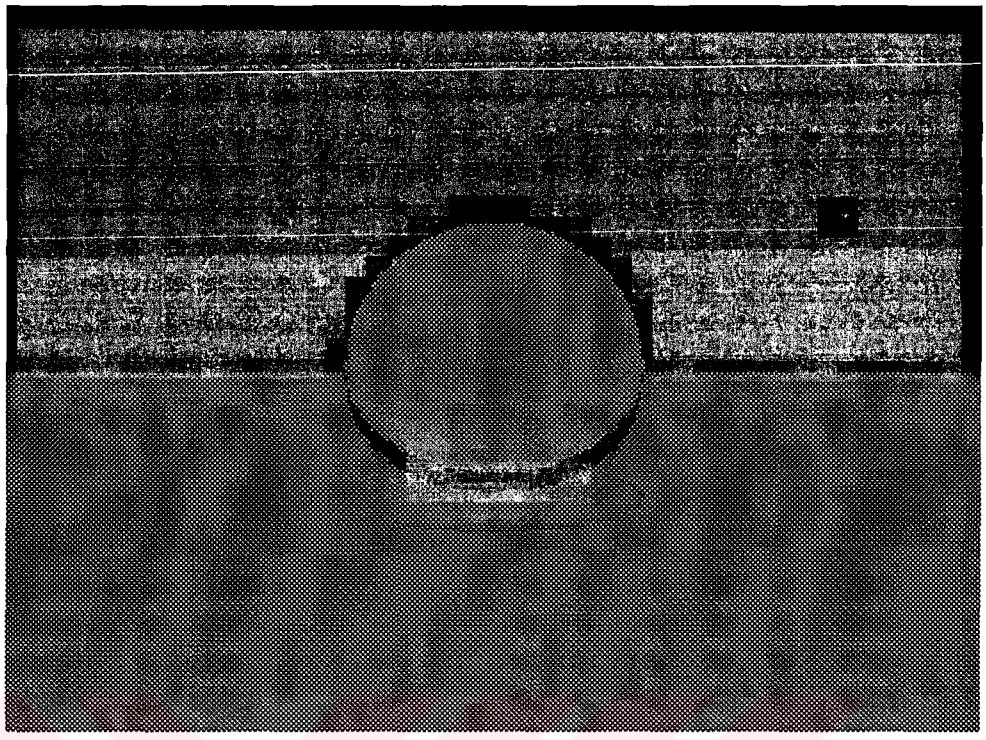
Sisteme veri olarak girilen bütün sahne tanım dosyaları bir adet kamera tanımı içermelidir. Aksi taktirde sistemin çalışması mümkün değildir. Aynı şekilde, birden fazla kamera tanımı da sistem tarafından hata olarak değerlendirilir.

**İlgili diğer komutlar**

screen, location, lookAt, right, up

**Örnek**

```
// Camera.TRC

// Bakılacak bir cisim tanımlayalım

object { sphere { <0 0 0> 1 } color <1 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 5> lookAt <0 0 0> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.3. Camera komutuna örnek

### 13.3.4. Checker

**Tanım**

Bu komut, tanımlandığı yüzey üzerinde iki farklı renkten oluşan kareli bir renk yapısı oluşturur.

**Yazım**

*checker { color <renk1> color <renk2> [scale büyütme_faktörü] }*

**Açıklama**

Bu komut, belirtildiği yüzey üzerinde, üç boyutlu iki farklı renkli küplerden oluşmuş bir renk düzeni uygular. Küpler eksenlere paralel olacak şekilde birim uzunlukta olarak tanımlanırlar. Ancak "scale" terimi ile belirtilen faktör ile küplerin boyutu değiştirilebilir. Böylece kareli renk düzeninin istenilen büyüklükte olması sağlanır.

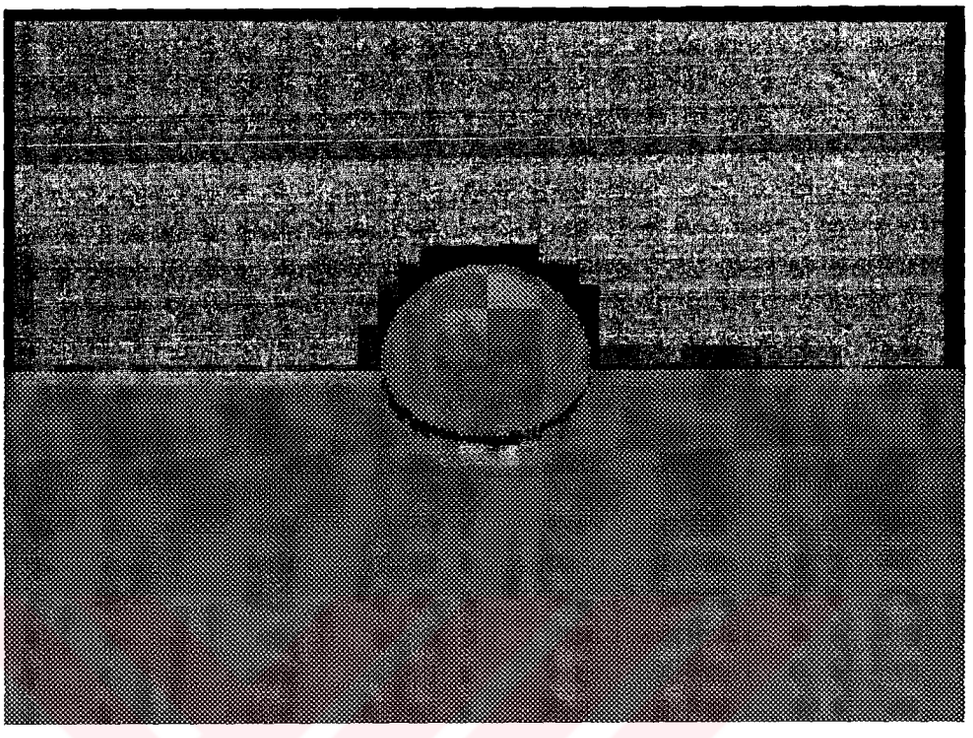
**İlgili diğer komutlar**

color, texture

**Örnek**

```
// Checker.TRC

object { sphere { <0 0 0> 1 }

        checker { color < 1.0 0.0 0.0> color <0.0 1.0 0.0> scale 0.5 } }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı
```

*screen { width 320 height 240 }*



Şekil 13.4. Checker komutuna örnek

### 13.3.5. Color

**Tanım**

Bu komut, tanımlandığı yüzeyin rengini kırmızı, yeşil ve mavi bileşenleri cinsinden ifade eder.

**Yazım**

*color < kırmızı_bileşen yeşil_bileşen mavi_bileşen >*

**Açıklama**

Sistem içerisinde yüzeylere ait bütün renk tanımları, bu komut kullanılarak gerçekleştirilir. Tanımlanan bütün renk değerleri üç bileşenden oluşmaktadır. Bileşenlere ait renk değerleri 0.0 ile 1.0 arasında herhangi bir sayı olabilir. Bu değerin 0.0 olması, ilgili bileşenin yüzeyin renginde hiçbir katkısının olmadığını gösterir. Bu komut diğer bazı komutlarla (checker, texture, vs.) ortak olarak da kullanılabilir. Bu gibi komutlarla ortak kullanım konusuda daha detaylı bilgi için, ilgili komutun tanımına bakılmalıdır.

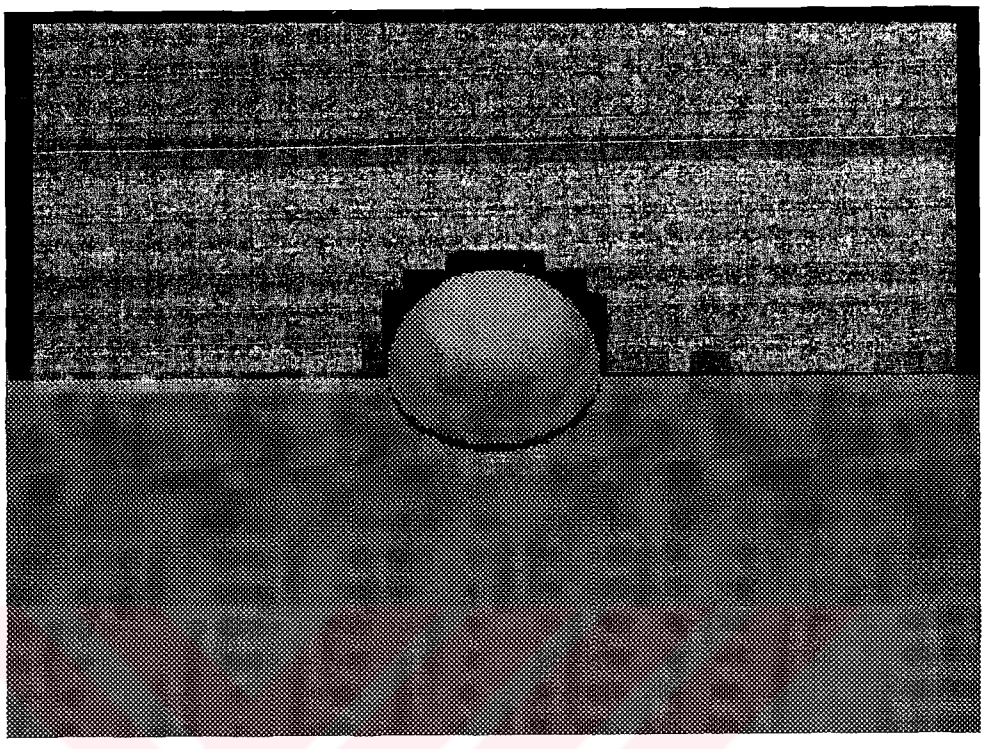
**İlgili diğer komutlar**

checker, colorMap, object, texture

**Örnek**

```
// Color.TRC

object { sphere { <0 0 0> 1 } color < 1.0 0.8 0.1> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı
```

*screen { width 320 height 240 }*



Şekil 13.5. Color komutuna örnek

### 13.3.6. ColorMap

**Tanım**

Bu komut tanımlandığı yüzeyin rengini değişik aralıklarla belirleme imkanı sağlar.

**Yazım**

*colormap { { alt_değer1 üst_değer1 color <renk1_1> color <renk1_2> }*

*{ alt_değer2 üst_değer2 color <renk2_1> color <renk2_2> } ...}*

**Açıklama**

Bu komut yüzeyler üzerinde değişik aralıklarla renk tanımı yapılabilmesini olası kılar. Color komutu, bir yüzeyin rengini tanımlarken özel bazı renk efektlerine imkan tanımaz. Örneğin bir nesnenin rengi color komutu yardımıyla yeşil olarak tanımlanırsa, yüzey, ışığın az geldiği yerlerde koyu yeşil, fazla geldiği yerlerde de parlak yeşil olacak şekilde görünür. Ancak, ışığın geldiği açıya göre renk değişimi gösteren bir nesne tanımlamak gerekirse, color komutunun yetersiz kaldığı görülecektir. Örneğin ışığı dik olarak gören bölümleri kırmızı, dar açıyla gören bölümleri mavi ve diğer bölümleri yeşil olacak bir sanal yüzey tanımlamak ancak colorMap komutu yardımıyla gerçekleştirilebilir.

Komuta parametre olarak girilen alt/üst değerler ve bunlara karşılık gelen renk değerleri kullanılarak, ilgili yüzeyin ışık değeri, yüzeye vuran ışık ışınlarının açısına göre belirlenir.

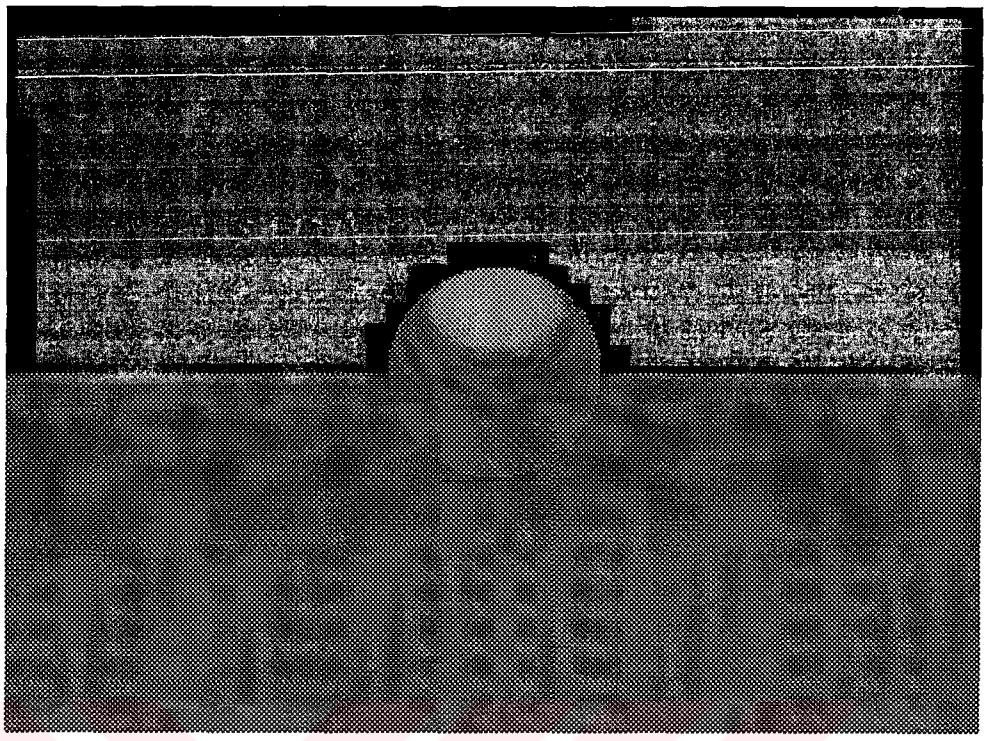
**İlgili diğer komutlar**

color, granite, marble, texture, wood

**Örnek**

```
// ColorMap.TRC

object { sphere { <0 0 0> 1 }

        colorMap { { 0.0 0.2 color <1.0 0.0 0.0> color <0.7 0.0 0.7> }

                   { 0.2 0.4 color <0.7 0.0 0.7> color <0.0 0.0 1.0> }

                   { 0.4 0.6 color <0.0 0.0 1.0> color <0.0 0.7 0.7> }
```

```
                    { 0.6 0.8 color <0.0 0.7 0.7> color <0.0 1.0 0.0> }

                    { 0.8 1.0 color <0.0 1.0 0.0> color <1.0 0.0 0.0> } } }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.6. ColorMap komutuna örnek

### 13.3.7. Diffuse

**Tanım**

Bu komut, cisim üzerinde ışık kaynağından direkt olarak etkilenen bölgelerde oluşan aydınlanma miktarını belirlemek için kullanılır.

**Yazım**

*object { <nesne_tipi> { <nesne_tanımı> } diffuse <aydınlanma_miktarı> }*

**Açıklama**

Bu komut, bir yüzeyin ışık kaynakları tarafından direkt olarak görülen bölgelerindeki aydınlanma miktarını belirlemek için kullanılır. Bu tip aydınlanma, çevrede bulunan diğer yüzeyler üzerinden her yönden gelen ışık ışınları sayesinde olan aydınlanmadan oldukça farklı bir aydınlanma şeklidir. Yüzeyde oluşan bu tip aydınlanma, ilgili yüzey noktası ve ışık kaynağının geometrik pozisyonuna bağlıdır. Işık ışını ile çarpma noktasındaki yüzey normali arasındaki açı ne kadar az olursa (ışık ışını yüzeye ne kadar dik gelirse), yüzeyde oluşan aydınlanma o kadar fazla olur. Eğer bir yüzey ışık kaynağını herhangi bir nedenle göremez ve gölgede kalırsa, ilgili yüzeydeki yayınım (diffuse) aydınlanma değeri sıfır olur. Komuta parametre olarak yayınım aydınlanma değeri girilmelidir. Yayınım aydınlanma değeri 0.0 (hiç direkt aydınlanma yok, sadece gölgeler) ile 1.0 (tamamen aydınlanmış, hiç gölge yok) arasında herhangi bir değer olabilir. Eğer bu komut kullanılmazsa, sistem otomatik olarak 0.6 değerini kabul edecektir. Bu değer, yüzey üzerinde gerçeğe yakın bir direkt aydınlanma sağlayacaktır. Gerçekçi bir resim elde edebilmek için, ambient, diffuse ve reflection terimlerine ait değerlerin toplamı 1.0'ı geçmemelidir.

**İlgili diğer komutlar**

ambient, lightSource, reflection

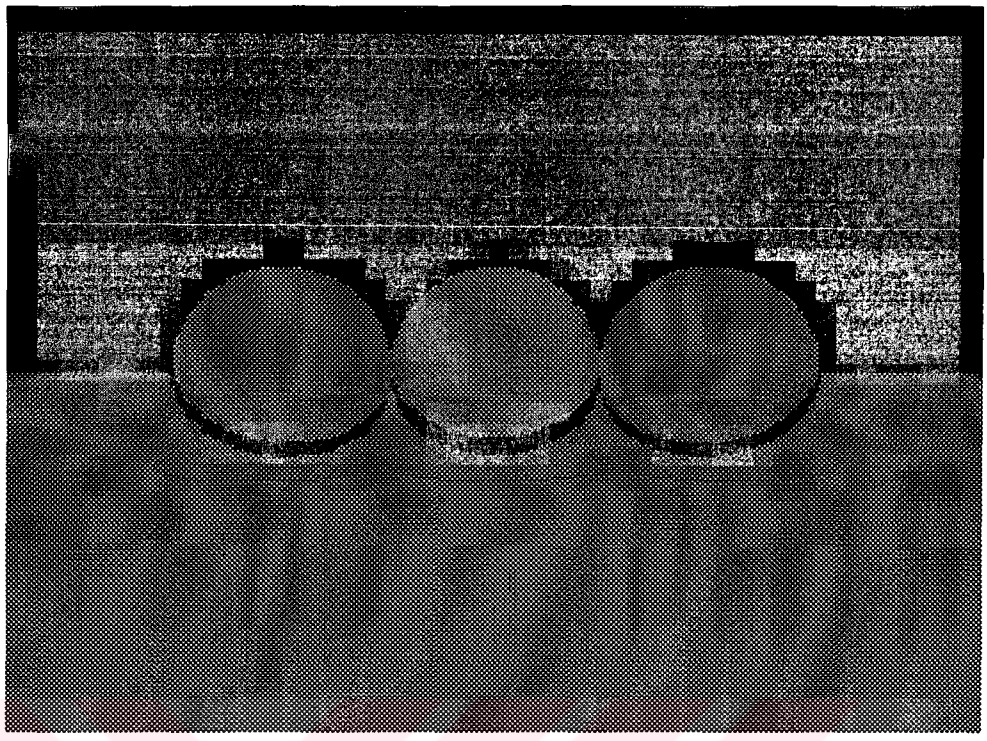
**Örnek**

*// Diffuse.TRC*

*// İlk küre, sistem tarafından tanımlı değerleri kullanmaktadır*

*object { sphere { <-2 0 0> 1 }*

```
        color < 1.0 0.0 0.0> }

// İkinci kürenin diffuse değeri düşük olarak seçildi

object { sphere { < 0 0 0 > 1 }

        color < 1.0 0.0 0.0 > diffuse 0.2 }

// Yüksek diffuse değerli bir nesne

object { sphere { < 2 0 0 > 1 }

        color < 1.0 0.0 0.0 > diffuse 0.8 }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.7. Diffuse komutuna örnek

### 13.3.8. Granite

**Tanım**

Bu komut, cisim üzerinde granite benzeyen bir renk dağılımı oluşturur.

**Yazım**

*texture { granite color <taban_rengi> color <damar_rengi> }*

**Açıklama**

Bu komut, tanımlandığı yüzey üzerinde gerçek bir granit görüntüsünü simüle

etmek amacıyla kullanılır. Sistem bu efekti verebilmek için özel olarak tanımlanmış bir gürültü fonksiyonu kullanır. Komuta parametre olarak iki renk değeri girilmelidir. İlk değer taban rengini oluştururken, ikinci değer de "damar" rengini belirtir.

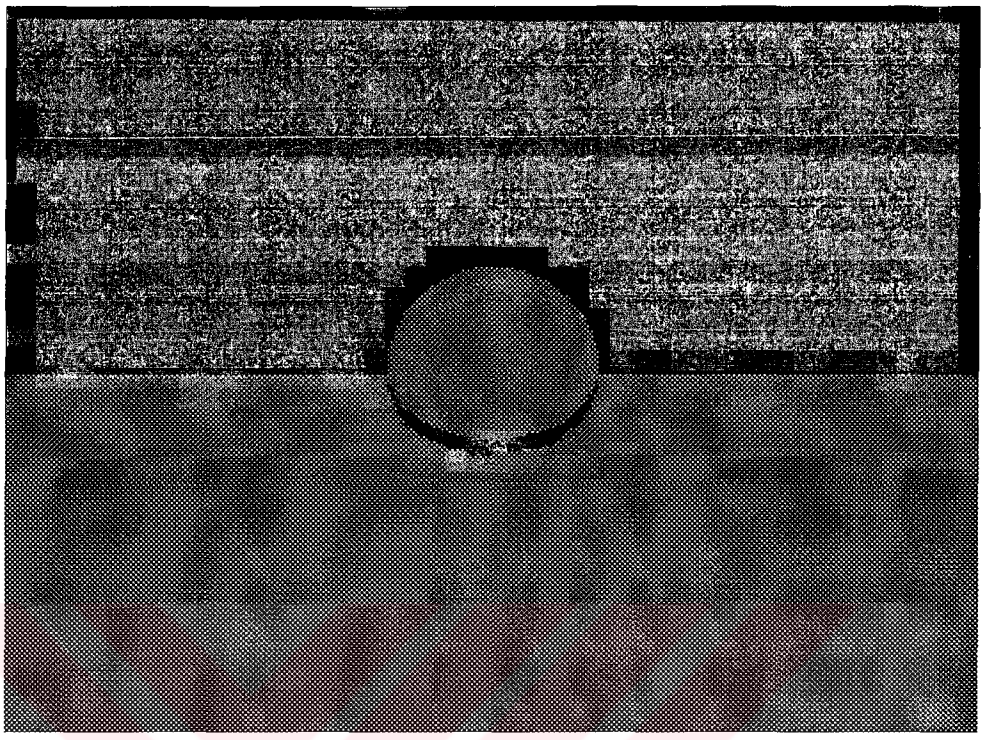
**İlgili diğer komutlar**

colorMap, marble, texture, wood

**Örnek**

```
// Granite.TRC

// Granit bir küre

object { sphere { < 0 0 0 > 1 }

        texture { granite color <0.051 0.224 0.060>

                          color <0.773 0.727 0.050> } }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı
```

*screen { width 320 height 240 }*



Şekil 13.8. Granite komutuna örnek

### 13.3.9. Height

**Tanım**

Bu komut, oluşturulacak görüntü dosyasının yüksekliğini (y boyutu) piksel sayısı cinsinden ifade etmek için kullanılır.

**Yazım**

*screen { width <genişlik_değeri> height <yükseklik_değeri> }*

**Açıklama**

Bu komut, tanımlanan sahnenin görüntüsü oluşturulurken, diske yazılacak Targa dosyasının yükseklik değerini piksel olarak belirlemek amcıyla kullanılır. Komuta parametre olarak yükseklik değeri girilmelidir. Bu değer sıfırdan büyük herhangi bir tamsayı olabilir. Ancak bu değer ile width komutunun değeri arasındaki oran ile kamera tanımında bulunan up ve right komutlarının arasındaki oran aynı olması, görüntünün geçekçiliği açısından önem taşır. Örneğin "up" değeri (0, 1, 0), "right" değeri (1.333, 0, 0) olan bir kamera tanımına uygun olarak tanımlanacak "screen" komutunun parametreleri arasındaki oran 1.333 olmalıdır. Yani,

$$\frac{|right|}{|up|} = \frac{width}{height}$$

oranı korunursa, elde edilen görüntüler daha gerçekçi olur.

**İlgili diğer komutlar**

camera, right, screen, up, width

**Örnek 1**

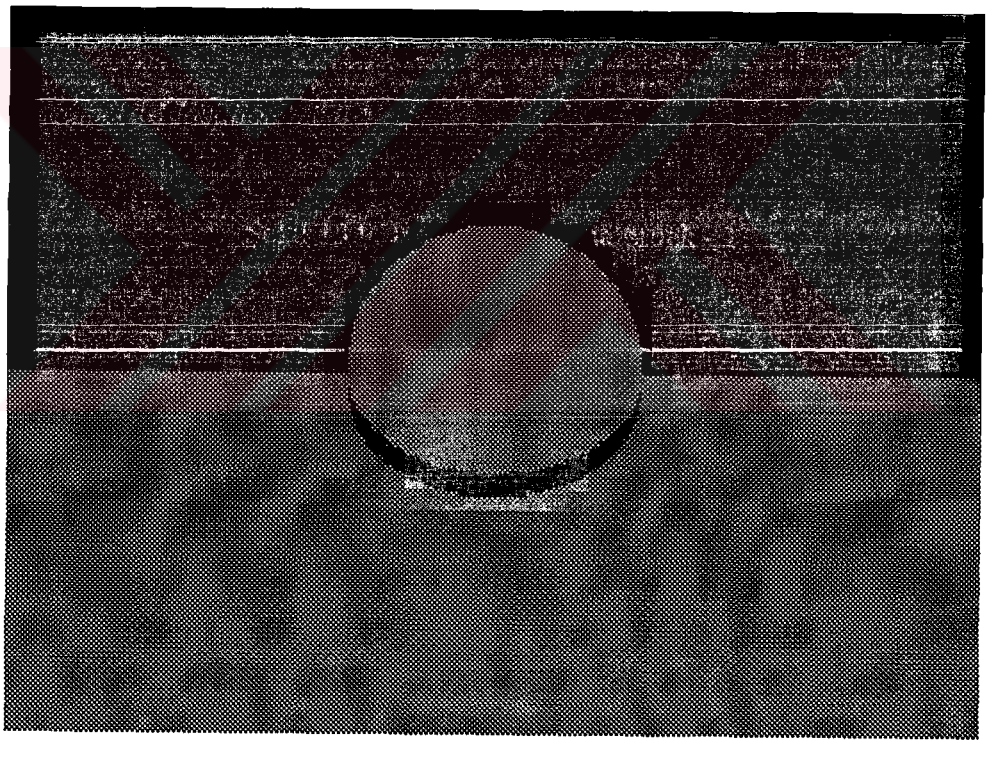
*// Height1.TRC*

*// Bir küre*

*object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }*

*// Gölgelerin görünmesi için taban düzlemi*

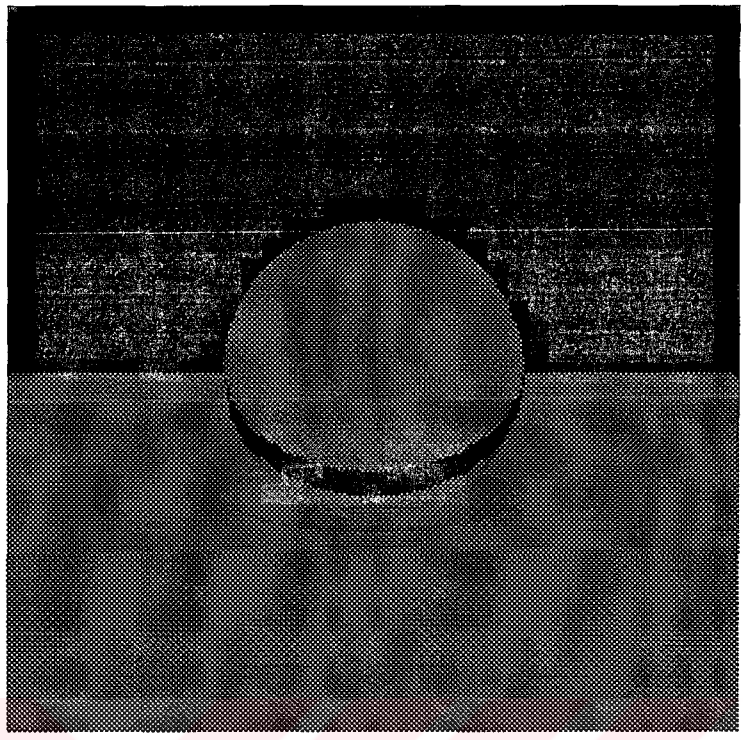
```
object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 5> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 up <0 1 0> right <1.333 0 0> }
```



Şekil 13.9. Height komutuna örnek - 1

**Örnek 2**

```
// Height2.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 5> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 320 up <0 1 0> right <1 0 0> }
```



Şekil 13.10. Height komutuna örnek - 2

### 13.3.10. LightSource

**Tanım**

Bu komut, tanımlanan sahneyi aydınlatmada kullanılan ışık kaynaklarını belirtmek için kullanılır.

**Yazım**

*lightSource { location <ışık_kaynağı_yeri> color <renk_değeri> }*

**Açıklama**

Bu komut, görüntüsü oluşturulmak istenen sahneyi aydınlatan ışık kaynaklarını tanımlamak amacıyla kullanılır. Bütün sahne tanıtım dosyaları en az bir adet ışık kaynağı tanımı içermelidir. Aksi taktirde program uygun hata mesajı vererek sona erer.

Işık kaynağı tanımı içinde, kaynağın yeri ve rengi belirlenebilir. Kaynağın rengi seçimlik bir parametredir. Eğer bu parametre belirtilmezse, sistem otomatik olarak beyaz rengi seçer.

**İlgili diğer komutlar**

color, location

**Örnek**

```
// Light.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85  0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Sarı renkli bir ışık kaynağı
```

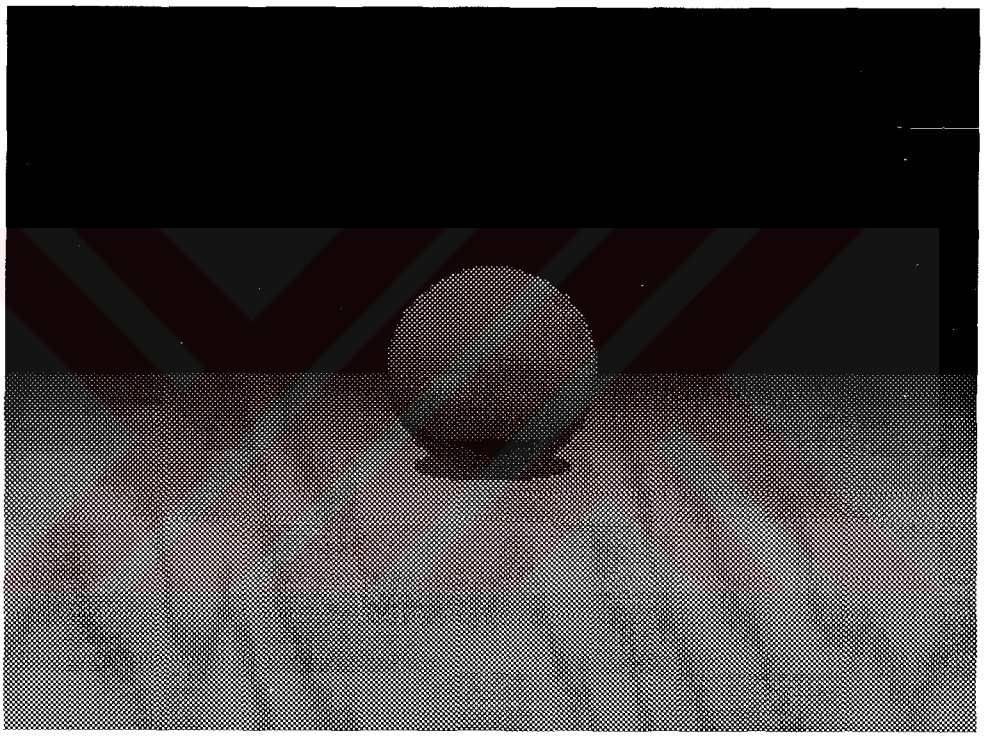
*lightSource { location <-2 5 5> color <1 1 0> }*

*// Renksiz (beyaz) bir ışık kaynağı*

*lightSource { location <2 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.11. LightSource komutuna örnek

### 13.3.11. Location

**Tanım**

Bu komut, tanımlandığı kavramın yerini üç boyutlu koordinatları cinsinden

belirtir.

**Yazım**

*camera { location <kameranın_yeri> ... }*

*lightSource { location <ışık_kaynağının_yeri> ... }*

**Açıklama**

Bu komut, sistemin kullandığı özel bazı kavramların (kamera ve ışık kaynağı) yerini belirtmek için kullanılır. Sistemdeki kamera ve ışık kaynağı nokta olarak kabul edildiği için, bu komuta parametre olarak sadece bu noktanın koordinatlarını girmek yeterlidir.

**İlgili diğer komutlar**

camera, lightSource

**Örnek 1**

*// Locatn1.TRC*

*// Bir küre*

*object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }*

*// Gölgelerin görünmesi için taban düzlemi*

*object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }*

*// Sisteme bakılacak nokta*

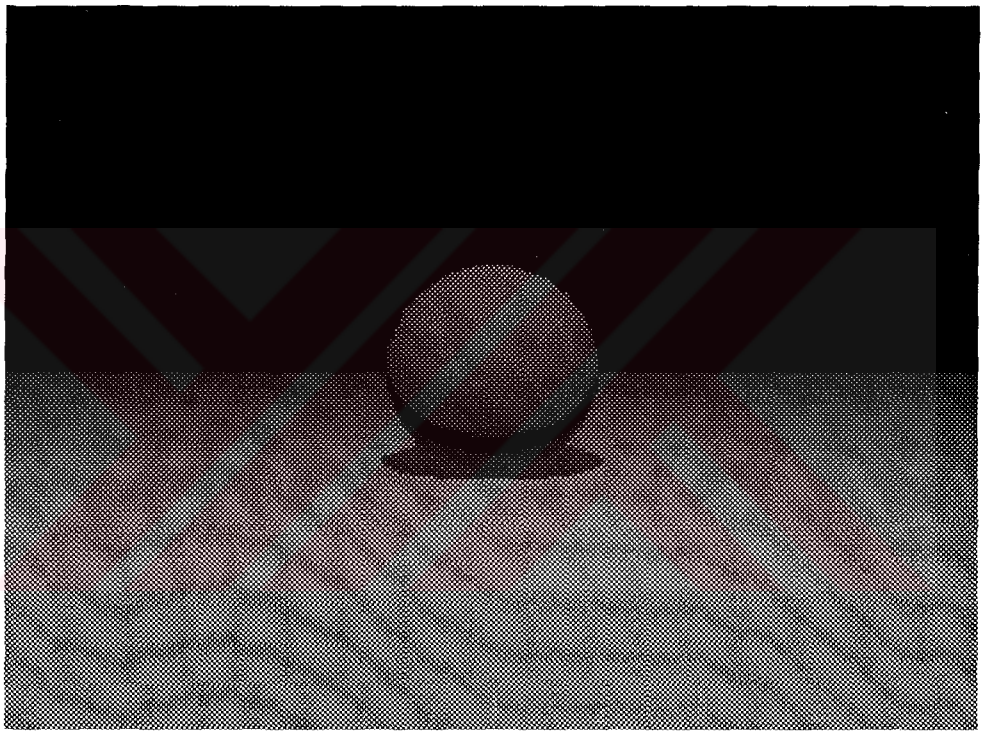
*camera { location <0 0 7> }*

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.12. Location komutuna örnek - 1

**Örnek 2**

*// Locatn2.TRC*

*// Bir küre*

```
object { sphere { < 0 0 0 > 1 } color <0.85  0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta biraz geriye alındı

camera { location <0 0 8> }

// Işık kaynağı tanımı - biraz sağa kaydırıldı

lightSource { location <2 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```

Şekil 13.13. Location komutuna örnek - 2

### 13.3.12. LookAt

**Tanım**

Bu komut, kameranın bakacağı noktayı belirtmek için kullanılır.

**Yazım**

*camera { ... lookAt <bakılacak_nokta> }*

**Açıklama**

Bu komut, tanımlanan sahnenin görüntüsü oluşturulurken, kamera tarafından

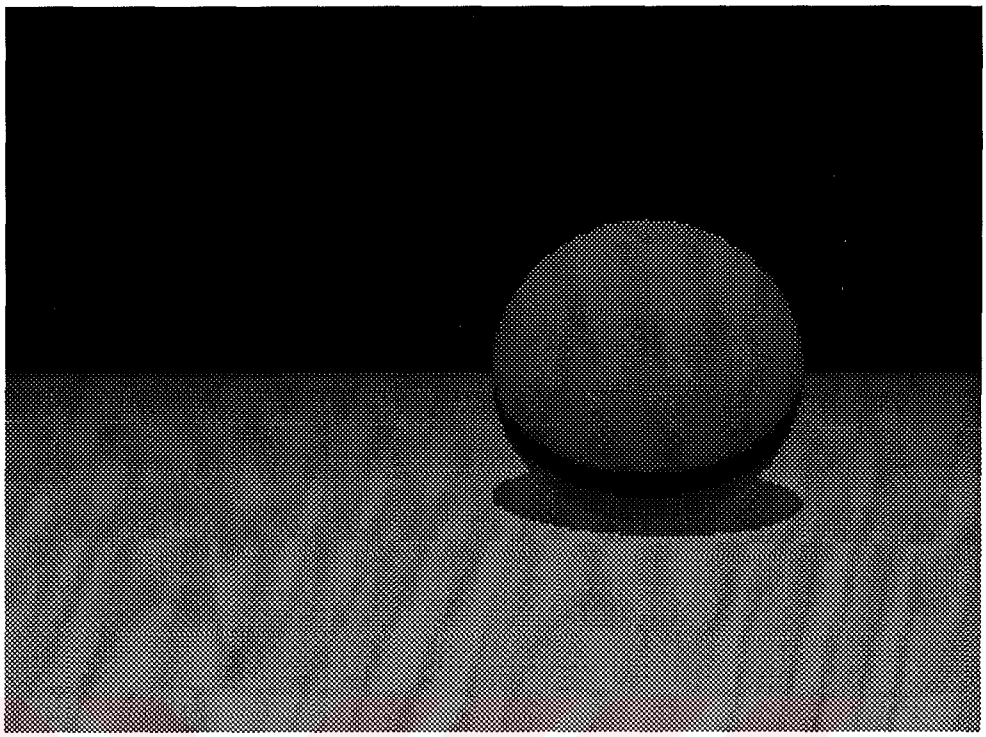
bakılacak noktanın koordinatlarını belirlemek için kullanılır. Kameranın yeri sabit bırakılarak, bu parametrenin değeri ile oynamak suretiyle, tanımlanan sahnenin aynı noktadan görülebilen değişik bölgelerinin görüntüsü elde edilebilir.

**İlgili diğer komutlar**

camera

**Örnek 1**

```
// LookAt1.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85  0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 5> lookAt <-1 0 0> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.14. LookAt komutuna örnek - 1

**Örnek 2**

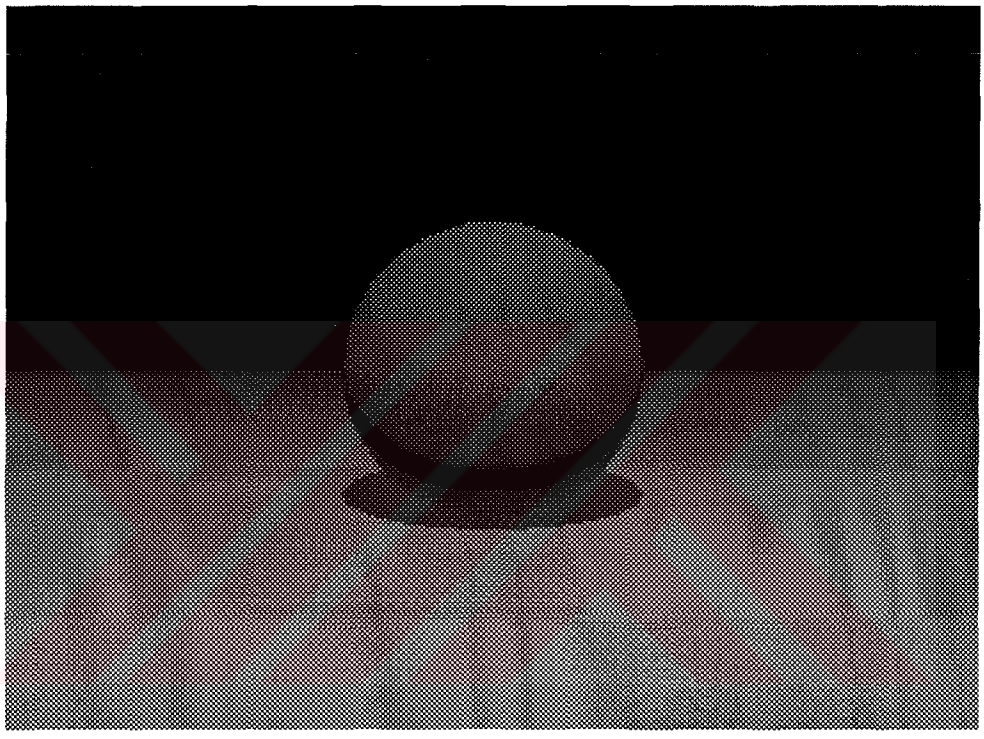
```
// LookAt2.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85  0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 5> lookAt <0 0 0> }
```

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.15. LookAt komutuna örnek - 2

### 13.3.13. Marble

**Tanım**

Bu komut, cisim üzerinde mermere benzeyen bir renk dağılımı oluşturur.

**Yazım**

*texture { marble color <taban_rengi> color <damar_rengi> }*

**Açıklama**

Bu komut, tanımlandığı yüzey üzerinde gerçek bir mermer görüntüsünü simüle etmek amacıyla kullanılır. Sistem bu efekti verebilmek için özel olarak tanımlanmış bir gürültü fonksiyonu kullanır. Komuta parametre olarak iki renk değeri girilmelidir. İlk değer taban rengini oluştururken, ikinci değer de "damar" rengini belirtir.

**İlgili diğer komutlar**

colorMap, granite, texture, wood

**Örnek**

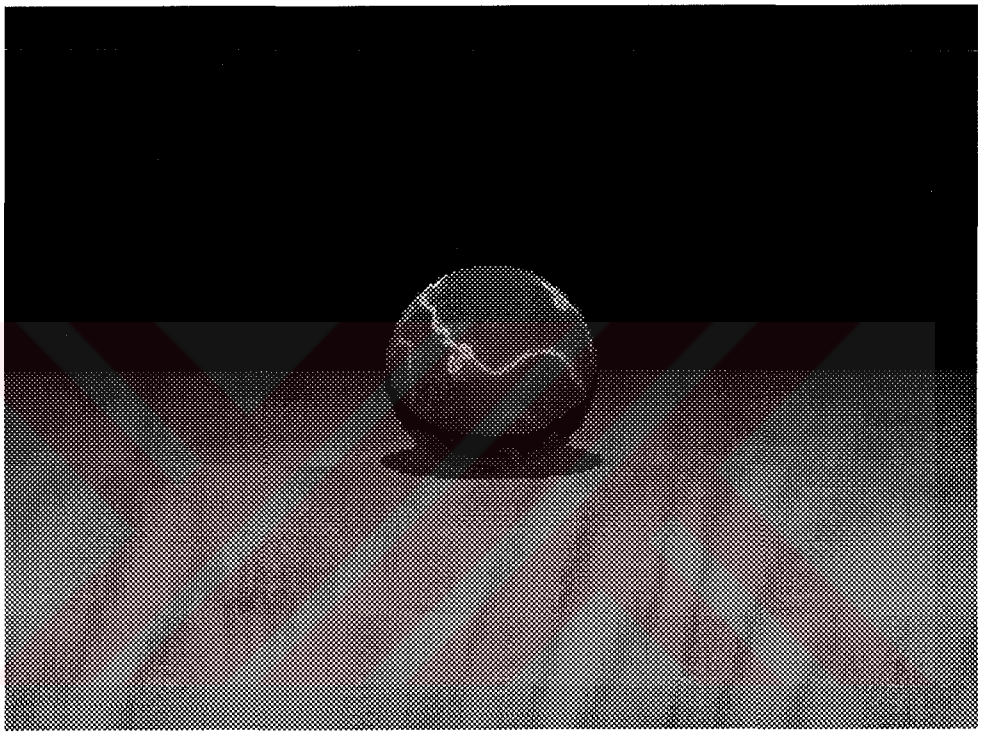
```
// Marble.TRC

// Mermer bir küre

object { sphere { < 0 0 0 > 1 }

        texture { marble color <0.6 0 0>  color <0.85  0.85  0.10> } }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }
```

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.16. Marble komutuna örnek

### 13.3.14. Object

**Tanım**

Bu komut, sisteme işlenecek bir nesne tanımlamak amacıyla kullanılır.

**Yazım**

*object { <nesne_tipi> { <nesne_tanımı> } ... }*

**Açıklama**

Bu komut, sisteme işlemek üzere bir nesne tanımlamak için kullanılır. Komut, nesneyle ilgili bütün özelliklerin bir arada bulunmasını sağlar. Bu özellikler arasında nesnenin tipi, geometrik tanımı, yüzey yapısı, renk bilgisi, geometrik dönüşüm değerleri, aydınlanma efektleri, vs sayılabilir. Her object komutu sadece bir adet nesnenin tanımlanması için kullanılabilir. Her sahne için en az bir adet nesne tanımlanmalıdır. Aksi taktirde sistem uygun hata mesajını vererek sonlanır.

**İlgili diğer komutlar**

box, color, color_map, polygonal, rotate, scale, sphere, texture, translate

**Örnek**

```
// Object.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 }

        color <0.85  0 0>

        scale 0.8

        rotate <-20 -60 0>

        translate <0.5 0.5 0> }
```

*// Gölgelerin görünmesi için taban düzlemi*

*object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }*

*// Sisteme bakılacak nokta*

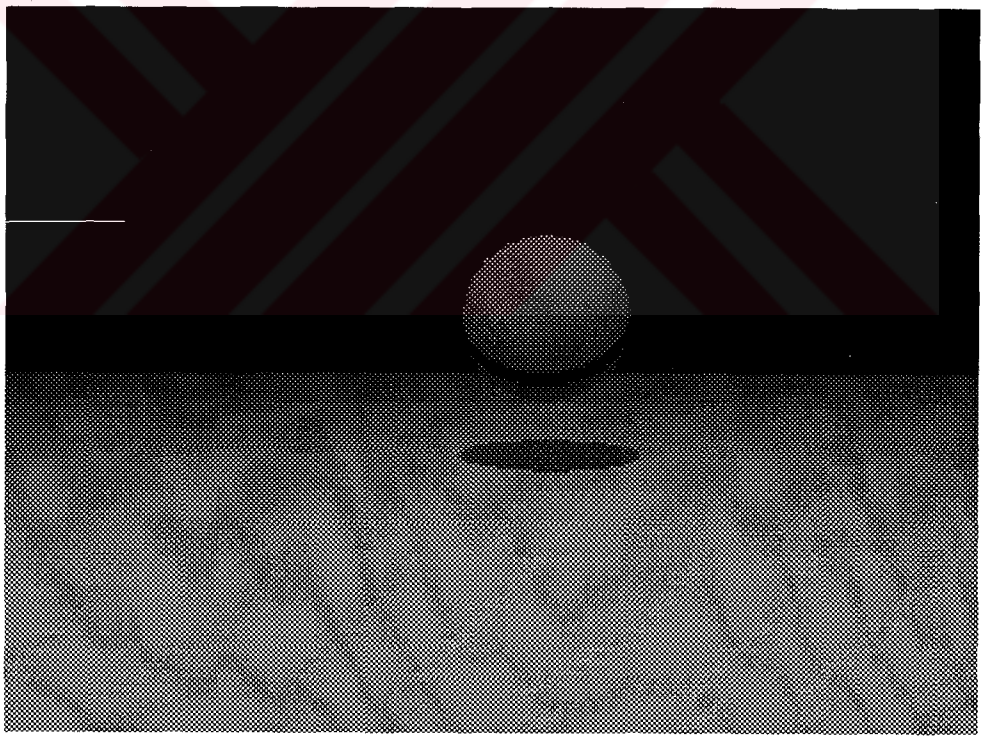
*camera { location <0 0 7> }*

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.17. Object komutuna örnek

### 13.3.15. Phong

**Tanım**

Bu komut, tanımlandığı cisim üzerinde phong aydınlanma modeline göre bir renk dağılımı oluşturur.

**Yazım**

*object { ... phong <phong_şiddeti> <phong_büyüklüğü> }*

**Açıklama**

Bu komut, tanımlandığı yüzey üzerinde phong aydınlanma modeline göre bir renk dağılımı oluşturur. Bu modele göre ışık kaynağı, çarptığı yüzey üzerinde parlak bir nokta oluşturacak şekilde bir aydınlanma oluşturur. Phong komutuna parametre olarak girilen şiddet değeri, oluşan bu parlak noktanın parlaklık seviyesini kontrol eder. Bu değer 0.0 (hiç phong aydınlanması yok) ile 1.0 (çok parlak) arasında herhangi bir sayı olabilir. Burada oluşan parlak alanın rengi ışık kaynağının rengiyle aynı olacaktır. Komuta parametre olarak girilen ikinci değer ise bu parlak bölgenin büyüklüğünü kontrol eder. Bu değer 1.0 (çok büyük) ile 100 (çok küçük) arasında herhangi bir sayı olabilir. Ancak bu parametre seçimlik bir parametredir. Eğer belirtilmezse sistem otomatik olarak 40 değerini seçecektir. Bu modele göre gerçekleştirilen aydınlanma şu formüle göre hesaplanır:

$$i = a \times \cos(t)^{s}$$

Bu formülde kullanılan "i" yüzeydeki aydınlanma miktarını, "a" phong

şiddetini, "s" phong büyüklüğünü, "t" ise yüzeye göre ışık kaynağı doğrultusuyla kamera doğrultusu arasındaki açıyı temsil etmektedir.

**İlgili diğer komutlar**

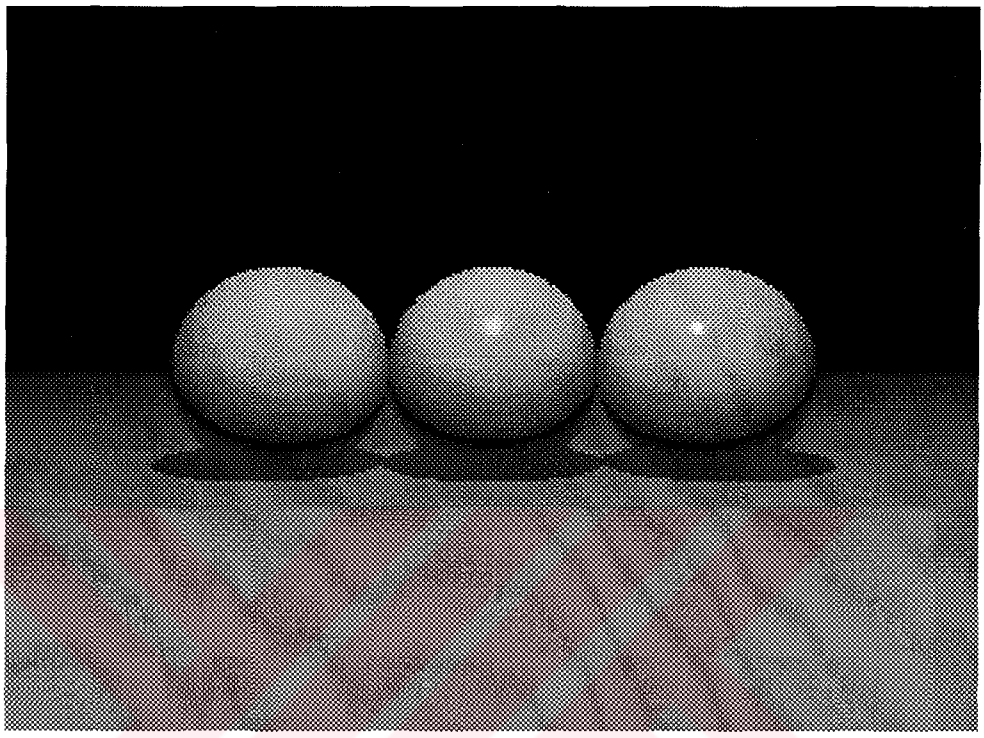
diffuse, lightSource

**Örnek 1**

```
// Phong1.TRC

// Birinci küre

object { sphere { < -2 0 0 > 1 } color <1 1 0> phong 0 }

// İkinci küre

object { sphere { < 0 0 0 > 1 } color <1 1 0> phong 0.5 }

// Üçüncü küre

object { sphere { < 2 0 0 > 1 } color <1 1 0> phong 1.0 }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }
```

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.18. Phong komutuna örnek - 1

**Örnek 2**

*// Phong2.TRC*

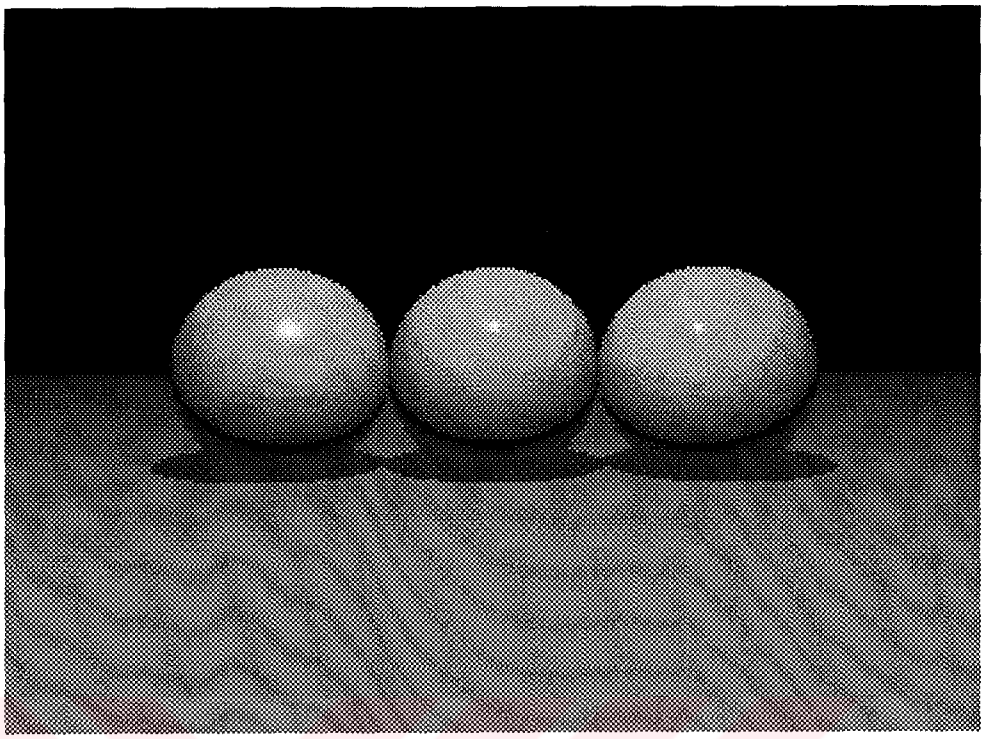
*// Birinci küre*

*object { sphere { < -2 0 0 > 1 } color <1 1 0> phong 1 10 }*

*// İkinci küre*

*object { sphere { < 0 0 0 > 1 } color <1 1 0> phong 1 40 }*

```
// Üçüncü küre

object { sphere { < 2 0 0 > 1 } color <1 1 0> phong 1 80 }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.19. Phong komutuna örnek - 2

### 13.3.16. Plane

**Tanım**

Bu komut, sisteme, işlenecek cismin şeklinin sonsuz bir düzlem olduğunu anlatmak için kullanılır.

**Yazım**

*object { plane { <normal_vektörü> öteleme_faktörü } ... }*

**Açıklama**

Bu komut sisteme bir düzlem tanımlar. Komuta parametre olarak girilen normal vektörü, düzlem yüzeyine ait normal vektörünü belirtir. Bu vektör, düzleme dik olan herhangi bir vektör olabilir. Öteleme değeri ise düzlemin (0,0,0) noktasına göre (normmal doğrultusundaki) uzaklığını belirtmek için kullanılır. Bir düzlem şekli sistem tarafından iki boyutta sonsuz büyüklükte ve üçüncü boyutta da sonsuz ince olacak şekilde algılanır.

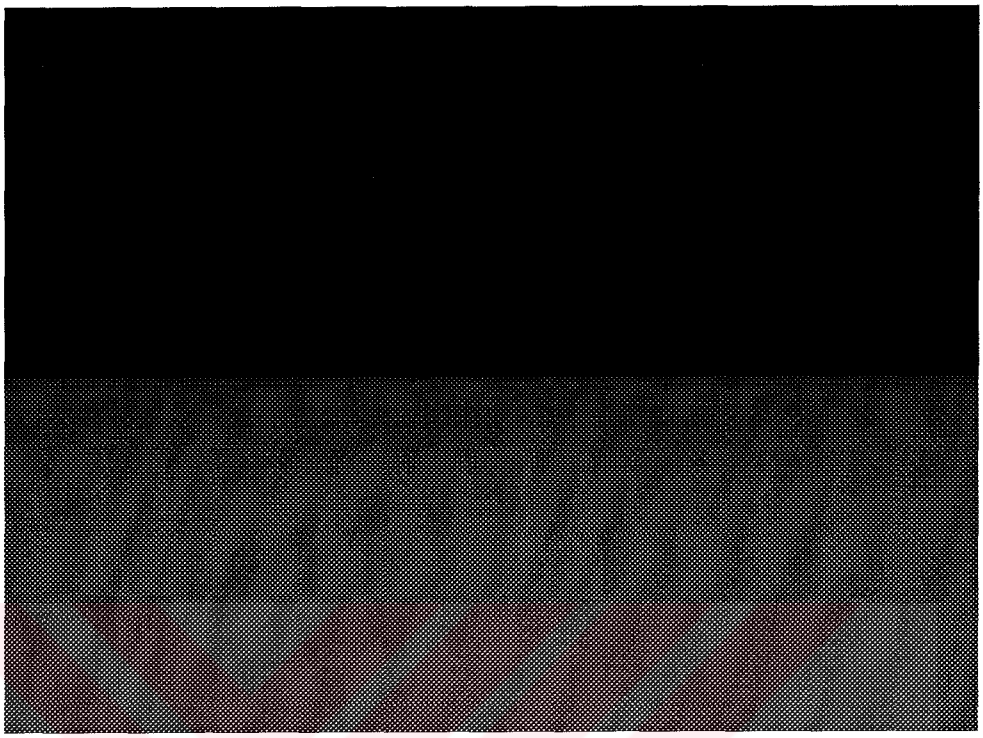
**İlgili diğer komutlar**

object

**Örnek**

```
// Plane.TRC

// Bir düzlem tanımı.

// Normal vektörü +y doğrultusunda

// Düzlem, y doğrultusunda -2 birim ötelenecek şekilde tanımlandı

object { plane { < 0 1 0 > -2 } color <0.85  0 0> ambient 0.2 }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı
```

*screen { width 320 height 240 }*



Şekil 13.20. Plane komutuna örnek

### 13.3.17. Polygonal

**Tanım**

Bu komut, sisteme, işlenecek cismin şeklinin bir çokgenler kümesi olduğunu anlatmak için kullanılır.

**Yazım**

*object { polygonal {*

*vertices { <köşe1> <köşe2> ... <köşeN> }*

*surfaces K* *// K=kenar_sayısı*

*{ <1.köşe 2.köşe ... K.köşe>* *// 1. yüzeyi oluşturan köşeler*

*<1.köşe 2.köşe ... K.köşe>* *// 2. yüzeyi oluşturan köşeler*

...

*<1.köşe 2.köşe ... K.köşe> }* *//M. yüzeyi oluşturan köşeler*

*smoothness yumuşaklık_faktörü } ... }*

**Açıklama**

Bu komut sisteme kullanıcı tarafından yaratılmış bir şekli çokgenler topluluğu olarak tanımlama imkanı verir. Komuta parametre olarak üç tanımlayıcı komut girilebilir.

İlk parametre vertices komutudur. Bu komut yardımıyla poligonlar topluluğunu oluşturan bütün köşe noktalarının tanımı yapılır. İlk nokta 1 olacak şekilde sırasıyla bütün noktalar üç boyutlu koordinatlarıyla tanımlanır.

İkinci parametre ise surfaces komutudur. Bu komut yardımıyla şekli oluşturan çokgenlerin (yüzey parçalarının) tanımı yapılır. Komuta ilk parametre olarak yüzeylerin kenar sayıları girilir. Daha sonrada esas şekli oluşturan çokgenler kenar sayıları bu sayıya eşit olacak şekilde tanımlanır. Bu çokgen tanımında köşe noktaları indeks değerleri saat yönü sırasına göre kullanılmalıdır. Aksi taktirde bu çokgene ait normal vektörü doğru olarak hesaplanamayacağı için cismin görüntüsü doğru olarak elde edilemeyebilir.

Üçüncü parametre ise smoothness komutudur. Bu komut oluşturulan cismin kenarlarına sanal bir yumuşaklık etkisi vermek için kullanılır. Komuta parametre olarak 0.0 (hiç yumuşaklık yok, sert kenarlar) ile 1.0 (tamamen

yumuşak kenarlar) arasında herhangi bir değer girilebilir. Komut seçimlik bir parametre olduğu için, girilmediği taktirde sistem otomatik olarak 0.0 değeri girilmiş gibi kabul eder.

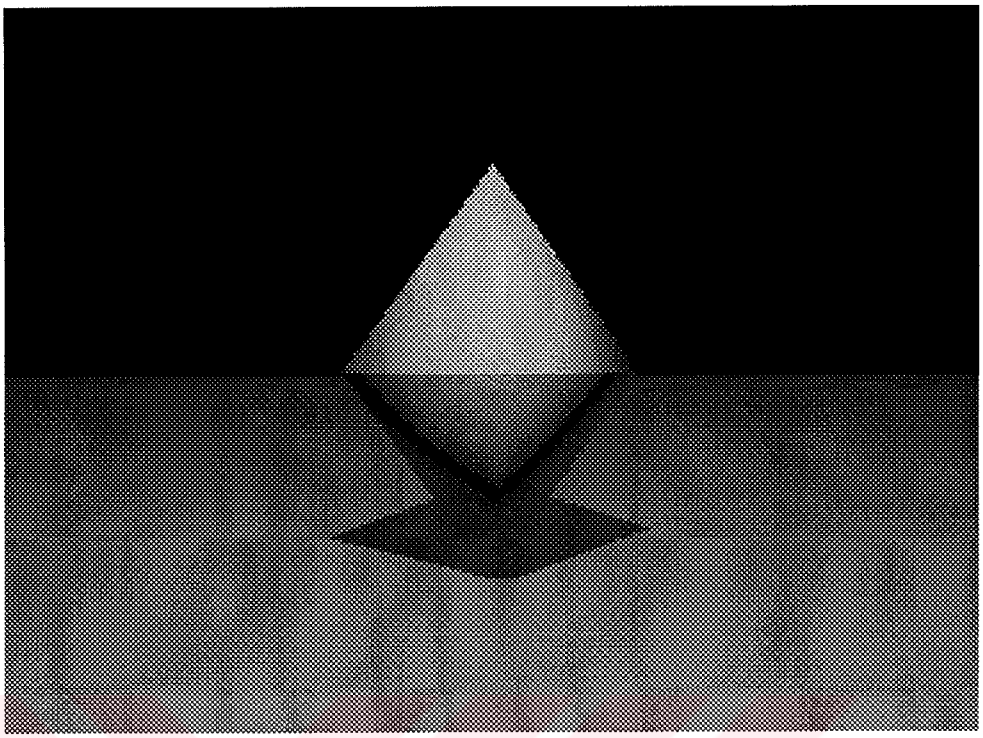
**İlgili diğer komutlar**

object, smoothness, surfaces, vertices

**Örnek**

```
// Polygon.TRC

// Üçgenlerden oluşmuş bir şekil tanımı

object { polygonal {

        vertices {

            < 1 0 1 >          // 1. nokta

            < 1 0 -1 >         // 2. nokta

            < 0 2 0 >          // 3. nokta

            < -1  0  1 >       // 4. nokta

            <  0 -2  0 >       // 5. nokta

            < -1  0 -1 > }     // 6. nokta

        surfaces 3 { // üç kenarlı yüzeyler

            < 1 2 3 >          // 1. yüzey

            < 1 3 4 >          // 2. yüzey
```

```
                < 6 4 3 >              // 3. yüzey
                < 3 2 6 >              // 4. yüzey
                < 4 5 1 >              // 5. yüzey
                < 2 1 5 >              // 6. yüzey
                < 6 2 5 >              // 7. yüzey
                < 5 4 6 > }            // 8. yüzey
            smoothness 0.5 }
        rotate <0 38 0>
         color  <0.85 0.85 0.09>
         diffuse 0.8
         phong 1.0 }
// Gölgelerin görünmesi için taban düzlemi
object { plane { <0 1 0> -2 } color < 0.0 0.7 0.7 > }
// Sisteme bakılacak nokta
camera { location <0 0 7> }
// Işık kaynağı tanımı
lightSource { location <0 5 5> }
// Ekran tanımı
screen { width 320 height 240 }
```



Şekil 13.21. Polygonal komutuna örnek

### 13.3.18. Reflection

**Tanım**

Bu komut, cisim yüzeyinde oluşan yansıma miktarını kontrol etmek için kullanılır.

**Yazım**

*object { <nesne_tipi> { <nesne_tanımı> } reflection <yansıma_miktarı> }*

**Açıklama**

Işın-izleme metodlarının diğer görüntü oluşturma yöntemlerine göre üstünlüğü, yüzeyler üzerinde oluşan yansıma değerlerini çok kesin olarak bulabilme özelliğidir. Bu bölümde tanıtılan "reflection" komutu da, bu yöntemle elde edilen yansıma ışınlarının miktarını kontrol etmek amacıyla kullanılır. Komuta parametre olarak yansıma değeri girilmelidir. Bu değer 0.0 (hiç yansıma yok) ile 1.0 (tamamen yansımalı aynalı yüzey) arasında herhangi sayı olabilir. Ancak bir yüzey üzerinde bu özelliğin tanımlanması ekstra yansıma ışınlarının da hesaba katılmasını doğuracağından, toplam hesap zamanı oldukça yavaşlayacaktır.

Burada dikkat edilecek bir nokta vardır. Işın-izleme sistemlerinde görüntü oluşturmak için kullanılan ışınlar, kamerdan başlayarak tanımlanan nesneler üzerine, oradan da diğer yüzeylere veya ışık kaynaklarına doğru yönelecek şekilde oluşturulur. Gerçek hayatta ışık kaynaklarından yayılarak varolan nesneleri aydınlatan yayınımın, bu yapıya göre gerçeklenmesi imkansızdır. Dolayısıyla, ışık kaynaklarından çıkıp aynalı yüzeyler üzerinden yansıyarak diğer nesneleri aydınlatan ışık ışınları bu sistemde gerçekleştirilemez. Buna göre, sistem yardımıyla oluşturulan görüntülerde aynalı yüzeyli cisimler görüntülenebilmesine rağmen, bu tip yüzeyler yardımıyla başka yüzeylerin aydınlatılması sağlanamaz. Bu tip aydınlanma "radiosity" adlı başka bir yöntem yardımıyla gerçekleştirilebilir. Bu yöntem oldukça farklı bir hesaplama şekli içerir ve sistem tarafından desteklenmemektedir.

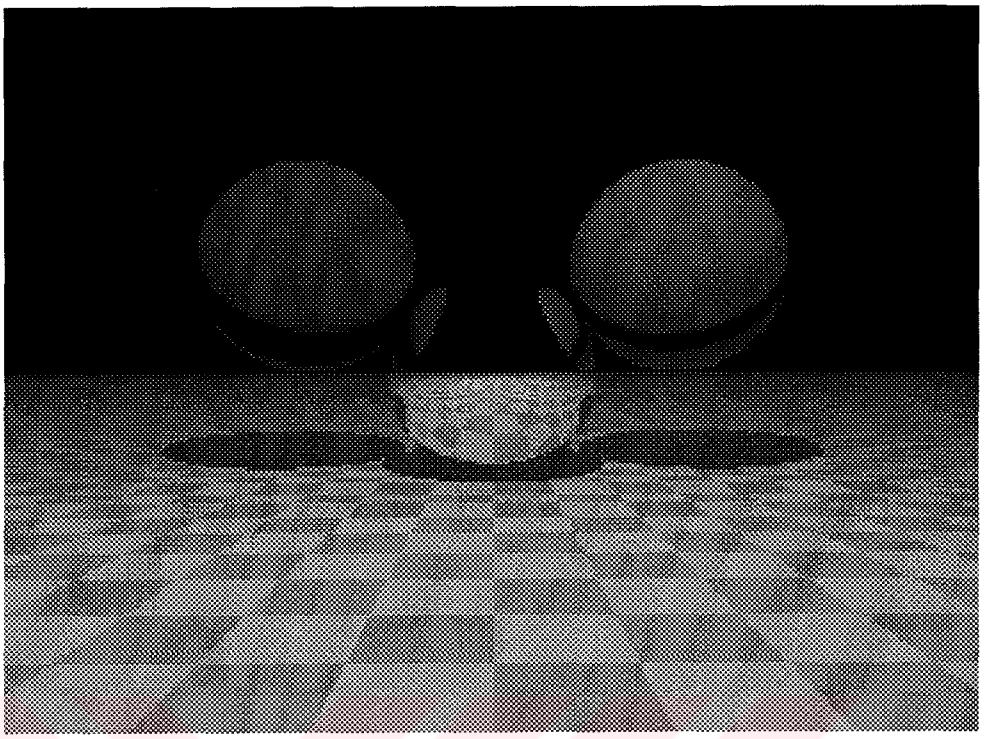
**İlgili diğer komutlar**

ambient, diffuse, lightSource

**Örnek**

*// Reflect.TRC*

```
// Normal bir küre - yansıma değeri = 0.0

object { sphere { < -1.75 1 0 > 1 } color <0 0 0.85>  }

// Tamamen aynalı bir küre - yansıma değeri 1.0

object { sphere { < 0 0 0 > 1 } color <0.85  0 0> reflection 1.0 }

// Normal bir küre - yansıma değeri = 0.0

object { sphere { < 1.75 1 0 > 1 } color <0.85  0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 }

        checker { color < 0.0 0.7 0.7 > color < 0.7 0.7 0.0> scale 0.5 } }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.22. Reflection komutuna örnek

### 13.3.19. Right

**Tanım**

Bu komut, görüntü düzleminin yatay eksenini belirtmek için kullanılır.

**Yazım**

*screen { ... right <ekrana_ait_yatay_eksen_vektörü> }*

**Açıklama**

Bu komut, görüntü düzleminin yatay eksenini ve birim genişliğini belirtmek

için kullanılır. Eğer bu komut kullanılmazsa sistem otomatik olarak <1.33 0 0> değerini kabul edecektir. Bu değer "up" değerinin <0 1 0> olduğu varsayılarak 4:3 oranını koruyabilmek için seçilmiştir. Bu oran kişisel bilgisayarlarda genellikle kullanılan ekran çözünürlüklerinde bulunan oranla aynıdır. Ancak kullanıcı bu komuta parametre olarak istediği vektör değerini girmekte serbesttir. Sistem bu değeri kullanılan ekran moduna veya piiksel boyutuna bağlı olmadan işler. Ama ortaya çıkacak resmin anlamlı bir görüntü içerebilmesi için kullanıcı bu komuta ve "up" komutuna vereceği parametrelere dikkat etmelidir. Örneğin kare bir görüntü elde edebilmek için "up <0 1 0>" ve "right <1 0 0>" değerleri girilmelidir.

**İlgili diğer komutlar**

screen, up

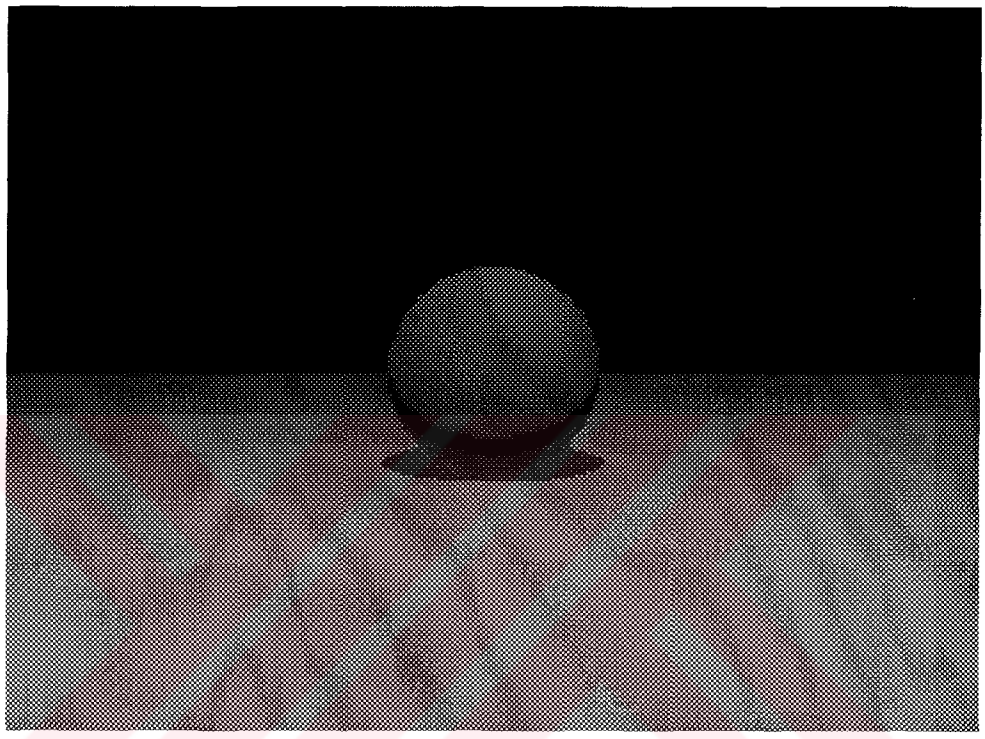
**Örnek 1**

```
// Right1.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı
```

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 up <0 1 0> right <1.33 0 0> }*



Şekil 13.23. Right komutuna örnek - 1

**Örnek 2**

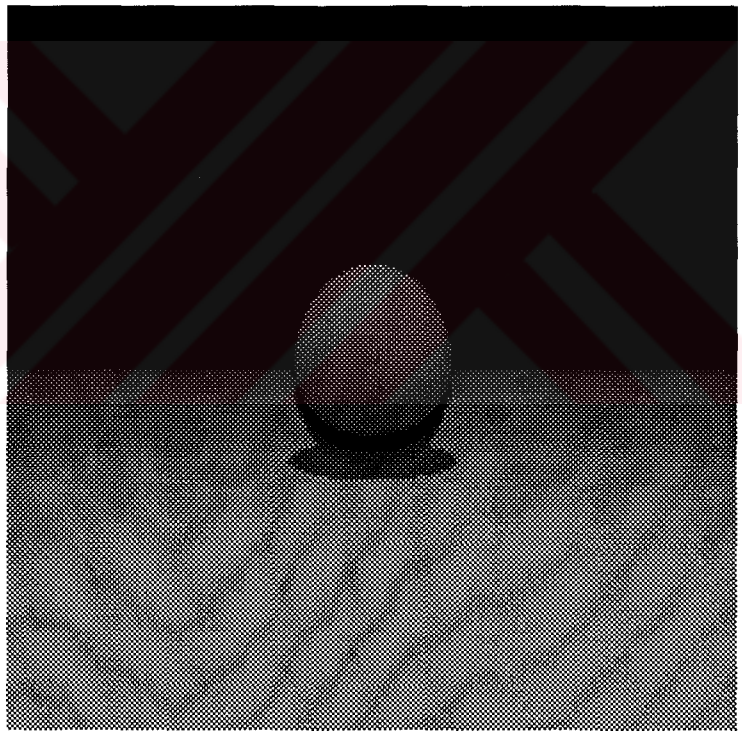
*// Right2.TRC*

*// Bir küre*

*object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }*

*// Gölgelerin görünmesi için taban düzlemi*

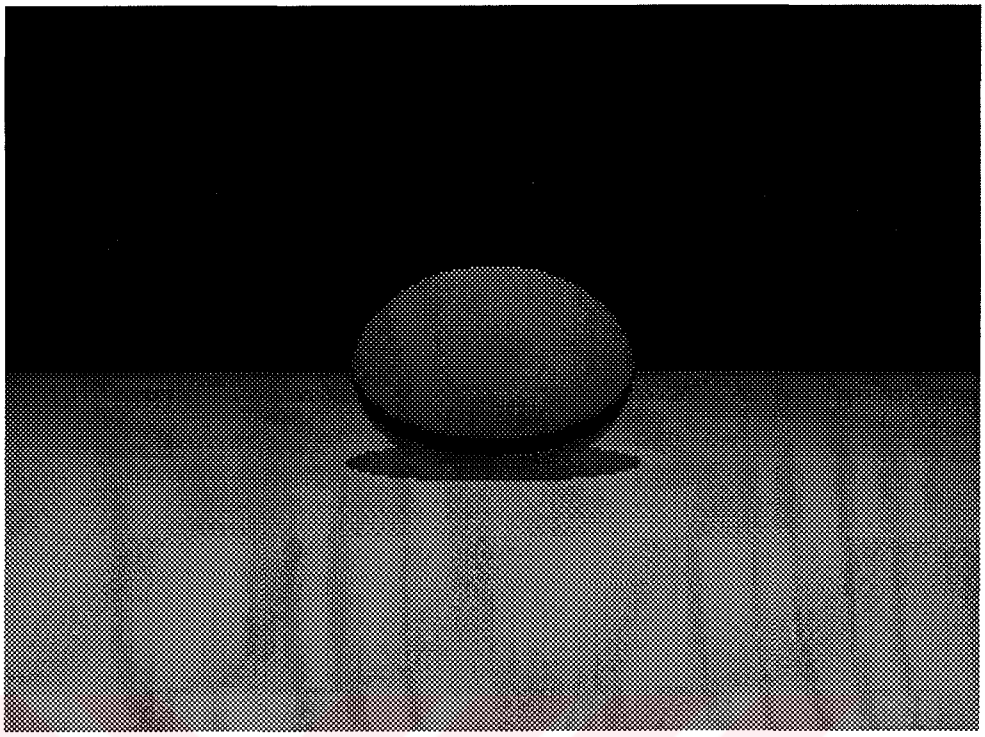
```
object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 240 height 240 up <0 1 0> right <1.33 0 0> }
```



Şekil 13.24. Right komutuna örnek - 2

**Örnek 3**

```
// Right3.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 up <0 1 0> right <1 0 0> }
```



Şekil 13.25. Right komutuna örnek - 3

**Örnek 4**

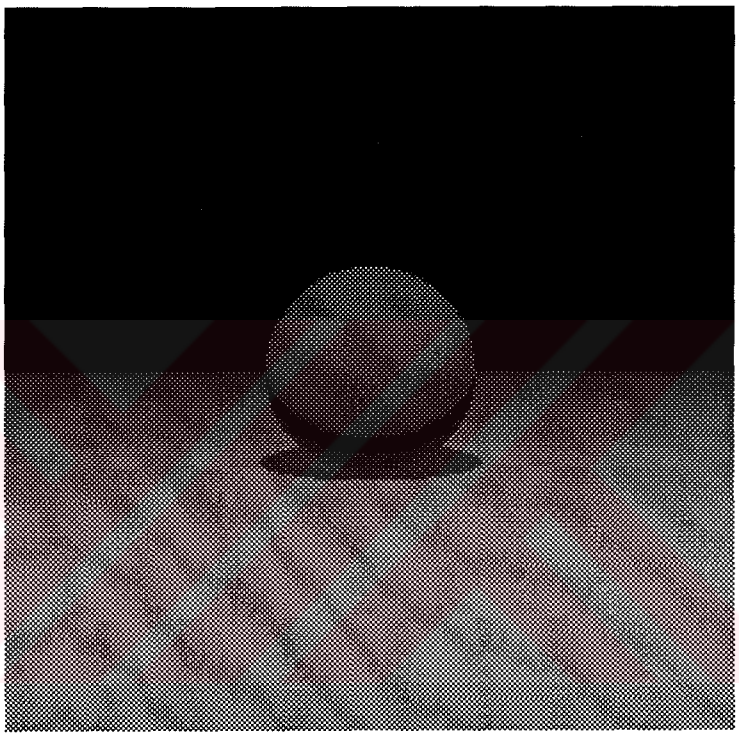
```
// Right4.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }
```

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 240 height 240 up <0 1 0> right <1 0 0> }*



Şekil 13.26. Right komutuna örnek - 4

### 13.3.20. Rotate

**Tanım**

Bu komut, bir cismi belirtilen doğrultuda ve açıda döndürmek için kullanılır.

**Yazım**

*object { ... rotate < x_açısı y_açısı z_açısı > }*

**Açıklama**

Bu komut, tanımlanan bir nesneyi x, y ve z eksenleri etrafında döndürmek için kullanılır. Burada dikkat edilecek nokta, dönme işlemlerinin ilgili cismin merkezi etrafında değil daima koordinat eksenleri etrafında olduğudur. Dolayısıyla (0,0,0) noktasından uzakta olan cisimler üzerinde uygulanan dönme işlemleri sonucunda elde edilecek sonuç, cismin merkezi etrafında yapacağı dönüşümden oldukça farklı olacaktır. Bu sorunu çözebilmek için cismin merkezi (0,0,0) noktası üzerinde olacak şekilde tanımlanır. Daha sonra dönme işlemi bu nokta üzerindeyken gerçekleştirildikten sonra "translate" komutu yardımıyla cisim istenilen pozisyona ötelenir.

Komuta parametre olarak girilen açı değerleri derece cinsinden verilmelidir. Genellikle bu parametre içinde girilen değerlerin en az biri, veya daha iyi sonuçlar için ikisi 0.0 olmalıdır. Dönme işlemleri sırasıyla x, y ve z eksenleri etrafında gerçekleştirilir. Dolayısıyla üç açı değeri de 0'dan büyük olursa, bu tip üç boyutlu dönüşümler tahmin edilemeyen görüntüler oluşturabilir. Eğer ortaya çıkacak sonuçlardan emin olunamıyorsa, bir cisim için birden fazla sayıda ayrı "rotate" komutları kullanılabilir.

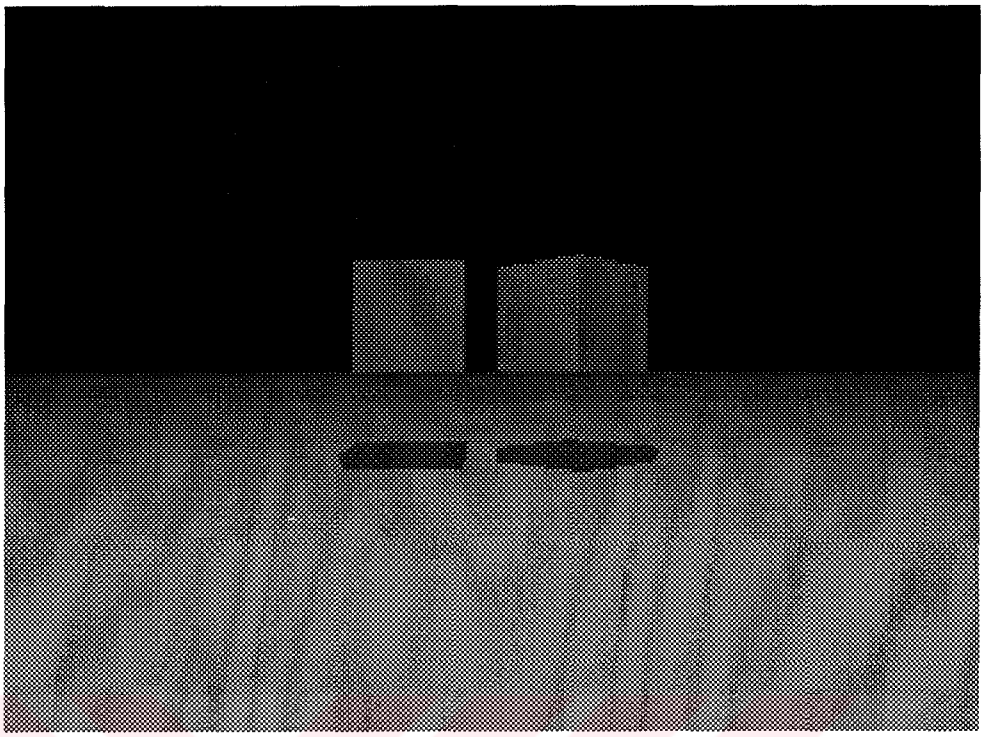
**İlgili diğer komutlar**

object, scale, translate

**Örnek**

*// Rotate.TRC*

```
// Normal bir kutu

object { box { <-0.5 -0.5 -0.5> <0.5 0.5 0.5> }

        color < 1.0 0.0 0.0> rotate < 0 0 0 > translate <-0.75 0.5 0 > }

// Dönmüş bir kutu

object { box { <-0.5 -0.5 -0.5> <0.5 0.5 0.5> }

        color < 1.0 0.0 0.0> rotate < 0 45 0 > translate <0.75 0.5 0 > }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.27. Rotate komutuna örnek

### 13.3.21. Scale

**Tanım**

Bu komut, bir cismin büyüklüğünü belirtilen ölçüde değiştirmek için kullanılır.

**Yazım**

*object { ... scale <büyüklük_faktörü> }*

**Açıklama**

Bu komut, tanımlandığı cismin büyüklüğünü belirtilen faktör oranında

değiştirmeye yarar. Komuta parametre olarak sıfırdan büyük herhangi bir sayı girilebilir. Bu değer cismin büyüklüğünün çarpılacağı oranı göstermektedir. Örneğin bir cismin büyüklüğünü iki katına çıkarabilmek için komuta parametre olarak 2.0 değeri girilmelidir.

**İlgili diğer komutlar**

object, rotate, translate

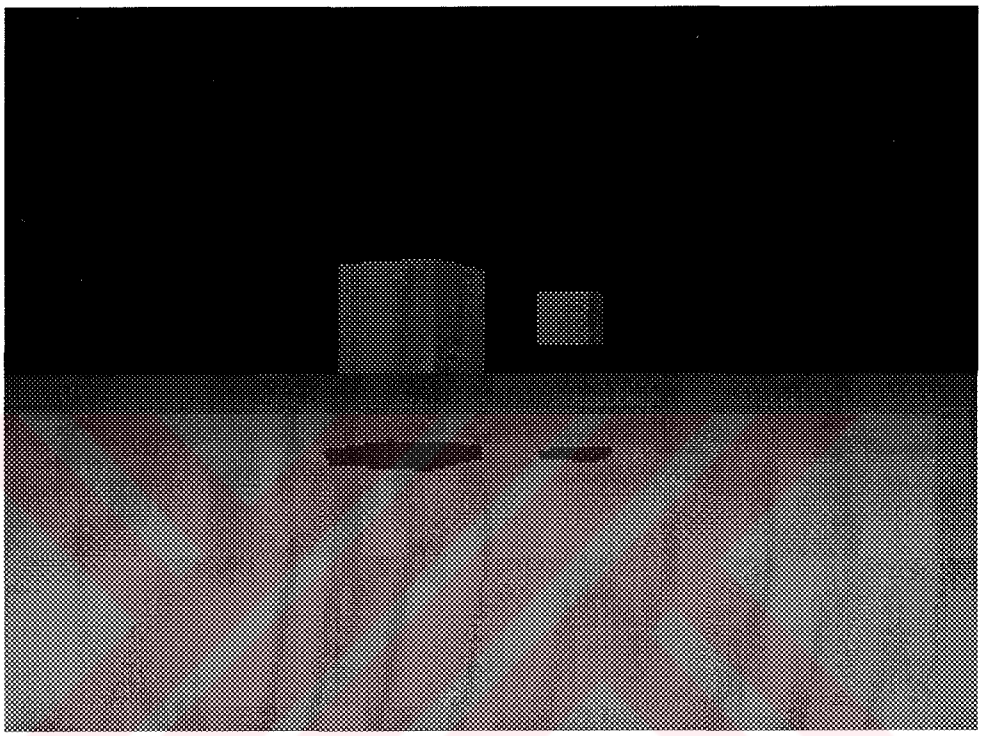
**Örnek**

```
// Scale.TRC

// Normal bir kutu

object { box { <-0.5 -0.5 -0.5> <0.5 0.5 0.5> }

        color < 1.0 0.0 0.0> scale 1 rotate <0 25 0> translate <-0.75 0.5 0 > }

// Yarıyarıya küçültülmüş bir kutu

object { box { <-0.5 -0.5 -0.5> <0.5 0.5 0.5> }

        color <1.0 0.0 0.0> scale 0.5 rotate <0 25 0> translate <0.75 0.5 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı
```

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.28. Scale komutuna örnek

### 13.3.22. Screen

**Tanım**

Bu komut, sisteme görüntü düzlemini tanımlamak için kullanılır.

**Yazım**

*screen { width <genişlik> height<yükseklik>*

*up<dikey_vektör> right<yatay_vektör> }*

**Açıklama**

Bu komut, görüntü düzleminin tanımlanması için kullanılır. Komuta parametre olarak dört parametre girilebilir. Bu parametrelerin tanımında dikkat edilecek nokta, tanımlama sırasının olmamasıdır.

Birinci parametre, "width" komutudur. Bu komut, tanımlanan sahnenin görüntüsü oluşturulurken, diske yazılacak Targa dosyasının genişlik değerini piksel olarak belirlemek amcıyla kullanılır. Komuta parametre olarak genişlik değeri girilmelidir. Bu değer sıfırdan büyük herhangi bir tamsayı olabilir. Ancak bu değer ile height komutunun değeri arasındaki oran ile kamera tanımında bulunan up ve right komutlarının arasındaki oran aynı olması, görüntünün geçekçiliği açısından önem taşır. Örneğin "up" değeri (0, 1, 0), "right" değeri (1.333, 0, 0) olan bir kamera tanımına uygun olarak tanımlanacak "screen" komutunun parametreleri arasındaki oran 1.333 olmalıdır. Yani,

$$\frac{|right|}{|up|} = \frac{width}{height}$$

oranı korunursa, elde edilen görüntüler daha gerçekçi olur.

İkinci parametre ise "height" komutudur. Bu komut, tanımlanan sahnenin görüntüsü oluşturulurken, diske yazılacak Targa dosyasının yükseklik değerini piksel olarak belirlemek amcıyla kullanılır. Komuta parametre olarak yükseklik değeri girilmelidir. Bu değer sıfırdan büyük herhangi bir tamsayı olabilir. Ancak

bu değer ile width komutunun değeri arasındaki oran ile kamera tanımında bulunan up ve right komutlarının arasındaki oran aynı olması, görüntünün geçekçiliği açısından önem taşır. Örneğin "up" değeri (0, 1, 0), "right" değeri (1.333, 0, 0) olan bir kamera tanımına uygun olarak tanımlanacak "screen" komutunun parametreleri arasındaki oran 1.333 olmalıdır. Yani,

$$\frac{|right|}{|up|} = \frac{width}{height}$$

oranı korunursa, elde edilen görüntüler daha gerçekçi olur.

Üçüncü parametre ise "up" komutudur. Bu komut, seçimlik bir parametre olup, görüntü düzleminin dikey eksenini ve birim yüksekliğini belirtmek için kullanılır. Eğer bu komut kullanılmazsa sistem otomatik olarak <0 1 0> değerini kabul edecektir. Bu değer "right" değerinin <1.33 0 0> olduğu varsayılarak 4:3 oranını koruyabilmek için seçilmiştir. Bu oran kişisel bilgisayarlarda genellikle kullanılan ekran çözünürlüklerinde bulunan oranla aynıdır. Ancak kullanıcı bu komuta parametre olarak istediği vektör değerini girmekte serbesttir. Sistem bu değeri kullanılan ekran moduna veya piiksel boyutuna bağlı olmadan işler. Ama ortaya çıkacak resmin anlamlı bir görüntü içerebilmesi için kullanıcı bu komuta ve "right" komutuna vereceği parametrelere dikkat etmelidir. Örneğin kare bir görüntü elde edebilmek için "up <0 1 0>" ve "right <1 0 0>" değerleri girilmelidir.

Dördüncü parametre ise "right" komutudur. Bu komut, seçimlik bir parametre olup, görüntü düzleminin yatay eksenini ve birim genişliğini belirtmek için kullanılır. Eğer bu komut kullanılmazsa sistem otomatik olarak <1.33 0 0> değerini kabul edecektir. Bu değer "up" değerinin <0 1 0> olduğu varsayılarak 4:3 oranını koruyabilmek için seçilmiştir. Bu oran kişisel bilgisayarlarda genellikle kullanılan ekran çözünürlüklerinde bulunan oranla aynıdır. Ancak

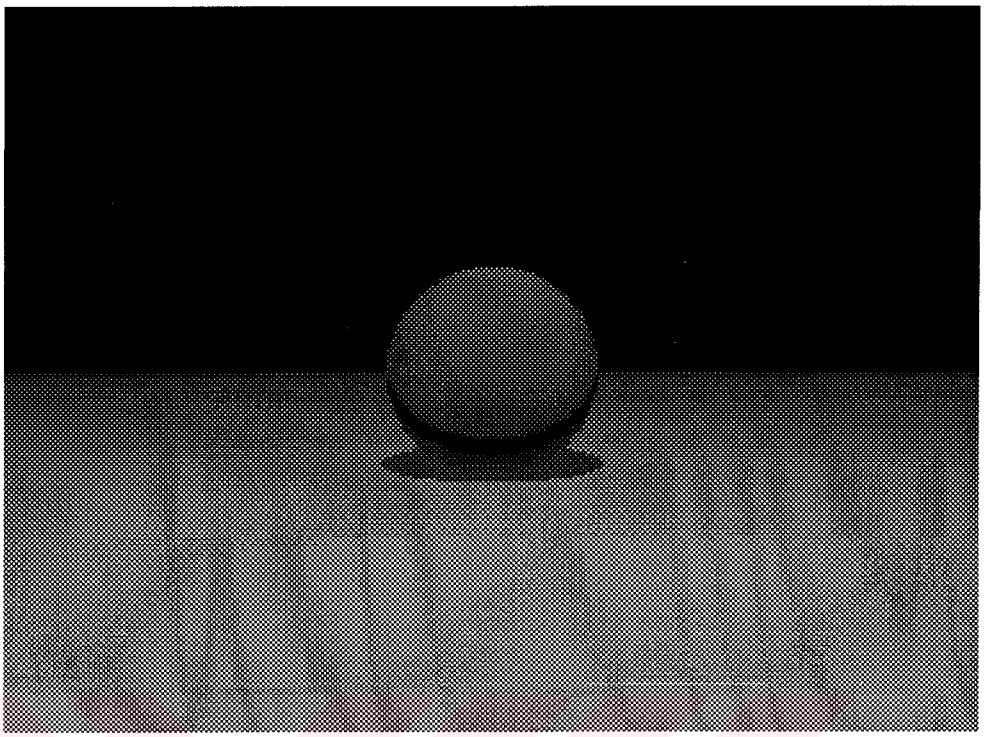
kullanıcı bu komuta parametre olarak istediği vektör değerini girmekte serbesttir. Sistem bu değeri kullanılan ekran moduna veya piiksel boyutuna bağlı olmadan işler. Ama ortaya çıkacak resmin anlamlı bir görüntü içerebilmesi için kullanıcı bu komuta ve "up" komutuna vereceği parametrelere dikkat etmelidir. Örneğin kare bir görüntü elde edebilmek için "up <0 1 0>" ve "right <1 0 0>" değerleri girilmelidir.

**İlgili diğer komutlar**

height, right, up, width

**Örnek**

```
// Screen.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 up <0 1 0> right <1.33 0 0> }
```



Şekil 13.29. Screen komutuna örnek

### 13.3.23. Smoothness

**Tanım**

Bu komut, tanımlanmış bir poligonun aydınlatılması gerçekleştirilirken kenarların ne kadar yumuşatılacağını belirtmek için kullanılır.

**Yazım**

*object { polygonal { ... smoothness yumuşaklık_faktörü } ... }*

**Açıklama**

Bu komut "polygonal" komutuyla oluşturulan cismin kenarlarına sanal bir yumuşaklık etkisi vermek için kullanılır. Komuta parametre olarak 0.0 (hiç yumuşaklık yok, sert kenarlar) ile 1.0 (tamamen yumuşak kenarlar) arasında herhangi bir değer girilebilir. Komut seçimlik bir parametre olduğu için, girilmediği taktirde sistem otomatik olarak 0.0 değeri girilmiş gibi kabul eder.

**İlgili diğer komutlar**

object, polygonal, surfaces, vertices

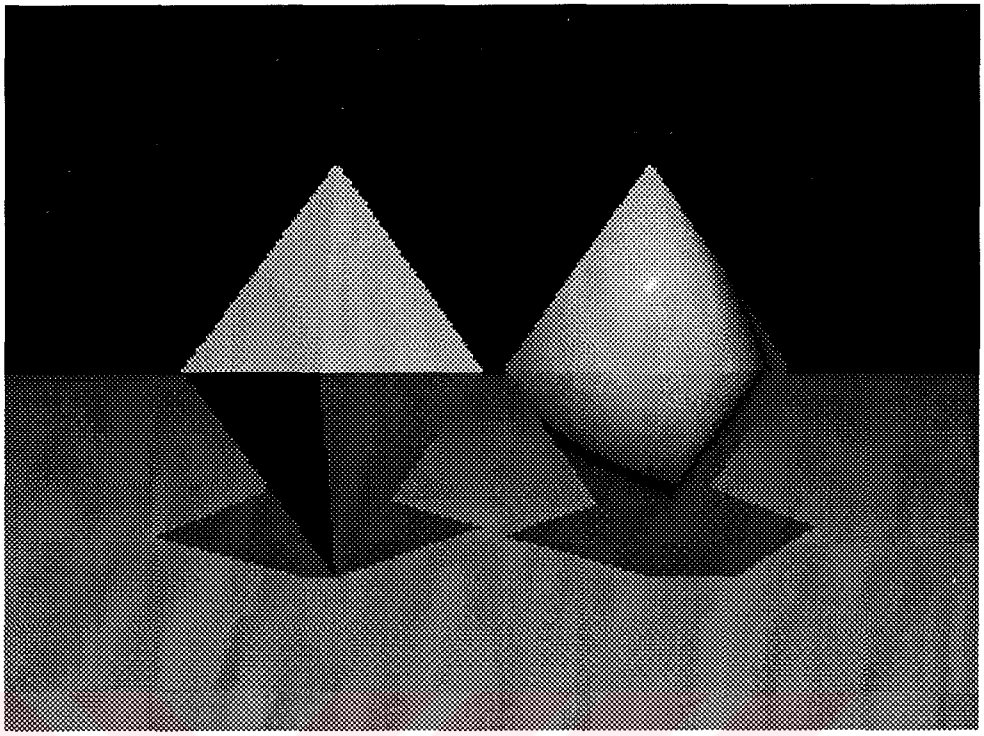
**Örnek**

```
// Smooth.TRC

// Üçgenlerden oluşmuş bir şekil tanımı - yumuşaklık faktörü = 0.0

object { polygonal {

        vertices {

            < 1 0 1 >          // 1. nokta

            < 1 0 -1 >         // 2. nokta

            < 0 2 0 >          // 3. nokta

            < -1  0  1 >       // 4. nokta

            <  0 -2  0 >       // 5. nokta

            < -1  0 -1 > }     // 6. nokta
```

```
        surfaces 3 { // üç kenarlı yüzeyler
            < 1 2 3 >          // 1. yüzey
            < 1 3 4 >          // 2. yüzey
            < 6 4 3 >          // 3. yüzey
            < 3 2 6 >          // 4. yüzey
            < 4 5 1 >          // 5. yüzey
            < 2 1 5 >          // 6. yüzey
            < 6 2 5 >          // 7. yüzey
            < 5 4 6 > }        // 8. yüzey
        smoothness 0.0 }
    rotate <0 38 0>
    translate <-1.5 0 0>
     color  <0.85 0.85 0.09>
     diffuse 0.8
    phong 1.0 }
// Üçgenlerden oluşmuş bir şekil tanımı - yumuşaklık faktörü = 0.8
object { polygonal {
        vertices {
            < 1 0 1 >          // 1. nokta
```

```
< 1 0 -1 >              // 2. nokta

< 0 2 0 >               // 3. nokta

< -1  0  1 >            // 4. nokta

<  0 -2  0 >           // 5. nokta

< -1  0 -1 > }          // 6. nokta

surfaces 3 { // üç kenarlı yüzeyler

< 1 2 3 >               // 1. yüzey

< 1 3 4 >               // 2. yüzey

< 6 4 3 >               // 3. yüzey

< 3 2 6 >               // 4. yüzey

< 4 5 1 >               // 5. yüzey

< 2 1 5 >               // 6. yüzey

< 6 2 5 >               // 7. yüzey

< 5 4 6 > }             // 8. yüzey

smoothness 0.8 }

rotate <0 38 0>

translate <1.5 0 0>

color  <0.85 0.85 0.09>

diffuse 0.8
```

```
        phong 1.0 }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -2 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.30. Smoothness komutuna örnek

### 13.3.24. Sphere

**Tanım**

Bu komut, sisteme, işlenecek cismin şeklinin bir küre olduğunu anlatmak için kullanılır.

**Yazım**

*object { sphere { <merkez_koordinatları> yarıçap_değeri } ... }*

**Açıklama**

Bu komut, sisteme bir küre tanımlamak için kullanılır. Komuta parametre olarak küre merkezinin üç boyutlu koordinat değeri ve kürenin yarıçapı girilir. Yarıçap değeri olarak sıfırdan büyük herhangi bir gerçel sayı girilebilir.

**İlgili diğer komutlar**

object

**Örnek**

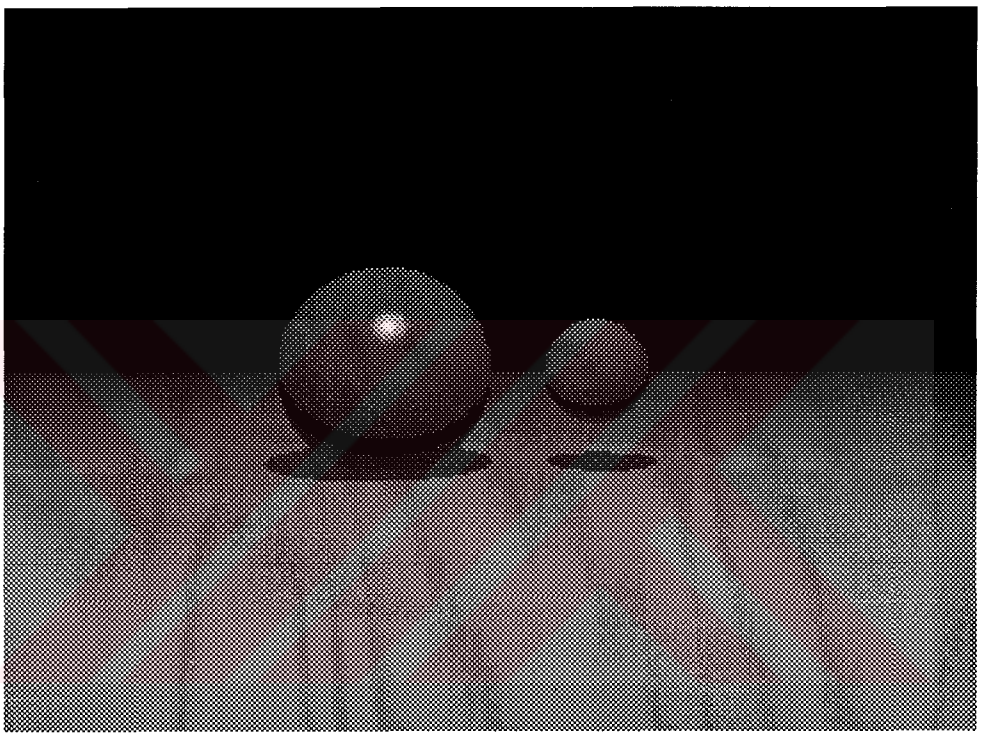
```
// Sphere.TRC
// Bir küre
object { sphere { < 0 0 0 > 1 }
        color <0.85 0 0> translate <-1 0 0> phong 1.0}
// Başka bir küre
object { sphere { < 0 0 0 > 1 }
        color <0.85 0 0> translate <1 0 0> scale 0.5 phong 1.0}
// Gölgelerin görünmesi için taban düzlemi
object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }
// Sisteme bakılacak nokta
camera { location <0 0 7> }
```

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.31. Sphere komutuna örnek

### 13.3.25. Surfaces

**Tanım**

Bu komut, işlenmek üzere tanımlanan çokgenler kümesinin yüzey bilgilerini sisteme belirtmek için kullanılır.

**Yazım**

*object { polygonal {*

*<köşelerin_tanımı>*

*surfaces K* *// K=kenar_sayısı*

*{ <1.köşe 2.köşe ... K.köşe>* *// 1. yüzeyi oluşturan köşeler*

*<1.köşe 2.köşe ... K.köşe>* *// 2. yüzeyi oluşturan köşeler*

...

*<1.köşe 2.köşe ... K.köşe> }* *// M. yüzeyi oluşturan köşeler*

*... } ... }*

**Açıklama**

Bu komut, sisteme bir çokgenler kümesinin yüzey bilgilerini tanımlamak için kullanılır. Bu komut yardımıyla poligonlar topluluğunu oluşturan bütün yüzey bilgilerinin tanımı yapılır. Bu tanımda, "vertices" komutu yardımıyla belirtilen köşe noktalarının indeks değerleri kullanılır.

Bu komut yardımıyla şekli oluşturan çokgenlerin (yüzey parçalarının) tanımı yapılır. Komuta ilk parametre olarak yüzeylerin kenar sayıları girilir. Daha sonrada esas şekli oluşturan çokgenler kenar sayıları bu sayıya eşit olacak şekilde tanımlanır. Bu çokgen tanımında köşe noktaları indeks değerleri saat yönü sırasına göre kullanılmalıdır. Aksi taktirde bu çokgene ait normal vektörü doğru olarak hesaplanamayacağı için cismin görüntüsü doğru olarak elde edilemeyebilir.

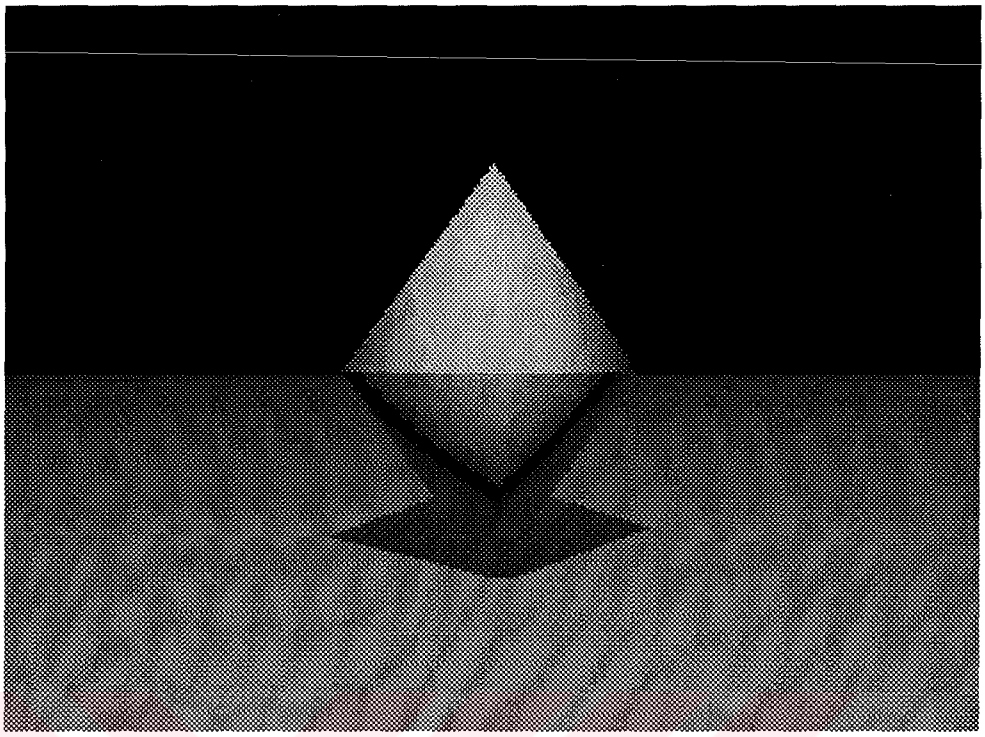
**İlgili diğer komutlar**

object, polygonal, smoothness, vertices

**Örnek**

```
// Surfaces.TRC
// Üçgenlerden oluşmuş bir şekil tanımı
object { polygonal {
        vertices {
            < 1 0 1 >          // 1. nokta
            < 1 0 -1 >         // 2. nokta
            < 0 2 0 >          // 3. nokta
            < -1  0  1 >       // 4. nokta
            <  0 -2  0 >       // 5. nokta
            < -1  0 -1 > }     // 6. nokta
        surfaces 3 { // üç kenarlı yüzeyler
            < 1 2 3 >          // 1. yüzey
            < 1 3 4 >          // 2. yüzey
            < 6 4 3 >          // 3. yüzey
            < 3 2 6 >          // 4. yüzey
```

```
                < 4 5 1 >              // 5. yüzey
                < 2 1 5 >              // 6. yüzey
                < 6 2 5 >              // 7. yüzey
                < 5 4 6 > }            // 8. yüzey
            smoothness 0.5 }
        rotate <0 38 0>
        color  <0.85 0.85 0.09>
        diffuse 0.8
        phong 1.0 }
// Gölgelerin görünmesi için taban düzlemi
object { plane { <0 1 0> -2 } color < 0.0 0.7 0.7 > }
// Sisteme bakılacak nokta
camera { location <0 0 7> }
// Işık kaynağı tanımı
lightSource { location <0 5 5> }
// Ekran tanımı
screen { width 320 height 240 }
```



Şekil 13.32. Surfaces komutuna örnek

### 13.3.26. Texture

**Tanım**

Bu komut, cisim üzerinde özel yüzey yapılarını simüle etmek için kullanılır.

**Yazım**

*object { <nesne_tanımı> texture { <yüzey_özelliği_bilgisi> } ... }*

**Açıklama**

Bu komut, tanımlandığı yüzey üzerinde özel yüzey yapılarını simüle etmek

amacıyla kullanılır. Sistem bu tip efektleri verebilmek için özel olarak tanımlanmış gürültü fonksiyonları kullanır. Komuta parametre olarak kullanılmak istenilen yüüzey özelliği tanımı girilmelidir. Sistem şu andaki yapısıyla, "granite", "marble" ve "wood" komutlarını parametre olarak kabul eder. Bu parametrelerle ilgili daha detaylı bilgi ilgili komutun tanıtıldığı bölümde bulunabilir.

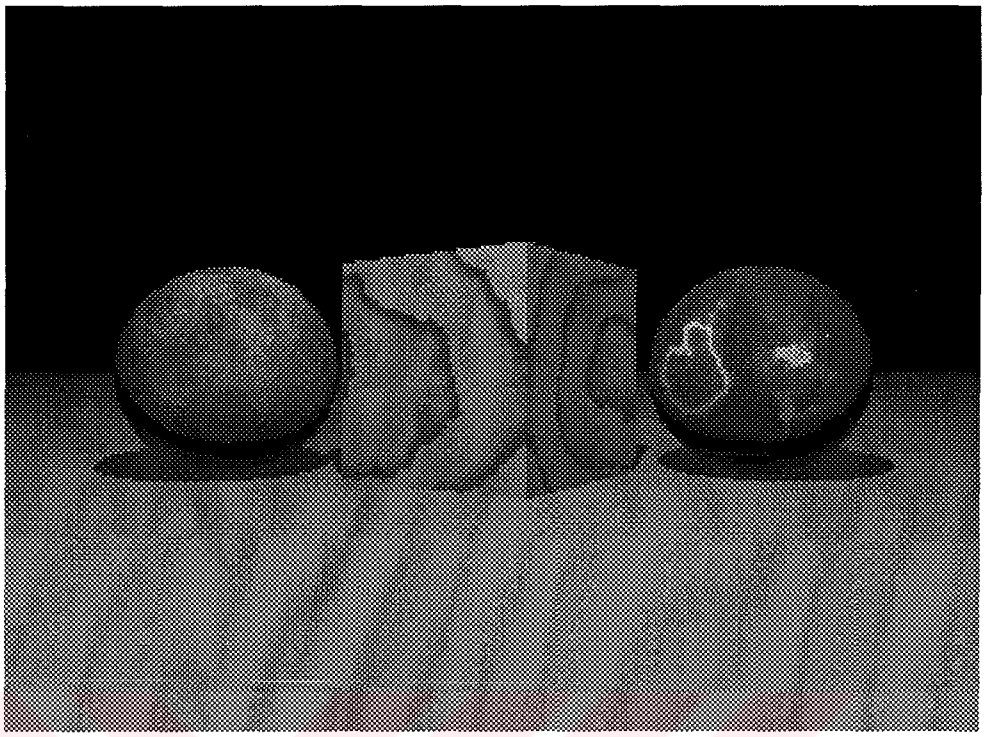
**İlgili diğer komutlar**

colorMap, granite, marble, wood

**Örnek**

```
// Texture.TRC

// Granit bir küre

object { sphere { < -2.5 0 0 > 1 }

        texture { granite color <0.051 0.224 0.060>

                color <0.773 0.727 0.050> } }

// Ağaçtan kesilmiş bir kutu

object { box { < -1 -1 -1> <1 1 1> }

        rotate <0 35 0>

        ambient 0.3

        texture { wood color <0.858824 0.576471 0.439216>

                color <0.647059 0.164706 0.164706> } }
```

```
// Mermer bir küre

object { sphere { < 2.5 0 0 > 1 }

        texture { marble color <0.6 0 0>  color <0.85  0.85  0.10> } }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.33. Texture komutuna örnek

### 13.3.27. Translate

**Tanım**

Bu komut, bir cismin yerini değiştirmek için kullanılır.

**Yazım**

*object { ... translate <öteleme_vektörü> }*

**Açıklama**

Bu komut, tanımlandığı cismin pozisyonunu belirtilen vektör kadar ötelemeye

yarar. Komuta parametre olarak üç boyutlu herhangi bir vektör değeri girilebilir.

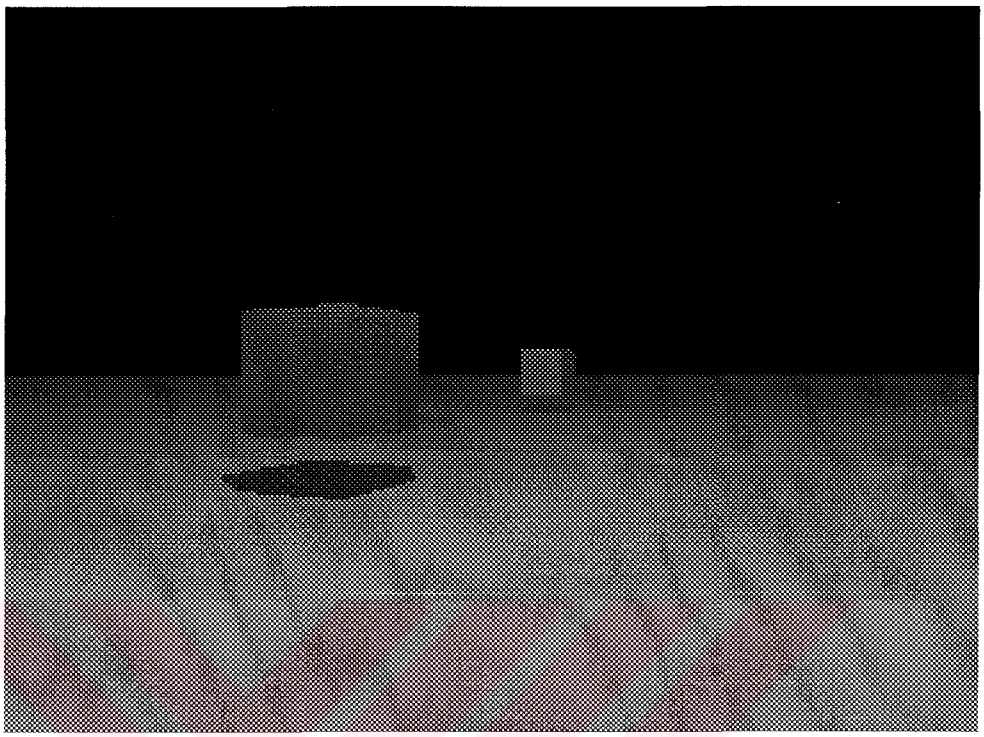
**İlgili diğer komutlar**

object, rotate, scale

**Örnek**

```
// Translat.TRC

// Sola ve öne ötelenmiş bir kutu

object { box { <-0.5 -0.5 -0.5> <0.5 0.5 0.5> }

        color < 1.0 0.0 0.0> rotate <0 25 0> translate <-1.3 0 1> }

// Sağa ve geriye ötelenmiş bir kutu

object { box { <-0.5 -0.5 -0.5> <0.5 0.5 0.5> }

        color < 1.0 0.0 0.0> rotate <0 25 0> translate <1.3 0 -10 > }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı
```

*screen { width 320 height 240 }*



Şekil 13.34. Translate komutuna örnek

### 13.3.28. Transparency

**Tanım**

Bu komut, cisimlerde şeffalık miktarını kontrol etmek için kullanılır.

**Yazım**

*object { <nesne_tipi> { <nesne_tanımı> } transparency <şeffaflık_miktarı> }*

**Açıklama**

Bu komut, tanımlandığı cismin şeffaflığını kontrol etmek için kullanılır. Komuta parametre olarak şeffaflık miktarı girilmelidir. Bu değer 0.0 (şeffaflık özelliği hiç yok) ile 1.0 (tamamen şeffaf) arasında herhangi sayı olabilir.

**İlgili diğer komutlar**

ambient, diffuse, lightSource, reflection

**Örnek**

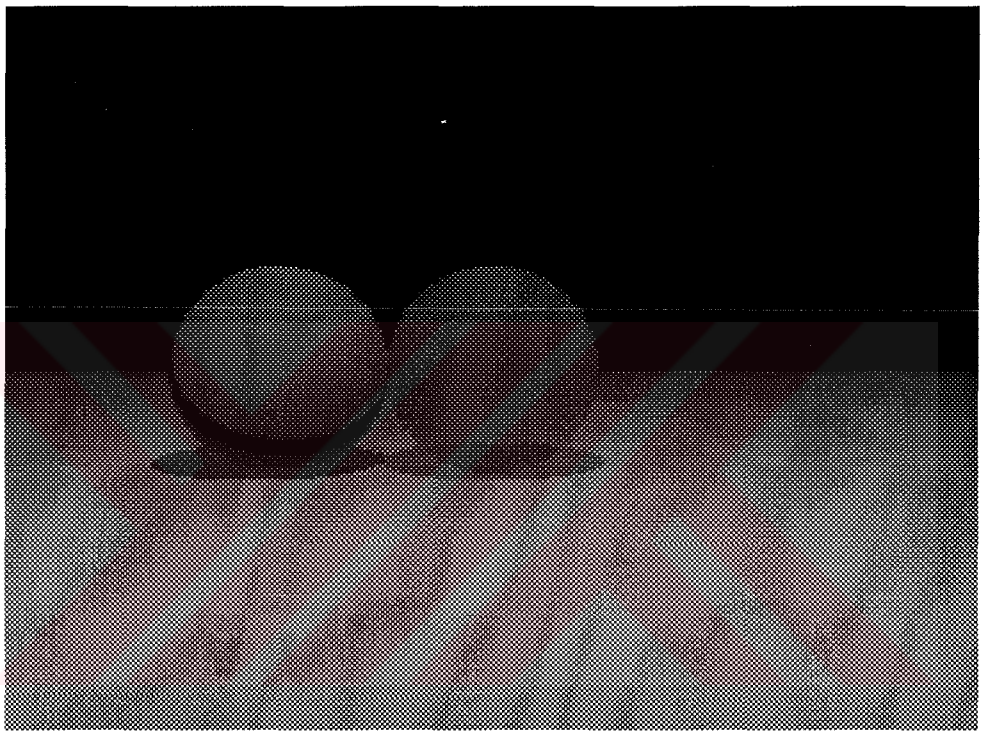
```
// Transprn.TRC

// Normal bir küre - şeffaflık değeri = 0.0

object { sphere { < -2 0 0 > 1 } color <0.85  0 0> transparency 0.0 }

// Yarı-şeffaf bir küre - şeffaflık değeri 0.5

object { sphere { < 0 0 0 > 1 } color <0.85  0 0> transparency 0.5 }

// Tamamen şeffaf bir küre - şeffaflık değeri = 1.0

object { sphere { < 2 0 0 > 1 } color <0.85  0 0> transparency 1.0 }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }
```

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 }*



Şekil 13.35. Transparency komutuna örnek

### 13.3.29. Up

**Tanım**

Bu komut, görüntü düzleminin dikey eksenini belirtmek için kullanılır.

**Yazım**

*screen { ... up <ekrana_ait_dikey_eksen_vektörü> }*

**Açıklama**

Bu komut, görüntü düzleminin dikey eksenini ve birim yüksekliğini belirtmek için kullanılır. Eğer bu komut kullanılmazsa sistem otomatik olarak <0 1 0> değerini kabul edecektir. Bu değer "right" değerinin <1.33 0 0> olduğu varsayılarak 4:3 oranını koruyabilmek için seçilmiştir. Bu oran kişisel bilgisayarlarda genellikle kullanılan ekran çözünürlüklerinde bulunan oranla aynıdır. Ancak kullanıcı bu komuta parametre olarak istediği vektör değerini girmekte serbesttir. Sistem bu değeri kullanılan ekran moduna veya piiksel boyutuna bağlı olmadan işler. Ama ortaya çıkacak resmin anlamlı bir görüntü içerebilmesi için kullanıcı bu komuta ve "right" komutuna vereceği parametrelere dikkat etmelidir. Örneğin kare bir görüntü elde edebilmek için "up <0 1 0>" ve "right <1 0 0>" değerleri girilmelidir.

**İlgili diğer komutlar**

screen, right

**Örnek 1**

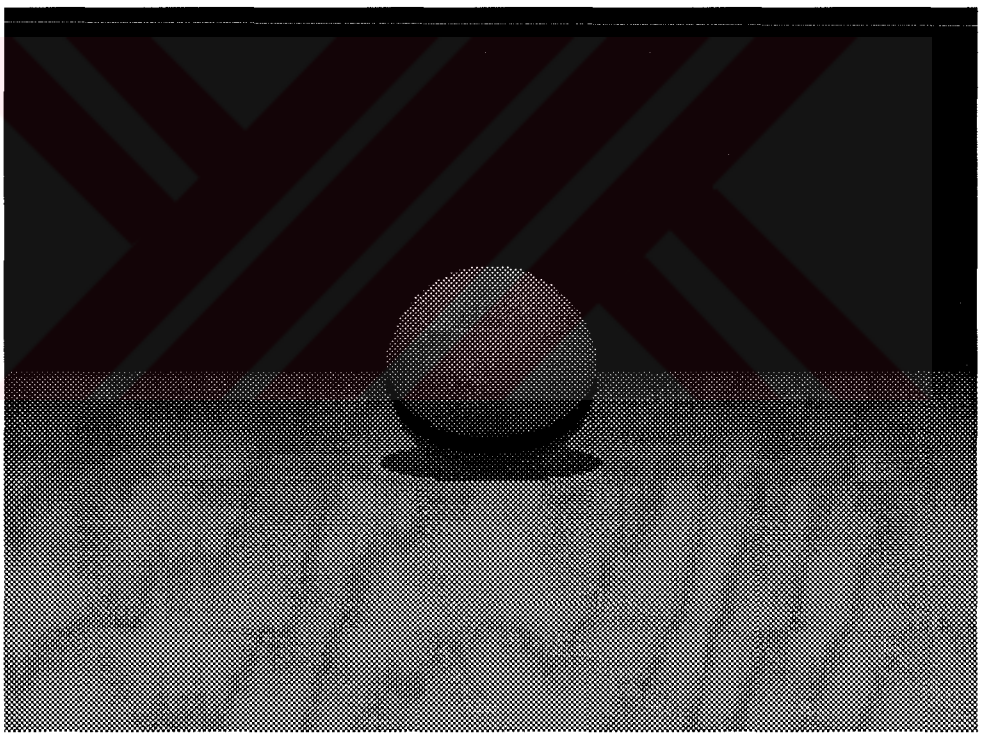
*// Up1.TRC*

*// Bir küre*

*object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }*

*// Gölgelerin görünmesi için taban düzlemi*

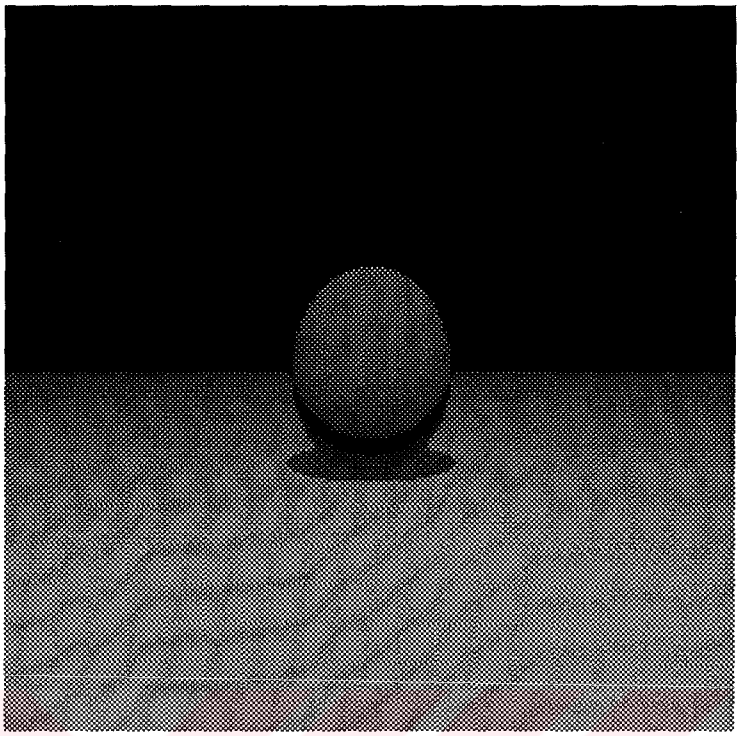
```
object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 320 height 240 up <0 1 0> right <1.33 0 0> }
```



Şekil 13.36. Up komutuna örnek - 1

**Örnek 2**

```
// Up2.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 240 height 240 up <0 1 0> right <1.33 0 0> }
```



Şekil 13.37. Up komutuna örnek - 2

**Örnek 3**

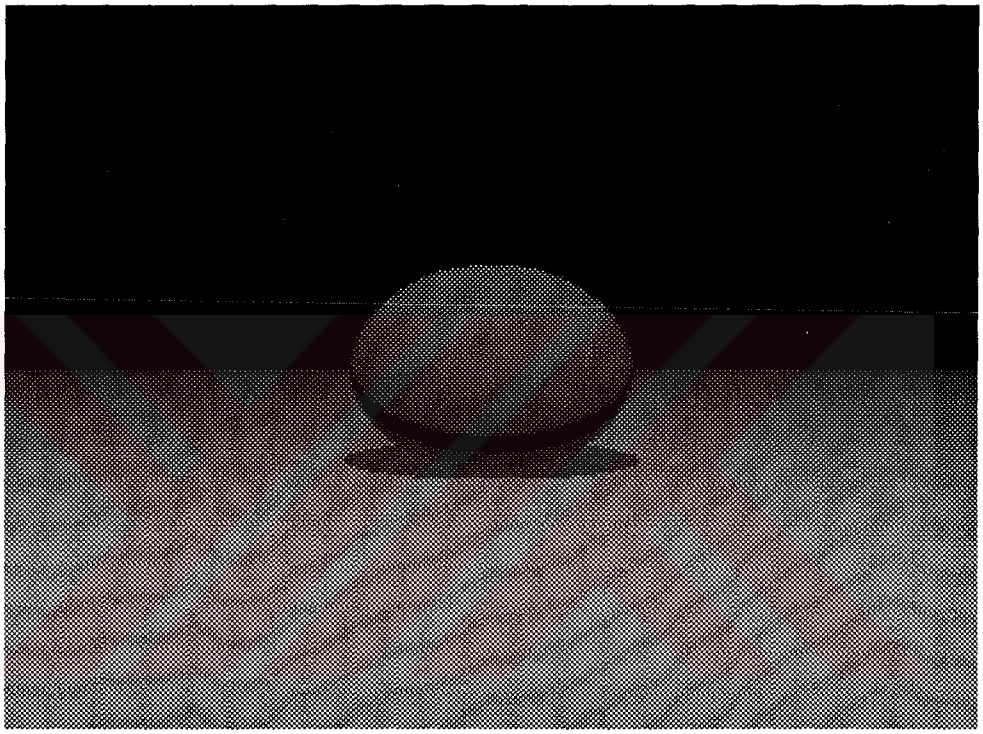
```
// Up3.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }
```

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

*screen { width 320 height 240 up <0 1 0> right <1 0 0> }*



Şekil 13.38. Up komutuna örnek - 3

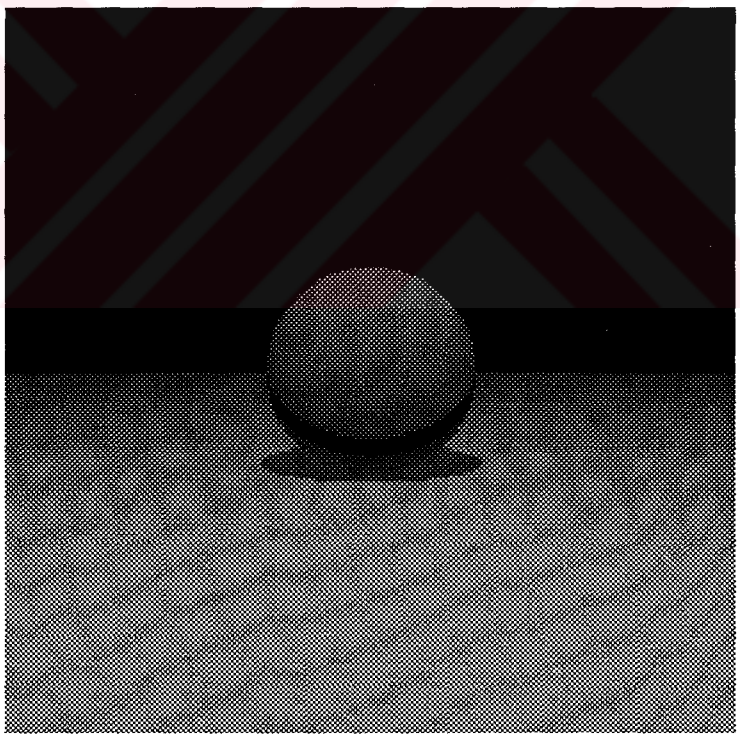
**Örnek 4**

*// Up4.TRC*

*// Bir küre*

*object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }*

```
// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 7> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 240 height 240 up <0 1 0> right <1 0 0> }
```



Şekil 13.39. Up komutuna örnek - 4

### 13.3.30. Vertices

**Tanım**

Bu komut, işlenmek üzere tanımlanan çokgenler kümesinin köşe noktalarını sisteme belirtmek için kullanılır.

**Yazım**

*object { polygonal {*

*vertices { <köşe1> <köşe2> ... <köşeN> }*

*<yüzeylerin_tanımı> } ... }*

**Açıklama**

Bu komut, sisteme bir çokgenler kümesinin köşe noktalarını tanımlamak için kullanılır. Bu komut yardımıyla poligonlar topluluğunu oluşturan bütün köşe noktalarının tanımı yapılır. İlk nokta 1 olacak şekilde sırasıyla bütün noktalar üç boyutlu koordinatlarıyla tanımlanır. Burada noktaların tanımlanma sırası önemli değildir.

**İlgili diğer komutlar**

object, polygonal, smoothness, surfaces

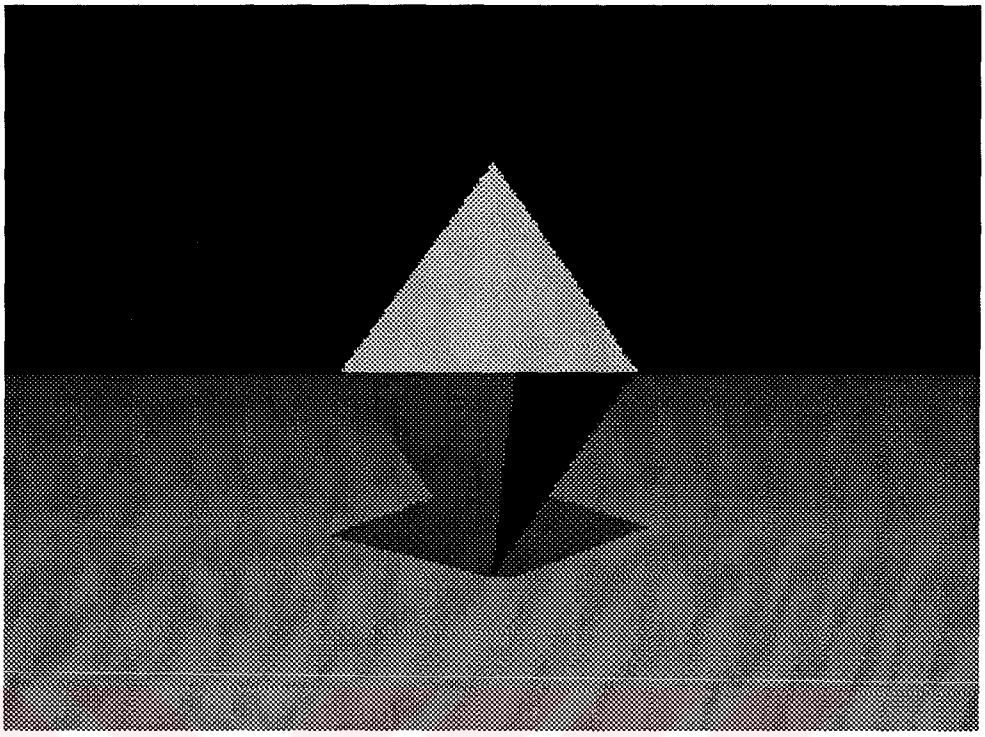
**Örnek**

*// Vertices.TRC*

*// Üçgenlerden oluşmuş bir şekil tanımı*

```
object { polygonal {
        vertices {
            < 1 0 1 >          // 1. nokta
            < 1 0 -1 >         // 2. nokta
            < 0 2 0 >          // 3. nokta
            < -1  0  1 >       // 4. nokta
            <  0 -2  0 >       // 5. nokta
            < -1  0 -1 > }     // 6. nokta
        surfaces 3 { // üç kenarlı yüzeyler
            < 1 2 3 >          // 1. yüzey
            < 1 3 4 >          // 2. yüzey
            < 6 4 3 >          // 3. yüzey
            < 3 2 6 >          // 4. yüzey
            < 4 5 1 >          // 5. yüzey
            < 2 1 5 >          // 6. yüzey
            < 6 2 5 >          // 7. yüzey
            < 5 4 6 > }        // 8. yüzey
        smoothness 0.0 }
    rotate <0 38 0>
```

```
            color  <0.85 0.85 0.09>
            diffuse 0.8
            phong 1.0 }
// Gölgelerin görünmesi için taban düzlemi
object { plane { <0 1 0> -2 } color < 0.0 0.7 0.7 > }
// Sisteme bakılacak nokta
camera { location <0 0 7> }
// Işık kaynağı tanımı
lightSource { location <0 5 5> }
// Ekran tanımı
screen { width 320 height 240 }
```



Şekil 13.40. Vertices komutuna örnek

### 13.3.31. Width

**Tanım**

Bu komut, oluşturulacak görüntü dosyasının genişliğini (x boyutu) piksel sayısı cinsinden ifade etmek için kullanılır.

**Yazım**

*screen { width <genişlik_değeri> height <yükseklik_değeri> }*

**Açıklama**

Bu komut, tanımlanan sahnenin görüntüsü oluşturulurken, diske yazılacak Targa dosyasının genişlik değerini piksel olarak belirlemek amcıyla kullanılır. Komuta parametre olarak genişlik değeri girilmelidir. Bu değer sıfırdan büyük herhangi bir tamsayı olabilir. Ancak bu değer ile height komutunun değeri arasındaki oran ile kamera tanımında bulunan up ve right komutlarının arasındaki oran aynı olması, görüntünün geçekçiliği açısından önem taşır. Örneğin "up" değeri (0, 1, 0), "right" değeri (1.333, 0, 0) olan bir kamera tanımına uygun olarak tanımlanacak "screen" komutunun parametreleri arasındaki oran 1.333 olmalıdır. Yani,

$$\frac{|right|}{|up|} = \frac{width}{height}$$

oranı korunursa, elde edilen görüntüler daha gerçekçi olur.

**İlgili diğer komutlar**

camera, height, right, screen, up

**Örnek 1**

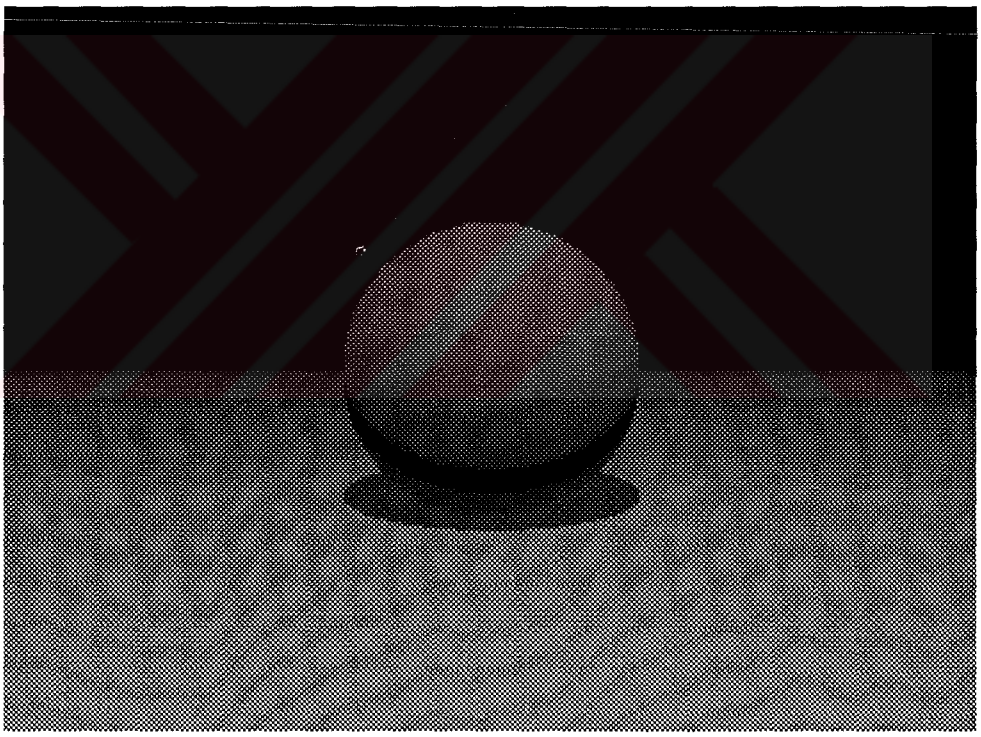
*// Width1.TRC*

*// Bir küre*

*object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }*

*// Gölgelerin görünmesi için taban düzlemi*

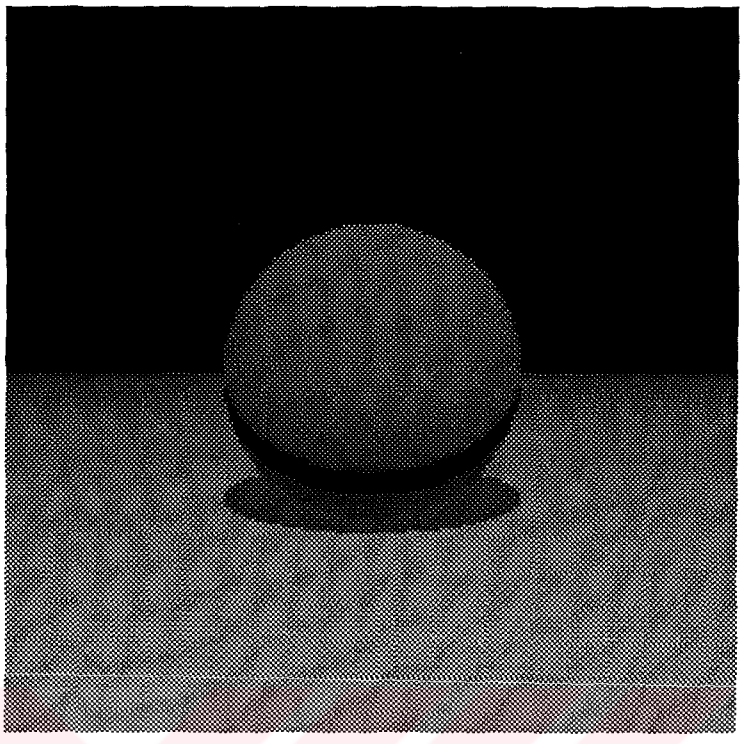
```
object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 5> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> up <0 1 0> right <1.333 0 0> }

// Ekran tanımı

screen { width 320 height 240 }
```



Şekil 13.41. Width komutuna örnek - 1

**Örnek 2**

```
// Width2.TRC

// Bir küre

object { sphere { < 0 0 0 > 1 } color <0.85 0 0> }

// Gölgelerin görünmesi için taban düzlemi

object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }

// Sisteme bakılacak nokta

camera { location <0 0 5> }

// Işık kaynağı tanımı

lightSource { location <0 5 5> }

// Ekran tanımı

screen { width 240 height 240 up <0 1 0> right <1 0 0> }
```



Şekil 13.42. Width komutuna örnek - 2

### 13.3.32. Wood

**Tanım**

Bu komut, cisim üzerinde kesilmiş ağaç yüzeyine benzeyen bir renk dağılımı oluşturur.

**Yazım**

*texture { wood color <taban_rengi> color <damar_rengi> }*

**Açıklama**

Bu komut, tanımlandığı yüzey üzerinde gerçek bir kesilmiş ağaç yüzeyi görüntüsünü simüle etmek amacıyla kullanılır. Sistem bu efekti verebilmek için özel olarak tanımlanmış bir gürültü fonksiyonu kullanır. Komuta parametre olarak iki renk değeri girilmelidir. İlk değer taban rengini oluştururken, ikinci değer de "damar" rengini belirtir.

**İlgili diğer komutlar**

colorMap, granite, marble, texture

**Örnek**

```
// Wood.TRC
// Ağaçtan kesilmiş bir kutu
object { box { < -1 -1 -1> <1 1 1> }
        rotate <0 35 0>
        ambient 0.3
        texture { wood color <0.858824 0.576471 0.439216>
                        color <0.647059 0.164706 0.164706> } }
// Gölgelerin görünmesi için taban düzlemi
object { plane { <0 1 0> -1 } color < 0.0 0.7 0.7 > }
// Sisteme bakılacak nokta
```

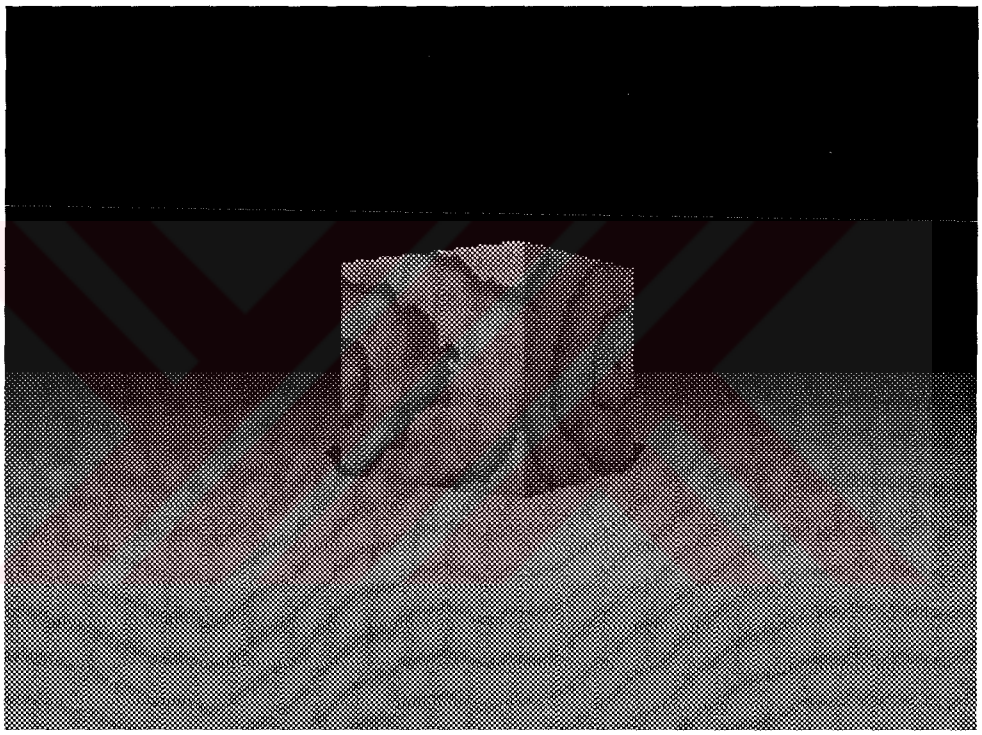
*camera { location <0 0 7> }*

*// Işık kaynağı tanımı*

*lightSource { location <0 5 5> }*

*// Ekran tanımı*

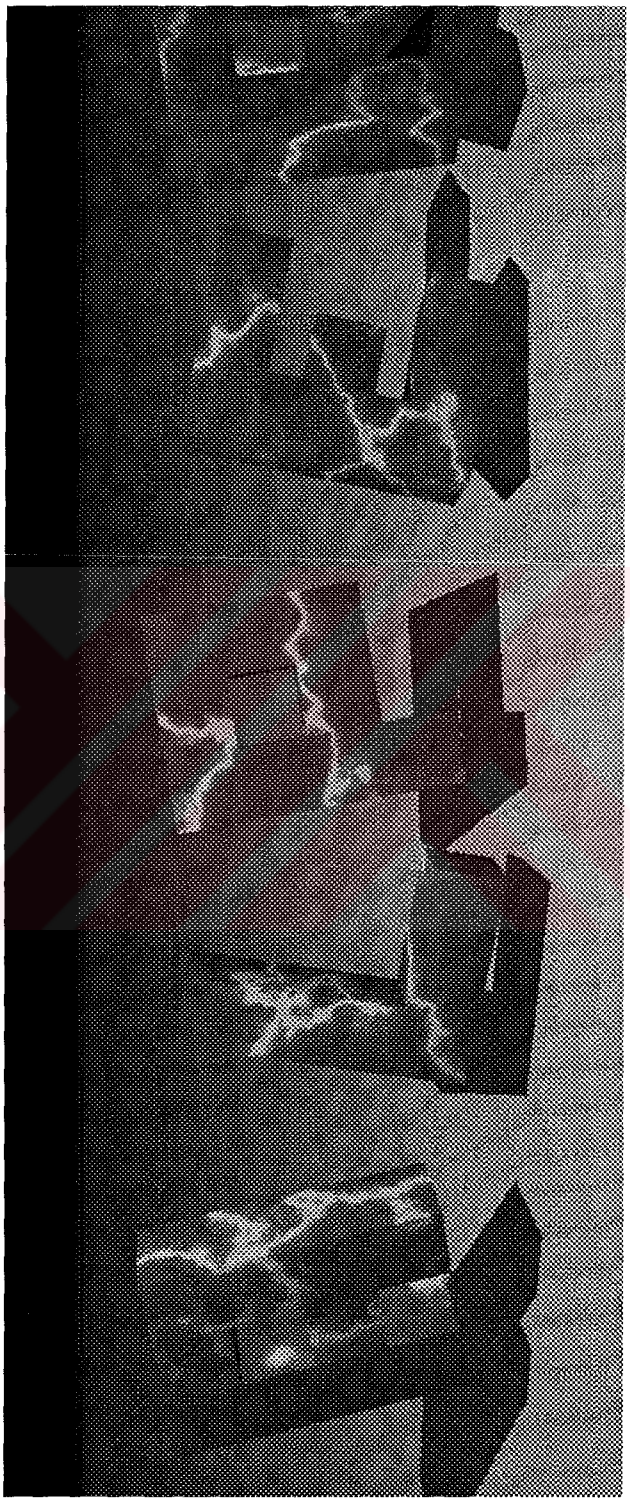
*screen { width 320 height 240 }*

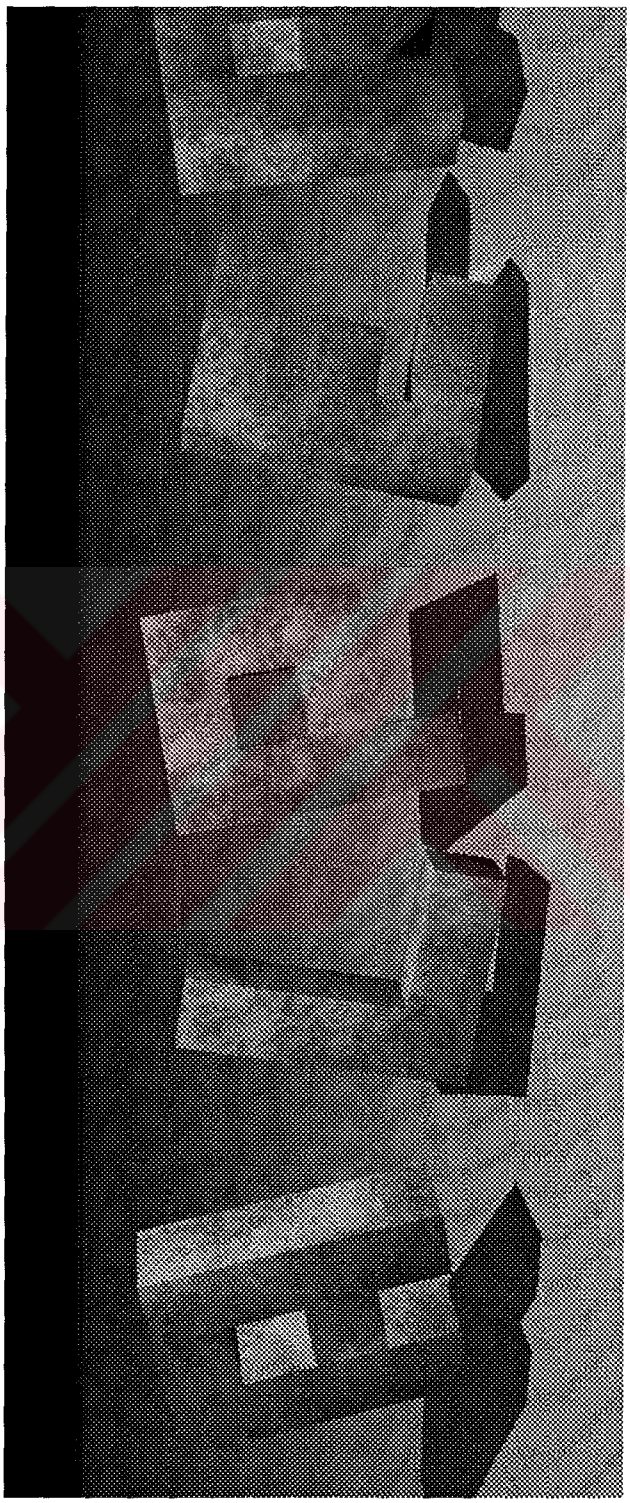


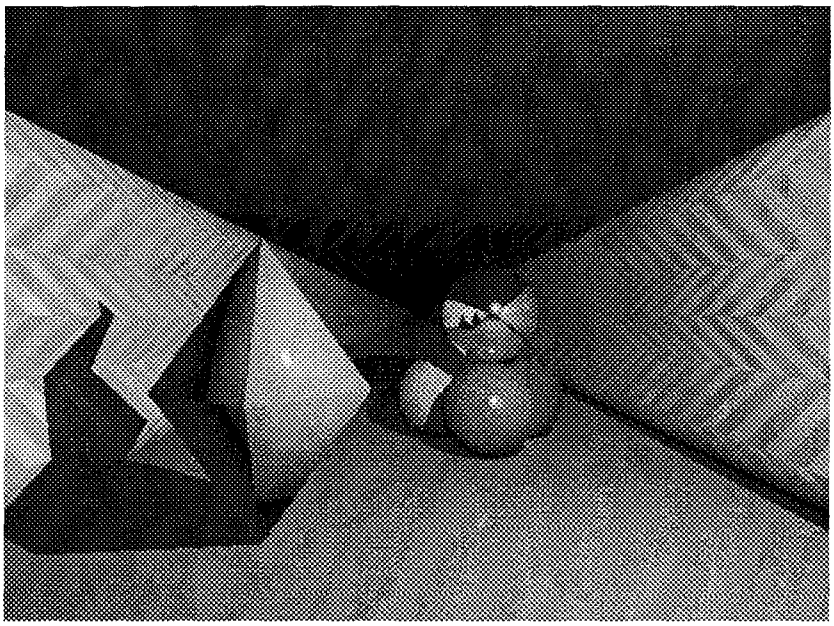
Şekil 13.43. Wood komutuna örnek

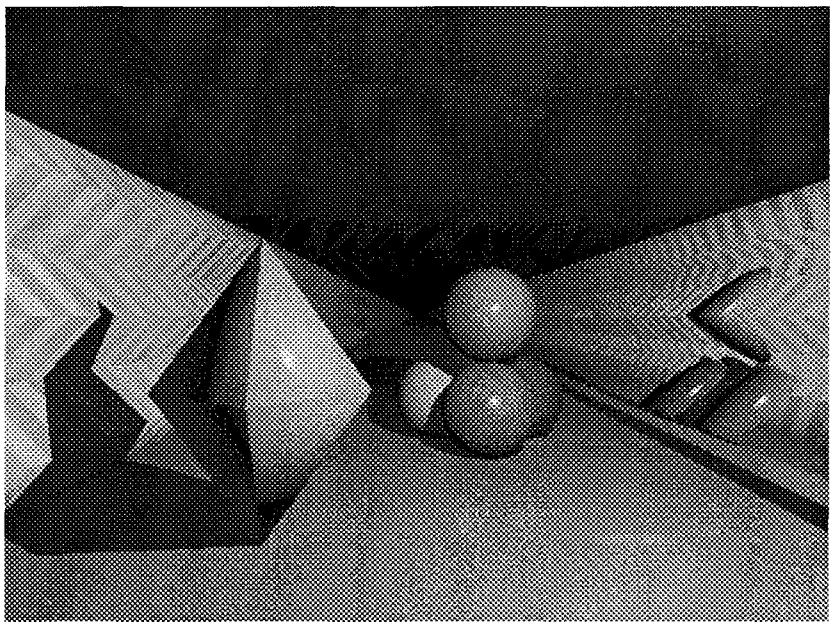
# 14. ÖRNEKLER

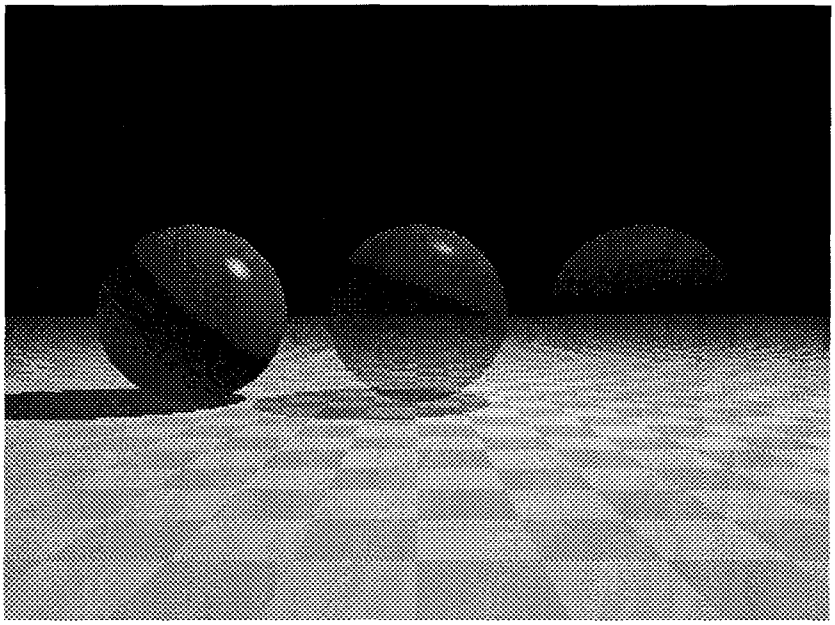
Bu bölümde, geliştirilen sistem sayesinde elde edilen bazı örnek görüntüler sergilenecektir. Bu örnekleri oluşturmak için kullanılan sahne tanıtım dosyaları bu kitap ile birlikte sunulan disket içinde bulunabilir. Aynı disket içerisine burada görüntüleri sergilenen sahnelerin orjinal versiyonları TGA ve GIF formatlarında olacak şekilde yerleştirilmiştir. Bu resimlerdeki gerçek renk dağılımlarını görebilmek için, disket üzerindeki resimler 24 bit rengi destekleyen bir grafik kartı yardımıyla incelenmelidir.

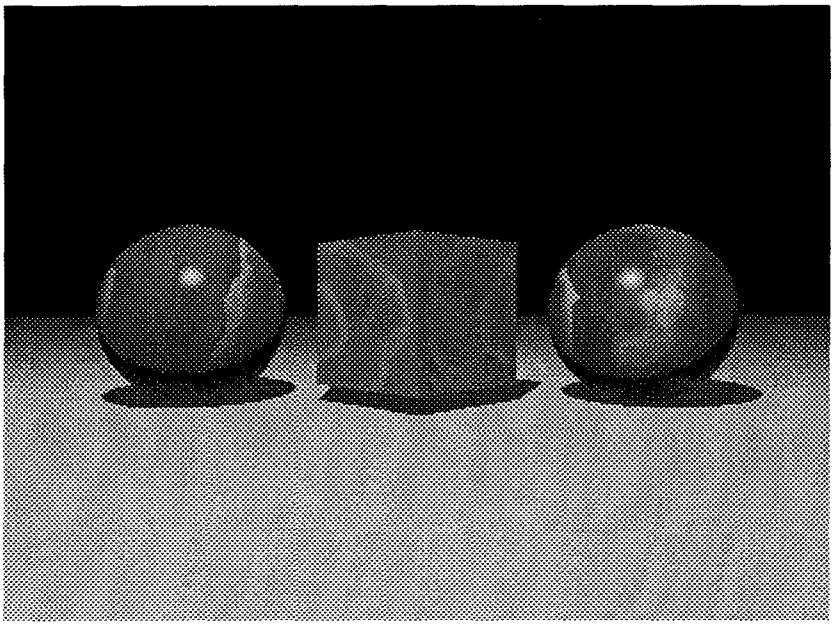


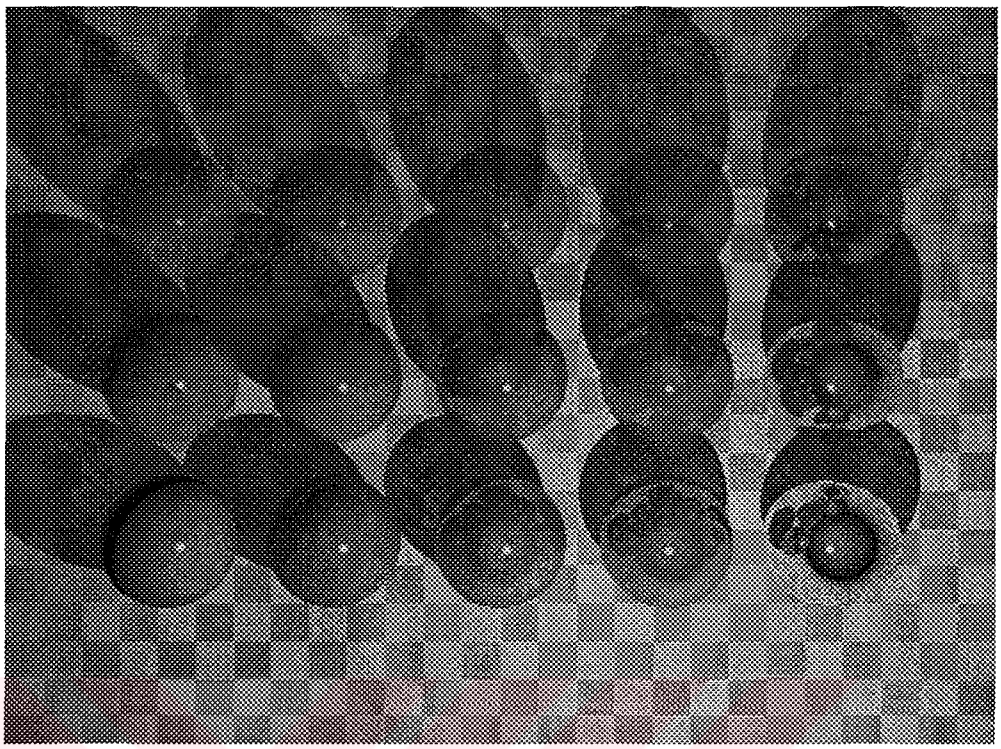


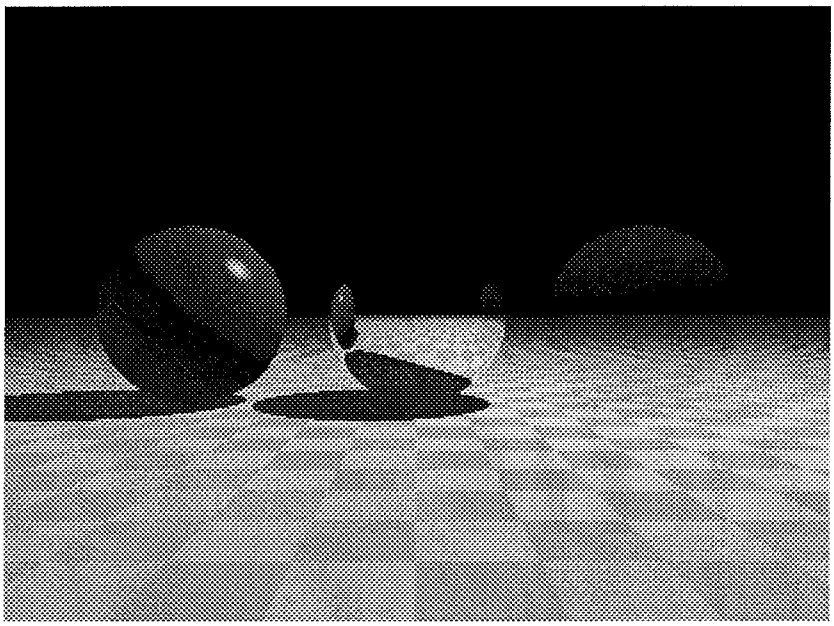


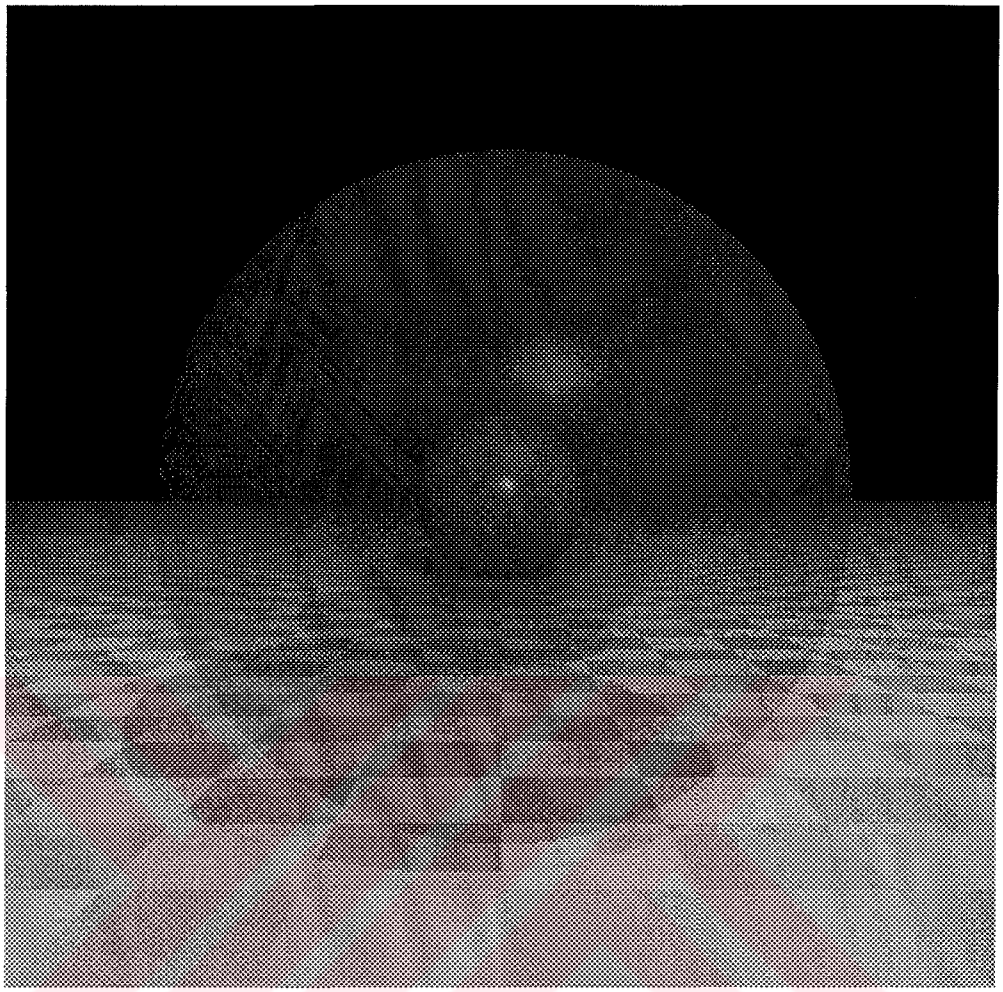


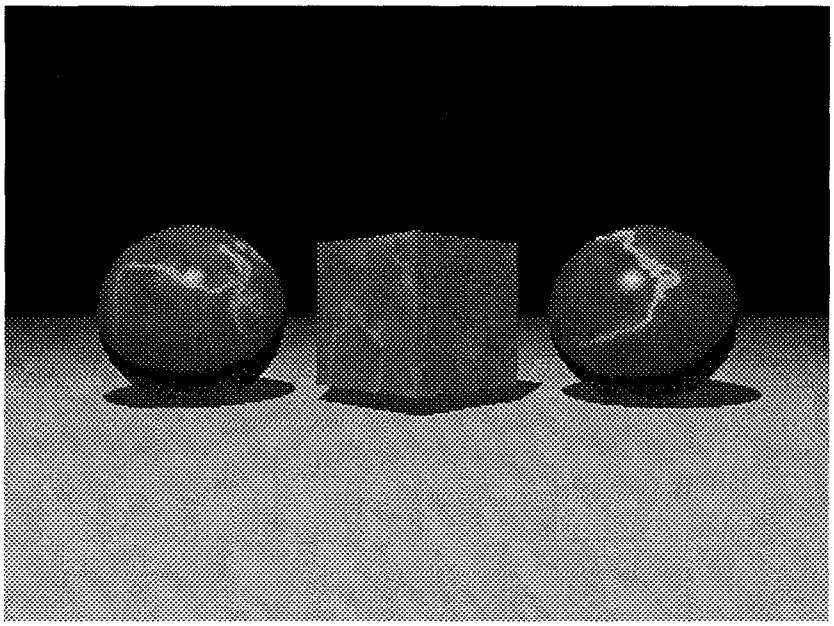


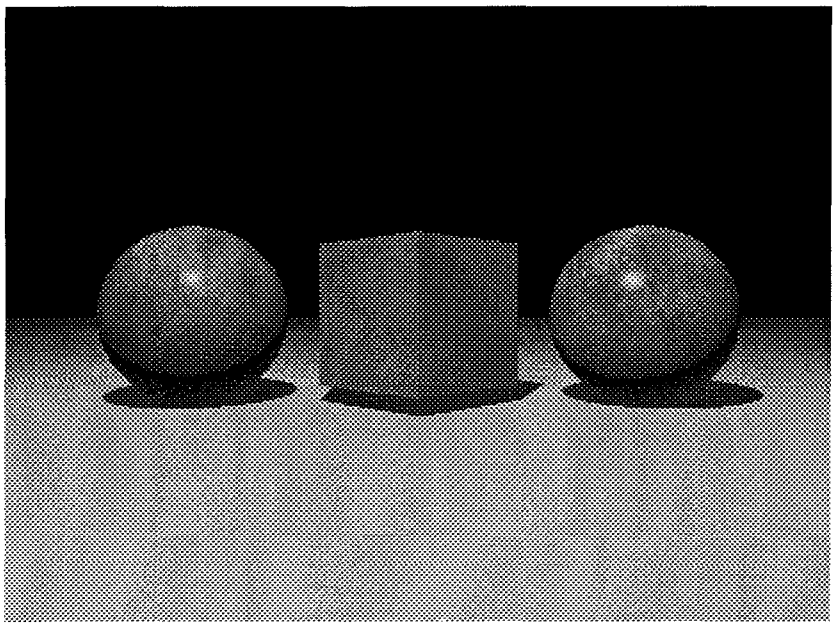












## SONUÇ

Bu çalışmada önceki bölümlerde anlatılan teorik yapılar incelenmiş ve yapılan bu incelemelerin sonuçları bir yazılım paketinde birleştirilmiştir. Böylece savunulan yöntemlerin teoride kalmaması sağlanmıştır.

Geliştirilen yazılım sistemi, kullanıcı tarafından herhangi bir ASCII yazım programında oluşturulan sahne tanıtım dosyasını analiz ederek bu dosyada tasarlanan sahnenin görüntüsünü ışın-izleme yöntemleri yardımıyla elde eder. Kullanıcı sahne tanıtım dosyasını oluştuturken, kuralları 13. Bölüm'de anlatılan oldukça basit bir dil kullanır. Hayal edilen sahne bu dil yardımıyla sisteme aktarıldıktan sonra yapılacak tek şey sistemin tasarlanan sahnenin fotoğrafını oluşturmasını beklemektir. Görüntünün oluşturulması için beklenmesi gereken süre, kullanıcının istediği detay ve kaliteyle doğru orantılıdır. Tanımlanan nesnelerin çokluğu, çokgenlerden oluşan nesnelerde çokgen sayısı, yansımalı yüzeylerin sayısı ve yansımalarda istenilen detay , resmin büyüklüğü süreyi etkileyen faktörlere örnek olarak verilebilir.

Sistem geliştirilirken nesne yönelimli bir yapı ve bu amaç için tasarlanmış bir dil olan C++ yazılım geliştirme dili kullanılmıştır. Bu da sisteme yeni bir özellik tanımlama işini oldukça kolaylaştırmaktadır. Örneğin yeni bir nesne tipi tanımlamak için, sadece o nesneyi ifade edecek parametreleri (küre için merkez ve yarıçap bilgisi, düzlem için taban noktası ve yüzey normali, çokgenler için köşe noktaları ve yüzey bilgisi, gibi) belirlemek ve bu nesnenin herhangi bir üç boyutlu ışın ile kesişimini bulan bir yöntem tanımlamak yeterli olacaktır. Sistem bundan öncesini ve sonrasını kendi kendine gerçekleştirecektir.

Sistem, kitap yazıldığı andaki haliyle, teorik olarak sonsuz sayıda nesne tanımını destekleyebilecek kapasitededir. Bu, sistemin çokgenleri görüntüleme özelliğinden kaynaklanmaktadır. Yaşadığımız ortamda bulunan ve küre, düzlem, koni gibi genel bir yapıyla modellenemeyen her tür yapı, çokgenler yardımıyla kolaylıkla modellenebilir. Daha sonra bu çokgen topluluğu sisteme veri olarak girildiği taktirde, nesnenin ekran

üzerindeki görüntüsü kolaylıkla elde edilebilir. Dolayısıyla bu sistem üç boyutlu modelleme ihtiyacı duyan birçok uygulama alanında rahatlıkla kullanılabilir.

Veri olarak sahne tanıtım dosyalarını kabul eden sistem, çıkış olarak 24 bitlik gerçek renk özelliği içeren bir TARGA dosyası oluşturur. Bu dosya 24 bit desteği veren herhangi bir grafik ortamda işletim sisteminden bağımsız olarak rahatlıkla kullanılabilir. Dolayısıyla örneğin kişisel bilgisayarınızda bu sistem yardımıyla oluşturduğunuz görüntü dosyasını UNIX ortamında rahatlıkla inceleyebilirsiniz. İşletim sisteminden bağımsız görüntü oluşturma özelliği, sistemden birçok platformda yararlanılabilmesini sağlamaktadır.

## KAYNAKLAR

1. Anderson et al, 1994. PC Graphics Unleashed. SAMS Publishing, USA.

2. Glassner, S. A., 1990. Graphics Gems, Academic Press Inc., USA.

3. Watt, A., 1996. 3D Computer Graphics. Addison-Wesley, England.

4. Watt, A., and Watt, M., 1995. Advanced Animation and Rendering Techniques. Addison-Wesley, ACM Press, England.

## ÖZGEÇMİŞ

| | |
|---|---|
| **Adı/Soyadı** | Alp-er Yurdakul |
| **Doğum Tarihi** | 02 Şubat 1972 |
| **Doğum Yeri** | Ankara |
| **Bitirdiği Okullar** | Bilgisayar Bilimleri ve Mühendisliği Bölümü, Lisans Programı, Yıldız Teknik Üniversitesi, İstanbul, 1993.<br>Kadıköy Anadolu Lisesi, İstanbul, 1989. |
| **Verdiği Seminerler** | VM/CMS ve VSE/ESA sistemleri tanıtımı, Yıldız Teknik Üniversitesi, İstanbul 1993.<br>Paralel Programlama Dilleri / Derleyicileri ve JADE dili tanıtımı, Yıldız Teknik Üniversitesi, İstanbul 1994. |
| **Gerçekleştirdiği Projeler** | İstatistiksel verilerin haritalar üzerinde gösterilimi ve 3-boyut simülasyonu, 1993.<br>Sayfalama (paging) mantığıyla çalışan bir işletim sistemi modeli tasarımı ve gerçeklenmesi, 1992.<br>Sayısal metod simülasyonu sağlayan bir sistem tasarımı, 1991. |