KARADENİZ TEKNİK ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

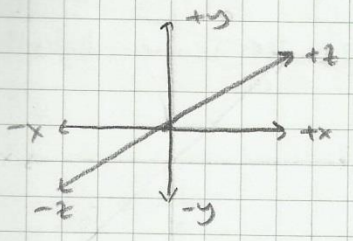
BIL 320 BİLGİSAYAR GRAFİKLERİ-I DERS NOTLARI

2012-2013 Bahar Dönemi

Üç boyutlu gerçekçi görüntüler elde edebilmek için kullanılan yönteme ray tracing denir.

1. Rotation (Dönme)
2. Translation (Öteleme)
3. Scaling (ölçekleme)

RAY TRACING (IŞIN İZLEME)



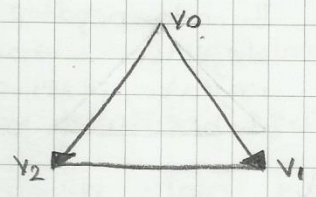
Bu koordinat sistemi kullanılacak.

Herhangi bir vektörün boyunu 1 br yapma işlemine <u>normalizasyon</u> denir.

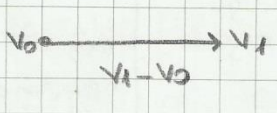
X : Vektörel çarpımın gösterimi (Cross Product)

* : Skaler çarpımın gösterimi (Dot Product)

- Vektörel çarpım yüzey normalini hesaplamak için kullanılır.
- Yüzey denklemi; $Ax+By+Cz+D=0$ dır. Bunun yüzey normali $V(A,B,C)$ dir.
- Herhangi bir üçgenin normali hesaplanırken iki farklı fark denklemi kullanılır.



$$N = (V_1 - V_0) \times (V_2 - V_0)$$



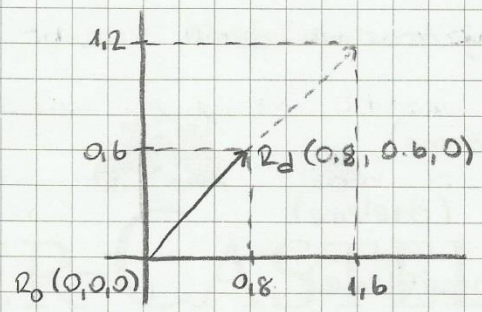
- Skaler çarpım yapılacak vektörler birim vektörse sonuç bize iki vektör arasındaki açının cosünüsünü verir.

$$R_1 * R_2 = R_{1x}R_{2x} + R_{1y}R_{2y} + R_{1z}R_{2z} = |R_1| * |R_2| * \cos\beta$$

Işın bir vektör olduğu için bir başlangıç ve bitiş noktası olmalıdır

$R = R_o + t R_d$

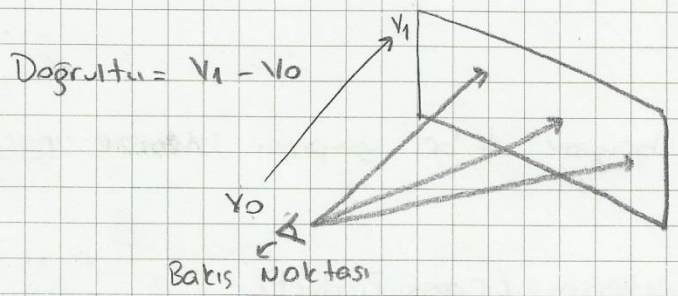
- R_o: Origin (başlangıç noktası)
- R_d: Direction (doğrultu)
- t: Çizgiye veya küreye olan uzaklık

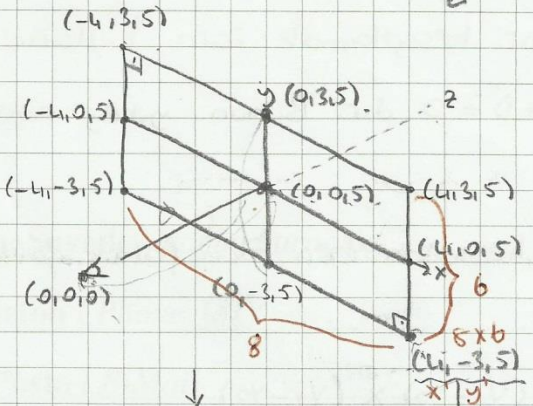


$R = (0,0,0) + 2(0,8, 0,6, 0)$

$R = (1,6, 1,2, 0)$

Doğrultu = pixel - bakış noktasının koordinatı nın normalizesi ile ışının doğrultusu hesaplanır.



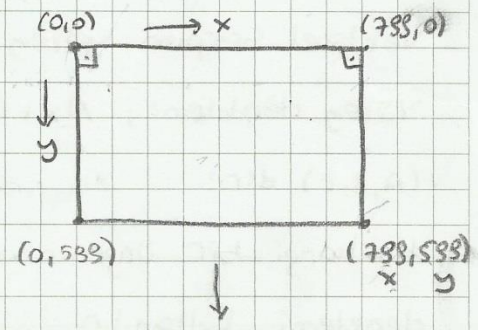


Burada 3 boyutlu olduğu için x, y değerleri negatif alınabilir.

Böyle hesaplanır

$x' = (8 * x / 788) - 4$

$y' = 3 - (y * 6 / 588)$

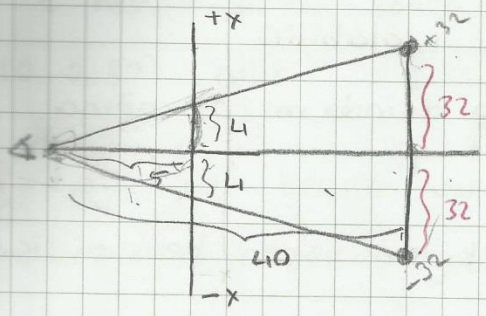


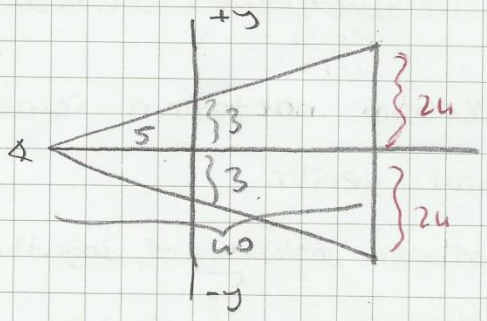
Burada negatif x, y değerleri alınamaz. Buradaki pixel değerleri 1 br ilerler yani int dir

Görüntü Düzlemi

Burada pixeller 0,01 br olarak ilerler. Yani float 'dır

Yukarıdaki ilk şekildeki eğik x eksenini düz olarak düşündüğümüzde, bakılan noktadan 40 br uzaklıktaki bir üçgeni yukarıdaki görüntü düzlemine sığdırabileceğimiz max boyutu bulabilmek için benzer üçgenleri kullanınız.



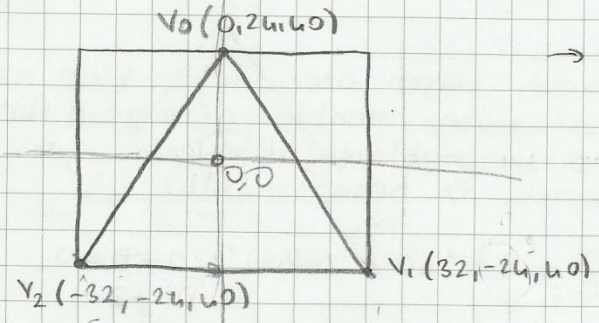


$\frac{5}{40} = \frac{4}{x}$

$x = 32$ hesaplanır.

$\frac{5}{40} = \frac{3}{y}$

$y = 24$ hesaplanır.



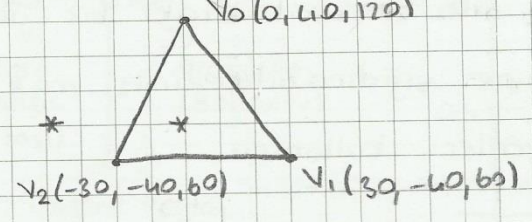
→ Şekildeki gibi üçgen görüntü düzlemine bu boyutta yerleşir.

Sanal olarak oluşturulan görüntü düzlemi sayesinde koddaki x,y koordinat değerlerinin değişmesiyle üçgen boyutu da buna bağlı olarak kolaylıkla değişir.

Işın - Üçgen Kesişim Testi

3D cisimler çoğunlukla üçgenler ile temsil edilirler. Burada anlatılacak olan ışın-üçgen kesişim testi iki aşamadan oluşmaktadır.

1) Işın ile üçgenin yüzey arasında kesişim test:



→ Elde edilecek kesişim noktası üçgenin içinde de olabilir dışında da olabilir.

2) Kesişim noktasının içindemi yoksa dışında mı olacağına karar verilir.

* Başlangıç noktası ve doğrultusunu bilmek ışın üçgen kesişimi için yeterlidir.

$$R_o + t.R_d$$

→ Işının üçgen ile kesişimini bulmak için yüzey denklemini çıkarmak gerekir.

$$Ax + By + Cz + D = 0$$
$$N(A,B,C) = (0,\ 0.6,\ -0.8)$$

Buradaki üçgen üzerindeki köşe noktaları yüzey denklemini sağlar. Herhangi bir noktanın değerlerini yerine koyarak D hesaplanabilir.

$$0*0 + 0.6*40 + (-0.8)*120 + D = 0$$
$$D = 72$$

$$\boxed{0,6y - 0,8z + 72 = 0}$$ → Yüzey Denklemi

→ Işın yüzeyle kesişiyorsa kesişim noktasındaki ışının x, y, z değerleri yüzey denklemini sağlar.

$$A(R_{ox} + t.R_{dx}) + B(R_{oy} + t.R_{dy}) + C(R_{oz} + t.R_{dz}) + D = 0$$

Amacımız t'yi bulmak.

$$t = -\frac{AR_{ox} + BR_{oy} + CR_{oz} + D}{AR_{dx} + BR_{dy} + CR_{dz}} = -\frac{N * R_o + D}{N * R_d}$$

→ $R_0 = (0,0,0)$, $R_d = (0,0,1)$ olarak alınırsa;

$$t = -\frac{(0, 0.6, -0.8) * (0,0,0) + 72}{(0, 0.6, -0.8) * (0,0,1)} = \frac{-72}{-0.8} = 90$$

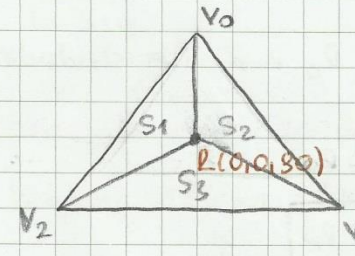
t hesaplandıktan sonra yüzey koordinatları hesaplayabiliriz.
$(0, 0, 80)$ gelir.

✓ $t > 0$ ise kesişim var demektir.

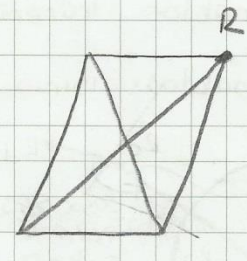
$t < 0$ ise kesişim yok demektir.

$t = \infty$ ise yani $N * R_d = 0$ ise ışın yüzeye paralel demektir.

Buraya kadar 1. aşama sonlanır. 2. aşamada da kesişim noktasının nerede olduğu bulunur.



$S = S_1 + S_2 + S_3$ ise kesişim noktası üçgenin içindedir.



$S_1 + S_2 + S_3 > S$ ise nokta üçgenin dışında demektir.

Unutmaaaa yoksa normal olur =)

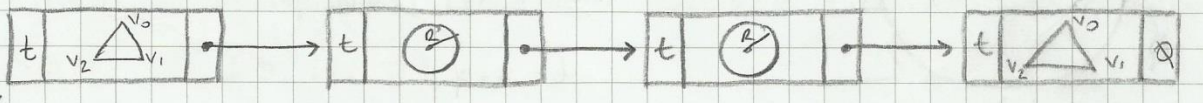
$$S_1 = |(R - V_0) \times (V_2 - V_0)|$$ → 2 boyu

$$S_2 = |(R - V_0) \times (V_1 - V_0)|$$

$$S_3 = |(R - V_1) \times (V_2 - V_1)|$$

$$S = |(V_1 - V_0) \times (V_2 - V_0)|$$

$R = R_0 + t * R_d$ → Işının yüzey üzerindeki koordinatlarını hesaplar.



↳ Herbir pixel için kesişim sayısı değişebileceği için dinamiklik sağlamak amacıyla bağlı liste tutulur.

→ Ard arda gelen cisimlerden en yakın olanı ekrana nasıl çizilebilir?

Görüntü düzleminde geçen ışınlardan herhangi biri gittiği doğrultu boyunca 1'den fazla cisimle kesişiyorsa ekrana bunlardan görüntü düzlemine en yakın olanı çizilmelidir. Bunun için kesişim testleri ile hesaplanan t uzaklıkları sıralanır ve en küçük t uzaklığına sahip cismin görüntüsü çizilir. Diğer cisimler görünmeyen yüzey olarak adlandırılırlar.

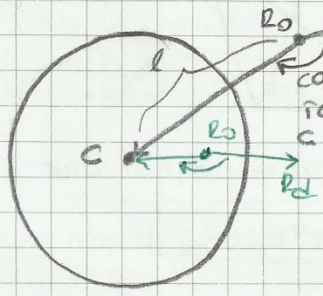
Işın - Küre Kesişim Testi

Işın küre kesişim testi yapılırken 3 durum göz önüne alınmalıdır:

(a) Küre ışının gerisinde kaldığı için kesişim yoktur.

(b) Kesişim noktası kürenin içindedir

(c) Kesişim noktası ışının başlangıç noktası R_o'nun ilerisindedir.



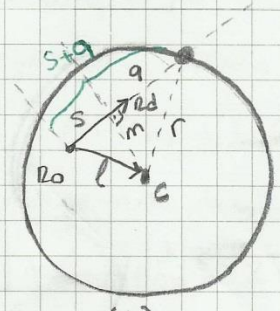
cosα > 90 olduğu için l negatif çıkar.

(a)

$l = c - R_o$

$\boxed{s = l * R_d}$

* $s < 0$ ise küre ışının arkasında kalır. s kürenin içindeyken de negatif çıkabilir. Bu durumda bir test daha yapmak gerekir. ($l^2 < r^2$)

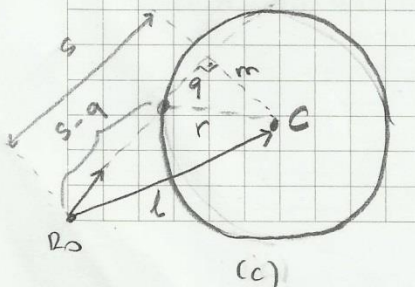


(b)

$l^2 = l * l$

$l^2 = |l| . |l| . \cos\beta$

$t = s + q$



(c)

$l^2 = l^2 > r^2$

$t = s - q$

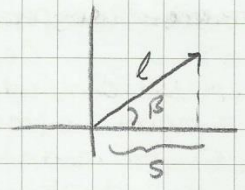
$q = \sqrt{r^2 - m^2}$

S aslında ℓ vektörünün yatay bileşenidir.

$S = \ell * R_d$

$= |\ell| . |R_d| . \cos\beta$

$= |\ell| . \cos\beta$



→ Şekil (b) için:

$m^2 = \ell^2 - s^2$ hesaplanır. m^2 ile ℓ^2 karşılaştırılır.

✓ $m^2 > \ell^2$ ise kesişim yoktur.

✓ $m^2 < \ell^2$ ise kesişim vardır.

$q = \sqrt{r^2 - m^2}$ → Kesişimin içeriden mi yoksa dışarıdan mı olduğunu anlamak için hesaplanır.

$t = s - q$

$t = s + q$

06.03.2013

Phong Boyama Modeli:

Renkler Kırmızı, Yeşil ve Mavi ana renklerinden oluşturulur. C# da ilgili renkleri basarken bir color değişkeni tutulmuştur. Bu değişken bu 3 renkten oluşturulur.

R → 0-255
G → 0-255
B → 0-255

} 24 bit

$2^{24} = 2^{20} * 2^4$

$= 1MB * 2^4 = \underline{16MB}$

Kırmızı → (255, 0, 0) → Bu değer düştükçe kırmızının tonu açılır.

Yeşil → (0, 255, 0)

Mavi → (0, 0, 255)

Beyaz → (255, 255, 255)

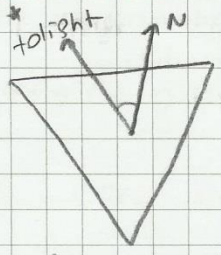
Siyah → (0, 0, 0)

Phong boyama modelinin bileşenleri:

$0{,}2 * \text{Ambient}_{(R,G,B)} + 0{,}5 * \text{diffuce renk}_{(R,G,B)} + 0{,}3 * \text{specular}_{(R,G,B)} = \text{Phong Boyama}$ (renk)

↳ Toplamlar 255 sınırını aşmasın diye belirli kat sayılarla çarpılırlar. (Katsayılar toplamı 1 olmalıdır)

Diffuce Renk



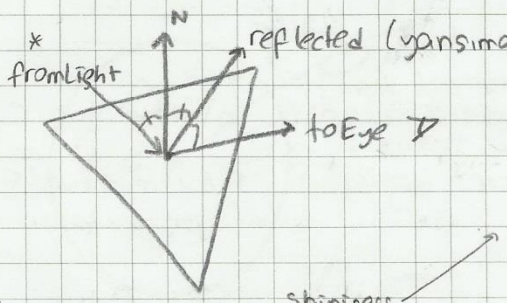
toLight * N = diffuce katsayı

diffuce renk = (toLight * N) (255,0,0)

(diffuce katsayı; toLight * N = cosθ)

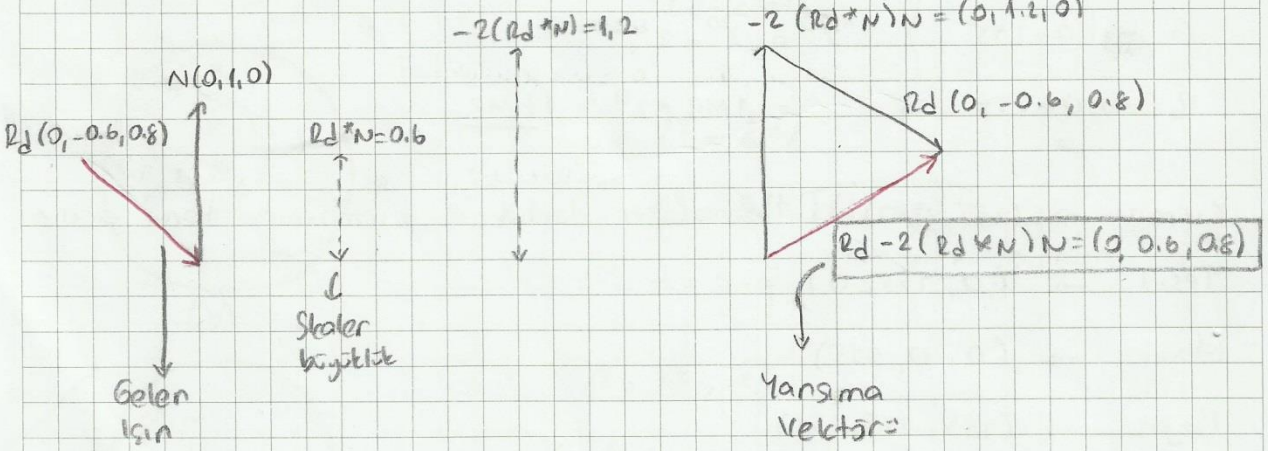
Specular Renk

Specular Renk: Işık kaynağının cisim üzerinden yansıyıp gözümüze gelen ışıktır.



Değer büyüdükçe yansıma boyutu küçülür.

(reflected * toEye)



Ambient Bileşen

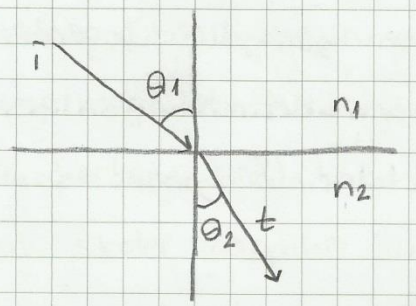
Işık kaynağından doğrudan ışın almayan noktalar vardır. Buralara duvarlardan veya eşyalardan yansıyan ışıklardan dolayı az da olsa aydınlatılmaktadır. Phong boyama sonucu oluşan renk katsayıları sıfır alınsa bile ambient bileşenden dolayı (ambient bileşen 0--1 arasında bir katsayı ile cismin kendi renginin çarpımı sonucu oluşur) aydınlanma olur.

Farklı kaynaklardan yansıyan ışınlar için ray tracing in bir çözümü yoktur. Bunun için Photon Mapping yöntemi vardır.

Yansıma (Reflection)

Yüzey normaline bağlı olarak yansıma doğrultusuna bakılır. Yansıma doğrultusu hesaplandıktan sonra yeni doğrultu boyunca yollanan ışın ile cisimler üzerinde kesişim testleri yapılarak en yakın cisim belirlenir. Onun rengi 0..1 arası bir katsayı ile çarpılarak yansıma ile görülen cismin rengi hesaplanır. Cisim saydam ise transmitted direction l_d olarak alınıp aynı işlemler tekrarlanır.

Refractions (Kırılma) → Sınavda çıkmayabilir



$$\frac{n_1}{n_2} = \frac{\sin\theta_2}{\sin\theta_1}$$

$r = \frac{n_1}{n_2}$ → Kırılma indislerinin oranı

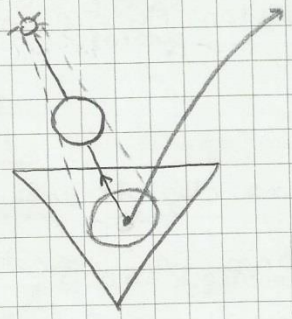
$t = ri + (w - k)n$

$w = -(i * n)r$

$$k = \sqrt{1 + (w-r)(w+r)}$$

Shadows (Gölge Testi)

Herhangi bir yüzeyin başka bir yüzeyin gölgesinde kalıp kalmadığını belirlemek için yüzey üzerindeki kesişim noktasından ışık kaynağına doğru gölge test etme ışını yollanır. Bu ışın ile üçgenler/küreler arasında kesişim testleri yapılarak en yakın üçgen/küre belirlenir. Hesaplanan uzaklık değeri ışık kaynağına olan uzaklıktan küçük ise yüzey bu kürenin/üçgenin gölgesinde kalıyor demektir.

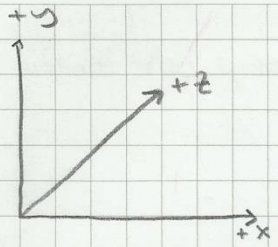


Bu noktanın başka bir cismin gölgesinde kalıp kalmadığına bakmak için o noktadan cisme bir ışın gönderilir. Sonrasında ışık kaynağına ray tracing koşar. Ortamdaki cisimlere göre yakınlık testi yapılıp en yakın olana bakılır.

En yakın cisim < Işık kaynağı ile kesişim noktası arasındaki fark ise True döner.

Arkayüz Kaldırma (Backface Culling)

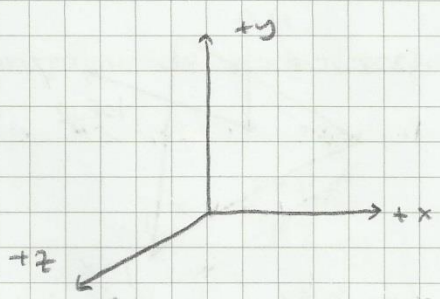
Arkayüz olan yüzey bakış noktası ile arasında başka yüzeyler olmasa bile ters durduğu için görünmeyen yüzeydir. Üçgenlerin köşe noktalarına bağlı olarak yüzey normallerinin yönleri değişebilir. Genellikle üçgenlerin köşe noktalarının yönünü saat yönü olarak alırız.



Sol El Kuralı

Direct-X

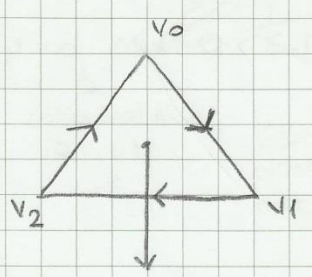
↳ CW (clock wise)
Saat yönünde alınır.



Sağ El Kuralı

Oper GL

↳ CCW (Counter clock Wise)
Saat yönünün tersinde alınır.



→ Eğer normalin bize bakmasını istersek köşelerin yönleri saat yönünde olmalıdır.

Eğer normal bize bakıyorsa biz o cismi görüyoruz demektir. Eğer normal bize bakmıyorsa o cismi göremeyiz.

fromL − 2 (fromL * N) N → Yansıma vektörü

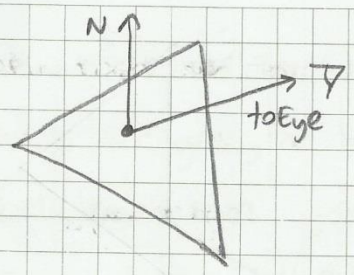
Diffuse katsayısı: Yüzey normali ile bakış noktasına doğru olan vektörlerin skaler çarpımı ile bulunur.

Arkayüz kaldırmada iki yöntem kullanılır.

Skaler Çarpım ile Arkayüz Kaldırma:

Herhangi bir yüzeyin arkayüz olup olmadığını belirlemek için o yüzey üzerindeki herhangi bir noktadan (üçgen için köşe noktalarından herhangi biri) bakış noktasına doğru olan vektör ile yüzey normali skaler çarpılır. Sonuç sıfırdan küçükse bu yüzey arkayüzdür.

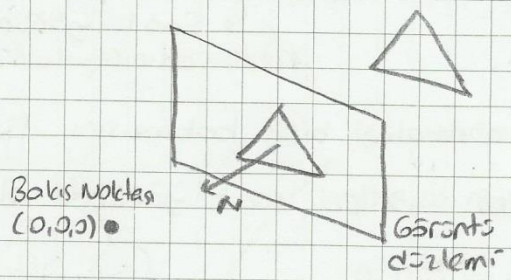
Üçgenin köşelerinden birinden bakış noktasına doğru olan vektör ile normalin skaler çarpımı yapılır. Sonuç <0 ise backface dir.



if (N · toEye < 0) then backface

Vektörel Çarpım ile Arkayüz Kaldırma

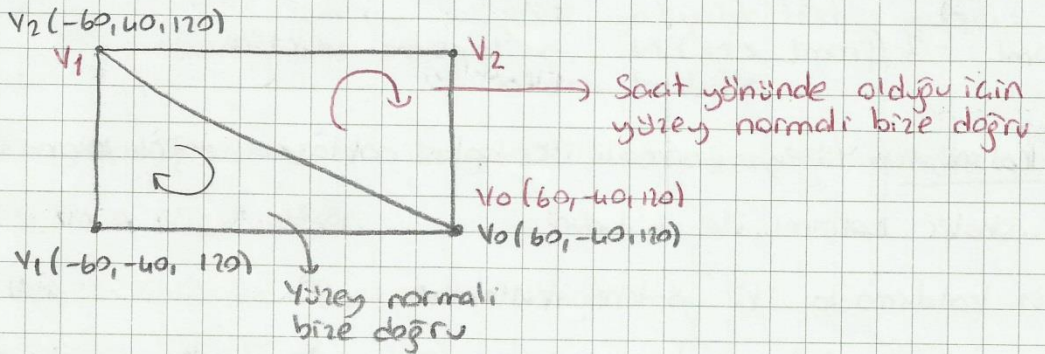
Üçgenin görüntü düzlemine izdüşümü alınır ve normali hesaplanır. Eğer normalin z bileşeni 0'dan büyükse arkayüzdür.



if N.z > 0 then backface

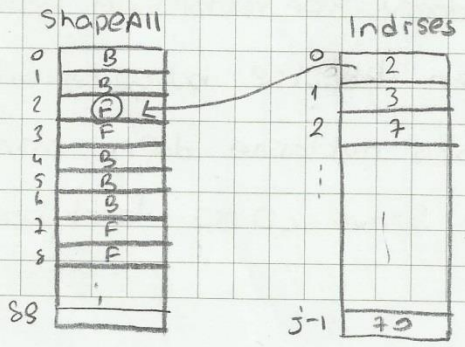
→ Skaler Çarpım ile Backface

Yüzey normali ile bakış noktasına olan vektör arasındaki açı 90°'yi geçtiğinde biz o yüzeyi göremeyiz yani backface olur.



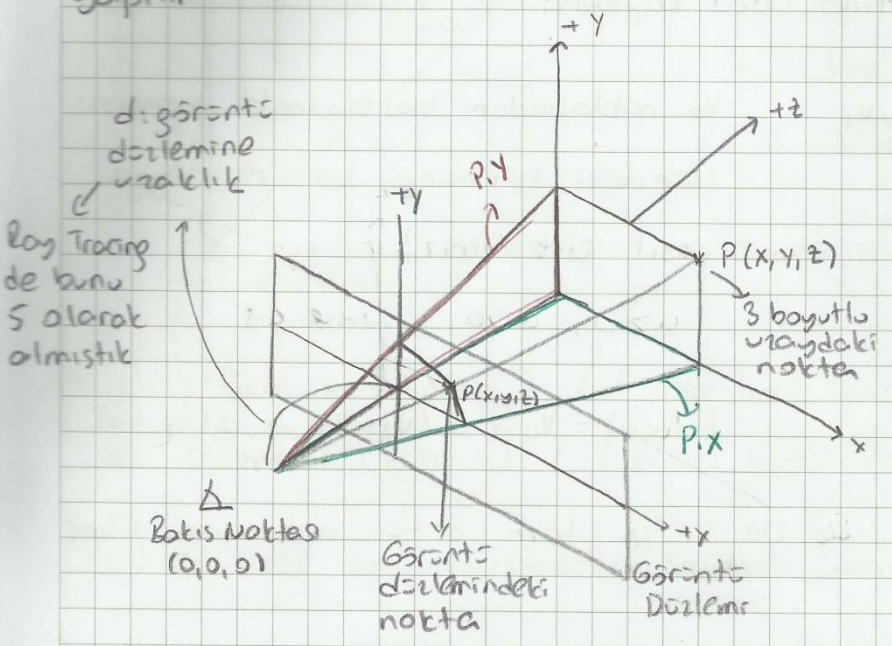
* Diffuse bileşende bakış noktasını dikkate almıyoruz.

Bakış noktasına olan vektörü hesaplamanın en kolay yolu köşe noktalarının herhangi birinden hesaplamaktır.



Vektörel Çarpım ile Backface

Görüntü düzlemine perspective izdüşüm alınır ve normalizasyon yapılır.



$$P'.x = \frac{d * P.X}{P.Z}$$

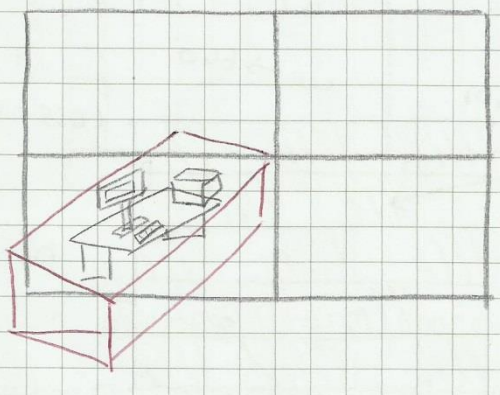
$$\frac{d}{P.Z} = \frac{P'.X}{P.X}$$

$$P'.X = \frac{d * P.X}{P.Z}$$

$$P'.y = \frac{d * P.Y}{P.Z}$$

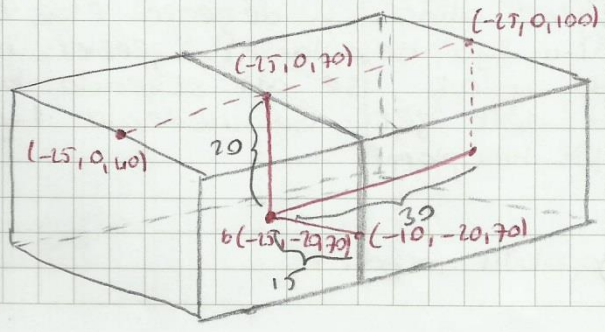
$$P'.z = \frac{d * P.Z}{P.Z} = d$$

P'.z nin d'ye eşit olduğunu ispatlamak için yazılmıştır.

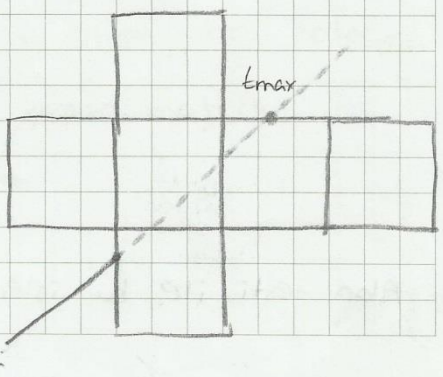


→ Poligonal yoğunluk olan kısmı daha basit bir cisme alıyoruz. Sanal bir dikdörtgen prizma tanımlanır. Bunun özelliği o yoğunluğu içine almaktır. Eğer ışınımız dikdörtgen prizma ile kesişiyorsa prizma içinde kalan cisimlerle olan kesişim testi o zaman yapılır.

Axis Aligned Bounding Box (AABB)

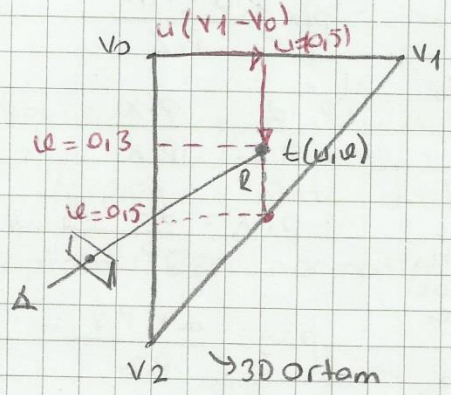


→



Doku Kaplama

Düzlemsel Yüzey Üzerine Doku Kaplama



V_0 noktasından başlayarak üçgenin içindeki herhangi bir noktaya nasıl gidebiliriz?

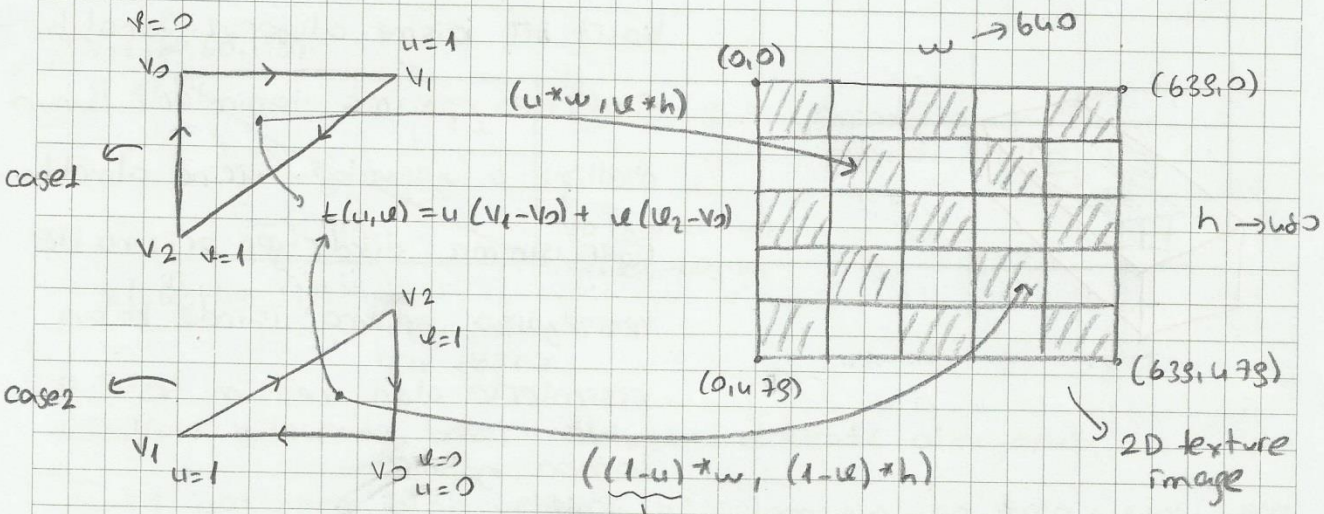
$u \geq 0, \quad v \geq 0, \quad u+v \leq 1$

$$t(u,v) = V_0 + u(V_1 - V_0) + v(V_2 - V_0)$$

* Mesela $u=0,5$ için V_0 ile V_1 in tam orta noktasına gitmiş oluruz.

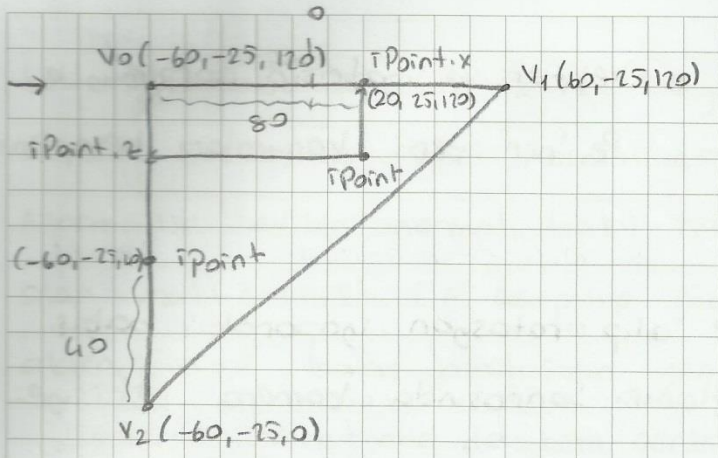
→ $u+v > 1$ olduğunda üçgenin dışına çıkarız.

→ $t(u,v)$ noktasının kesişim testinin bize döndürdüğünü varsayıyoruz.



Üst üçgende alınan u değerinin izdüşümünü alt üçgene alınca u değerini elde etmek için 1-u 'yu hesaplamak gerekir. Üst üçgende $u=0,3$ ise alt üçgende bu 0,7 ye denk gelir. Bu nedenle 1-u ile 0,7 hesaplanır.

Alan testi ile bu işlemleri yapabilir miydik?



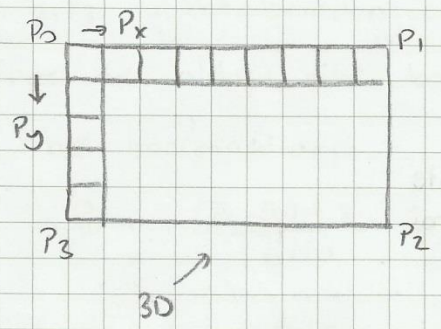
Zemin kaplamada y değerini kullanmadığımız için koordinat bileşenleri x ve z'dir. y değeri zemin boyunca sabittir. u için x'i, v için z'yi kullanırız.

$u = (iPoint.x - S.V_0.x) / (S.V_1.x - S.V_0.x) = 80/120 = 0,67$

$v = (iPoint.z - S.V_0.z) / (S.V_2.z - S.V_0.z) = -80/-120 = 0,67$

Küre üzerine doku kaplamadan sınavda sorumlu değiliz.

Generate Rays



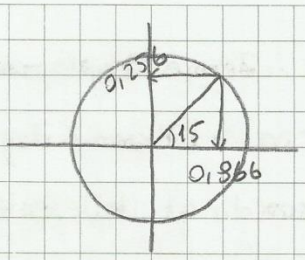
$P_x = (P_1 - P_0) / 639$

$P_y = (P_3 - P_0) / 479$

Her dönmede köşe koordinatları ve pixel değerleri yeniden hesaplanır. Başlangıçta pixel koordinatı sabittir. 3 boyutlu ortamda ilerlerken tüm pixel koordinatları değişir. Bunu hesaplamak için tüm pixeli değilde köşe noktalarına göre işlem yapılır.

$$[Pe.x \quad Pe.y \quad Pe.z] \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

→ y ekseninde rotasyon yapan matris

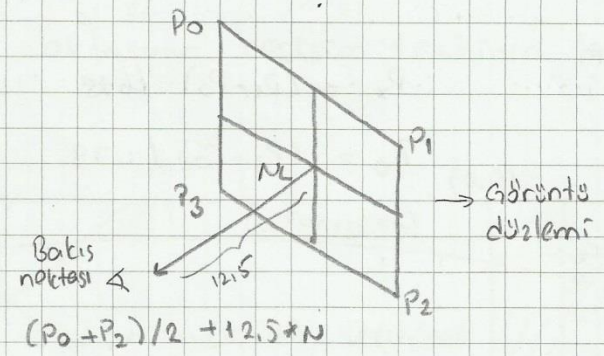


*Vektör ve matrisin çarpımıyla Pe'nin yeni konumları hesaplanır.

Köşe noktalarını orjine alıp rotasyon yaparak bakış noktasına göre görüş sağlanır. Sonrasında kamera eski yerine alınmalıdır.

ETKİLEŞİMLİ IŞIN İZLEME

Etkilesimli ışın izleme, belli bir bakış noktasındaki gözlemcinin klavye/mouse kullanarak belli dönme, ilerleme hareket komutlarıyla gözlemlediği ortamla arasındaki etkileşimdir.



Kullanılan görüntüleme modelinde dönme hareketinin bakış noktası etrafında olacağı varsayıldığından öncelikle P_0, P_1, P_2, P_3 ten bakış noktası çıkarılır. Elde edilen koordinatlar dönme işlemi için gereken rotasyon matrisi ile çarpılıp bakış noktası tekrar eklenir. Böylece dönme hareketi sonucu görüntü düzleminin yeni P_0, P_1, P_2, P_3 noktaları elde edilir.

Görüntü düzleminin koordinatları değişince bakış noktasının koordinatları da değişir. Önce yeni düzlemin merkez koordinatları

bulunmalıdır. Bu da $(P_0+P_2)/2$ veya $(P_1+P_3)/2$ ile bulunur. Görüntü düzleminin yeni normali de $(P_1-P_0)\times(P_2-P_1)$ ile bulunur. Normalize edilen normal, bakış noktasının görüntü düzlemine olan uzaklığı (12,5) ile çarpılıp görüntü düzleminin merkezine eklendiğinde bakış noktasının yeni koordinatları bulunur.

İlerleme hareketinde de hem görüntü düzleminin hem de bakış noktasının koordinatları değişmektedir. Bu koordinatlar ilerleme doğrultusuna göre belirlenir. İlerleme doğrultusu ise geri yönde ilerlemede görüntü düzleminin normali doğrultusunda; ileri yönde ilerlemede ise normalin tersi doğrultusunda olur. İleri yönde k birim kadar ilerlenecekse hesaplanan normal k ile çarpılıp. P_0, P_1, P_2, P_3 noktalarından çıkarılır.(geri yön için toplanır). Sonra da dönme de olduğu gibi bakış noktasının yeni koordinatları hesaplanır.

Piksel koordinatları:

$$P_0 + \frac{(P_1-P_0)*x}{640} + \frac{(P_3-P_0)*y}{480}$$

27.03.2013

DIRECT X

DirectX 11 de tüm setlemeleri bizim yapmamız gerekir.

- Device
- Immediate Context
- Swap Chain

} olmak üzere 3 tane nesne oluşturulmalıdır.

✓ DirectX 10'da device nesnesi çizim işlemleri için, resource kaynakları oluşturmak için kullanılır.

✓ DirectX11 de backbufferla çizim yapma işinden immediate context yapmış oldu.

✓ Device → Resource creation

✓ Immediate Context → Render auto backbuffer

✓ Swap chain → Bufferları swap yapar. Front buffer'ın içeriğini ekranda görüntüler.

* DirectX 11'de kodlama kısmı bize ait. Ekran kartını istediğimiz gibi kodlayabiliyoruz.

* Immediate context'in draw komutuyla çizim yapılır.

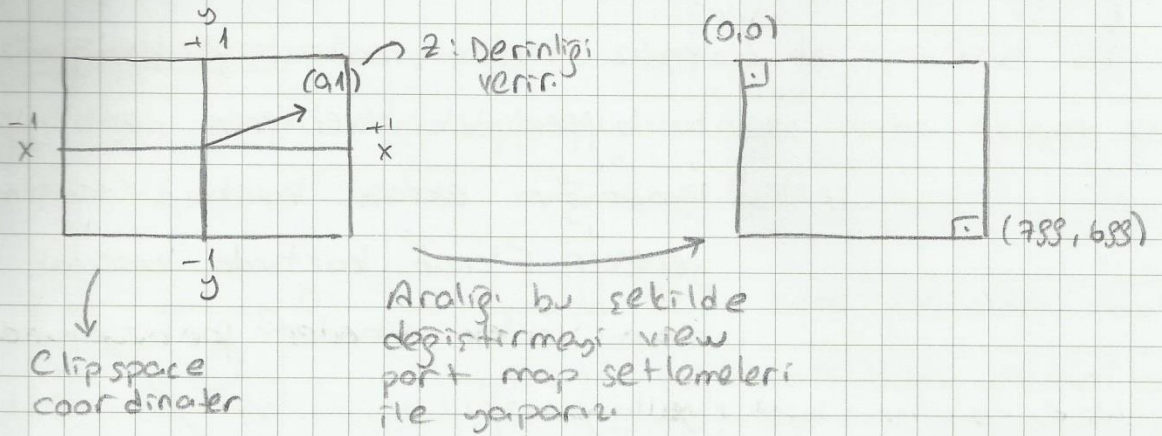
✓ O anda görünen buffer front buffer'dır. Arka planda yazım yapılan da backbuffer'dır. Swap chain arka tarafta koşan backbuffer ile frontbuffer'ın yerdeğiştirmesini sağlar. Böylece ekranda görüntüyü sürekli olarak görebiliyoruz. Kısacası iki tane buffer var. Birisi ekranda görünen diğeri de arkada üzerine yazım yapılan buffer. Bu işleri present fonksiyonu yapar.

✓ g_hwnd = Create window (---) → Açılan pencereyi gösteren pointer g_hwnd'dir.

✓ render targed view → Çizim yapacağımız yeri setlememiz gerekir.

Bu bufferlardan hangisine çizim yapacağımızı render targed view ile setlememiz gerekir. Hangi amaçla bellek alanının kullanacağımızı belirlemek içinde bunu tanımlamamız gerekir.

O bellek alanına çizim yapabilmek için orayı render targed olarak setlememiz gerekir. Bunu da render targed view ile yaparız. Kullanacağımız buffer backbuffer'dır.



GetMessage() → Mesaj varsa işlem yapar. Yoksa pencerenin bloklanmasına sebep olur. Yani işleme izin vermez.

PeekMessage() → Eğer mesaj yoksa bloklanma yerine return yapar. Yani else bloğu koşar.

Render() → Çizim fonksiyonu.

→ .cpp'nin içinde tanımlanan tüm veriler .fx uzantılı dosya içinde, DirectX 11 kodlarıda .cpp'nin içinde koşulur.

DirectX 11 (.cpp)

HLSL (.fx)
(High Level Shading Language)

→ <u>Vertex Shader</u> fonksiyonu üçgen çizme, küp çizip döndürme gibi uygulamalarda kullanılacak matrislerdeki dönüşüm için kullanılır. Pozisyon döndürür. Koordinatlar üzerinde işlem yapar. 3 boyutlu ortamdan 2 boyutluya izdüşüm yapar. World, view, projection matrislerini kullanır.

→ Pixel Shader, geriye renk döndürür.

```
Struct SimpleVertex
{
    XMFLOAT3 Pos;
};
```

Bu cpp dosyasında tanımlı olan structer'dır. Köşe noktalarını tuttuğumuz Vertex matrisi gönderdiğimizde, bu buffer'daki verilerin nasıl yorumlanacağını ekran kartına tanıtmamız gerekir. Ekran kartında koşacak olan Vertex Shader fonksiyonuna yolluyoruz.

→ Vertex Shader semantiği gördüğünde kendine gelecek verilerin formatının nasıl olduğunu anlar. Formatı Vertex Shader'ın anlayacağı şekilde yollamalıyız.

XMFLOAT3 POS(float x, float y, float z)

→ Köşe noktalarının herbiri için bir input tanımladık. Ancak bu köşe noktalarından oluşacak üçgeni henüz oluşturmadık.

Birden fazla buffer tanımlamak istediğimizde g_pVertexBuffer ile bunu tanımlayabiliriz.

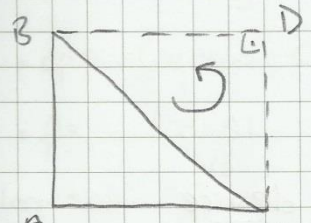
→ Buffer'ları ekrana çizdirmek için Immediate nesnesi ile draw(..) fonksiyonu kullanılır.

→ Vertex buffer ile backbuffer içine yazacağımız buffer'ları tanımlıyoruz.

→ Tek bir buffer kullanılarak aynı cisim farklı noktalara transfer edilebilir.

→ Vertex buffer'a birden fazla üçgen setlediğimizde primitive topology diye bir topology tanımlamamız gerekir.

→ Köşe noktalarına birtanesi daha eklenerek yeni üçgen oluşturma Triangel Stript ile yapılır. Bu topology de primitive dir. Triangel stript olan buffer'a bir üçgen daha eklemek için tek bir köşe noktası eklememiz yeterli.

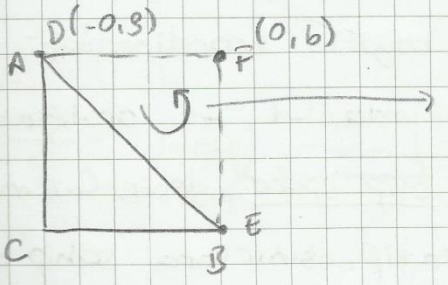


Triangle Stript
(Primitive Topology)

→ Burada BCD köşe noktaları saat yönünü sağlamaz. Bunu ekran kartı otomatik olarak swap yapar. 2., 4., 6., 8.. üçgenlerde swap yapılır. Çünkü o üçgenlerde yön saat yönünün tersindedir.

Burada 3 tane köşe noktası kullanılarak iki üçgenle kare oluşturulur.

$$\begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix}$$



Burada 6 tane köşe noktası kullanılır.

→ Triangle List

→ DirectX sol el kuralını kullandığı için saat yönünün tersi olduğunda bunu backface olarak algıladığı için çizmez. Noktaların yönünü değiştirdiğimizde çizmesini sağlayabiliriz.

winMain() → DirectX ile ilgili fonksiyonların çağrıldığı fonksiyondur.

- DirectX uygulamalarında Render ve initDevice içeriğini güncelleyeceğiz.

1. device
2. immediate context } device (DirectX10)
3. swap chain

- ✓ Vertex buffer oluşturma
- ✓ Index buffer oluşturma
- ✓ Buffer setlemeleri

} Bunlardan device sorumlu

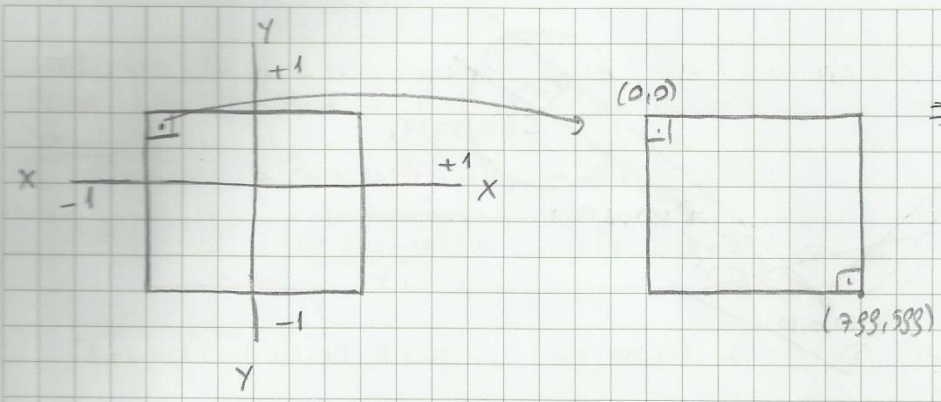
→ Backbuffer üzerine çizim yapmaktan immediate context sorumlu.

→ O anda ekranda görüntülenen buffer'a frontbuffer denir. Ekranda görüntüleme işleminden swap chain sorumludur.

→ Backbuffer'la çizim yapıldığını, o alanda ekranda gösterileceklerin çiziminin yapıldığını render da belirleriz.

→ ViewPort: Bu değişken bizim 3 boyutlu koordinatdaki cisimler ekran düzlemine taşındığında -1, +1 aralığındaki koordinatlara izdüşüm alınır. Buna clip space denir. Çizim yaptığımız penceredeki değerler negatif sınırlara sahip değil.

→ Vertex Shader'ın en önemli görevi transformasyon işlemidir. (3B'dan 2B'lu ekran koordinatlarına dönüşüm)



⇒ Ray Tracing'de bunun tersi yapılmıştır. Formdan görüntü düzlemine taşınmıştı.

→ Vertex buffer setlemelerinden önce input layout tanımlamaları yapılır.

- .cpp (Visual C++, DirectX 11)
- .fx (HLSL, C++)
  VS
  PS

→ Vertex buffer aslında bir pointer verilen köşe noktalarını ifade eder.

→ Input Layout tanımlamasından sonra köşe noktaların tutan bir vertex buffer tanımlanmalıdır.

InitData → Vertex buffer'a ilk değer atamada kullanılır.

SwapChain: present fonksiyonu ile backbuffer'ın içeriğini ekranda görüntüler.

17.04.2013

→ Render işleminden önce Vertex Shader ve Pixel Sheder setlenmelidir.

- .cpp kodları yeterli değil. .fx uzantılı kodların da yazılması gerekir.

High Level Shading Language

✓ Vertex Shader'ın en önemli görevi transformasyon işlemidir. Transformasyon işlemi için world matrisi kullanılır. Transformasyondan kastımız 3 boyutlu uzaydan 2 boyutlu uzaya dönüşümdür.

World, View, Projection } Koordinatlar bu matrisler ile carpılır. (Dönüşüm)

✓ Vertex Shader'ın çıktısı olan koordinatlar Pixel Shader'a gider. İki boyutlu düzleme aktarılan koordinatlar pixel shader ile boyanır. Renk hesaplama işlemleri pixel shader da yapılır. Doku kaplama işi de yine pixel shader içinde olur.

→ Vertex ve Pixel Shader .fx içinde tutulur.

→ DirectX'te poligonların ekran düzlemine izdüşümü alıp ordan cisimleri z koordinatına göre sıralayıp yakın cismi belirliyoruz.

→ CompileShaderFromFile içinde D3DX11CompileFromFile fonksiyonu çağrılır.

↓

Bu fonksiyon Vertex Shader ya da Pixel Shader'ın derlenmesini sağlar. Ancak hatayı göstermez. Hatayı göstermesi için bazı setlemeleri yapmak gerekir.

3D Space

1) Object Space
2) World Space ↓ Bu sırayla gider.
3) View Space
4) Projection Space
5) Screen Space

Object Space: Çizeceğimiz cisme ait koordinatların olduğu uzaydır. Örneğin bir insan çizimi. Çizim programlarında cisim (0,0,0) noktasına çizilir. DirectX'de modelleri hep (0,0,0)'a çizersek üst üste biner veya boyut değişebilir. (MAYA)

World Space:

- ✓ Scaling
- ✓ Rotation
- ✓ Translation

} işlemleri söz konusudur. World matrisi vardır. World matrisi cisimlerin 3 boyutlu ortamdaki konumu belirlenir.

Nesneler arası ilişkilere bakar. Ortamda nesneleri birbirleri ile ilişkilendirerek tutarız. Bunların birbiri ile uyumlu olmaları için bazı translation, scaling, rotation işlemleri yapılır.

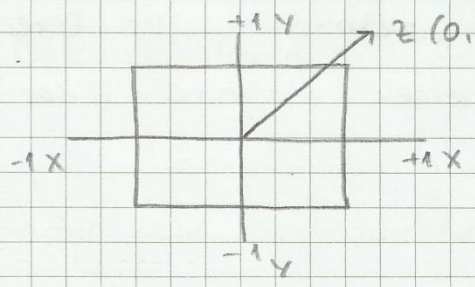
View Space: Orijin bakış noktasıdır. Önemli olan bakış noktasından cisimlerin nasıl göründüğüdür. Bunların setlenmesi view space içindedir.

View matrisi vardır. Eye, At, Up vektörleri kullanılarak bu matris setlenir.

Eye → Bakış Noktası
At → Hangi doğrultuda bakılır
Up → Ne şekilde bakılır

Projection Space: Perspective izdüşüm view space'den yapılır. Bu haline projection space denir. Projection matrisini önce world ile çarpıyoruz. View matrisi ile hangi noktalara bakıp nereleri seçtiğimize bakarız. Projection matrisi ise görüntünün iki boyutlu düzlemdeki izdüşümüne bakar. Z değerleri 0-1 aralığında olur. Bu Z değerleri, birden fazla poligonu rengi söz konusu ise bunlardan hangisini göreceğimize bakarken en yakın olanı alırız. Yani Z değeri en küçük olan ekranda görünür.

Depth Buffer

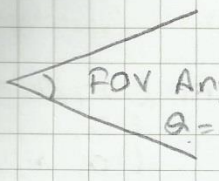


Screen Space: Bunun orjini 0,0 değeri olur. Projection space'deki değerler matrise taşındığında screen space elde edilir. Bunu ekran koordinatlarına dönüştürmek için VIEWPORT tanımlanır. Cismin ekrandaki konumunu bulur. (frame buffer)

Bu uzaylardan birbirlerine dönüşümde herbiri için bir matris tutulur.

Object Space
↓
World Matrix → Başlangıçta birim matris alınır. Daha sonra cisim üzerinde ölçekleme, rotasyon translation işlemi yapıldıkça değişir.
↓
World Space
↓
View Matrix
↓
View Space
↓
Projection Matrix
↓
Projection Space
↓
Screen Space

→ field of view (FOV) bakış açısı demektir. Bunu genelde 45° alırız.

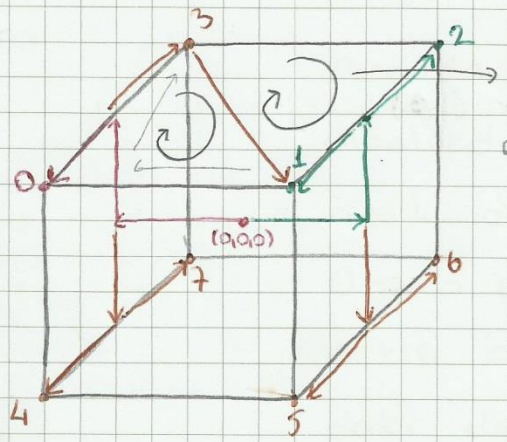


Perspective İzdüşüm (FOV Angle, width/height, Yakın z düzlemi, Uzak z düzlemi)

- FOV Angle → Bir görüş piramidi oluşturuyoruz. (Bu sonsuz olabilir)
- width/height → Aspect ratio
- Yakın z düzlemi, Uzak z düzlemi → Piramidin sonsuz olmasını engellemek için kullanılan sınırlardır.

→ Küpü çizme işlemini nasıl yapabiliriz?

Küpün 6 yüzü vardır. Her yüzü 2 üçgen ile ifade edersek 12 üçgen oluşur. Herbir üçgen için 3 köşe noktası olduğuna göre 36 köşe noktası vertex buffer'la yazılır. Burada 36 köşe noktasından bazıları ortaktır.



Saat yönünde olmalıdır.

8 köşe noktası birçok üçgende ortak kullanılır. Herbir üçgen için bunları tekrarlayarak Vertex buffer'da gereksiz yere alan harcamış oluruz. Ancak 8 köşe noktası tutup bunları 3'erli 3'erli eşleştirerek üçgen oluşturulabilir. Bu 3'erli yapıyı tutan buffer Index Buffer'dır.

→ Index Buffer'da alırken 3'erli şekilde alıyoruz. 12 üçgen olduğu için 12 satır var.

```
indices[] = {  3,1,0,    3,4,7,    2,7,6,
               2,1,3,    0,4,3,    3,7,2,

               0,5,4,    1,6,5,    6,4,5,
               1,5,0,    2,6,1,    7,4,6 };
```

→ Simple Vertex'de artık hem köşe noktası hem de rengi tutulur.

```
vertices[] = {
    { XMFLOAT3 (-1, 1, -1)   , XMFLOAT4 (0, 0, 1, 1)}, → 0
    {    ''    ( 1, 1, -1)   ,    ''    (0, 1, 0, 1)}, → 1
    {    ''    ( 1, 1, 1)    ,    ''    (0, 1, 1, 1)}, → 2
    {    ''    (-1, 1, 1)    ,    ''    (1, 0, 0, 1)}, → 3
    {    ''    (-1, -1, -1)  ,    ''    (1, 0, 1, 1)}, → 4
    {    ''    (1, -1, -1)   ,    ''    (1, 1, 0, 1)}, → 5
    {    ''    (1, -1, 1)    ,    ''    (1, 1, 1, 1)}, → 6
    {    ''    (-1, -1, 1)   ,    ''    (0, 0, 0, 1)}, → 7
                                                  };
```

→ Birden fazla index ve vertex buffer setlemek mümkündür

→ Index buffer kullanarak üçgenler setlenmişse bunu çizmek için DrawIndexed kullanılır

Constant Buffer → Hem .fx hem de .cpp içinde kullanılır. Vertex Shader'dan önce tanımlanır.

UpdateSubresource ile buffer'ın içeriği Vertex Shader'a gönderilir.

```
cbuffer ConstantBuffer : register (b0)
{
    matrix World;
    matrix View;
    matrix Projection;
}
```

→ Birden fazla Constant buffer kullanmak mümkündür

3D Transformation

World matrisi ile ilişkilidir. Bu matrisi setlerken 3 matris etkilidir. Scale, Translate, Rotate

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix} \rightarrow \text{Translation}$$

$a \downarrow x \quad b \downarrow y \quad c \downarrow z$

$$\begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow y \text{ ekseninde Rotation}$$

$$\begin{bmatrix} p & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \text{Scale (ölçekleme)}$$

Depth Buffer: Screen space'de Translate ettiğimiz x, y (-1, +1), z ise (0, 1) arasında değişir. Burada ekranda görünecek olanlar z değeri küçük olanlardır. Ekrana çizilen her pixelin derinlik değerini kontrol etmekte kullanılır. O anda çizilen pixelin değeri <= ise eskisi atılıp yenisi yazılır. Depth buffer'daki değerinden büyük bir renk değeri yazılmaya çalışılırsa değer değiştirilmez. Büyük olan renk değerini yazamayız.

→ Art arda birkaç tane draw yapabiliriz. En son present ile ekrana çizilir. Üçgenler mavi – yeşil – kırmızı olsun. Kırmızı çizildi, ardından mavi geliyor. Backbuffer'da kırmızı vardı, mavi ve kırmızı için depth'lere bakılır. Mavi küçükse backbuffer'a mavi üçgen yazılır.

Transformation

1- g_world = mScale * mTranslate * mRotate;

World matrisi güncellerken matrisleri yukarıdaki sırada çarparız. Rotate ve Scale işlemi orjine göre yapılır. Yani bir cisim rotate yapılacağı zaman (0,0) noktasında değilse ötelenen miktardaki nokta etrafında döner. (0,0)'da olmayan cisme scale yapıldığında boyut ne kadar küçüldüyse başlangıca olan mesafe de o kadar küçülür.

↳ Backbufferla istediğimiz kadar çizim yapabiliriz. En son onlara present yaptığımızda ekranda görebiliriz.

2- g_world = mScale * mRotate * mTranslate
↳ küçültülmüş küp -ü noktasında döner.

3- g_world = mTranslate * mScale * mRotate
↳ Önce öteleme sonra küçülme yapılır. Scale orjine göre yapıldığı için aradaki mesafede küçülür. Ortadaki büyük küpe yakın bir şekilde rotatate yapar.

§ Backbuffer'a en son hangisi yazıldıysa depth buffer'da o çizilir. Depth buffer olmazsa büyük ve küçük küpün kesiştikleri noktada küçük küpü çizen uzaklık testi yapmaz.

4- g_world = Translate * Rotate * Scale → 3. ile aynı

5- g_world = Rotate * Scale * Translate → 2. ile aynı

6- g_world = Rotate * Translate * Scale → Büyük küpe yakın, kendi ekseni etrafında döner.

Lighting

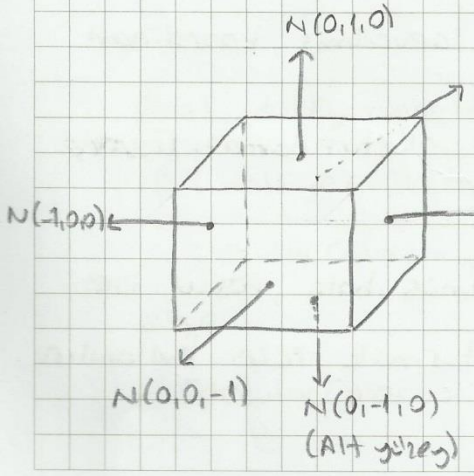
24.04.2013

Kendi ekseni etrafında dönen 2 üçgenden oluşan bir zemin ve zeminin içinden geçecek şekilde dönen biri beyaz diğeri sarı 2 küp ışık kaynağı var.

→ Işık kaynağımızı 2 şekilde tutuyoruz.

1) Buffer'lar ile küp olarak tutuyoruz.
2) Köşe noktalarını tutuyoruz.

* Bir küpün 6 farklı yüzeyi olduğundan ve herbirinin normali farklı olduğundan o yüzeye ait köşe noktalarını bulmak zorundayız.



§ Vertex Buffer'da 4 köşe noktasını tutuyoruz. 4 köşe noktasını kullanarak index buffer'da 2 farklı üçgen oluşturuyoruz.

1. küp olarak ışık kaynağını temsil ettiğimizde küpün tek fonksiyonu ışık kaynağının nerede olduğunu bildirir.

→ Diffuse ve specular bileşen için bir noktanın 3 boyutlu konumda merkez koordinatları kullanılır.

vMeshcolor → zeminin rengi

✓ Küplerin önce transformasyonunu gerçekleştiriyoruz.
✓ Küplerden biri x, biri z ekseni etrafında rotasyon yapıyor
✓ Vertex ve pixel shader'la küpün rengini yollayarak render ediyoruz
✓ Böylece ışık kaynaklarını temsil eden küpler çizilmiş oldu
! Ancak diffuse ve specular bileşen için ışık kaynağının x, y, z koordinatlarını kullanacağız. Yani merkez koordinatları.

XMFLOAT4 (--, --, --, --)
↳ 4 sütunlu olmasının nedeni: dönüşüm matrislerinin 4x4 lük olmasıdır.

output.PosH = mul(input.Pos, world);

⁉ vertex shader'ın outputu olan durumlar pixel shader'ın inputu olur.

posH → 3B'li koordinatların 2B'liye çevrilmiş koordinatı

→ posH ile diffuse ve specular bileşenler oluşturulamaz. Çünkü z'ye de ihtiyaç vardır.
→ posH ekran koordinatlarına dönüştürülmüş hali, posW ise diffuse ve specular bileşenleri oluşturmak için kullanılır.

```
output.PosWV = mul(input.Pos, world);
output.Norm = input.Norm;
```

→ Zemin normali neden transform edilmemiştir?

Zemin y ekseni etrafında döndüğü için normali değişmez.

(Pixel Shader)

→ PS'ye PS-INPUT gelir. Ek olarak gelen EyePos da var.

Bunu constat buffer üzerinden yollayabiliriz.

→ Eye vektörünü view matriste kullanıyoruz.

Diffuse

```
toLight = normalize ( vLightPos[1] - input.PosWV);
float dotLightNorm = dot(toLight, input.Norm);
if (dotLightNorm > 0.0F) → yüzey backface değilse
    diffuseColor = dotEyeNorm * vLightColor[1];
else
    diffuseColor = float3(0,0,0);
```

Specular

```
fromLight = normalize (input.PosW - vLightPos);
toEye = normalize( EyePos - input.PosWV)
float3 reflected = reflect (fromLight , input.Norm)

float dotEyereflected = dot ( toEye, reflected);
if ( dotEyereflected > 0.0F)
    specularColor = pow (dotEye reflected, 32.0f ) * vLight
else
    specularColor = Float3 (0,0,0);
```

Doku Kaplama

→ g_pTextureRV → ile hangi yüzeye doku kaplayacağımızı belirler.

→ Doku kaplama işlemi pixel shader içinde yapılır. Pixel shader da doku kaplama yapılacaksa kaplanacak dokuyu işaret eden pointer g_pTextureRV dir.

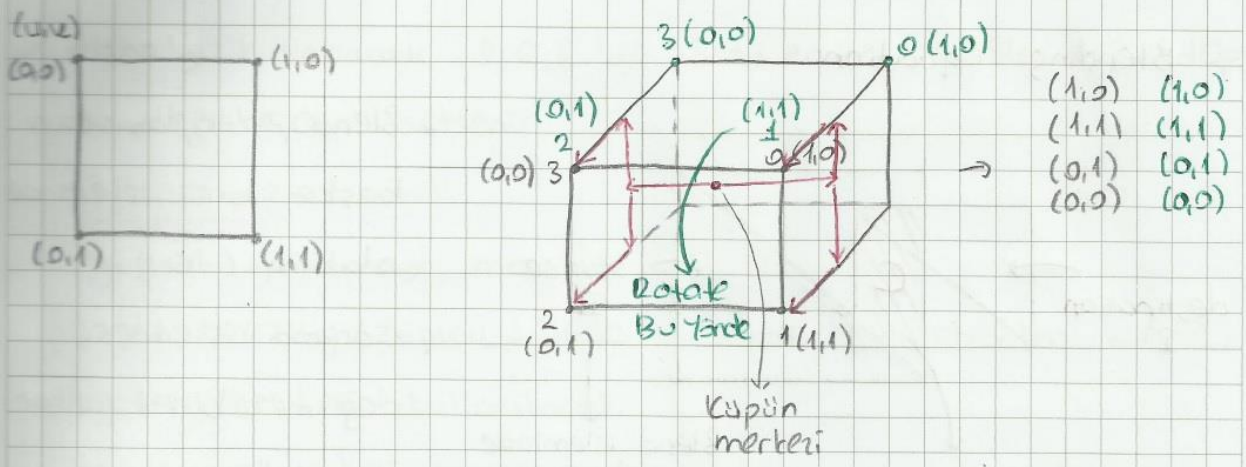
→ Dokuyu kaplarken dokudaki renk geçişlerinin daha yumuşak olması için g_pSamplerLinear kullanılır. Doku üstünde filtreleme kullanılır.

→ Pixel shader öncesi tüm setlemeler yapılır. Doku içindeki rengi pixel shader'daki kosan kod ile belirleriz.

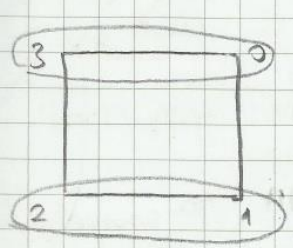
→ Pixel shader içinde sample fonksiyonu kullanılır. Bu doku içinden renk değerini (doku koordinatlarını) alır. Texture koordinatları u,v koordinatlarıdır. O koordinatlara bağlı olarak köşe noktalarının arasındaki değerler için ilgili pixelin texture değerini dokudan okur.

Texture koordinatları → Ray tracing deki barisentrik koordinatlardır.

* Bir önceki konuda ışık kaynağı küçük küplerin merkezinde idi. Normalde bu durumda küpün bütün yüzeylerinin backface olması gerekirdi. Bunun önüne 2 tane PS kullanarak geçiyoruz. PSSolid ile küçük küpleri yeni renklerine boyuyoruz. PSSolid ile diffuse ve specular bileşen hesaplamamış oluyoruz.



* Önce köşe noktalarının sıralamasını kada göre buluyoruz. Sonra rotate yönüne göre yüzeyin köşe noktalarını bulurur.



→ 0,3 ve 1,2 noktaları swap yapılırsa üzerindeki dokunun simetriği (aynaya göre yansıması) oluşmuş olur.

B İnit Device ve Render önemli

Blending

08.03.2013

Alpha Blending

✔ Herhangi bir rengin 3 bileşeni vardı. Buna bir de Alpha eklenir.

(R, G, B, A) → Alpha

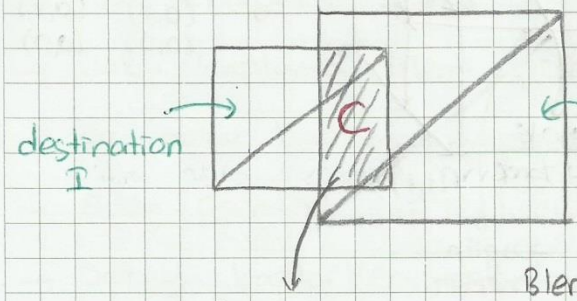
0,5 G + 0,3 R + 0,1 1+B

✔ R,G,B değerleri 8 bit olduğu için 24 bit toplam oluşur. Alpha'da eklenince 32 bite çıkar.

24 bit → 16 milyon renk eder

✔ İki dokunun çakışması sonucu kesişen alanların görünürlüğü alpha bileşeni ile sağlanır.

Blending Equation



Kesisim Bölgesi

Blend işleminde source'un Alpha bileseni kullanilir.

* BlendOp toplamadan başka bir işlem de olabilir. (-) gibi.
* Carpma işlemleri değişmez. Blend factor ile renk mutlaka carpilir ama blend değişebilir.

BlendOp

$C = C_{src} \otimes F_{src} \boxplus C_{dst} \otimes F_{dst}$

SrcBlend: SRC_ALPHA → Source'un alpha bileseni: (0,3)

DestBlend: INV_SRC_ALPHA → Source'un alpha bileseninin 1 eksigi: (1-0,3)=0,7

Örnek koddaki ifadeleri bu sekildedir.

✓ Backbuffer'a ilk yazilmis poligona destination, ikinci yazilana ise source denir. Destination'nin dokusu root tur yani agactir. Source ise izgara sekli (örnek kod için)

✓ Transprency yaparken backbuffer'da destination var. Sonrasında source'da yazildiğinda kesisim noktalarini bulma islemi blending'dir.

✓ Önce birinci doku buffer'a cizilir. İkinci dokuyu cizerken kesistikleri noktalarin rengini hesaplarken amSetBlendState setlemeleri kullanilir.

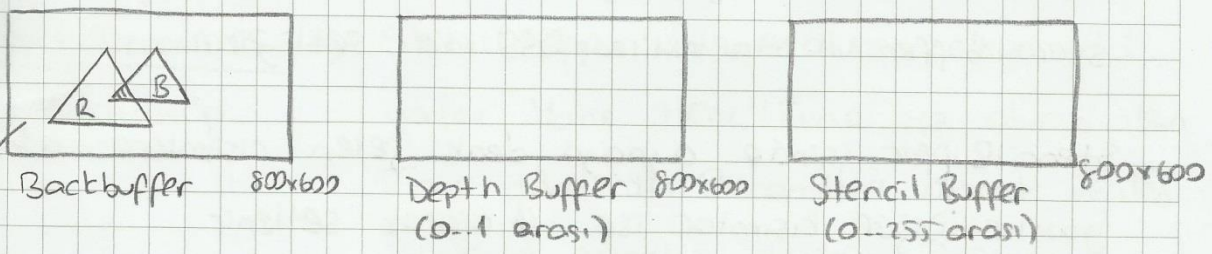
✓ Alpha'yı kullanarak R,G,B'yi SrcBlend, DestBlend, BlendOp bu değişkenler setlenir.

Stenciling

Aynasal yansımayı modeller

D3D11_DEPTH_STENCIL_DESC isminde bir structer var. Bu structer iki kez kullanılacak.

→ Backbuffer'a bir üçgen çizdiğimizde bunu ekranda görüntülemeden önce backbuffer'a birden fazla şekil çizilebilir



Buradaki kesişim bölgesinin ne renk olacağına depth buffer'a göre karar veriyoruz. Mesela mavinin derinlik değeri daha yakınsa kesişim o renk olur ve depth buffer'a da mavinin yakın olduğu setlenir.

Stencil buffer için yapılan setlemeler aynasallık için daha önemlidir. Yansıma olduğunda stencil buffer setlenir.

✓ Depth buffer'ın bir kopyası olarak stencil buffer tutulur.

→ StencilPassOp

→ StencilFunc

✓ if (StencilRef & StencilReadMask ⊵ Value & StencilReadMask)
    accept pixel
  else
    reject pixel

⊵ → StencilFunc

Value → 61

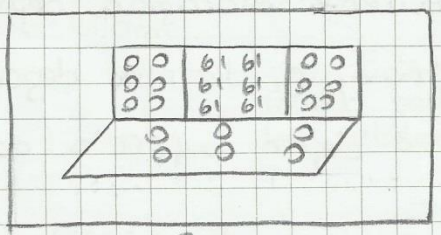
if ifadesi True döndürdüğünde ne yapacağımız StencilPassOp'dır.

Backbuffer'a veya stencil buffer'a bir değer setlenmiş. Mesela başlangıçta 0 olsun. Bu değere stencilRef denir. Karşılaştırmada kullandığımız ikinci değere (bu ikinci değeri poligonu çizmeden hemen önce kullanıyoruz) value denir.

Aynasallığı modellerken ortamdaki tüm nesneler stencilbuffer setlemesi yapılmadan backbuffer'a çizilir.

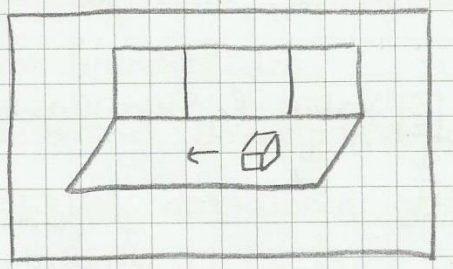
- Zemin, aynanın sağ ve solundaki duvarları çizerken stencil buffer'a setleme yapmıyoruz. Ancak aynayı çizerken stencilbuffer'ı MarkMirrorDSS ile setliyoruz.

→ StencilBuffer içinde aynaya denk gelen cisimlere 61 yazılır. Diğer kısımlar ise 0 olarak setlenir.



→ Bunun bir kopyası backbuffer da da vardır. Burada RGB olarak var, stencil Buffer da int olarak

→ Aynanın yansıması için eğer stencil buffer'daki değer 61 ise yansıma hesaplanır. Ama çizerken stencilbuffer daki yere göre çizer.



* Yansıma hesaplanırken küpü world matrisi ile çarparak aynanın diğer tarafına öteliyoruz.

→ Ayna karşısındaki yansıma için mirrorPlane kullanılır.

XMVECTOR mirrorPlane = XMVectorSet(0.0f, 0.0f, 1.0f, -6.0f)

$Ax + By + Cz + D = 0$

$z - 6 = 0$

$z = 6$ → Tanımlanan plane $z=6$'da sonsuz bir plane'dir.

Aynayı çizerken stencil buffer'a 61 yazmıştık. Yansımasını çizerken 61'e eşit olan noktalara yansıma çizilir.

✓ MarkMirrorDSS ⇒ Stencil buffer'da ne olursa olsun 61 i yazar. Yani StencilFunc ne olursa olsun True döndürür. Parametre olarak verilen değerle sonraki değerin değişimini StencilPassOp yazar. Bunu replace ile sağlar.

StencilPassOp
StencilFunc

✓ drawReflectionDesc ⇒ Yansımayı stencil buffer'daki 61 olan yerlere çizer. Yansımayı backbuffer'a çizerken StencilFunc == ise True döndürür. Yani daha öncededen stencil bufferdaki değer 61 ise demektir. StencilPassOp ise 61 olan yerlere çizimi yapar. Bunu keep ile sağlar.

Draw emri hem backbuffer'a hem de stencil buffer'a yazma yapar. Mutlaka kullanılmalıdır. Draw hem back buffer'a yazar hemde setlemeye bağlı olarak stencil buffer'a yazar. Draw'dan önce DepthStencilState setlenir. Backbuffer'a çizerken stencil buffer'daki ilgili pixellerde setlenir.

15.05.2013

MAYA

- Maya'da export etmeden önce cisimleri üçgenlere bölmek gerekir. Çünkü vertex buffer'da üçgenler üzerinden işlem yapılmaktadır.

Window
↳ Setting/Preferences
↳ Plug-in Manager
↳ ObjExport seçilir

→ Obj dosyaları DirectX içine kopyalanır. Kod çalıştırıldığında console'dan hangi dosyanın seçileceği sorulur ve sonrasında model.txt dosyası oluşur. Sonrasında model.txt dosyası render ile okunur ve vertex buffer'a render edilir.

→ Grand.txt'de 6 tane köşe noktası vardır. Çünkü bölge iki üçgene bölünmüştür. Veriler index buffer'da vertex buffer'da tutulur.