PÜRÜZLÜ YÜZEY ÜRETİMİ

1. Giriş

Yüzey dokusu, yüzeye küçük ayrıntılar eklemek için kullanılabilmesine rağmen, bu yöntem portakal veya çilek gibi cisimlerin pürüzlü yüzey görünümünü modellemek için etkin olamamaktadır. Bu tür cisimlerin doku bilgisinde var olan ışık-şiddeti ayrıntısı (ışık-kaynağının doğrultusu gibi) aydınlatma parametrelerinden bağımsızdır. Yüzey pürüzlülüğünü (bumpiness) modellemenin en iyi yolu, yüzey normaline bozma (perturbation) fonksiyonu uygulamak ve daha sonra bu **bozulmuş normal vektörünü aydınlatma-modeli** hesaplamalarında kullanmaktır. Buna pürüz izdüşüm (bump mapping) tekniği denir.

Cisimlerin yüzeyleri üçgen kafes kullanılarak temsil edilir. Her pikseldeki parlaklık, aydınlatma bağıntısının değerlendirilmesi esnasında normal vektörü tarafından belirlenir. Bu normal vektörler, çoğu zaman üçgen kafesin köşelerindeki normaller veya bunların düzgün interpolasyon yapılmış değerleridir. Bu kaba çözünürlük, kafesteki üçgenden daha küçük boyutlardaki ayrıntıların aydınlatılmasını engeller. Pürüz izdüşüm tekniği, her pikselde normal vektörünü, doku bilgisi yardımıyla bozarak daha fazla ayrıntı yanılgısı oluşturmayı amaçlar.

2. Pürüz Matematiği

Parametrik yüzey üzerindeki bir nokta **P**(s, t) ise, bu noktadaki yüzey normali aşağıdaki bağıntı ile hesaplanabilir.

$$\mathbf{n} = \mathbf{P}_s \times \mathbf{P}_t \qquad (1)$$

$\mathbf{P}_s$ ve $\mathbf{P}_t$ vektörleri, **P** noktasının sırasıyla s ve t parametrelerine göre kısmi türevleridir. Yüzey normalinde değişimler yaratmak için **P** yüzey konum vektörüne, pürüz fonksiyonu adı verilen b(s, t) küçük bozma fonksiyonu eklenir.

$$\mathbf{P}'(s, t) = \mathbf{P}(s, t) + b(s, t)\,\mathbf{N} \qquad (2)$$

Bu bağıntı, yüzeyin **N**=**n**/ |**n**| birim normali doğrultusunda yüzeye pürüz ekler. Bozulmuş yüzey normali aşağıdaki bağıntı ile verilir.

$$\mathbf{n}' = \mathbf{P}_s' \times \mathbf{P}_t' \qquad (3)$$

$\mathbf{P}'$ noktasının s parametresine göre kısmi türevi

$$\mathbf{P_s}' = \frac{\partial}{\partial s}(\mathbf{P} + b\mathbf{N})$$
$$= \mathbf{P_s} + b_s\mathbf{N} + b\mathbf{N}_s \quad (4)$$

b-pürüz fonksiyonunun küçük olduğu varsayılırsa son terim ihmal edilebilir.

$$\mathbf{P_s}' \approx \mathbf{P_s} + b_s\mathbf{N} \quad (5)$$

Benzer şekilde aşağıdaki bağıntı yazılabilir.

$$\mathbf{P_t}' \approx \mathbf{P_t} + b_t\mathbf{N} \quad (6)$$

O zaman bozulmuş yüzeyin normali

$$\mathbf{n}' = \mathbf{P_s} \times \mathbf{P_t} + b_t(\mathbf{P_s} \times \mathbf{N}) + b_s(\mathbf{N} \times \mathbf{P_t}) + b_s b_t(\mathbf{N} \times \mathbf{N})$$

yazılabilir**.** $\mathbf{N} \times \mathbf{N} = 0$ olduğundan

$$\mathbf{n}' = \mathbf{n} + b_t(\mathbf{P_s} \times \mathbf{N}) + b_s(\mathbf{N} \times \mathbf{P_t}) \quad (7)$$

bulunur. Bu süreçteki son adım, aydınlatma modeli hesaplamalarında kullanılmak üzere $\mathbf{n}'$ normalini normalize etmektir.

3. Pürüz Haritasının Oluşturulması

Normal vektörünü bozacak yüksek çözünürlüklü bilgi, 3-boyutlu vektörlerin 2-boyutlu dizisi halinde saklanır. Bu bilgiye **pürüz haritası (bump map)** veya **normal haritası (normal map)** denir. Pürüz haritasındaki her vektör bir yön gösterir. Bu yön, üçgenin içindeki bir noktada interpolasyonlu normal vektörüne göre yeni normal vektörünün yönüdür**.** <0, 0 ,1> vektörü, bozulmamış (unperturbed) normali gösterir; halbuki herhangi başka bir vektör ise bu normale göre bir değişimi gösterir, ve bu yüzden ışıklandırma (lighting) bağıntısının sonucuna etki eder.

Pürüz haritası, yükseklik (height) haritasından normal vektörler çekilerek inşa edilir**.** Yükseklik haritası ise her pikselde düz (flat) yüzeyin yüksekliğini gösterir. Yükseklik haritasında herhangi bir piksele karşı düşen normal vektörünü türetmek için, yüzeyin bu noktasında s ve t doğrultularındaki teğetleri hesaplanmalıdır. Bu teğetler, seçilen piksele komşu olan piksellerin yüksekliklerindeki (height) farka dayanır. Boyutu w*h piksel olan bir yükseklik haritası alınsın ve bu haritanın <i, j> koordinatlarında saklanan yükseklik değeri H(i, j) ile gösterilsin. s ve t doğrultularındaki teğet vektörler sırasıyla **S**(i,j) ve **T**(i,j) ile gösterilirse, k yükseklik ölçeklendirme katsayısı olmak üzere aşağıdaki bağıntılar yazılabilir.

$$\mathbf{S}(i, j) = < 1, 0, kH(i+1, j) - kH(i-1, j) >$$

$$\mathbf{T}(i, j) = < 0,1, kH(i, j+1) - kH(i, j-1) > \quad (8)$$

S(i, j), **T**(i, j) vektörlerinin z bileşenleri sırasıyla S_z ve T_z ile gösterilirse, o zaman **N**(i, j) normal vektörü aşağıdaki gibi hesaplanabilir.

$$\mathbf{N}(i, j) = \frac{< -Sz, -Tz, 1 >}{\sqrt{Sz^2 + Tz^2 + 1}} \quad (9)$$

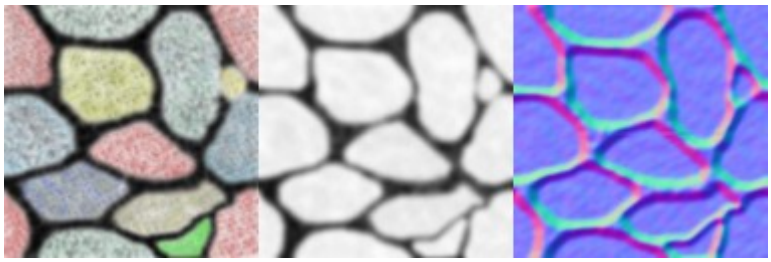
Her normal vektörün bileşenleri aşağıda verilen bağıntılar kullanılarak RGB rengi olarak kodlanır. Şekil 1'de **doku haritası**, gri kodlanmış **yükseklik haritası** ve bu yükseklik haritasından elde edilen **pürüz haritası** (yeni normal haritası) gösterilmiştir.

$$\text{Kırmızı} = \frac{x+1}{2} \quad (10)$$

$$\text{Yeşil} = \frac{y+1}{2}$$

$$\text{Mavi} = \frac{z+1}{2}$$

x, y, z koordinat bileşenleri [-1, 1] aralığında değiştiği halde renk bileşenleri ise [0, 1] aralığında değerler alır. Negatif x değerlerine +1 eklenerek pozitif alana taşındığı zaman x=+1 değerleri +2'ye yükselir. Bu yüzden ötelenmiş x'ler tekrar 2'ye bölünerek [0, 1] aralığında gösterecek şekilde normalize edilir.



Şekil 1. Doku, Yükseklik, ve Normal haritaları

4. Teğet Uzayı

Pürüz haritasındaki <0,0,1> vektörü bozulmamış (unperturbed) normali gösterdiğinden, ışıklandırma (lighting) bağıntısındaki interpolasyonlu normal vektöre bozulmamış vektör karşı düşürülmelidir**.** Bu iş her köşede (vertex) bir koordinat sistemi kurarak başarılabilir. Köşelerdeki köşe normali daima pozitif z eksenini gösterir. **Ortonormal** yapıyı oluşturmak için bu normal vektöre ilaveten her köşede yüzeye teğet iki vektöre daha ihtiyaç vardır. İşte bu şekilde elde edilen koordinat sistemine **teğet uzayı** veya **köşe uzayı** denir.

Üçgen kafesin her köşesinde teğet-uzayı koordinat sistemi kurulduktan sonra, **L ışık vektör**ünün doğrultusu her köşede hesaplanır ve teğet uzayına transform edilir. Daha sonra teğet uzayındaki **L** vektörü bir üçgenin yüzü üzerinden interpolasyon yapılır. Teğet uzayındaki <0, 0, 1> vektörü, normal vektörüne karşı düştüğünden; **L** ışığına göre teğet-uzayının doğrultusu ile pürüz haritasından gelen numunenin (sample) çarpımı, **Lambertian** yansıma terimini üretir.

Her köşedeki teğet vektörler, pürüz haritasının doku uzayı ile çakışacak (align) şekilde seçilmelidir. Parametrik fonksiyonlarla üretilen yüzeyler için, bu teğetler, parametrelerin her birine göre türev alınarak hesaplanabilir. Keyfi üçgen kafesler ise kendilerine herhangi bir yönde uygulanmış pürüz haritalarına sahip olabilir. Bu durum her köşede teğet doğrultularını belirleyen bir yöntemi gerekli kılar.

5. Yazılım Gereksinimleri

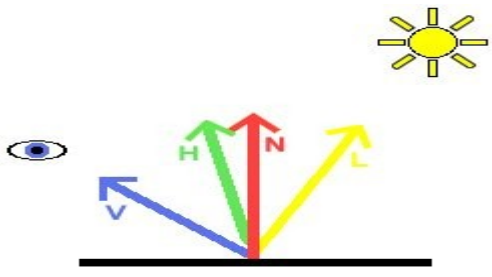
Grafik API'lerinin çoğu poligona dayalı çalışır ve poligon üretir. Işık kaynağının konumuna bağlı olarak sadece köşe noktalarının (vertex) renk değerlerini hesaplar. Poligonların içindeki noktaların rengi interpolasyonla üretilir. Bu yüzden API'ler boyama işlemini poligon başında yaparlar. Pürüzlü yüzey üretiminde ise piksel boyutunda boyama yapılır ve ışık kaynağına bağlı olarak her pikseldeki renk ayrı ayrı hesaplanarak pikseller boyanır.

Grafik kartlarında **Köşe (vertex) işlemci** ve **Piksel (fragment) işlemci** olmak üzere iki işlemci vardır. Köşe işlemci köşelerin dünya koordinatlarından ekran koordinatlarına izdüşümü, görünmeyen yüzeylerin atılması (backface culling), kırpma (clipping) ve z-tamponlama gibi işler yapar. **OpenGL** ve **DirectX** gibi **API**'ler grafik kartının sadece köşe işlemcisini programlayabilir. Grafik kartın her iki işlemcisini programlayabilmek için geliştirilmiş dillere "**Tonlama Dili (shading language = SL)**" denir. Bu dillerin köşe işlemciye yönelik kısmına "**Köşe Tonlayıcı**" ve piksele yönelik kısmına da "**Piksel Tonlayıcı**" denir. OpenGL API'sinin Tonlama Diline **GL Shading La**nguage yani kısaca **GLSL** denir.

6. OpenGL API'sinde Aydınlatma

Pürüzlü yüzey üretimi aydınlatma bağıntısını her piksel için değerlendirmeyi gerektirir. Poligon içindeki noktalarda yüzeyin normali değiştirilerek tümsek etkisi oluşturulur. OpenGL'de aydınlatma bağıntısı aşağıdaki gibidir.

```
Vertex Color = emission + globalAmbient + sum(attenuation * spotlight
* [lightAmbient + (max {L.N, 0} * diffuse) + (max {H.N, 0} ^
shininess)*specular])                                              (11)
```



V: Gözlemci vektörü
H: L ile V arasındaki normalizeli yarım yol vektörü
N: Birim uzunluklu normal vektör
L: Normalizeli (birim uzunluklu) ışık kaynağı vektörü

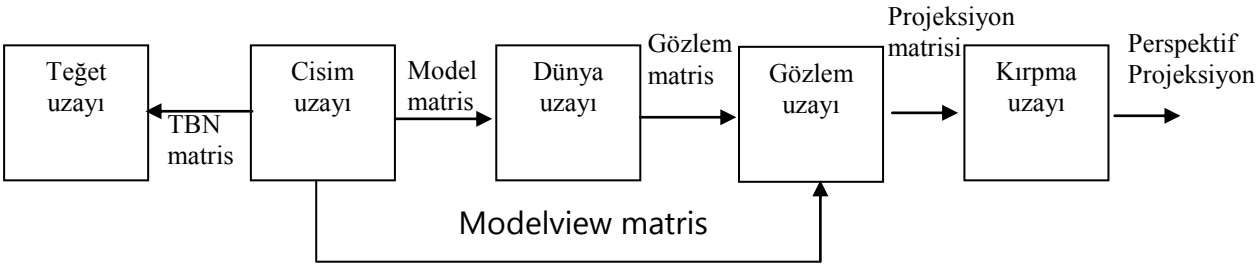
Şekil 2. Aydınlatmada kullanılan vektörler

(11) bağıntısını herhangi bir piksel programı kullanmadan değerlendirmek zordur. Bu yüzden bağıntıdaki bazı bileşenler atılarak, aşağıdaki biçimi kullanılacaktır.

```
color = max{L.N, 0}*diffuse
```
(12)

7. Koordinat Uzayları

Bu deneyde kullanılacak 5 koordinat uzayı aşağıdaki diyagramda verilmiştir. Uzaylar arası geçişler oklar üzerinde gösterilen dönüşümler ile yapılmaktadır.

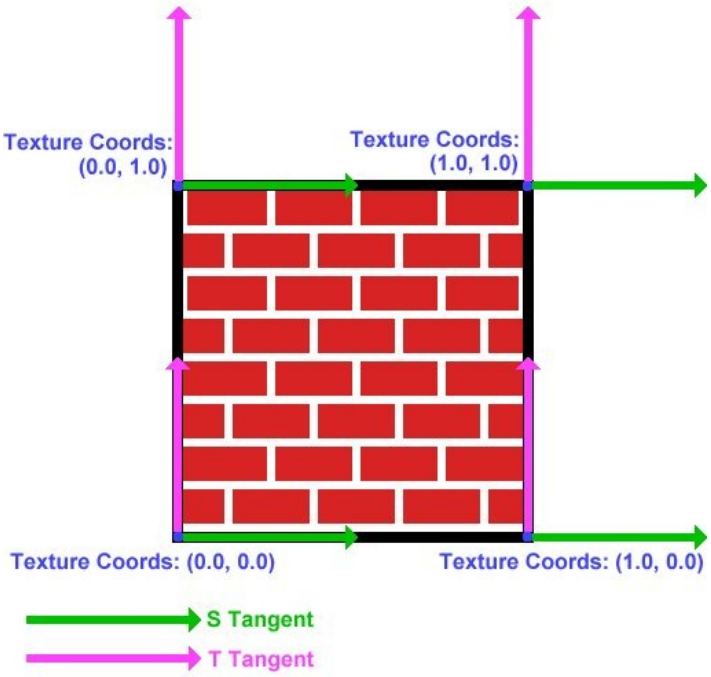


Şekil 3. Koordinat uzayları diyagramları

OpenGL programlayıcıları OpenGL'deki modelview ve projeksiyon matrislerini bilir. Şekil 3'ten görüleceği üzere **modelview matrisi** cisim uzayındaki koordinatları alır ve onları **gözlem uzayına** dönüştürür. Projeksiyon matrisi ise gözlem uzayındaki koordinatları kırpma uzayına iz düşürür. Kırpma uzayından sonra bu koordinatlar perspektif bölmeye ve **gözlem kapısı** (viewport) transformasyonuna tabi tutulur ve pencere üzerindeki cisim koordinatları bulunur.

Köşelerin (vertices) tanımlanması cisim uzayında yapılır. Modelview ve projeksiyon matrisleri birlikte bu köşeleri cisim uzayından **kırpma uzayına** iz düşürür. Kırpma uzayından geri izdüşüm yapmak için **modelview** matrisin tersine ihtiyaç vardır.

Teğet uzayı (veya doku uzayı) modelimizin yüzeyine yerel olan bir uzay olup tek bir dörtgen için Şekil 4'te gösterilmiştir.



Şekil 4. Teğetler

Normalin, dörtgenden dışarı doğru çıkan bir vektör olduğu kolayca görülmektedir. Yani her köşede normal ekrandan dışarı doğru çıkmaktadır ve z- eksenine karşı düşer. S-teğetinin x-eksenine ve T-teğetinin de y-eksenine karşı düştüğü düşünülebilir.

Yüzey üzerindeki her köşenin kendi teğet uzayı vardır. Üç eksen bir köşede bir basis oluşturur ve bu eksenler teğet uzayı (veya doku uzayı) adı verilen bir koordinat sistemi tanımlar. Eğer bu eksenler bir matrise koyulursa, TBN (Tangent, Binormal, Normal) matrisi ortaya çıkar.

$$[TBN] = \begin{bmatrix} S_x & S_y & S_z \\ T_x & T_y & T_z \\ N_x & N_y & N_z \end{bmatrix}$$

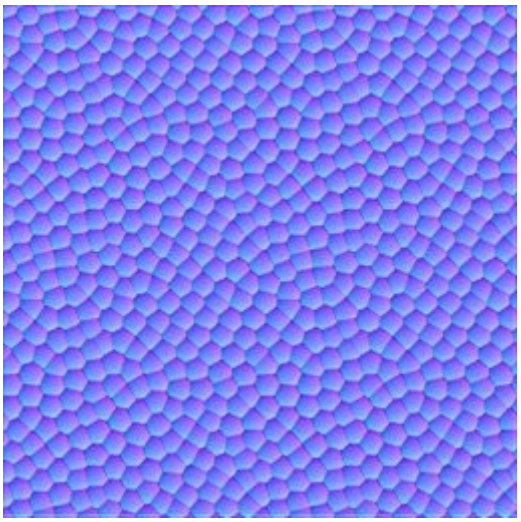
Burada S-teğetin bileşenleri S_x, S_y, S_z, T-teğetin bileşenleri T_x, T_y, T_z ve N-normalin bileşenleri de N_x, N_y, N_z ile gösterilmiştir. Binormal, T-teğetin bir başka adıdır. Cisim uzayındaki bir vektör TBN matrisi ile çarpılırsa, teğet uzayındaki, yani doku uzayındaki karşılığı bulunmuş olur.

8. Normal Haritaları

Burada her piksel için normallerin nasıl değiştirileceği incelenecektir. Değiştirme işlemi, doku haritası kullanılarak her pikselin renginin değiştirilmesine dayanır. Normal haritası, renkleri değil normal vektörlerini saklamalıdır. Normallerin biçimi aşağıda verilmiştir.

$$[x \ y \ z]^T$$

Normaller birim uzunluğa sahip olduklarından $x^2+y^2+z^2 = 1$ dir ve x, y, z $\in \{-1, 1\}$ dir. Bu koordinatlar doku haritasında kırmızı, yeşil ve mavi renklere karşı düşürülerek temsil edilebilir. Doku haritasına koyulan normallerin keza teğet uzayında olacağı unutulmamalıdır. **Olağan normal yani pürüzsüz yüzey normali teğet uzayında ise z-yönündedir** ve bu yüzden x=0, y=0, z=1 bileşenlerine sahip olduğundan (10) bağıntıları (0.5, 0.5, 1.0) renklerini verir. Normal haritalarının mavimsi renkli olması işte bu yüzdendir.

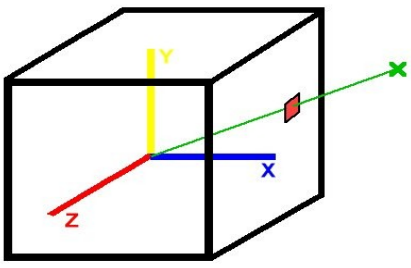


Şekil 5. Normal haritası

9. Küp Haritaları

2D doku, düzlemsel dörtgendir ve renk bilgisinin 2 boyutlu bir dizisidir. Her biri iki boyutlu 6 kare, küpün yüzeylerini oluşturacak şekilde birleştirilsin. Buna **küp haritası** denir. Küp haritasındaki doku renklerine erişmek için 3-bileşenli doku koordinatlarına ihtiyaç vardır, yani **g1TexCoord2f'**den ziyade **g1TexCoord3f** fonksiyonunu veya bu örnekte buna karşı düşen köşe dizi yapısı kullanılmalıdır.

Eğer greenx'in koordinatları **g1TexCoord3f**'ye iletilirse; OpenGL, kırmızı doku rengini kullanır. Bu renk orijine yerleştirilmiş birim küpün merkezini gözlemcinin bulunduğu konuma birleştiren yeşil doğrunun küpü kestiği yer olarak hesaplanır.



Şekil 6. Küp doku haritası

10. Normalizeli Küp Haritası

Bu deneyde normalizeli küp haritası kullanılacaktır. Bunun anlamı, küp haritasında renkleri saklamak yerine vektörleri saklamak anlamına gelir, yani tam normal haritasında olması gereken durum oluşur. (ka, kb, kc) ile indekslenen tekselde şu vektör saklanır:

Normalize(a, b, c)

Bu yüzden küpümüze iletilen herhangi bir vektör, o vektörün normalizeli uyarlamasının geri döndürülmesine sebep olur. Her pikselde ışık vektörünün normalize edildiğinden emin olmak için bu yol kullanılacaktır.

Aydınlatma bağıntımızın **color = max {l.n, 0}*diffuse** olduğunu hatırlayınız. Dünya uzayında ışığın konumu bilinmektedir. **Ters model matrisini** kullanarak ışık konumu cisim uzayına taşınır. Daha sonra, ele alınan köşeden ışık kaynağına giden vektör hesaplanır. Ardından TBN matrisi kullanılarak ışık kaynağı **teğet uzayına** taşınır ve köşe yapıtında saklanır. Daha sonra torus çizilir. Bu ışık vektörü, normalizeli küp haritası kullanılarak normalize edilir. Normal haritasındaki normaller ile ışık vektörü teğet uzayında deklere edildiğinden skaler çarpımları şimdi anlam taşıyacaktır. Daha sonra bu çarpımın sonucu diffuse materyal rengi ile çarpılır.

Bu işleri başarabilmek için **Architecture Review Board**'ın onayladığı OpenGL uzantıları kullanılacaktır.

ARB_multitexture
ARB_texture_cube_map
ARB_texture_env_combine
ARB_texture_env_dot3

OpenGL'in en güçlü yönlerinden biri gelişebilme yeteneğidir. Gelişmeler, donanım özelliklerinden yararlanmak için donanım satıcıları tarafından yazılabilir. Burada kullanılacak uzantılar ARB uzantılardır. Bu uzantılar OpenGL'in en son uyarlamasının çekirdek özellikleridir. Microsoft, Windows için OpenGL'in 1.1. uyarlamasının ilerisi için kütüphaneleri vermemeye karar vermiştir. OpenGL 1.4. uyarlaması indirilebilir, ama bu fonksiyonları kullanmak yerine burada OpenGL 1.1. uzantıları kullanılacaktır.

11. Programın Açıklaması

11.1. Uzantıların Kurulması (Setting Up Extentions)

Açıklamaya önce ARB_multitexture dosyası ile başlanacaktır. Bu OpenGL uzantısının desteklenip desteklenmediği tutulacak bir boole değişkeni ile belirlenecektir.

```
bool ARB_multitexture_supported=false;
```

Aşağıdaki fonksiyon ARB_multitexture'ın desteklenip desteklenmediğini test eder. Eğer desteklerse onu başlatır (initialize). Sizin OpenGL gerçeklemeniz tarafından desteklenen

tüm uzantıların adları, uzantı dizesinde (string) tutulur. "**GL_ARB_multitexture**"i bulmak için bu dize araştırılır.

```
bool SetUpARB_multitexture()
{
```

Uzantı dizesini al ve "**GL_ARB_multitexture"**ini ara. Eğer orada varsa, "ARB_multitexture_ supported"i doğruya (true) setle.

```
char * extensionString=(char *)glGetString(GL_EXTENSIONS);
char * extensionName="GL_ARB_multitexture";

char * endOfString; //store pointer to end of string
unsigned int distanceToSpace; //distance to next space

endOfString=extensionString+strlen(extensionString);

//loop through string
while(extensionString<endOfString)
{
//find distance to next space
distanceToSpace=strcspn(extensionString, " ");

//see if we have found extensionName
if((strlen(extensionName)==distanceToSpace) &&
(strncmp(extensionName, extensionString, distanceToSpace)==0))
{
ARB_multitexture_supported=true;
}

//if not, move on
extensionString+=distanceToSpace+1;
}
```

Eğer "GL_ARB_multitexture" bulunamazsa, bir hata mesajı verilir ve yanlış (false) geri döndürülür.

```
if(!ARB_multitexture_supported)
{
printf("ARB_multitexture unsupported!\n");
return false;
}

printf("ARB_multitexture supported!\n");
```

Eğer bu noktaya erişilirse, ARB_multitexture destekleniyor demektir. ARB_multitexture fonksiyonlarını kullanmak için her fonksiyona ilişkin **fonksiyon göstericileri** (pointer) yaratılmalıdır. O zaman fonksiyon göstericilerini başlatmak için **wglGetProcAddress** kullanılır.

```
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB =NULL;

glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
wglGetProcAddress("glActiveTextureARB");
```

Bu fonksiyon göstericiler global olarak deklere edilir, ve header dosyalarda "extern" olarak deklere edilirler. Doğru header dosya kapsatılarak ve programın başında "**SetUpARB-multitexture**" çağrılarak bu fonksiyonlar şimdi kullanılabilir.

Diğer üç uzantı da aynı şekilde kurulabilir (setup), ama onlar daha basittir, çünkü fonksiyon pointeri gerektirmezler.

11.2. Normalizasyon Küp Haritasının Oluşturulması (Creating the Normalisation Cube Map)

GenerateNormalisationCubeMap fonksiyonu tamamen ismine uygun şekilde davranır.

```
bool GenerateNormalisationCubeMap()
{
```

Birinci olarak, bir küp yüzü için gerekli datayı tutacak uzay oluşturulur. Her yüz 32*32 büyüklüğündedir, ve her noktada R,G,B bileşenleri saklamalıdır.

```
unsigned char * data=new unsigned char[32*32*3];
if(!data)
{
printf("Unable to allocate memory for texture data for cube map\n");
return false;
```

Bazı faydalı değişkenler deklere edilecektir.

```
//some useful variables
    int size=32;
    float offset=0.5f;
    float halfSize=16.0f;
    VECTOR3D tempVector;
    unsigned char * bytePtr;
```

Her yüz için aşağıdaki kısım bir kere yapılacaktır. Burada pozitif x yüzü için bunun nasıl yapılacağı gösterilecektir. Diğer yüzler de oldukça benzerdir. Ayrıntılar için kaynağa (source) bakınız.

```
 //positive x
 bytePtr=data;
```

Bu yüzdeki pikseller üzerinden döngü yapınız.

```
for(int j=0; j<size; j++)
{
for(int i=0; i<size; i++)
{
```

Küpün merkezinden bu teksele (texel) giden bir vektör hesaplayınız.

```
tempVector.SetX(halfSize);
tempVector.SetY(-(j+offset-halfSize));
tempVector.SetZ(-(i+offset-halfSize));
```

Bu vektör normalize edilir ve renk cinsinden saklanmak üzere [0,1] aralığında paketlenerek doku datasında tutulur.

```
tempVector.Normalize();
tempVector.PackTo01();

bytePtr[0]=(unsigned char)(tempVector.GetX()*255);
bytePtr[1]=(unsigned char)(tempVector.GetY()*255);
bytePtr[2]=(unsigned char)(tempVector.GetZ()*255);

bytePtr+=3;
}
}
```

Şimdi küpün bu yüzü OpenGL'e yüklenir (upload). Mevcut küp haritasının pozitif x yüzü olduğu ve ayrıca bu datayı nerede bulacağı da OpenGL'e söylenir.

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB, 0, GL_RGBA8,
32, 32, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
```

Küpün her yüzü için bu iş tekrarlanır. Geçici data saklayıcının silinmesi işi unutmamalıdır.

```
delete [] data;

return true;
}
```

11.3. Torusun Oluşturulması

Torus bilgisi bir sınıfta (class) saklanacaktır. İlk olarak, bir köşeye (vertax) ilişkin datayı saklamak için basit bir sınıf (class) kuracağız. Önce **konum (position)** ve **doku (texture)** koordinatları ve ardından da teğetler ve normal bilgileri saklanır. Son olarak da her köşe için **teğet uzayı ışık vektörü** (tangent space light vector**)** saklanır. Bu vektör teğet uzayında, köşeden ışık kaynağına giden vektördür.

```
class TORUS_VERTEX
{
public:
VECTOR3D position;
float s, t;
VECTOR3D sTangent, tTangent;
VECTOR3D normal;
VECTOR3D tangentSpaceLight;
};
```

Şimdi **ana torus** sınıfı (class) kurulacaktır. Bu sınıf sadece **köşelerin listesini**, **indislerin listesini**, ve bu **listelerin boylarını** saklayacaktır. Bu sınıf ayrıca bu listeleri dolduracak **InitTorus()** adlı bir fonksiyon da içerir.

```
class TORUS
{
public:
TORUS();
~TORUS();
```

```
bool InitTorus();

int numVertices;
int numIndices;

unsigned int * indices;
TORUS_VERTEX * vertices;
};
```

Şimdi 48 doğruluğuna yani halka başına 48 köşeye sahip bir torus tanımlanacaktır.

```
const int torusPrecision=48;
```

Yapıcı (constructor), torus için **Init** fonksiyonunu çağırır.

```
TORUS::TORUS()
{
InitTorus();
}
```

Torus yıkıcı (destructor), belleği serbest duruma geçirmek için indeks ve köşe listelerini siler.

```
TORUS::~TORUS()
{
if(indices)
delete [] indices;
indices=NULL;

if(vertices)
delete [] vertices;
vertices=NULL;
}
```

Aşağıda torus başlatma fonksiyonu verilmiştir.

```
bool TORUS::InitTorus()
{
```

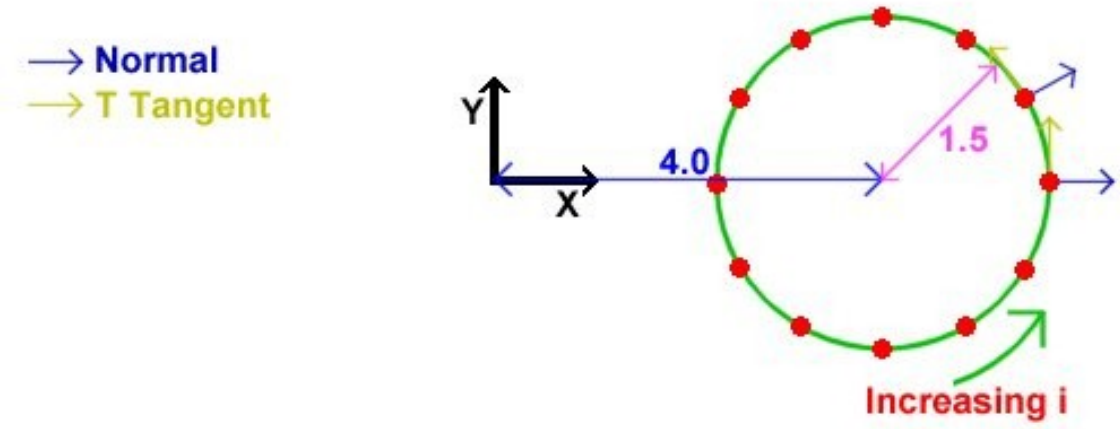
Daha sonra köşelerin ve indislerin sayısı hesaplanır, ve indeks listeleri için gerekli bellek uzayı oluşturulur.

```
numVertices=(torusPrecision+1)*(torusPrecision+1);
numIndices=2*torusPrecision*torusPrecision*3;

vertices=new TORUS_VERTEX[numVertices];
if(!vertices)
{
printf("Unable to allocate memory for torus vertices\n");
return false;
}

indices=new unsigned int[numIndices];
if(!indices)
{
printf("Unable to allocate memory for torus indices\n");
return false;
}
```

Şimdi torusun köşeleri, normalleri, vs, hesaplanabilir. İlk olarak XY düzleminin sağ tarafında 48 köşeden ibaret bir halka üretilecektir.



Şekil 7. Halkanın köşelerle tanımlanması

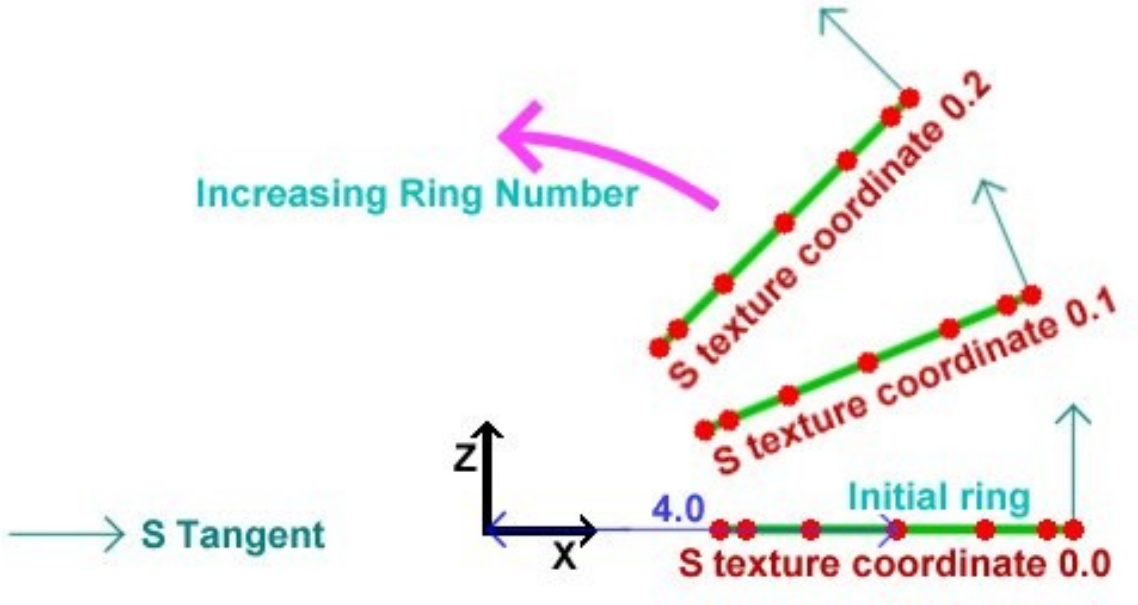
Bu köşelere ilişkin normaller şekilde gösterildiği gibidir. Ayrıca T doku koordinatı i ile lineer artacak şekilde, ve S koordinatını da sıfır olacak şekilde setlenir. Bu yüzden T teğeti artan i yönünü gösterirken S teğeti de ekranın içini işaret eder.

```
//calculate the first ring - inner radius 4, outer radius 1.5
for(int i=0; i<torusPrecision+1; i++)
{
vertices[i].position=VECTOR3D(1.5f, 0.0f, 0.0f).GetRotatedZ(
i*360.0f/torusPrecision)+VECTOR3D(4.0f, 0.0f, 0.0f);

vertices[i].s=0.0f;
vertices[i].t=(float)i/torusPrecision;

vertices[i].sTangent.Set(0.0f, 0.0f, -1.0f);
vertices[i].tTangent=VECTOR3D(0.0f, -1.0f,
0.0f).GetRotatedZ(i*360.0f/torusPrecision);
vertices[i].normal=vertices[i].tTangent.CrossProduct(vertices[i].sTangent);
}
```

Daha sonra, tüm halkayı 2*PI/48 radyanlık adımlarla Y ekseni etrafında döndürürüz. Bu iş torusu oluşturan diğer halkaların üretilmesini sağlar. Tüm normaller, teğetler, vs, döndürülmüş olur, ve T doku koordinatları, orijinal halkanın koordinatları ile aynı kalır. S doku koordinatları ise bir halkadan diğerine artar. İşte birinci halkanın S teğetinin ekranın içini işaret etmesi bu sebeptendir. Bu resim, yukarıdaki resmin üstten görünüşüdür.



Şekil 8. Halkalardaki köşelerin üstten görünüşü

```
//rotate the first ring to get the other rings
for(int ring=1; ring<torusPrecision+1; ring++)
{
for(i=0; i<torusPrecision+1; i++)
{
vertices[ring*(torusPrecision+1)+i].position=
vertices[i].position.GetRotatedY(ring*360.0f/torusPrecision);

vertices[ring*(torusPrecision+1)+i].s=2.0f*ring/torusPrecision;
vertices[ring*(torusPrecision+1)+i].t=vertices[i].t;

vertices[ring*(torusPrecision+1)+i].sTangent=
vertices[i].sTangent.GetRotatedY(ring*360.0f/torusPrecision);
vertices[ring*(torusPrecision+1)+i].tTangent=
vertices[i].tTangent.GetRotatedY(ring*360.0f/torusPrecision);
vertices[ring*(torusPrecision+1)+i].normal=
vertices[i].normal.GetRotatedY(ring*360.0f/torusPrecision);
}
}
```

Şimdi bu konumlandırılan (yerleştirilen) köşelerden üçgenleri kuracak (construct) indisler hesaplanır.

```
//calculate the indices
for(ring=0; ring<torusPrecision; ring++)
{
for(i=0; i<torusPrecision; i++)
{
indices[((ring*torusPrecision+i)*2)*3+0]=ring*(torusPrecision+1)+i;
indices[((ring*torusPrecision+i)*2)*3+1]=(ring+1)*(torusPrecision+1)+i;
indices[((ring*torusPrecision+i)*2)*3+2]=ring*(torusPrecision+1)+i+1;
indices[((ring*torusPrecision+i)*2+1)*3+0]=ring*(torusPrecision+1)+i+1;
indices[((ring*torusPrecision+i)*2+1)*3+1]=(ring+1)*(torusPrecision+1)+i;
indices[((ring*torusPrecision+i)*2+1)*3+2]=(ring+1)*(torusPrecision+1)+i+1;
}
}
```

Böylece torus tamamlanmış oldu.

```
return true;
}
```

10.4. Ana Dosya

İlk olarak gerekli header dosyaları eklenir. Bunlardan "WIN32_LEAN_AND_MEAN", çok sayıda kullanılmayacak gereksiz şeylerin eklenmemesini Windows.h'a söyler. Ardından "**gl.h**" ve "**glu.h**" dosyalarını kapsayan "**glut.h**" eklenir. "**glext.h**" dosyası kullanılacak uzantılar için gerekli simgeleri (tokens) içerir.

Daha sonra uzantılar için yaptığımız headerlar eklenir. **"IMAGE.h"** dosyası, **normal** haritamızı ve **doku** haritamızı yüklemek için gerekli bir görüntü sınıfıdır (image class). "**Maths.h**" dosyası, kullanacak **vektör** ve **matris** sınıflarını içerir. **"Torus.h"** çizilecek torusun ayrıntılarını ve "**Normalisation cube map.h**" ise diğer şeyleri içerir, o şeylerin neler olduğu okuyucuya bırakılmıştır.

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <GL/glut.h>
#include <GL/glext.h>
#include "Extensions/ARB_multitexture_extension.h"
#include "Extensions/ARB_texture_cube_map_extension.h"
#include "Extensions/ARB_texture_env_combine_extension.h"
#include "Extensions/ARB_texture_env_dot3_extension.h"
#include "Image/IMAGE.h"
#include "Maths/Maths.h"
#include "TORUS.h"
#include "Normalisation Cube Map.h"
```

Şimdi bazı global **nesneler** (object) deklere edilecektir. Bu nesneler, **pürüz haritası** ve **renkli dokunun** (color texture) çizilip çizilmemesine ilişkin boole değişkenleridir.

```
//Our torus
TORUS torus;

//Normal map
GLuint normalMap;

//Decal texture
GLuint decalTexture;

//Normalisation cube map
GLuint normalisationCubeMap;

//Light position in world space
VECTOR3D worldLightPosition=VECTOR3D(10.0f, 10.0f, 10.0f);

bool drawBumps=true;
bool drawColor=true;
```

Şimdi programın başında bir kere çağrılan "**Init**" adlı fonksiyon oluşturulacaktır., Bu çağırma pencere oluşturulduktan SONRA olur.

```
//Called for initiation
void Init(void)
{
```

İlk olarak kullanılacak uzantılar kurulacaktır. Eğer bir uzantı kurma işlemi **yanlış** (false) geri döndürürse, o uzantı desteklenmiyor demektir. O zaman demo koşulamaz, hata mesajı verilerek çıkılır.

```
//Check for and set up extensions
if( !SetUpARB_multitexture() || !SetUpARB_texture_cube_map() ||
!SetUpARB_texture_env_combine() || !SetUpARB_texture_env_dot3())
{
printf("Required Extension Unsupported\n");
exit(0);
}
```

Şimdi OpenGL durumları (state) setlenecektir. Birim **modelview** matrisi yüklenerek **renk** ve **derinlik** durumları (states) setlenir. Daha sonra **arkayüz ayıklama** (backface culling) işlemi yetkilendirilir.

```
//Load identity modelview
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

//Shading states
glShadeModel(GL_SMOOTH);
glClearColor(0.2f, 0.4f, 0.2f, 0.0f);
glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

//Depth states
glClearDepth(1.0f);
glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);

glEnable(GL_CULL_FACE);
```

Şimdi bizim dokularımız yüklenecektir (load). İlk olarak **IMAGE** sınıfı kullanılarak **decal harita** yüklenir. Doku haritamız paletli olduğundan (piksel başına 8-bit), onu OpenGL'e göndermeden önce 24bpp alacak şekilde genişletilir. Bu datanın gönderilmesi için **glTexImage2D** kullanılır ve ardından doku parametreleri setlenir. Normal haritası için de tamamen aynı şey yapılır.

```
//Load decal texture
IMAGE decalImage;
decalImage.Load("decal.bmp");
decalImage.ExpandPalette();

//Convert normal map to texture
glGenTextures(1, &decalTexture);
glBindTexture(GL_TEXTURE_2D, decalTexture);
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, decalImage.width,
decalImage.height,
0, decalImage.format, GL_UNSIGNED_BYTE, decalImage.data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

//Load normal map
IMAGE normalMapImage;
normalMapImage.Load("Normal map.bmp");
normalMapImage.ExpandPalette();

//Convert normal map to texture
```

```
glGenTextures(1, &normalMap);
glBindTexture(GL_TEXTURE_2D, normalMap);
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, normalMapImage.width,
normalMapImage.height,
0, normalMapImage.format, GL_UNSIGNED_BYTE, normalMapImage.data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Şimdi normalizasyon küp haritası oluşturulacaktır. Doku daha önce anlatıldığı gibi oluşturulur, ama "**GL_Texture_2D**" yerine, "**GL_Texture_CUBE_ARB**" kullanılır. Sondaki "ARB", bu simgenin bir ARB uzantısı olduğunu gösterir. **LoadTexture** rutinini kullanmak yerine, küp haritası için data üretilir ve yukarıda bahsedilen "**GenerateNormalizationCubeMap**" fonksiyonunda OpenGL'e gönderilir.

```
//Create normalisation cube map
glGenTextures(1, &normalisationCubeMap);
glBindTexture(GL_TEXTURE_CUBE_MAP_ARB, normalisationCubeMap);
GenerateNormalisationCubeMap();
glTexParameteri(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_R,
GL_CLAMP_TO_EDGE);
}
```

Başlatma (initiation) işlemi için bu kadar yeter. Şimdi bir şeyler çizmeye başlanabilir.

```
//Called to draw scene
void Display(void)
{
```

İlk olarak **renk** ve **derinlik** tamponları silinir (clear), ve **modelview** matrisi birim matrise resetlenir.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
```

Şimdi **modelview** matrisi kurulur. İlk olarak yukardaki diyagramda var olan gözleme transformasyonu için **view** matrisi uygulanır. Bu transformasyon esasen kamerayı manzarada belirli bir yere oturtur. **gluLookAt** kullanarak kamera (0.0, 10.0, 10.0) noktasına yerleştirilerek orijinden bakılır.

Daha sonra, model matrisi kurulur. Bu matris esasen ardından gözlem matrisinin geldiği model matristir. Ama, OpenGL bu transformasyondaki matrislerin oluşum sırasının tersinde belirlenmelerini gerektirmektedir. Torus Y ekseni etrafında döndürmek istendiğinden, statik değişken artırılarak rotasyon OpenGL'e gönderilir.

```
//use gluLookAt to look at torus
gluLookAt(0.0f,10.0f,10.0f,
0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f);

//rotate torus
static float angle=0.0f;
```

```
angle+=0.1f;
glRotatef(angle, 0.0f, 1.0f, 0.0f);
```

Şimdi dünya uzayından cisim uzayına dönüşüm yapan ters model matrisi hesaplanacaktır. Önce mevcut **modelview** matrisini koruma altına alınır ve daha sonra birim matrise resetlenir. Bir **a** açısı kadar rotasyonun tersi, aynı eksen etrafında **–a** kadar rotasyona eşdeğerdir. Bu rotasyon OpenGL'e gönderilir, ve bu matrisi elde etmek için **GetFloatv** kullanılır. Daha sonra koruma altına alınan **modelview** matrisi restore edilir.

```
//Get the inverse model matrix
MATRIX4X4 inverseModelMatrix;
glPushMatrix();
glLoadIdentity();
glRotatef(-angle, 0.0f, 1.0f, 0.0f);
glGetFloatv(GL_MODELVIEW_MATRIX, inverseModelMatrix);
glPopMatrix();
```

Şimdi ışık konumunun cisim uzayına dönüştürülmesi için biraz önce hesaplanan matris kullanılır.

```
//Get the object space light vector
VECTOR3D objectLightPosition=inverseModelMatrix*worldLightPosition;
```

Şimdi her köşe için teğet uzayı ışık vektörünün hesaplanması gerekir. Tüm köşeler üzerinden bir döngü yaparak bu vektör doldurulur.

Yukarıdaki diyagrama (tekrar) göz atılırsa, TSB matrisinin, cisim uzayındaki noktaları aldığı ve onları teğet uzayına dönüştürdüğü görülür.

Işık kaynağının cisim uzayındaki konumu bilinmektedir. Köşeden kaynağa giden vektörü elde etmek için ışık konumundan köşe konumu çıkarılır. Bunu teğet uzayına dönüştürmek için TSB matrisi kullanabilir. TSB matrisi aşağıdaki biçime sahiptir.

$$\begin{bmatrix} S_x & S_y & S_z \\ T_x & T_y & T_z \\ N_x & N_y & N_z \end{bmatrix}$$

Bu yüzden teğet uzayı ışık vektörünün ilk bileşeni $(S_x, S_y, S_z).(L_x, L_y, L_z)$ skaler çarpımı ile verilir, burada köşeden ışığa giden vektör L ile gösterilmiştir. Benzer şekilde, diğer iki bileşen de skaler çarpımlar ile verilir.

```
//Loop through vertices
for(int i=0; i<torus.numVertices; ++i)
{
VECTOR3D lightVector=objectLightPosition-torus.vertices[i].position;

//Calculate tangent space light vector
torus.vertices[i].tangentSpaceLight.x=
torus.vertices[i].sTangent.DotProduct(lightVector);
torus.vertices[i].tangentSpaceLight.y=
torus.vertices[i].tTangent.DotProduct(lightVector);
torus.vertices[i].tangentSpaceLight.z=
torus.vertices[i].normal.DotProduct(lightVector);
}
```

Pürüzlü torus iki safhada çizilir. Birinci olarak pürüzler çizilir ve daha sonra renk dokusu (color texture) ile çarpılır. Bu safhalar, farklı efektler için **on** ve **off** yapabilir. İlk olarak, gerekiyorsa pürüz geçişi (bump pass) çizilir.

```
//Draw bump pass
if(drawBumps)
{
```

Kullanılacak dokular birleştirilir (bind). Normal harita ünite-0 ile birleştirilir, ve 2D dokulaması yetkilendirilir. Daha sonra normalizasyon küp haritası, doku ünite-1 ile birleştirilir; küp-harita dokulaması yetkilendirilir ve mevcut doku ünitesi ünite-0 resetlenir.

```
//Bind normal map to texture unit 0
glBindTexture(GL_TEXTURE_2D, normalMap);
glEnable(GL_TEXTURE_2D);

//Bind normalisation cube map to texture unit 1
glActiveTextureARB(GL_TEXTURE1_ARB);
glBindTexture(GL_TEXTURE_CUBE_MAP_ARB, normalisationCubeMap);
glEnable(GL_TEXTURE_CUBE_MAP_ARB);
glActiveTextureARB(GL_TEXTURE0_ARB);
```

Şimdi kullanılacak köşe dizisi kurulacaktır. Konumlara işaret etmek için köşe işaretçisi (pointer) kurulacak ve yetkilendirecektir.

Doku koordinat dizileri her doku ünitesi için kurulur. Ünite-0'a ilişkin doku koordinat seti sadece s ve t doku koordinatlarıdır. Bu yüzden, normal haritası, tam standart doku haritası gibi torusa uygulanacaktır. Ünite-1'e ilişkin doku koordinat seti sadece teğet uzay ışık vektöründen ibarettir**.** Doku-1'i normalizasyon küp haritası olacak şekilde setlediğimizden, doku-1 her piksel için normalizeli teğet uzay ışık vektörünü içerecektir.

```
//Set vertex arrays for torus
glVertexPointer(3, GL_FLOAT, sizeof(TORUS_VERTEX),
&torus.vertices[0].position);
glEnableClientState(GL_VERTEX_ARRAY);

//Send texture coords for normal map to unit 0
glTexCoordPointer(2, GL_FLOAT, sizeof(TORUS_VERTEX), &torus.vertices[0].s);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

//Send tangent space light vectors for normalisation to unit 1
glClientActiveTextureARB(GL_TEXTURE1_ARB);
glTexCoordPointer(3, GL_FLOAT, sizeof(TORUS_VERTEX),
&torus.vertices[0].tangentSpaceLight);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glClientActiveTextureARB(GL_TEXTURE0_ARB);
```

Normal vektörünü doku-0'da ve normalizeli teğet uzay ışık vektörünü de ünite-1'de elde ettik. Bizim amacımızın aşağıdakini değerlendirmek olduğunu hatırlayınız.

```
color = max{l.n, 0}*diffuse
```

Diffuse ile çarpma ikinci geçişte (pass) yapılacaktır. Bu yüzden tex0 dot tex1'i değerlendirmemiz gerekir. Bunu başarmak için ARB_texture_env_combine ve

ARB_texture_env_dot3 uzantıları kullanılır. Birinci doku ünitesi, piksel (fragment) renginin yerine doku rengini, yani normal haritayı koyacaktır.

İkinci doku ünitesi, bunu alır ve texture-1 ile yani ışık vektörü ile skaler çarpar (dot). Bunların her ikisi de teğet uzayında bulunduğundan, skaler çarpımın bir anlamı vardır. Ayrıca bu sonuç renk olarak dışarı çıkarıldığından, [0,1] aralığına sıkıştırılacaktır (clamped). Bu yüzden, denklemimizin birinci terimini değerlendiririz. **Demoda 2'ye** basarak bu terimin neye benzediğini görebilirsiniz.

```
//Set up texture environment to do (tex0 dot tex1)*color
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB, GL_REPLACE);

glActiveTextureARB(GL_TEXTURE1_ARB);

glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB, GL_DOT3_RGB_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB, GL_PREVIOUS_ARB);

glActiveTextureARB(GL_TEXTURE0_ARB);
```

Dikkat edilirse dokularda, vektörlerin [0, 1] aralığında paketlendiğini ve onların [-1, 1] aralığına tekrar geri genişletilmediği görülebilir. Endişelenmeye gerek yoktur, çünkü "**DOT3_RGB_ARB**" doku ortamı (environment) sizin için bu görevi otomatik olarak yapacaktır.

Şimdi torus çizilebilir.

```
//Draw torus
glDrawElements(GL_TRIANGLES, torus.numIndices, GL_UNSIGNED_INT,
torus.indices);
```

Bu setup ile ihtiyacımız olan her şeyi yaptık, bu yüzden dokuları ve doku dizilerini (array) yetkisiz kılabiliriz (**disable**). Ayrıca doku ortamını standart "**modulate**"e resetleriz.

```
//Disable textures
glDisable(GL_TEXTURE_2D);

glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_CUBE_MAP_ARB);
glActiveTextureARB(GL_TEXTURE0_ARB);

//disable vertex arrays
glDisableClientState(GL_VERTEX_ARRAY);

glDisableClientState(GL_TEXTURE_COORD_ARRAY);

glClientActiveTextureARB(GL_TEXTURE1_ARB);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glClientActiveTextureARB(GL_TEXTURE0_ARB);

//Return to standard modulate texenv
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
}
```

Bir sonraki adım **diffus**e rengi çizmektir, bu bizim dokulu torusumuzdur. Hem **pürüzlü harita geçişi** (bump map pass) ve hem de **dokulu geçiş** çiziliyorsa, bu işlemler çarpma ile birleştirilebilir. Bu yüzden, çarpımsal **karıştırma** (blending) işlemi yetkilendirilmelidir.

```
//If we are drawing both passes, enable blending to multiply them together
if(drawBumps && drawColor)
{
//Enable multiplicative blending
glBlendFunc(GL_DST_COLOR, GL_ZERO);
glEnable(GL_BLEND);
}
```

Şimdi dokulu geçiş çizilebilir.

```
//Perform a second pass to color the torus
if(drawColor)
{
```

Eğer pürüzlü harita geçişi çizilmiyorsa, pürüzsüz haritalı torus ile pürüzlü haritalı torusu karşılaştırmamıza imkan sağlamak için standart OpenGL ışığını yetkilendireceğiz. **Demoda 3'e** basarak standart aydınlatılmış torusu görebilirsiniz.

```
if(!drawBumps)
{
glLightfv(GL_LIGHT1, GL_POSITION, VECTOR4D(objectLightPosition));
glLightfv(GL_LIGHT1, GL_DIFFUSE, white);
glLightfv(GL_LIGHT1, GL_AMBIENT, black);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, black);
glEnable(GL_LIGHT1);
glEnable(GL_LIGHTING);

glMaterialfv(GL_FRONT, GL_DIFFUSE, white);
}
```

Decal (color) doku, doku ünitesi-0 ile birleştirilir (bind) ve 2d dokulama yetkilendirilir.

```
//Bind decal texture
glBindTexture(GL_TEXTURE_2D, decalTexture);
glEnable(GL_TEXTURE_2D);
```

Şimdi köşe dizisi (array) ile doku koordinat dizisi yetkilendirilir. Bu kez, standart OpenGL ışıklandırma için **normal dizi** de (normal array) yetkilendirilir.

```
//Set vertex arrays for torus
glVertexPointer(3, GL_FLOAT, sizeof(TORUS_VERTEX),
&torus.vertices[0].position);
glEnableClientState(GL_VERTEX_ARRAY);

glNormalPointer(GL_FLOAT, sizeof(TORUS_VERTEX), &torus.vertices[0].normal);
glEnableClientState(GL_NORMAL_ARRAY);

glTexCoordPointer(2, GL_FLOAT, sizeof(TORUS_VERTEX), &torus.vertices[0].s);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

Şimdi torus çizilir ve OpenGL **durum**ları (ışıklandırma, dokulandırma vs.) resetlenir.

```
//Draw torus
glDrawElements(GL_TRIANGLES, torus.numIndices, GL_UNSIGNED_INT,
torus.indices);

if(!drawBumps)
```

```
glDisable(GL_LIGHTING);

//Disable texture
glDisable(GL_TEXTURE_2D);

//disable vertex arrays
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
}

//Disable blending if it is enabled
if(drawBumps && drawColor)
glDisable(GL_BLEND);
```

glFinish, tüm çizimleri tamamlamasını OpenGL'e söyler. **glutSwapBuffers**, ön ve arka renk tamponlarını takas eder, ve **glutPostRedisplay** de mümkün olur olmaz bir sonraki çerçeveyi (frame) çizmesinin istediğini glut'a söyler.

```
glFinish();
glutSwapBuffers();
glutPostRedisplay();
}
```

Tüm çizim bundan ibarettir. Şimdi **glut**'un birkaç tane daha ekstra fonksiyon üretmesine ihtiyaç vardır. Pencere yeniden boyutlandırılacağı (resize) zaman **Reshape()** çağrılır. Bu fonksiyon **viewport**u, tüm ekranı alacak şekilde resetler ve ayrıca doğru (correct) **aspect ratio** için projeksiyon matrisini de resetler.

```
//Called on window resize
void Reshape(int w, int h)
{
//Set viewport
if(h==0)
h=1;

glViewport(0, 0, w, h);

//Set up projection matrix
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective( 45.0f, (GLfloat)w/(GLfloat)h, 1.0f, 100.0f);

glMatrixMode(GL_MODELVIEW);
}
```

Bir tuşa basıldığı zaman **Keyboard()** fonksiyonu çağrılır.

```
//Called when a key is pressed
void Keyboard(unsigned char key, int x, int y)
{
```

Kullanıcı escape tuşlarsa programdan çıkılır.

```
//If escape is pressed, exit
if(key==27)
exit(0);
```

1-tuşuna basılırsa, pürüzlü harita geçişi ve renkli geçişin ikisi birden çizilir. Bu yüzden her iki boole değişkeni doğruya (true) setlenmelidir.

```
//'1' draws both passes
if(key=='1')
{
drawBumps=true;
drawColor=true;
}
```

2-tuşuna basılırsa, yalnız pürüzlü harita geçişi çizilmek istenir. Bu yüzden o boole değişkeni doğruya diğer geçiş değişkeni ise yanlışa (false) setlenmelidir.

```
//'2' draws only bumps
if(key=='2')
{
drawBumps=true;
drawColor=false;
}
```

3-tuşuna basılırsa, yalnız renkli geçiş çizilmek istenir, pürüzlü geçiş yetkisiz kılınır.

```
//'3' draws only color
if(key=='3')
{
drawBumps=false;
drawColor=true;
}
```

W-tuşuna basılırsa, mevcut çizim modu kafes (wireframe) moduna; F-tuşuna basılırsa, dolu poligon moduna setlenir.

```
//'W' draws in wireframe
if(key=='W' || key=='w')
glPolygonMode(GL_FRONT, GL_LINE);

//'F' return to fill
if(key=='F' || key=='f')
glPolygonMode(GL_FRONT, GL_FILL);
}
```

Son olarak standart **glut** işlerini yapan **ana fonksiyon** gelir. **glut** başlatılır, ve çift tamponlu RGB modlu (renk indeks modunun tersine) derinlik tamponlu bir pencere oluşturulur. Daha sonra yukardaki **Init** fonksiyonu çağrılır. Ardından **display**, **reshape** ve **keyboard** fonksiyonlarını nereden bulacağı **glut**'a söylenir. Bu işleme "**registering callback**" denir. Ve son olarak **glut**'a çizmeye başlaması söylenir.

```
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize(640, 480);
glutCreateWindow("Simple Bumpmapping");

Init();

glutDisplayFunc(Display);
glutReshapeFunc(Reshape);
glutKeyboardFunc(Keyboard);
glutMainLoop();
return 0;
}
```