2.1 OpenGL nedir ?

OpenGL dediğimiz şey, grafiksel donanıma arayüz olan bir yazılımdır.Bu arayüz, şimdiki araştırmalarıma göre yaklaşık olarak 250 tane birbirinden farklı olan OpenGL komutları içerir (bu komutların yaklaşık 200 tanesi OpenGL'e has olan ve kalan 50 tanesi ise OpenGL Utility Lbrary dediğimiz kütüphaneye aittir.) Bu komutları kullanarak objeleri yaratabilir ve interaktif bir şekilde üç boyutlu uygulamaları sağlayan işlemleri gerçekleştirebilirsiniz.

OpenGL, faklı donanım özelliklerine sahip platformlarda çalışabilen bir arayüz olarak tasarlanan, yani donanımdan bağımsız bir araçtır. OpenGL'in bu niteliğini geçekleştirebilmek için OpenGL'de kullanıcının dışarıdan bilgi girmesini sağlayan araçlar ve işletim sistemlerinde pencere yönetimini sağlayan komutlar yoktur. Bu yüzden programcı bulunduğu platformu iyi tanımalı ve kullandığı işletim sistemi komutlarını iyi bilmelidir. Aynı şey üç boyutlu nesneler için de geçerlidir.Yani söylemek istediğim şey, OpenGL'in üç boyutlu nesneler için sağlayacağı yüksek seviyeli komutları yoktur. Sizin yapmanız gereken şey OpenGL'in sağladığı ilkel (primitive) geometrik nesneleri kullanarak istediğiniz modelde iki veya üç boyutlu nesneleri yaratmanızdır. İlkel nesneler olarak bahsettiğim şeyler nokta, çizgi, poligon gibi objelerdir. Bu özellikleri sağlayabilen gelişmiş bir kütüphane OpenGL'in üstüne inşa edilmiştir. OpenGL Utility Library (GLU) quadric yüzeyleri, NURBS eğrileri ve yüzeyleri gibi bir çok özzelliği sağlamaktadır. GLU, her OpenGL implementasyonunun bir gerçekleştirim standardıdır.

2.2 Küçük bir OpenGL kodu

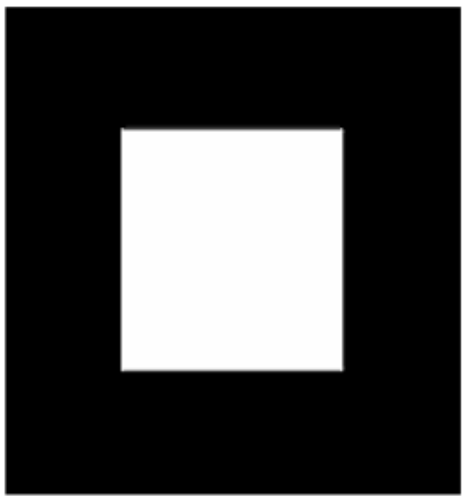
OpenGL grafik sistemi ile bir çok şey yapabildiğinizden dolayı, bir OpenGL programı karmaşık olabilir.Fakat, programın temel yapısı basit olabilir. OpenGL'in nasıl

reder işleminde kontrol görevini sağlayan belirli durumları ilklemek ve render edilecek olan nesneleri ise belirlemek OpenGL'in görevleri arasında yer alır.

OpenGL koduna bakmadan önce, birkaç terimi incelemekte fayda var. Mesela biraz önce kullandığım *render* kelimesinin anlamı bilgisayarın modellerden yarattığı resimleri işleme anlamına gelir. Bu modeller ve nesneler geometrik ilkel elemanlardan yaratılırlar (poligonlar, çizgiler ve noktalar).

Sonuç olarak üretilen resim yani render edilmiş olan resim ekranınızda çizilmiş olan pikselleri içerir. Pikselin anlamı, donanım tarafından ekrana koyulan en küçük görünür elementtir. Piksellerle ilgili bilgiler, memorideki bitplaneler içinde düzenlenmiştir. Bitplane bir memori bölgesidir. Bu bölge ekran üzerindeki her bir piksel ile ilgili tek bitlik bilgi tutmaktadır.Örnek olarak, bit herhangi bir pikselin nasılı kırmızı renkte olup olamayacağı ile ilgili bilgiyi tutar.Bitplanelerin kendileri ise bir framebuffer içerisinde organize olmuştur.Bu framebuffer, ekran üzerindeki piksellerin renk kontollerini ve yoğunlukarı ile ilgili ihtiyaç duyduğu bilgileri tutmaktadır.

Bir OpenGL kodunu genel yapısını incelersek, Örnek 2-1, arka planı siyah olan beyaz renkli bir karenin render edişini Resim 2-1' de göstermektedir.



Resim 2-1

Örnek 2-1

```
#include <istenile başlık dosyaları>

int main(int argc, char** argv) {

      PencereİlklemeİşlemleriYap();

      glClearColor(0.0, 0.0, 0.0, 0.0);
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(1.0, 1.0, 1.0);
      glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
      glBegin(GL_POLYGON);
            glVertex3f(0.25, 0.25, 0.0);
            glVertex3f(0.75, 0.25, 0.0);
            glVertex3f(0.75, 0.75, 0.0);
            glVertex3f(0.25, 0.75, 0.0);
      glEnd();
      glFlush();

      PencereyiGüncelleVeKontrolleriYap();

      return 0;
}
```

main() fonksiyonun ilk satırı, ekranda gözükecek olan pencere ile ilgili hazırlık işlemleri yapmaktadır. **PencereİlklemeİşlemleriYap()** fonksiyonu, OpenGL'e ait olamayan işletim sistemine has pencere sistemi ile ilgili rutinleri içeren bir yer tutucudur. Sonraki iki satır ise pencerenin arka planın siyah olarak temizleyen OpenGL komutlarıdır: **glClearColor()**, pencerenin hangi renk ile temizleneceğini belirler ve **glClear()** komutu ise pencereyi temizler. **glColor3f()** komutu çizilecek olan nesnenin rengini belirtir. Bu

örnek içerisinde çizilmiş olan karenin rengi beyaz olarak belirlenmiştir.Bu noktadan itibaren tüm nesneler bu komut tarafından yeni bir renk ayarlaması olana kadar bu rengi kullanırlar. Sonraki OpenGL komutu **glOrtho()**' dur. Bu komut pencerede nesne çizilmeden önce, koordinat sistemini belirtir. **glBegin()** ve **glEnd()** komutları tarafından sarılmış olan komutlar, çizilecek nesneyi tanımlamaktadır. Örnek içerisinde göreceğiniz gibi dört köşeli bir poligon çizilmiştir. Poligonu köşeleri, **glVertex3f()** komutları tarafından tanımlanmaktadır. **glVertex3f()** komutu koordinat sistemi ile alakalı üç tane parametre almaktadır. Bu parametreler x, y ve z dir. Bu örnek içerisinde z'nin değeri 0 olarak belirlenmiştir. Son olarak **glFlush()** komutu çizim komutlarını çalıştırır. **PencereyiGüncelleVeKontrolleriYap()** rutini ise pencere ile ilgili güncellemeri yapmaktadır ve diğer processler arasındaki mesaj alış-verişini dinlemektedir.

Aslında örnek 2-1, basit olarak OpenGL komut yapısını vermektedir. Ayrıntılara raporumun sonraki sayfalarında değineceğim.

2.3 OpenGL Komut Sentaksı

Örnek 2-1' de ki basit OpenGL programını incelediyseniz, her OpenGL komutunun önüne **gl** ön eki getirilmektedir. Aynı şekilde, tanımlanmış OpenGL sabitleri de GL ön eki kullanılmıştır (örneğin GL_COLOR_BUFFER_BIT gibi…). Dikkat edeceğiniz gibi bazı ekler de bu komutlara birer anlam katmak için kullanılmışlardır. (örneğin **glColor3f()** komutunun son eki olan **3f**). Yaratılmış olan nesnenin o anki rengini belirleyen komutun yani **glColor3f()** komutunun **Color** parçasına bakarak anlamak yeterlidir sanırım. Fakat, komutların farklı sayıda argüman almalarını sağlamak için, farklı komutlar tanımlanmıştır. Mesela **glColor3f()** komutunda aklınıza takılan 3 sayısı, bu komutun üç tane argüman aldığını bildirmektedir. Komutun **f** karakteri, alınacak olan parametrelerin **float** tipinde olmaları gerektiğini anlatmaktadır. Aslında kısacası, OpenGL komutlarını yazarken, fonksiyonların aşırı yüklenmesi söz konusu değildir. Bunu dikkate almakta fayda var.

Bazı OpenGL komutları, sekiz farklı tipte argüman almaktadır. Bir sonraki sayfadaki tablo, OpenGL' in ISO C gerçekleştirimi için belirlenmiş veri tiplerini önek olarak tanımlamaları ile beraber kullanımını göstermektedir. Bu veri tiplerini programlama dilinde kullanmakta fayda var, çünkü kullanılacak olan komutların argüman tipleri, programla dilinde kullanılan veri tipleri ile uyuşmayabilir. Bu yüzden, OpenGL komutlarının taşınabilirliği söz konusu ise bu komutları kullanmakta fayda var.

Tablo 2-1 Komut Önekleri ve Argüman Veri Tipleri

Önek	**Veri Tipi**	**C programlama dilinde Kullanılan Tipler**	**OpenGL Tip Tanımı**
B	8-bit integer	signed char	GLbyte
S	16-bit integer	Short	GLshort
İ	32-bit integer	int veya long	GLint, GLsizei
F	32-bit floating-point	Float	GLfloat, GLclampf
D	64-bit floating point	Double	GLdouble, GLclampf
Ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
Us	16-bit unsigned integer	unsigned short	GLushort
Ui	32-bit unsigned integer	unsigned int veya unsigned long	GLuint, GLenum, GLbitfield

Aşağıdaki iki komutu incelersek

```
glVertex2i(1, 3);
glVertex2f(1.0, 3.0);
```

her iki komut ta aynı işleve sahiptir. Fakat ilki, vertex'in koordinatları argüman olarak 32-bit integerlar ve diğe komut ise argüman olarak float tipindedir.

Bazı OpenGL komutları, son ek olarak **v** karakterini alır. Bu karakter, programcıya parametre olarak alınacak şeyin bir vectörü veya bir diziyi işaret eden bir işaretçi olduğunu bildirir.Birçok komutun parametre olarak vector alan veya almayan versiyonları bulunmaktadır. Aşağıdaki satırlar **glColor*()** komutunun vektörlü ve vektörsüz olarak parametre alımını göstermektedir.

```
glColor3f(1.0, 0.0, 0.0);

GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

Son olarak, OpenGL veri tipi olarak GLvoid'i tanımlar. Bu veri tipi dizileri işaret eden pointerları, argüman olarak alan OpenGL komutları için kullanılmaktadır.

2.4 GLUT, OpenGL Hizmet Aracı (OpenGL Utility ToolKit)

Bildiğiniz gibi, render komutlarını OpenGL içermektedir, fakat bir pencere sisteminden ve bir işletim sisteminden bağımsız olarak tasarlanmıştır. Sonuç olarak ne açılan pencerelere ait, ne de klavye ve mause olaylarına ait komutlar içermemektedir. Penceresiz bir grafik programı yazmak imkansızdır ve çok ilgi çeken programlar dışarıdan kullanıcı bilgisi alırlar veya işletim sisteminin ve pencere sisteminin hizmetlerinden yaralanırlar. GLUT kütüphanesini kullanarak, pencere yaratma dışarıdan bilgi alma gibi fonksiyonları kolayca kullanabiliriz. Raporumun ileriki sayfalarında Windows işletim sistemini pencere sistemi gerçekleştirimine değineceğim.

OpenGL' in ne olduğunu açıklarken, OpenGL'in basit ilkel nesneleri çizebilecek komutlarla sınırlı olduğunu söylemiştim, fakat GLUT sayesinde karmaşık üç boyutlu nesnelerimizi kolaylıkla yaratabiliriz.

2.5 Pencere Yönetimi

Pencere yaratabilmek için gerekli olan beş rutin yerine getirlmelidir.

- **glutInit**(int **argc*, char*** *argv*) fonksiyonu GLUT'ı ilkler ve komut satırı argümanlarını işlemektedir. Diğer GLUT rutinlerinden önce bu komutun yazılması zorunludur.

- **glutInitDisplayMode**(unsigned int *mode*) renk modunu belirlemektedir. Bu renk modu RGBA veya color-index mod olabilir. Ayrıca pencerenin single veya double buffered olacağını belirlersiniz. Eyer bir pencerenin RGBA ve double buffered modda olmasını istiyorsanız, bu fonksiyonu **glutInitDisplayMode**(GLUT_RGB | GLUT_DOUBLE) olarak çağırmalısınız.

- **glutInitWindowPosition**(int *x*, int *y*) fonksiyonu grafik penceresinin sol-üst köşesini baza alarak, ekrandaki yerini belirler.

- **glutInitWindowSize**(int *width*, int *height*), pencerenizin büyüklüğünü ayarlar. (piksel birimi olarak)

- int **glutCreateWindow**(char* *string*), OpenGL conteksli bir pencere yaratır. Yeni pencereye ait tek yani eşsiz bir satı dönderir. Not: glutMainLoop() çağrılana kadar pencere yaratılamaz.

2.6 Görüntüyü Geri Çağırma (The Display Callback)

glutDisplayFunc(void (**func*)(void)) fonksiyonu göreceğiniz ilk ve en önemli fonksiyon çağırma fonksiyonudur. Nezaman GLUT, pencerenizin içeriğinin yeniden

gösterileceğine ihtiyaç duyduğunu belirlerse, **glutDisplayFunc**() tarafından çalıştırılacak fonksiyon çağrılır. Anlaşılacağı gibi, glutDispalyFunc()' a parametre olarak aktarılan bir pointer to function tipinde bir işaretçidir. Bu nedenden ötürü, pencerenizi yeniden çizmeniz için gerekli gördüğünüz rutinleri çağıracağınız fonksiyon içerisine yani void (*func)(void) poiner to function içerisine koyun.

2.7 Renkli bir Üçken

Yapacağınız en son şey ise **glutMainLoop**(void) fınksiyonun çağırmak olacaktır. Bunu sayesinde yarattığınız pencere ekranda gösterilir. Bu fonksiyon GLUT olay işlem döngüsüne (the GLUT even processing loop) girer. Program içerisinde sadce bir kez çalıştırılmalıdır. Fonksiyon bir kere çağrıldığı zaman, fonksiyondan geri dönülmez. Kayıt edilen fonksiyonları çalıştırmak için bekleyecektir.

Aşağıdaki örnek 2-2 GLUT tarafından yaratılan bir pencerede renkli bir üçken göstermektedir. Programın verimliliğini arttırmak için, sadece bir kez çağrılması gereken fonksiyonlar, **init**() fonksiyonu içerisinde bulunmaktadır. Diğer render işlemleri ise **display**() fonksiyonu içerisine konulmuştur (üçkeni çizen fonksiyon) ve bu fonksiyonun ismi parametre olarak **glutDispalyFunc**() fonksiyonuna aktarılacaktır.

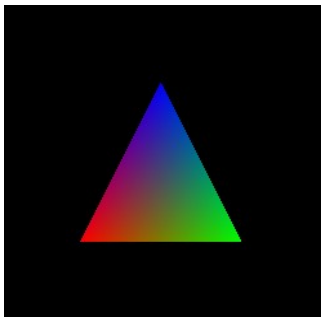
Örnek 2-2 GLUT kullanılarak yapılan basit bir OpenGL programı

```
#include <GL/glut.h>

void display(void) {
/* tum pikselleri temizle */
      glClear(GL_COLOR_BUFFER_BIT);
/* koordinatlari (0.25, 0.25),
 *  (0.75, 0.25), (0.50, 0.75) olan ucken ciz
 */
      glBegin(GL_POLYGON);
```

```
            glColor3f(1.0, 0.0, 0.0);
            glVertex2f(0.25, 0.25);
            glColor3f(0.0, 1.0, 0.0);
            glVertex2f(0.75, 0.25);
            glColor3f(0.0, 0.0, 1.0);
            glVertex2f(0.50, 0.75);
      glEnd();
      glFlush();
      return;
} // end of void display(void)

void init(void) {

/* arka plan rengini belirle */
      glClearColor(0.0, 0.0, 0.0, 0.0);
      glShadeModel(GL_SMOOTH);

/* bakis degerlerini ilkle */
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();

      return;
} // end of void init(void)

int main(int argc, char* argv[]) {

      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
      glutInitWindowSize(250, 250);
      glutInitWindowPosition(100, 100);
      glutCreateWindow(argv[0]);
      init();
      glutDisplayFunc(display);
      glutMainLoop();
```

```
        return 0;
} // end int main(int, char**)
```

Programın ekrandaki çıktısı aşağıdaki gibi olacaktır.



Resim 2-2

2.8 Dışarıdan Input Girişini Tutmak

Belli bir olay olduğunda, bu olayları tutabilmek için GLUT, bize aşağıdaki fonksiyonları sağlamıştır.

- **glutReshapeFunc**(void (**func*)(int *w*, int *h*)) fonksiyonu pencerenin büyüklüğü değişeceği zaman çalışır.

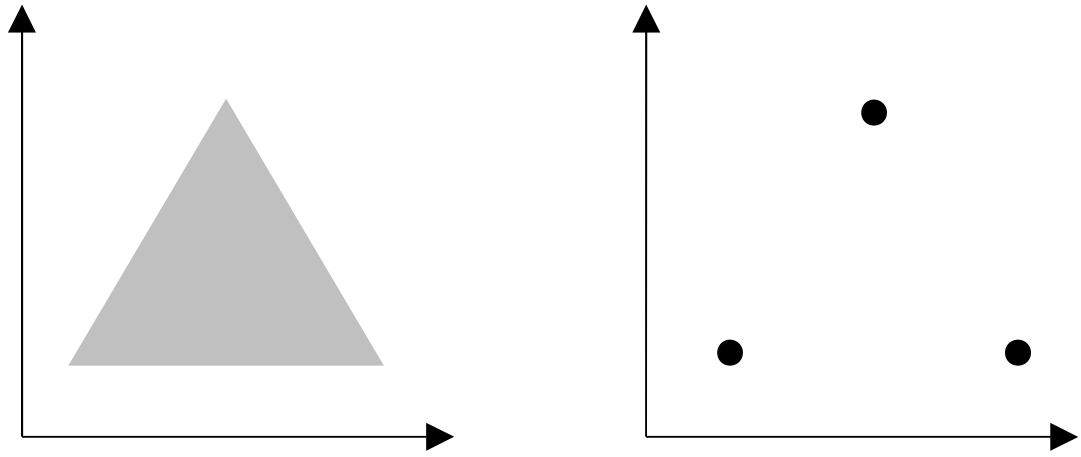
- **glutKeyboardFunc**(void (**func*)(unsigned char *key*, int *x*, int *y*)) ve **glutMouseFunc**(void (**func*)(int *button*, int *state*, int *x*, int *y*)) fonksiyonları, sizin mouse veya klavye ile ilgili işlemlerinizi tutar. Klavyenin veya farenin herhangi bir tuşuna bastığınızda veya basık tuttuğunuzda bu olaylar meydana gelir.
- **glutMotionFunc**(void (**func*)(int *x*, int *y*)) fonksiyonu, fare haraket ettmesi veya farenin düğmelerine basılması gibi olayları tutar.

- **glutIdleFunc**(void (**func*)(void)) , eyer arka planda bir olay olmuyorsa yani dışarıdan bir bilgi girişi yok ise, kendisine parametre olarak aktarılan fonksiyon ismini çalıştırır.Eyer **glutIdleFunc**() fonksiyonun inaktif yapmak istiyorsanız fonksiyona NULL değerini aktarın.

2.9 OpenGL İlkel Geometrik Nesneleri (OpenGL Primitives)

İlkel geometrik nesnelerden anlatmak istdiğimiz şey, OpenGL'in çizebildiği basit olan ilkel noktalar, çizgiler, poligonlar ve üçkenler gibi nesnelerdir. Bu nesnelerin birer koordinat bilgileri vardır. Bu koordinat bilgilerine vertex demekte fayda görüyorum. OpenGL bu nesnelerin vertex bilgileri sayesinde bu ilkel olan geometrik şekilleri çizebilmektedir. Fakat çizilecek olan nesnenin nokta, çizgi veya poligon olup olmadığını OpenGL'e bildirmek gerekir. Bu bildirim **glBegin**(Glenum *mode*) fonksiyonu tarafından gerçekleştirilir. Öncelikle nesnenin koordinat bilgileri OpenGL' e aktarılamadan önece **glBegin**() fonksiyonu çağrılıp çizim modu belirlenir. Ardından vertex bilgileri aktarılıp nesnenin çizimini ve çizme modunun bittiğini göstermek için glEnd() fonksiyonu kullanılır. Aşağıdaki **glBegin**() ve **glEnd**()'in kullanımını göstermektedir.

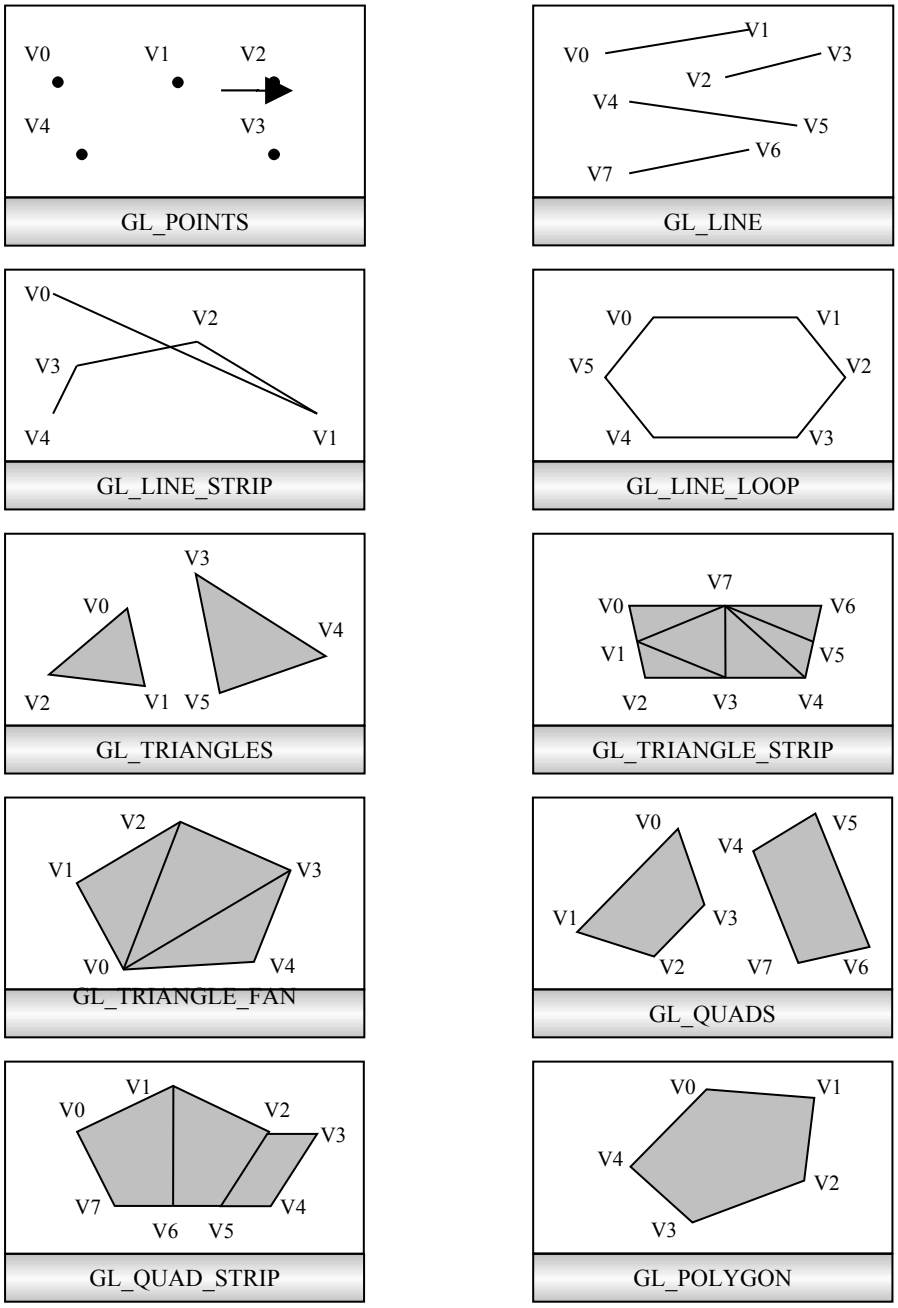
```
glBegin(GL_POLYGON);
      glVertex2f(0.25, 0.25);
      glVertex2f(0.75, 0.25);
      glVertex2f(0.50, 0.75);
glEnd();
```



GL_POLYGON GL_POINTS

Resim 2-3

Önceki sayfada gördüğünüz gibi, OpenGL'in çizeceği mode **glBegin** tarafından poligon olarak belirtilmiştir. Bunun için **glBegin**() fonksiyonuna parametre olarak aktarılan parametre GL_POLYGON dur. sayfada gördüğünüz resim 2-3' ün sol tarafında poligon modu geçerli olarak bir üçken çizilmiştir. Eğer **glBegin**()'e aktarılacak olan mod GL_POINTS olsaydı, aynı resmin sağında bulunan üç tane nokta gözükecekti. Resim 2-4, çizim modlarını göstermektedir.



Resim 2-4

Örnekte, gördüğünüz gibi, **glEnd**() fonksiyonunun parametresi yoktur. OpenGL'e belirttiğimiz modda, çizdiğimiz ilkel geometrik nesnenin çizimini birtirmesi gerektiğini haber vermekten başka, bir özel işlevi yoktur.

Şunu belirtmek gerekirse, **glBegin()** / **glEnd**() bloğu içerisinde, başka bir **glBegin**() / **glEnd**() bloğu bulunamaz. Ayrıca, tüm OpenGL fonksiyonları, bu bloğun içerisinde kullanılamayabilirler. Gerçekte, sadece **glVertex**(), **glColor**(), **glIndex**(), **glNormal**(), **glTextCoord**(), **glEvalCoord**(), **glEvalPoint**(), **glMaterial**(), **glEdgeFlag**(), **glCallList**() ve **glCallLists**() fonksiyonlarının çeşitli varyasyonları **glBegin**() / **glEnd**() bloğu içerisinde kullanılabilirler.

2.10 Vertexleri Belirleme

OpenGL ile her geometrik nesne düzgün bir sıraya sahip olan koordinat bilgileri ile ifade edilebilirler. Nesnenin bir koordinat bilgisini yani vertex'ini belirlemek için **glVertex*()** komutunu kullanmamız gerekir.

```
void glVertex[2, 3, 4][s, i, f, d][v]( Type coords );
```

Bu fonksiyon, nesnenin özel bir koordinat bilgisine göre en fazla dört parametre (parametre bilgisi olarak (x, y, z ,w) olabilir) ve en az iki parametre parametre (parametre bilgisi olarak (x, y)) alabilir.Eyer z veya w gibi parametreleri kullanmadığınız bir **glVertex*()** fonksiyonu kullanıyorsanız, z'nin değeri 0 ve w'nun değeri 1 olarak varsayılacaktır. **glVertex*()** fonksiyon çağrısı sadece **glBegin()** / **glEnd()** bloğu içerisinde verimlidir.

Aşağıdaki örnek **glVertex*()** kullanımını basitçe göstermektedir.

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);

Gldouble dVect[] = {5.0, 9.0, 1992.0};
glVertex3dv(dVect);
```

İlk satırdaki komut, üç boyutlu koordinat sistemine göre koordinat bilgisi (2, 3, 0) olan bir vertex'i göstermektedir.Bildiğiniz gibi bu komut, sadece iki parametre almaktadır. Yani koordinat biligileri olarak x = 2, y = 3 tür. Kullanmadığımız z'nin değeri 0 ve w'nun değeri 1 dir. İkinci komut koordinat bilgileri (0.0, 0.0, 3.1415926535898) değerlerine sahip olan bir vertex'i gösterir. Üçüncü satırdaki komut ise parametre olarak (2.3, 1.0, -2.2, 2.0) alır. Dikkat ederseniz w'nun değeri 2.0 dır ve bu yüzden demin belirttiğim parametre bilgilerini yeniden incelersek w değeri x, y ve z değerlerini bölecektir. Yani parametre aktarım (x/w, y/w, z/w) olarak aktarılır. Sonuç olarak OpenGL'in gördüğü koordinat bilgiler (2.3, 1.0, -2.2) değilde (1.15, 0.5, -1.1) olacaktır.
Son satırda ise, fonksiyona bir dizi parametre olarak atanmıştır.

Bazı donanımlarda **glVertex*()** fonksiyonunun vektör formlarını kullanmak daha etklidir çünkü sadece tek bir parametre, grafik alt sistemine geçirilmeye ihtiyaç duyar. Özel bir donanım, tek bir seri halinde, tüm koordinat bilgilerini gönderme yeteneğine sahip olabilir. Buda size iyi bir performans sağlar.

2.11 Nokta Büyüklüğünü Değiştirme

Nokta büyüklüğünü değiştirmek için aşağıdaki fonksiyonu kullanmak gerkir.

```
void glPointSize( GLfloat size );
```

Varsayılı olan nokta büyüklüğünün değeri yani size 1.0'a eşittir. Farklı nokta büyüklükleri için ekrana çizilecek olan piksel toplulukları antialiasing durumunun aktif veya pasif olamasına bağlıdır.(Antialiasing çizilecek olan nokta veya çizgileri pürüssüz bir şekilde çizmek için kullanılacak olan bir tekniktir.) Eyer antialiasing aktif değil ise ve verilen büyüklük değerleri virgüllü bir sayı ise bu sayı, tam sayıya dönüştürülür ve ekranda çizilecek olan noktalar kare şeklinde çizilirler. Mesela büyüklüğü 1.0 olan bir nokta, 1'e 1 genişliğinde bir piksel olarak çizilir.Eğer büyüklüğü 2.0 olan bir nokta enine vo boyuna 2 piksel olarak kare şeklinde çizilecektir

Antialiasing'in aktif olduğu durumlarda, çizilecek plan noktaların pikselleri dairesel bir şekilde çizilir buda nesnenin keskin köşelerinde yumuşak geçişler sağlayacaktır yani çizilecek olan nesne pürüssüz olacaktır. Bu modda büyüklüğü tamsayı olmayan bir sayı yuvarlanmayacaktır. Programın çalışması esnasında seçilmiş olan nokta büyüklüğünü **glGet**() fonksiyonuyla eldeedersiniz. Bu fonsiyona GL_PONIT_SIZE parametresini aktardığınızda noktanın büyüklüğü bulunur.

2.12 Çizgi Kalınlığını Değiştirme

Aşağıdaki fonksiyonu kullanarak bir çizginin kalınlığını belirleyebilirsiniz.

```
void glLineWidth( Glfloat width );
```

width yani genişlik 0.0 dan büyük olmalıdır ve varsayılı olan değeri 1.0 dır. Noktaların çiziminde antialiasing etkili olduğu gibi aynı şey çizgiler için de geçerlidir. Antialiasing olmadan, çizilecek olan çizgilerin kalınlıkları, sırasıyla 1, 2 ve 3 ise, piksel genişlikleri, yine aynı sırada, 1, 2 ve üç piksel genişliğinde olacaktır. Önemle belirtmem gerekirse, ekranınız düşük çözünürlükte çalışıyorsa piksel genişliği 1.0 olan çizgiler geniş olarak gözükecektir eğer çözünürlük yüksek ise çizilecek olan çizgi gözükmemeye yakın olabilir.

2.13 Pencere Temizleme

Bir resmi bilgisayar ekranına çizmek ile kağıda çizmek arasında çok fark vardır. Bilgisayarın belleğinde son çizdiğiniz resim tutulur. Yeni bir resim çizmeden önce arka planı temizlemeye ihtiyaç duyarsınız, tabi bunu yapmak için öncelikle arka planı temizleyecek renge sahip olamanız gerekir. Kullandığınız renk gerçekleştireceğiniz uygulamaya bağlıdır.

Bilmemiz gereken şey, bitplane olarak biline grafiksel donanımda her pikselin renginin nasıl tutulduğudur. Renklerin tutulmasına dair iki yönten mevcut.Birinci yöntem, bir pikselin kırmızı, yeşil, mavi ve alfa değerlerinin direk olarak bitplane içerisinde korunması ve diğer bir yöntem ise renk index modudur.

Aşağıdaki komut satırları RGBA modunda pencereyi siyaha temizlemektedir.

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

İlk satırdaki komut temizleme rengini siyah yapmaktadır ve ardından gelen komut ise seçilmiş olan temizleme rengini dikkate alarak pencereyi temizler. **glClear**() komutuna aktarılmış olan tek parametre hangi bufferın temizleneceğini OpenGL'e bildirmektedir. Bu durumda resmin saklı tutulduğu renk bufferı program tarafından temizlenecektir.

Aşağıda **glClear**() ile **glClearColor**() fonksiyonlarının prototipleri ve komutların işlevleri açıklanmaktadır.

- void **glClear**(Glbitfield *mask*) fonksiyonu mask olarak belirtilmiş parametreye göre bufferları temizler.

- void **glClearColor**(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*) fonksiyonu RGBA modda temizlemek için kullanılacak olan rengi belirler. Varsayılan temizleme değerleri (0.0, 0.0, 0.0, 0.0) olduğundan dolayı siyahtır.

2.14 Nesnenin Rengini Belirleme

OpenGL ile, çizilecek olan bir nesnenin şeklini ifade etmek, nesnenin rengini ifade etmekten bağımsızdır. Nezaman bir nesne çizileceğinde, o anki renk şeması kullanılarak nesne çizilir. Genel olarak OpenGL programcısının yapacağı ilk iş rengi veya renk şemasını ayarlamak ve sonra ise nesneyi çizmektir. O anki renk veya renk şeması değişmeyinceye kadar tüm nesneler bu rengi veya renk şemasını kullanrak çizilirler.

Aşağıdaki örneği incelerseniz, çizilecek olan nesnelerin yalancı kodunu göreceksiniz.

```
SetCurrentColor(RED);
DrawObject(A);
DrawObject(B);
SetCurrentColor(GREEN);
SetCurrentColor(BLUE);
DrawObject(C);
```

Yukarıdaki örnekte A ve B nesneleri kırmızı olarak ve C nesnesi ise mavi olarak çizilecektir. Dördüncü satırdaki SetCurrenColor(GREEN) komutu gereksiz yere kullanılmıştır.

Bir rengi belirlemek için, **glColor3f**() komutunu kullanın. Bu fonksiyon 0.0 ile 1.0 arasında üç tane argüman almaktadır. Parametreleri sırasıyla red, green ve blue dur. Bu parametrelerin değerlerinin karışımından yeni renkler yaratabilirsiniz. 0.0 değeri bileşeni yani rengi kullanmayacağımız anlamına gelmektedir. 1.0 ise tam tersidir. Mesela **glColor3f**(1.0, 0.0, 0.0); fonksiyonunu kulandığımızda sistemin bir nesneyi parlak kırmızı bir renkte çizeceği anlamına gelmektedir yeşil ve mavi renkler kullanılmaz. Tüm sıfırlar rengi siyah, tüm birler ise rengi beyaz yapar. Tüm renk değerle-

rini 0.5 olarak ayarlarsak rengimiz griye kaçacaktır. Bir sonraki sayfada sekiz tane renk ayarlaması gösterilmektedir.

```
glColor3f(0.0, 0.0, 0.0);     siyah
glColor3f(1.0, 0.0, 0.0);     kırmızı
glColor3f(0.0, 1.0, 0.0);     yeşil
glColor3f(1.0, 1.0, 0.0);     sarı
glColor3f(0.0, 0.0, 1.0);     mavi
glColor3f(1.0, 0.0, 1.0);     magenta
glColor3f(0.0, 1.0, 1.0);     çiyan
glColor3f(1.0, 1.0, 1.0);     beyaz
```

2.15 Koordinat Çevrimleri

Koordinat çevrimleri üç boyutlu bir ortamda varlıkları gerçekleştirme, döndürme ve haraket ettirme gibi olanaklar sağlar. Bu çevrimler, nesneleri ve nesnelerin koordinat sitemlerini direk olarak modifiye eder.(degistirir) Mesela, modelinizin koordinat siteminde bir
döndürme işlemi yaptığınızda, modelinizi çizeceğiniz zaman, o da değiştirmiş olduğunuz koordiant eksenine göre dönecektir.

Nesneler çizilmeden önce, üç tip transformasyondan yani dönüşümden geçer.

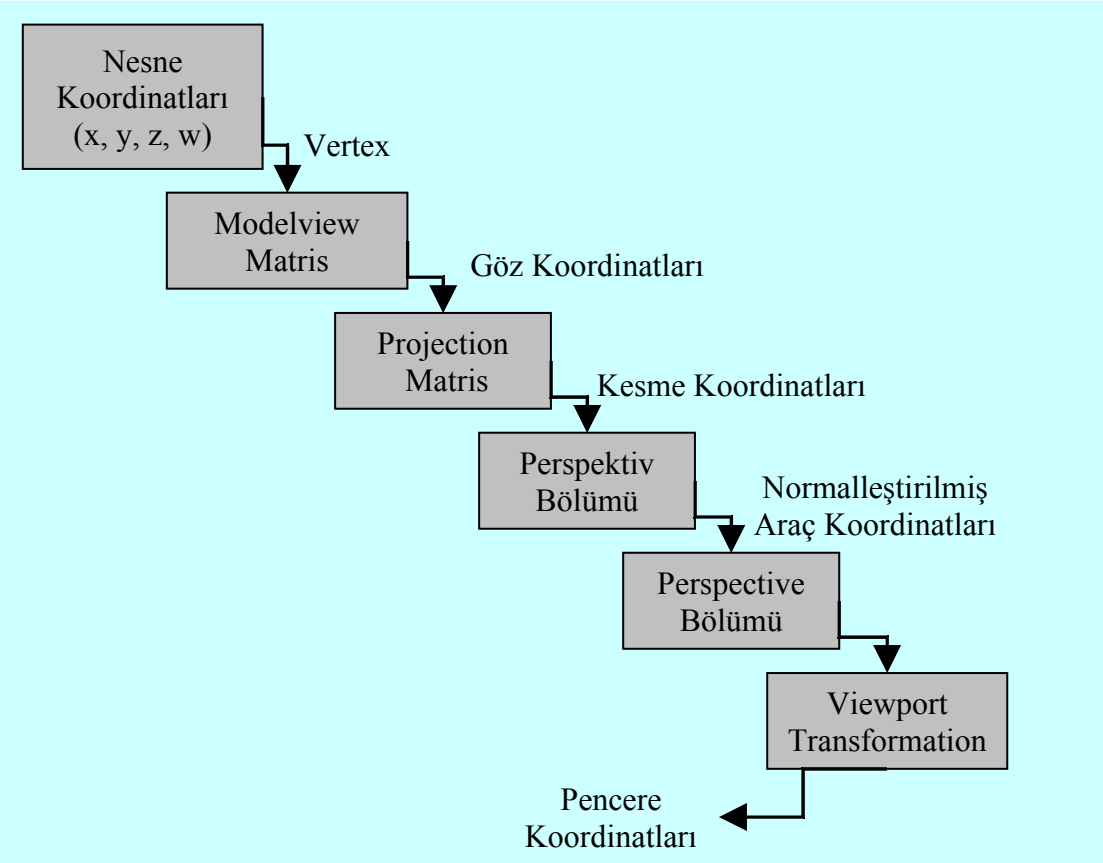
- **Viewing transformation:** Kameranın yerinin belirlendiği transformasyondur.

- **Modeling transformation:** Nesnenin ekranda hareketini sağlayan transformasyondur.

- **Projection transformation:** Kameranın bakış hacmini belirleyen transformasyondur.

Viewing transformation, yazdığınız kod içerisinde modelling transformation'dan önce gelmelidir fakat çizim yapmadan önce projection ve viewport transformationları yani dönüşümlerini belirleyebilirsiniz.

Aşağıdaki şekil resim 2-5 bilgisayarınızda hangi işlemlerin sırayla olduğunu göstermektedir.



Resim 2-5 Vertex Dönüşümünün Safhaları

Viewing, modeling ve projection transformationlarını belirlemek için, 4X4 lük bir **M** matris tanımlaması yapmanız gerekir.Bu matris, istenilen transformasyonları gerçekleştirebilmek için ekrandaki her bir v verteksinin koordinatlarıyla çarpılır.

$$v' = \mathbf{M}v$$

(Not: vertexlerin dört tane koordinatı olduğunu önemle vurgulamam gerekir. Fakat bir çok durumda w, 1'e eşit ve iki boyutlu koordinat sistemi için z değeri 0'a eşittir.)Viewing ve modeling transformasyonları otomatik olarak yüzey normal vektörlerine ve ek olarak vertexlere uygulanmaktadır.(Normal vektörleri sadece göz koordinatlarında kullanılır.Bu, normal vektörünün vertex verisi ile olan ilişkisini uygun bir şekilde sağladığın temin etmektedir.

Belirttiğiniz viewing ve modelling transformasyonları, modelview matrisini oluşturmak için bir araya getirilirler. Göz koordinatlarını vermek için, gelen nesnenin koordinat bilgilerine bu matris uygulanmaktadır.Eğer nesnenin ekranda gözükmeyen kısımlarını silmek gerekiyorsa, kesme yüzeyleri uygulanır. Bundan sonra, OpenGL, projection matrisini kesme koordinatlarını verebilmek için uygulayacaktır. Bu sayede kameranın nesneyi gösteremediği kısımlar kesilecektir, yani gözükmeyecektir. Bu noktadan sonra perspektif bölümü, normalleşmiş araç koordinatlarını üretebilmek için koordinat değerlerini w ile bölerek yerine getirilir. Son olarak ise, viewport transformasyonu uygulanarak, dönüştürülmüş koordinatlar pencere koordinatlarına çevrilir. Böylece viewport boyutları üzerinde, resim ile oynamak için, çeşitli manipulasyonlar uygulayabilirsiniz.

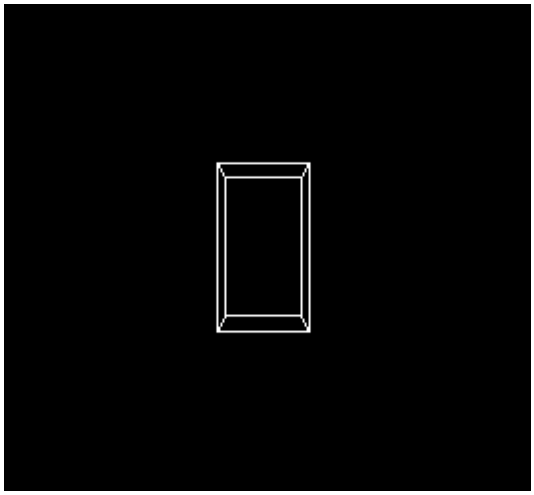
2.16 Basit bir Örnek: Bir Küp Çizme

Örnek 2-3, yarattığınız pencerede, wireframe olarak bir küp çizmektedir.Wireframe olarak söylenilen şey küpün sadece gözüken kenarlarını göstermek anlamına gelmektedir, küpün yüzeyleri çizilmez. Viewing transformasyonuda **gluLookAt**() ile kameranın nereye yerleştirileceği ve nasıl bir şekilde modeli izleyeceği belirtilir.
Ek olarak projection ve viewport transformasyonları (bezen transformation kelimesini tranformasyon veya dönüşüm olarak yazabilirim) belirlenecektir. Resim 2-6 ise ekranda oluşacak olan küpü göstermektedir.

Örnek 2-3 Dönüştürülmüş küp

```
#include <windows.h>
#include <GL/glut.h>

void init(void) {

	glClearColor(0.0, 0.0, 0.0, 0.0);
	glShadeModel(GL_FLAT);

} // end of void init(void)

void display(void) {

	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(1.0, 1.0, 1.0);
	glLoadIdentity();
		/* viewing transformation */
	gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
	//glTranslatef(0.0, 0.0, -5.0);
	glScalef(1.0, 2.0, 1.0);
	glutWireCube(1.0);
	glFlush();

} // end of void display(void)

void reshape(int w, int h) {

	glViewport(0, 0, (GLsizei) w, (GLsizei) h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glFrustum(-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
//	gluPerspective(60.0, 1.0, 1.5, 20.0);
	glMatrixMode(GL_MODELVIEW);

} // end of void reshape(int, int)

int main(int argc, char** argv) {
```

```
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(100, 100);
        glutCreateWindow(argv[0]);
        init();
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutMainLoop();

        return 0;
} // end of int main(int, char**)
```



Resim 2-6

2.17 Vieving Transformation

Viewing transformation kameranın yerinin belirlenmesi ve modele yönlendirilmesidir. Bu kodda, viewing transformation belirlenmeden önce, program içerisinde o anda kullanılan matris **glLoadIdentity**() fonksiyonu ile identity matris olarak yüklenir. Identity matris, diagonel elemanları 1'e ve diğer tüm elemanları 0'a eşit olan bir kare matristir. (Program içerisinde o anda kullanılmakta olan matrise current matris diyelim)

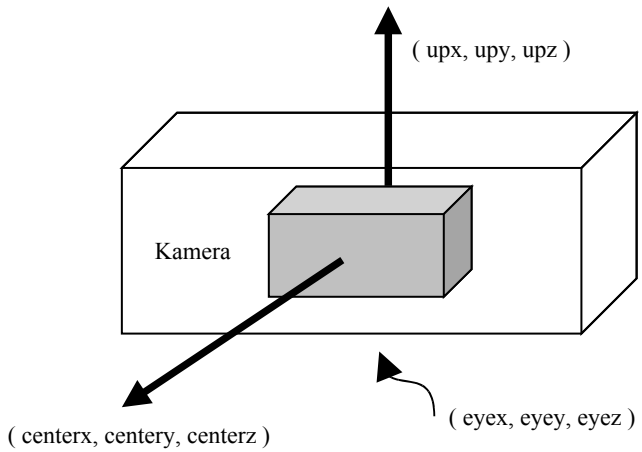
Bu aşama gereklidir, çünkü belirtilen bir matris current matris ile çarpılıp tekrar current matrise atanır. Dolayısı ile yapacağımız işlemlere dikkat etmemiz gerekir. Eğer current matrisini identity olarak ilklemezseniz yani temizlemezseniz, bazı istenmeyen sonuçlar elde edersiniz.

Örnekte, matris ilklendikten sonra**, gluLookAt**() fonksiyonu ile viewing transformasyonu belirlenmektedir. Fonksiyona aktarılan argümanlar, kameranın(göz koordinatları) nereye yerleştirileceğini, nereye yönlendirileceğini ve nasıl bir şekilde modeli göstereceğini belirtirler. Kodda kullanılan argümanlar kameranın (0, 0, 5) noktasında olduğunu, kameranın lensinin (0, 0, 0) noktasına yönlendirildiğini ve up-vectörü olarak (0, 1, 0) çekim yaptığını anlatmaktadır.

gluLookAt() fonksiyonunu kullanmamış olsaydık, kameranın varsayılı olan değerleri kullanılmış olurdu. Varsayılı olarak, kamera koordinat sistemini merkezinde, lensi negatif z eksenine çevrilmiş ve up vektör olarak (0, 1, 0) noktasına göre ayarlanmış olurdu. Örnekte ise kamerayı 5 birim, pozitif z ekseni doğrultusunda kaydırılmıştır.

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
               GLdouble centerx, GLdouble centery, GLdouble centerz,
               GLdouble upx, GLdouble upy, GLdouble upz);
```

(eyex, eyey, eyez) koordinatları kameranın yerini ayarlar. (centerx, centery, centerz) koordinatları kameranın hangi noktaya bakacağını tanımlamaktadır. Son olan koordinat parametreleri ise (upx, upy, upz) kameranın up-vektörünü belirtmektedir. Resim 2-7 bu parametrelerin, kamera üzerindeki kullanımını göstermektedir.



Resim 2-7

2.18 glRotate*() ve glTranslate*() Fonksiyonlarının Kullanımı

gluLookAt() fonksiyonunu kullanmadan, başka yöntemler ile bu fonksiyonu gerçekleştirebiliriz. Bu yöntem ise modelview transformasyon fonksiyonlarından olan **glRotate***() ve **glTraslate***() fonksiyonlarıdır. Bu fonksiyonlar durağan bir kameraya, üç boyutlu ortamdaki fonksiyonelliği sağlarlar. Kamerayı hareket ettirmek yerine kameranın gösterdiği modelleri hareket ettirmek de diğer bir yöntemdir. Örnek 2-3'teki koda tekrar bakacak olursanız, **display**() fonksiyonu içerisinde önü çift ters bölü işareti ile pasif hale getirilmiş **glTranslatef**(0.0, 0.0, -5.0) fonksiyonun göreceksiniz. Bu fonksiyonun önündeki ters bölü işaretlerini kaldırıp, **gluLookAt**() fonksiyonun pasif hale getirirseniz aynı etkiyi göreceksiniz. Bildiğiniz gibi **gluLookAt**() fonksiyonunu kullanmadığınızda kamera, koordinat sisteminin merkezine varsayılı olarak yerleştirilir. Koordinat sistemini, kameradan 5 birim uzağa yani negatif z ekseni doğrultusunda kaydırırsanız küpün görünümünde bir değişiklik olmayacaktır. Aşağıdaki **display**() fonksiyonu bu özeti göstermektedir.

```
void display(void) {

      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(1.0, 1.0, 1.0);
      glLoadIdentity();
            /* viewing transformation */
```

```
        //gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
        glTranslatef(0.0, 0.0, -5.0);
        glScalef(1.0, 2.0, 1.0);
        glutWireCube(1.0);
        glFlush();

} // end of void display(void)
```

2.19 Modeling Transformation

Modeling transformasyonları, nesneleri hareket ettirerek, döndürerek, boyutunda değişiklik yaparak, bu modellerin orientasyonunda ve pozisyonunda çeşitli değişiklikler yapmamıza izin verir. Bu işlemlerin birini veya bir çoğunu birlikte kullanarak gerçekleştirebilirsiniz.

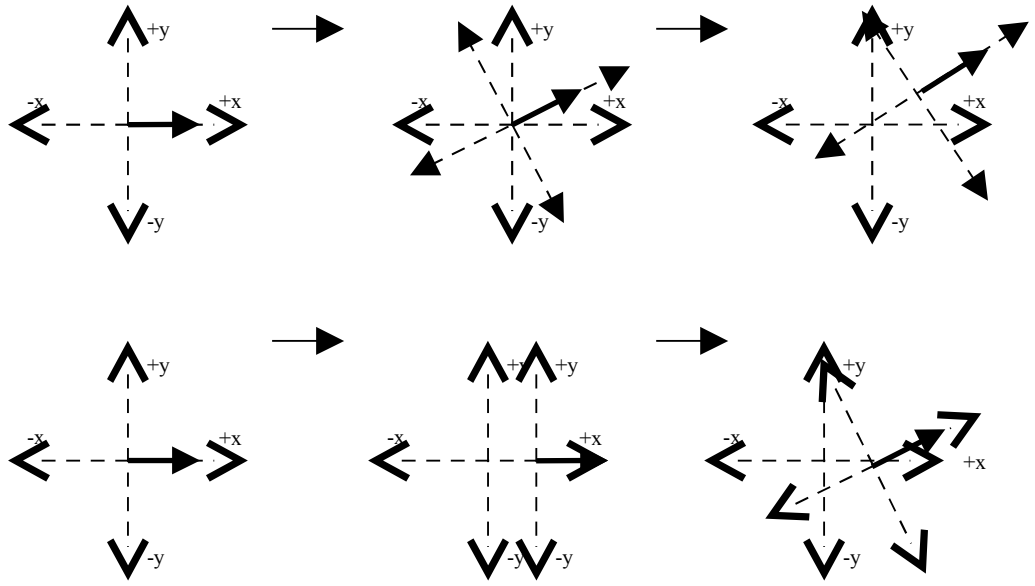
- **Translation:** Belirtilen bir eksen üzerinde nesnenin hareketini sağlar.

- **Rotating:** Bir eksen etrafında, modelin kendi çevresinde döndürme işlemidir.

- **Scalig:** Nesnenin boyutunu büyültmek veya küçültmek gibi fonksiyonelliklere sahip olan bir gerçekleştirimdir. Bu özellik sayesinde nesneye en veya boy gibi çeşitli boyut değişikliği kazandırabilirsiniz

Ekranınızda çizeceğiniz nesne için belirlemiş olduğunuz modeling transformasyon fonksiyonlarının sırası önemlidir. Örnek vermek gerekirse, Resim 2-8, translation işleminden sonra rotation işlemini ve rotation işleminden sonra translation işlemini göstermektedir. İki farklı işlemin etkileri birbirlerinden farklıdır. Farz edelim ki, koordinat sisteminin merkezine yerleşmiş olan bir ok olsun. Uygulayacağımız ilk transformasyon

işlemi bu oku z eksenine göre 30 derece döndürmektir yani rotation işlemidir. Ardından x ekseni doğrultusunda koordinat sistemini 5 birim taşıma yapalım. Eğer bu işlemi tersini yaparsak elde edeceğimiz sonuç ilkinden farklı olacaktır.



Resim 2-8

2.20 Modelview Matrisi

Modelview matrisi, nesneleri koordinat sisteminin istenilen bir yerine yerleştirmek ve onları yönetmek için koordinat sistemini tanımlar. 4'e 4'lük bir matristir verteksler ve transformasyonlar ile çarpılırlar. Bu çarpım işleminin sonunda, vertekslere uygulana transformasyonların sonuçlarını yansıtacak yeni bir matris üretilir.

OpenGL komutu olan **glMatrixMode**() fonksiyonu aracılığı ile modelview matrisini değiştirmek isteyebilirsiniz. Bu fonksiyonun prototipi aşağıdaki satırda belirtilmiştir

```
void glMatrixMode( GLenum mode );
```

Herhangi bir transformation işlemini çağırmadan önce, hangi matris modu üzerinde Modifikasyonlar uygulayacağınızı belirtmeniz gerekir. Modelview modunda işlem yapmak istiyorsanız parametre olarak GL_MODELVIEW parametresini aktarın.

glMatrixMode() için aktarılacak diğer parametreler ise GL_PROJECTION ve GL_TEXTURE argümanlarıdır. Projection matrisini belirtmek için GL_PROJECTION kullanılır ve texture matrisi için ise GL_TEXTURE bunu belirtmektedir.

Bir çok durumda, bu matrisi current matris olarak ayarladıktan sonra modelview matrisini identity matris olarak ilkleyin. Bunu yapabilmek için **glLoadIdentity**() fonksiyonu kullanın.

2.21 Translation

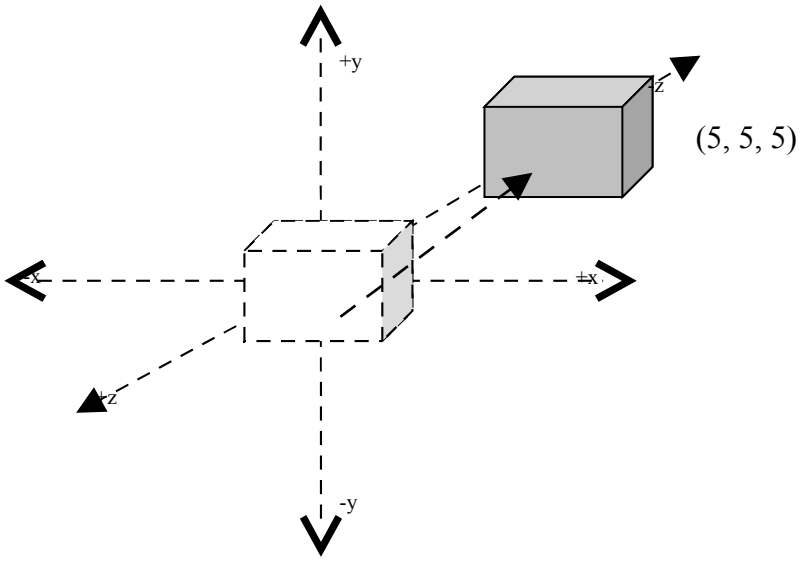
Üç boyutlu bir dünyada, nesneyi bir yerden başka bir yere taşımak için bu işlem kullanılmaktadır. Bunu yapabilmek için **glTanslatef**() ve **glTranslated**() fonksiyonları bu işlemi gerçekleştirir.

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
void glTranslated(GLdouble x, GLdouble y, GLdouble z);
```

Bu iki fonksiyon arasındaki tek fark aldıkları parametre tiplerinin birbirlerinden farklı olmalıdır. **glTranslatef**(), float tipinde ve **glTranslated**() fonksiyonu double tipinde parametre alır.

Farzedin ki, bir kübü koordinat sisteminin merkezinden (5, 5, 5) noktasına taşımak istiyorsunuz. İlk olarak modelview matrisini yükleyin ve sonra ilkleyin. Aşağıdaki kod yapacağımız işlemi gerçekleştirir ve resim 2-9 bunun sonucunu gösterir.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glTranslatef(5.0, 5.0, 5.0);
// kübü çiz
DrawCube();
```



Resim 2-9

2.22 Rotating

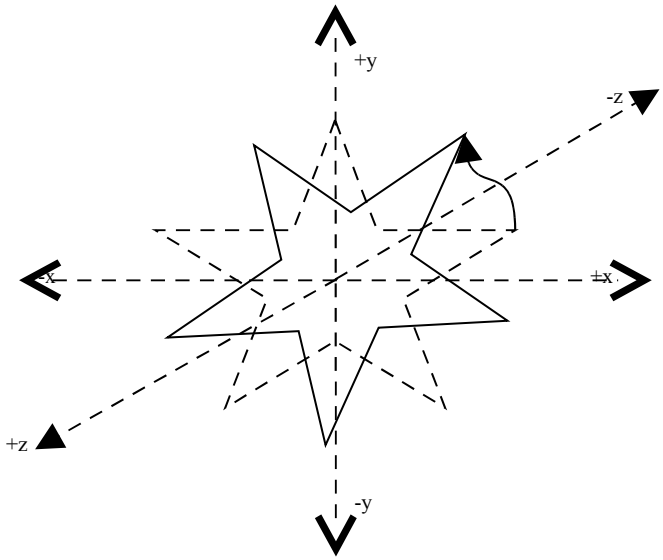
OpenGL'de rotating, **glRotate***() fonksiyonu ile gerçekleştirilmektedir.

```
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);
```

Tekrar söylemek gerekirse, **glTranslate***() fonksiyonunda olduğu gibi bu iki fonksiyon arasındaki tek fark aldıkları parametre tiplerinin birbirlerinden farklı olmalarıdır. x, y ve z parametrelerine göre, nesneyi belirli bir açı ölçüsünde bu koordinat eksenleri etrafında döndürebilirsiniz. Mesela çizmiş olduğunuz bir modeli y ekseni etrafında yelkovan yönüne, 135 derece döndürmek istiyoruz. Bunun için aşağıdaki bu işlemi bizim için yapmaktadır.

```
glRotatef(135.0, 0.0, 1.0, 0.0);
```

y argümanın aldığı 1.0 değeri, y ekseni yöndeki birim vektörü belirtmektedir. İstediğiniz eksene göre, rotate işlemini yapabilmeniz için sadece birim vektörünü belirtmeniz gerekir. Resim 2-10 glRotate*() fonksiyonunun nasıl çalıştığını göstermektedir.



Resim 2-10

Eğer şekli saat yönünün tersine çevirmek istiyorsanız, açıyı negatif olarak ayarlamanız gerekir. Bahsettiğimiz aynı rotating örneği için nesneyi saat yönünün tersinde 135 derece y ekseni çevresinde döndürmek istiyorsak aşağıdaki komutu yazmamız gerekmektedir.

```
glRotatef(-135.0, 0.0, 1.0, 0.0);
```

x, y, z parametrelerini kullanarak modeli istediğimiz yöne çevirebiliriz. Unutmayın ki koordinat ekseni bu dönüşten etkilenecektir.

2.23 Scaling

Modelin boyutundaki ayarlamaları yapmak için scaling işlemi kullanılmaktadır. Nesnenin boyutları eksenlere göre uzatılabilip kısaltılabilir. Bunu OpenGL'in sağladığı **glScale***() fonksiyonu ile yapabiliriz.

```
void glScalef(GLfloat x, GLfloat y, GLfloat z);
void glScaled(GLdouble x, GLdouble y, GLdouble z);
```

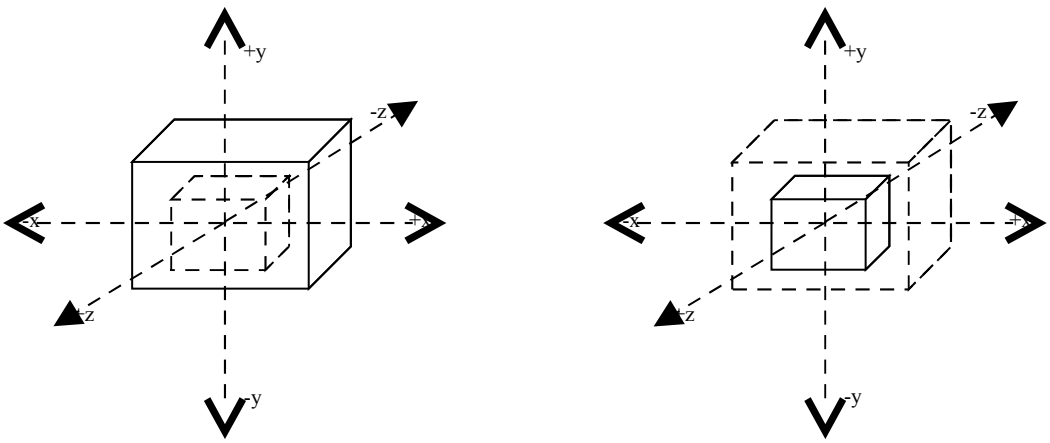
x, y, z parametrelerine geçirilen değerler her bir eksenin scale faktörünü belirler. Örnek olarak aşağıdaki satır nesnenin her boyutunu 2 katına çıkartmaktadır.

```
glScalef(2.0, 2.0, 2.0);
```

Mesela, çizdiğimiz bir kübün derinlik ve yüksekliğini değiştirmeden x ekseni üzerindeki genişlini 2 katına çıkartmak istiyoruz.

```
glScalef(2.0, 1.0, 1.0);
```

Eğer nesneyi küçültmek istiyorsanız, kullanacağınız değerler 1'den küçük olmalıdır.



glScalef(2.0, 2.0, 2.0); **glScalef(0.5, 0.5, 0.5);**

Aşağıda kübün hacmini 8 katına çıkartan örnek gösterilmektedir.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glScalef(2.0, 2.0, 2.0);
DrawCube();  // kübü çiz
```

2.24 Matris Yığıtları

Yarattığınız modelview ve projection matrisleri, yükleme ve çarpma işlemeleri sadece buz dağının görünen yüzünü göstermektedirler. Bu matrislerin her biri matrislerin yığıtının en üst elemanlarıdır.

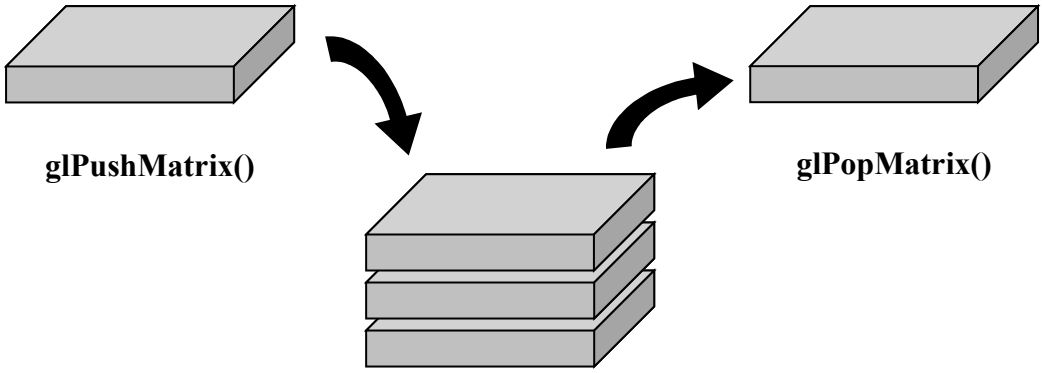
Matris yığıtı bizim için ideal bir mekanizma sağlamaktadır. Geçmişteki transformasyonları hatırlayabilmek için bu yapı kullanılmaktadır. Üç tip matris yığıtı vardır.

- Modelview Matris Yığıtı
- Projection Matris Yığıtı
- Texture Matris Yığıtı

Matrislerin yığıtları hiyerarşik olarak nesneleri meydana getirmek için kullanışlıdırkar. Bu karmaşık modeller basit olan modellerden yaratılırlar. Örneğin, kutulardan nasıl bir robot oluşturulduğunu düşünün. Eğer robotu bileşenlerine ayırırsanız, robotun bir gövdesinin, kollarının, başının ve bunun gibi bileşenlerinin olduğunu görürsünüz.

Bu ana kadar yaptığımız matris işlemleri hep current matris ile ilgiliydi. Aslında bu matris yığıtın en üst elemanıdır.Yığıt işlemlerini yerine getiren komutlar ile, yığıtın üstünde bulunan matrisi kontrol edebiliriz. Bildiğiniz gibi yığıt işlemelerinde en sıklıkla kullanılan iki komut vardır. Bu komutlar **push**() ve **pop**() komutlarıdır. Yığıta bilgi göndermek için yığıtın en üstüne bu bilgi kopyalanır. Bu işlem **push**() işlemidir. Yığıttın en üstünden bilginin atılabilmesi için **pop**() işlemi kullanılmaktadır. OpenGL bu gibi komutları bize sağlamıştır. **glPushMatrix**() ve **glPopMatrix**()… **glPushMatrix**()

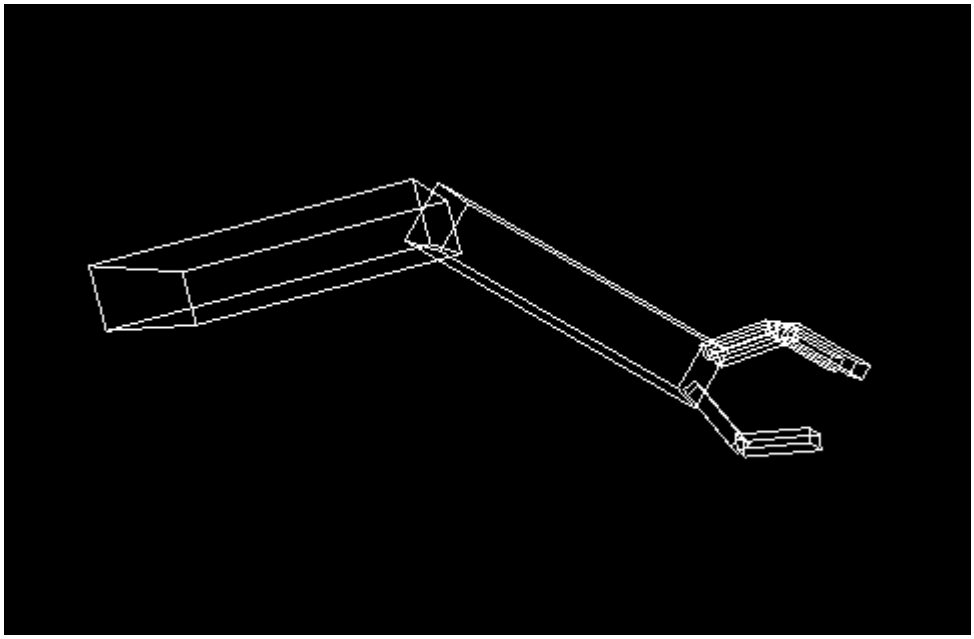
current matrisi kopyalar ve yığıtın en üstüne bu kopyalanmış bilgiyi yerleştirir. **glPopMatrix**() ise yığıtın en üstündeki bilgiyi çekip çıkartır. Hatırlayın: yığıtın en üstü current matristir. Resim 2-11 yığıt matrisini gösterir.



Matris Yığıtı

Resim 2-11

glPushMatrix() ve **glPopMatrix()** için yapacağımız en uygun örnek, bir robot kolu olacaktır. Bu örnek sayesinde bu iki fonksiyonun ne işe yaradığını bu örnek sayesinde anlayacaksınız. Bu bölümde robot kolunun ikiden fazla segmenti vardır. Aşağıdaki resim 2-12 çizilecek olan robot kolunu göstermektedir.



Resim 2-12

Kolun segmentlerini çizebilmek için öncelikle boyutlarını değiştirebileceğiniz bir kübe ihtiyacınız olmalı, fakat ilk olarak her segmentin kendi koordinat eksenlerine yerleştirmek için uygun modelleme transformasyonlarını çağırmak gerekir. İlk olarak yerel koordinat sisteminin merkezi çizilen kübün merkezinde olacağından dolayı, kübün bir kenarına yerel koordinat sistemini taşımaya ihtiyacınız olacak. Diğer bir koşulda, küp pivot noktasına göre değil de merkezine göre dönecektir.

Pivot noktasını yaratmak için **glTranslate*()** fonksiyonunu ve bu pivot noktasına göre **glRotate*()** fonksiyonunu çağırdıktan sonra, kübün merkezine doğru çevrimi yapın. Daha sonra küp çizilmeden önce, kübün boyutlarını değiştirin. **glScale*()** fonksiyonu diğer segmentleri etkileyebilir ve bunu engellemek için **glPushMatrix** ve **glPopMatrix** fonksiyonlarını kullanmak faydalı olacaktır. Aşağıda robot kolunun ilk segmenti için, program kodunun nasıl bir şeye benzediğini görün.

```
glTranslatef(-1.0, 0.0, 0.0);
glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef(1.0, 0.0, 0.0);
```

```
	glPushMatrix();
		glScalef(2.0, 0.4, 1.0);
		glutWireCube(1.0);
	glPopMatrix();
```

İkinci segmenti yaratabilmek için, lokal koordinat sistemini bir sonraki pivot nokta-Sına hareket ettirmemiz gerekiyor. Koordinat sistemi önceden döndüğüne göre, x ek-seni, dönmüş olan kolun uzunluğu doğrultusunda çoktan yerleşmiş olacaktır. Bu sebep-ten dolayıdır ki, x ekseni doğrultusunda transformasyon yapmak lokal koordinat siste-mini bir sonraki pivot noktasına hareket ettirecektir. Bu noktaya koordinat sistemin ta-şındıktan sonra, ilk olarak kullanmış olduğunuz kod bloğunu yeniden kullanarak, ikin-ci segmenti çizebilirsiniz. Bu işlem sonsuz sayıda segmentler için geçerlidir. Örnek 2-4 deki program kodunun tümü gösterilmektedir.

Örnek 2-4

```
#include <stdlib.h>
#include <GL/glut.h>

static GLfloat shoulder = 0.0, elbow = 0.0,
                    wrist1 = 0.0, finger1 = 0.0,
                    b_wrist = 0.0, b_finger = 0.0;

void init(void) {

	glClearColor(0.0, 0.0, 0.0, 0.0);
	glShadeModel(GL_FLAT);

	return;

} // end of void init(void)

void reshape(int w, int h) {
```

```
	glViewport(0, 0, (GLsizei) w, (GLsizei) h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(65.0, (GLfloat) w / (GLfloat) h, 1.0, 20.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	glTranslatef(-2.0, 0.0, -5.0);

	return;

} // end of reshape(int, int)

void display(void) {

	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(1.0, 1.0, 1.0);

	glPushMatrix();

		glTranslatef(-1.0, 0.0, 0.0);
		glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);
		glTranslatef(1.0, 0.0, 0.0);
		glPushMatrix();
			glScalef(2.0, 0.4, 1.0);
			glutWireCube(1.0);
		glPopMatrix();

		glTranslatef(1.0, 0.0, 0.0);
		glRotatef((GLfloat) elbow, 0.0, 0.0, 1.0);
		glTranslatef(1.0, 0.0, 0.0);
		glPushMatrix();
			glScalef(2.0, 0.4, 1.0);
			glutWireCube(1.0);
		glPopMatrix();


		glPushMatrix();
```

```
        glTranslatef(1.0, 0.125, 0.25);
        glRotatef((GLfloat)wrist1, 0.0, 0.0, 1.0);
        glTranslatef(0.25, 0.0, 0.0);
        glPushMatrix();
              glScalef(0.5, 0.1 , 0.25);
              glutWireCube(1.0);
        glPopMatrix();

        glTranslatef(0.25, 0.0, 0.0);
        glRotatef((GLfloat) finger1, 0.0, 0.0, 1.0);
        glTranslatef(0.25, 0.0, 0.0);
        glPushMatrix();
              glScalef(0.5, 0.1 , 0.25);
              glutWireCube(1.0);
        glPopMatrix();

  glPopMatrix();

  glPushMatrix();
        glTranslatef(1.0, 0.125, -0.25);

        glRotatef((GLfloat)wrist1, 0.0, 0.0, 1.0);
        glTranslatef(0.25, 0.0, 0.0);
        glPushMatrix();
              glScalef(0.5, 0.1, 0.25);
              glutWireCube(1.0);
        glPopMatrix();

        glTranslatef(0.25, 0.0, 0.0);
        glRotatef((GLfloat) finger1, 0.0, 0.0, 1.0);
        glTranslatef(0.25, 0.0, 0.0);
        glPushMatrix();
              glScalef(0.5, 0.1, 0.25);
              glutWireCube(1.0);
        glPopMatrix();

  glPopMatrix();
```

```
            glPushMatrix();
                  glTranslatef(1.0, -0.125, 0.0);

                  glRotatef((GLfloat)b_wrist, 0.0, 0.0, 1.0);
                  glTranslatef(0.25, 0.0, 0.0);
                  glPushMatrix();
                        glScalef(0.5, 0.1, 0.25);
                        glutWireCube(1.0);
                  glPopMatrix();

                  glTranslatef(0.25, 0.0, 0.0);
                  glRotatef((GLfloat) b_finger, 0.0, 0.0, 1.0);
                  glTranslatef(0.25, 0.0, 0.0);
                  glPushMatrix();
                        glScalef(0.5, 0.1, 0.25);
                        glutWireCube(1.0);
                  glPopMatrix();
            glPopMatrix();

      glPopMatrix();

      glutSwapBuffers();

      return;
} // end of void display(void)

void keyboard(unsigned char key, int x, int y) {

      switch (key) {
            case 's':
                  shoulder += 5;
                  if (shoulder > 360.0) shoulder -= 360.0;
                  glutPostRedisplay();
                  break;
            case 'S':
                  shoulder -= 5;
```

```
        if (shoulder < 360.0) shoulder += 360.0;
        glutPostRedisplay();
        break;
    case 'e':
        elbow += 5;
        if (elbow > 360.0) elbow -= 360.0;
        glutPostRedisplay();
        break;
    case 'E':
        elbow -= 5;
        if (elbow < 360.0) elbow += 360.0;
        glutPostRedisplay();
        break;
    case 'd':
        wrist1 += 5;
        if (wrist1 > 360.0) wrist1 -= 360.0;
        glutPostRedisplay();
        break;
    case 'D':
        wrist1 -= 5;
        if (wrist1 < 360.0) wrist1 += 360.0;
        glutPostRedisplay();
        break;
    case 'f':
        finger1 += 5;
        if (finger1 > 360.0) finger1 -= 360.0;
        glutPostRedisplay();
        break;
    case 'F':
        finger1 -= 5;
        if (finger1 < 360.0) finger1 += 360.0;
        glutPostRedisplay();
        break;
    case 'v':
        b_finger += 5;
        if (b_finger > 360.0) b_finger -= 360.0;
        glutPostRedisplay();
```

```
                  break;
            case 'V':
                  b_finger -= 5;
                  if (b_finger < 360.0) b_finger += 360.0;
                  glutPostRedisplay();
                  break;
            case 'c':
                  b_wrist += 5;
                  if (b_wrist > 360.0) b_wrist -= 360.0;
                  glutPostRedisplay();
                  break;
            case 'C':
                  b_wrist -= 5;
                  if (b_wrist < 360.0) b_wrist += 360.0;
                  glutPostRedisplay();
                  break;
            case 27:
                  exit(0);
                  break;
            default:
                  break;
      } // end of switch (key)
} // end of void keyboard(unsigned char, int , int)

int main(int argc, char** argv) {
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
      glutInitWindowSize(640, 480);
      glutInitWindowPosition(100, 100);
      glutCreateWindow(argv[0]);
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutKeyboardFunc(keyboard);
      glutMainLoop();
      return 0;
```

```
} // end of int main(int, int)
```

2.25 OpenGL' de Işıklandırma

Konu olarak, üç boyutlu grafik kavramları açısından nesneler için ışıklandırma büyük önem taşımaktadır. Işıklandırma bilgisayar ortamında, yarattığınız nesnelere bir gerçeklik katmaktadır. Şu ana kadar işlemiş olduğumuz konularda nesnelerin nasıl yaratıldığını, bu nesneleri nasıl hareket ettirdiğimizi ve boyutlarındaki değişimi nasıl gerçekleştirdiğimizi anlatmaya çalıştık. Şimdi ise bu nesneleri ışıklandırma fonksiyonları kullanarak canlandırmaya başlayacağız.

Gerçek dünyada nesneler nasıl ışığı yansıtarak, kendi renklerini bize gösteriyorsa, ışığı kırmızı, yeşil ve mavi bileşenlerine yaklaştırarak, OpenGL ışık ve ışıklandırmayı hesaplamaktadır. Burada söylenmek istenene şey, ışığın yaydığı kırmızı, yeşil ve mavi ışık miktarlarına bağlı olarak, bir ışığın yaydığı renk belirlenir. Işık bir yüzeye çarptığında, yüzey tarafından yansıtılacak olan kırmızı, yeşil ve mavi ışık tonlarının yüzdelerini belirtmek için, OpenGL yüzeyin materyalini kullanmaktadır.

OpenGL, dört ışık bileşenini kullanarak, gerçek dünyadaki ışıklandırmayı gerçekleştirmektedir.

- **Ambient Işıklandırma**, belirli bir yönden gelen ışık kaynağına benzememektedir. Bunun anlamı, ışığın hangi kaynaktan geldiğini belirlemenin imkansız olduğu anlamına gelir yani ışığın her yönden geldiğini ortaya koymaktadır.

- **Diffuse Işıklandırma**, belirli bir yönden gelmektedir, fakat ışık yüzeye çarptığında tüm yönlere eşit olarak yansıtılmaktadır. Gözünüzün hangi pozisyonda olduğuna bakılmadan, ışığın çarptığı yüzey, göze eşit olarak parlak görünür.

- **Specular Işıklandırma**, belirli bit yönden gelerek yüzeye çarpar ve önceden belirlenmiş bir yöne doğru yansımaktadır yani eşit olarak yansıma yoktur.

- **Emissive Işıklandırma**, nesneden çıkan ışık anlamına gelmektedir. OpenGL ı-şıklandırma yönteminde, nesnenin yüzeyinden yansıyan renk, nesneye daha çok yoğunluk katar, fakat bu diğer ışık kaynaklarını etkilemez.

2.26 Materyal Renkler

OpenGL ışıklandırma modeli, kendisinin yansıtmakta olduğu kırmızı, yeşil ve mavi Işıkların yüzdeleri materyalin rengini etkilemektedir. Örnek olarak, kırmızı bir top, tüm gelen kırmızı ışığı yansıtır ve diğer gelmekte olan yeşil ve mavi ışığı absorbe eder. Eğer bu topu beyaz ışık altında gözlüyorsanız, topun tüm kırmızı rengi yansıtılacak ve top kırmızı olarak görülecektir. Eğer top kırmızı ışığın az olduğu bir yerde bulunuyorsa, top yine kırmızı olarak, fakat az yeşil ışığın bulunduğu yerde top bulunuyorsa, aynı top si-yah olarak gözükür. Çünkü yeşil ışık absorbe edilmektedir ve orada kırmızı ışık kayna-ğı yoktur, bu yüzden ışık yansıtılamaz.

Işıklarda olduğu gibi, materyallerinde farklı ambient, diffuse ve specular renklere sahiptirler. Bu renkler, materyalin ambient, diffuse ve specular yansımalarını belirler. Bir materyali ambient yansıması, her gelen ışık kaynağının ambient bileşeni ile, diffuse yansıması ışığın diffuse bileşeni ile ve specular yansıması specular bileşeni ile bütün-leştrilmektedir. Ambient ve diffuse yansımaları materyalin rengini belirler. Specular yansıma genellikle beyaz yada gridir bu yüzden yüzey üzerindeki specular olan ışıklı ve detaylı olan kısım, orada ışık kaynağının specular yoğunluğunu olmasını sonlandırır. Eğer parlak bir kırmızı plastik topun beyaz ışık altında olduğunu düşünürseniz, bu top-un büyük bir yüzeyinin kırmızı, fakat parlaklığın fazla olduğu bölgenin beyaz olduğunu göreceksiniz.

Işıklar için belirlenmiş olan renk bileşenleri, materyaller için, daha farklı bir anlama gelmektedir. Bir ışığı ele alırsak, her renk için, sayılar tam yoğunluk yüzdesine uygun düşmektedir. Işığın kırmızı, yeşil ve mavi bileşenlerinin değerinin hepsi 1.0' a eşit ise, ışık parlak bir beyaz ışık kaynağı olmaktadır. Eğer tüm değerler 0.5' e eşit olduğunda, ışık kaynağı yine beyaz olur fakat yoğunlu %50 oranında azalcaktır. Işık bileşenlerinden kırmızı ve yeşilin değeri !.0' a, mavinin değeri ise 0.0' a eşit ise, ışık yeşil olarak görünür. Materyaller için, sayılar yansımakta ışığın bileşenler değeri ile çarpılmaktadır. Mesela materyalin kımızı değeri 1.0' a, yeşil değeri 0.5' e ve mavi değeri 0.0' a eşit oluyorsa, materyal gelen ışıklardan kırmızı ışığın %100' ünü, yeşil ışığın %50'sini ve mavi ışığın %0' ını yansıtmakta olduğunu belirtmektedir. Diğer bir anlatım şekli ile anlatmaya çalışırsak, elimizde bir ışık kaynaklarının değerleri olsun. Bu değerler (IK, IY, IM) ve materyalin değerleri ise (MK, MY, MM) olsun. Diğer yansıma etkilerini göze almasak, gözün göreceği değerler yani cismin yansıtacağı ışık değerleri (IK * MK, IY * MY, IM * MM) olmaktadır.

Benzer olarak, iki ışık kaynağı kullanıyorsanız, yani göze gönderilecek değerler (K1, Y1, M1) ve (K2, Y2, M2), OpenGL bu bileşenleri toplayacaktır (K1 + K2, Y1 + Y2, M1 + M2). Bu toplama işlemlerinden herhangi biri 1' den büyük çıkarsa toplama işlemi 1 olarak alınır.

2.27 Küçük Bir Örnek: Kırmızı Çaydanlık

Uygulamanıza, ışıklandırmayı eklemek için aşağıdaki aşamaları yapmakta fayda var.

1. Her nesnenin her verteksi için normal vektörleri tanımlayın. Bu normaller ışık kaynağıyla alakalı olarak nesnenin yerini belirlerler.

2. Bir veya birden fazla ışığı yaratın, seçin ve belirli konumlara getirin.

3. Bir ışıklandırma modelin yaratın.

4. Nesnenin materyal özelliklerini tanımlayın.

Aşağıdaki örnek 2-5 bu işlemleri gerçekleştirmektedir. Bu örnekte tek bir ışık kaynağı altında çizilmiş olan kırmızı bir çaydanlık çizilmektedir.

Örnek 2-5 Kırmızı Çaydanlık

```
#include <GL/glut.h>
#include <GL/glu.h>

GLfloat rotate_y = 0.0;


void draw_net(GLfloat size, GLint LinesX, GLint LinesZ) {

      int xc, zc;

      glBegin(GL_LINES);
            for (xc = 0; xc < LinesX; xc++)
            {
                  glVertex3f(
                        -size / 2.0 + xc / (GLfloat)(LinesX-1)*size,
                        0.0,
                        size / 2.0);
                  glVertex3f(
                        -size / 2.0 + xc / (GLfloat)(LinesX-1)*size,
                        0.0,
                        size / -2.0);
            } // end of for (xc = 0; xc < LinesX; xc++)

            for (zc = 0; zc < LinesZ; zc++)
            {
```

```
                glVertex3f(
                      size / 2.0,
                      0.0,
                      -size / 2.0 + zc / (GLfloat)(LinesZ-1)*size);
                glVertex3f(
                      size / -2.0,
                      0.0,
                      -size / 2.0 + zc / (GLfloat)(LinesZ-1)*size);
            } // end of for (zc = 0; zc < LinesZ; zc++)

      glEnd();

      return;
} // end of void draw_net(GLfloat, GLint, GLint)

void init(void) {

      GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
      GLfloat light_ambient[] = {1.0, 1.0, 1.0, 1.0};
      GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};
      GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};

      GLfloat material_ambient[] = {0.23, 0.0, 0.0, 1.0};
      GLfloat material_diffuse[] = {0.44, 0.0, 0.0, 1.0};
      GLfloat material_specular[] = {0.33, 0.33, 0.52, 1.0};
      GLfloat material_emission[] = {0.0, 0.0, 0.0, 0.0};
      GLfloat material_shininess = 10;

      glClearColor(0.0, 0.0, 0.0, 0.0);
      glShadeModel(GL_SMOOTH);

      glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
      glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
      glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

      glMaterialfv(GL_FRONT, GL_AMBIENT, material_ambient);
```

```
	glMaterialfv(GL_FRONT, GL_DIFFUSE, material_diffuse);
	glMaterialfv(GL_FRONT, GL_SPECULAR, material_specular);
	glMaterialfv(GL_FRONT, GL_EMISSION, material_emission);
	glMaterialfv(GL_FRONT, GL_SHININESS, &material_shininess);

	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);
	glEnable(GL_DEPTH_TEST);

	glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

	return;
} // end of init(void)

void display(void) {

	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glPushMatrix();

		glRotatef(rotate_y, 0.0, 1.0, 0.0);

		glPushMatrix();
			glTranslatef(0.0, 0.55, 0.0);
			glutSolidTeapot(0.75);
		glPopMatrix();

		glDisable(GL_LIGHTING);
		glColor3f(1.0, 1.0, 1.0);
		draw_net(5, 11, 11);
		glEnable(GL_LIGHTING);

	glPopMatrix();

	glutSwapBuffers();

	return;
} // end of void display(void)
```

```
void reshape(int w, int h) {

	glViewport(0, 0, (GLsizei)w, (GLsizei)h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	gluLookAt(0.0, 2.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

	return;
} // end of void reshape(GLfloat, GLfloat)

void rotate_teapot(void) {

	rotate_y += 0.5;
	if (rotate_y > 360.0)
		rotate_y -= 360.0;
	glutPostRedisplay();

	return;
} // end of rotate_teapot(void)

int main(int argc, char** argv) {

	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
	glutInitWindowSize(500, 500);
	glutInitWindowPosition(100, 100);
	glutCreateWindow("Teapot");
	init();
	glutDisplayFunc(display);
	glutReshapeFunc(reshape);
	glutIdleFunc(rotate_teapot);
	glutMainLoop();
```

```
    return 0;
} // end of int main(int, char**)
```

Örneği incelemeye başlayalım. Işık kaynaklarının bir çok özellikleri vardır, bunlar renk, pozisyon ve ışığın yönüdür. Bu tüm özellikleri belirlemek için kullanılan komut **glLight*()** fonksiyonudur. Fonksiyon üç argüman almaktadır. Fonksiyonun prototipi aşağıda belirtilmiştir.

```
void glLight[if]v( GLenum light, GLenum pname, Type *param );
```

İlk parametre olan *light* parametresi özelliği belirlenecek olan ışık kaynağını belirtir. Bu kaynaklar GL_LIGHT0, GL_LIGHT1, … GL_LIGH7 olan sekiz tane ışık kaynağı mevcuttur. Işığın karakteristiğini belirleyen parametre *pname* parametresidir ve tablo 2-2 bu parametrenin alacağı değerleri listelemektedir. *pname* parametresine göre *param* parametresinin alacağı değerler vardır. Bu parametre bir vektörü gösteren bir işaretçidir.

Tablo 2-2 glLight*() Parametreleri

Parametre	Varsayılı Değerler	Anlamı
GL_AMBIENT	(0.0, 0.0, 0.0, 0.0)	Ambient yoğunluğu
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	Diffuse yoğunluğu (light 0 için)
	(0.0, 0.0, 0.0, 0.0)	(Diğer ışıklar için)
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular yoğunluğu (light 0 için)
	(0.0, 0.0, 0.0, 0.0)	(Diğer ışıklar için)
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) pozisyon

Gösterilecek daha bir çok parametre var fakat şu anda bizim ilgileneceğimiz parametreler uygulamalarımız için yeterli. Örneği dikkatlice incelerseniz **init()** fonksiyonu içinde ışığın ambient, diffuse, specular ve pozisyon bilgileri tanımlanmıştır.

```
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
GLfloat light_ambient[] = {1.0, 1.0, 1.0, 1.0};
GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};
GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};

glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

Göreceğiniz gibi, parametre değerleri için diziler tanımlanmış ve **glLightfv()** fonksiyonu ışığın özelliklerini belirlemek için tekrar tekrar çağrılmıştır. Bu özellikler ayarlandıktan sonra, ışıklandırma olayını ve ışık kaynağını aktifleştirmeye geldi. Temel olarak ışıklandırmanın aktif hale gelebilmesi için **glEnable()** fonksiyonunu kullanmak gerekir.

```
glEnable(GL_LIGHTING);
```

Bu işlemin ardından hangi ışık kaynağını aktif hale getirmek istediğimizi OpenGL' e anlatmak içi yine aynı fonksiyonu yani **glEnable()** fonksiyonunu kullanmamız gerekir. Özelliklerini ayarladığımız ışık kaynağı light 0 olduğuna göre aşağıdaki kod bu ışık kaynağını aktif hale getirecektir.

```
glEnable(GL_LIGHT0);
```

Işığı aktif ettikten sonra kısaca anlatmak istediğimiz şey ışığın bulunduğu konum, yani ışığın pozisyonu ile ilgili önemli bir konuyu dile getirmek istiyoruz. Işık pozisyonu ayarlanırken kullanılan dört farklı değerler var (x, y, z, w). Bu değerleri ilk üçü yani x, y, z tahmin edeceğiniz gibi ışık kaynağının koordinatlarını belirtmektedir. Peki ya dördüncü parametre ne anlama gelmektedir ? Dördüncü parametreyi 0.0' a eşit olduğunda ışık kaynağının sonsuz uzaklıktan bir yerden geldiği anlamına gelmektedir ve bu sebepten dolayı ışık kaynağından yayılan ışınlar parelel olarak yayılırlar. Bu akla ilk somut olarak gelen örneğin güneş ışığı olduğudur. Güneş dünyadan çok uzak olduğundan dolayı bu uzaklığı sonsuz olarak alabiliriz. Uzaklık sonsuz olunca güneşten çıkan ışınlar dünya yüzeyine parelel olarak gelmektedir. Bu tip ışık kaynaklarına directional light source denmektedir. Light 0 için bu örneği komut olarak yazarsak aşağıdaki gibi kod bloğunu gösterebiliriz.

```
GLfloat light_position[] = {0.0, 0.0, 1.0, 0.0}
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Bu yukarıdaki örnekte z – ekseninden gelen ışık kaynağı tanımlanmıştır. Dördüncü parametreyi sıfırdan farklı bir değere sahip olduğunda bu tip ışık kaynağı positional light source olarak adlandırılır. Bunun anlamı (x, y, z) noktasında konumlandırılmış olan bir ışık kaynağının bulunması anlamına gelmektedir. Bu sanki masanın üstüne konmuş olan bir lambanın bulunması gibidir.

Işık kaynağının karakteristiklerini gördükten sonra materyallerin karakteristiklerini inceleyerek ikisi arasındaki benzerlikleri veya farklılıkları anlayacağız. En çok bilinen materyal özellikleri önceden görmüş olduğunuz ışık kaynağı özelliklerine benzemektedir. Bu özellikleri belirlemek için kullanılan mekanizma ışıkta kullanılan mekanizma ile benzerlik taşımaktadır. Fakat bu sefer materyaller için kullanılan komut **glMaterial*()** fonksiyonudur.

```
void glMaterial[if]v( GLenum face, GLenum pname, Type *param );
```

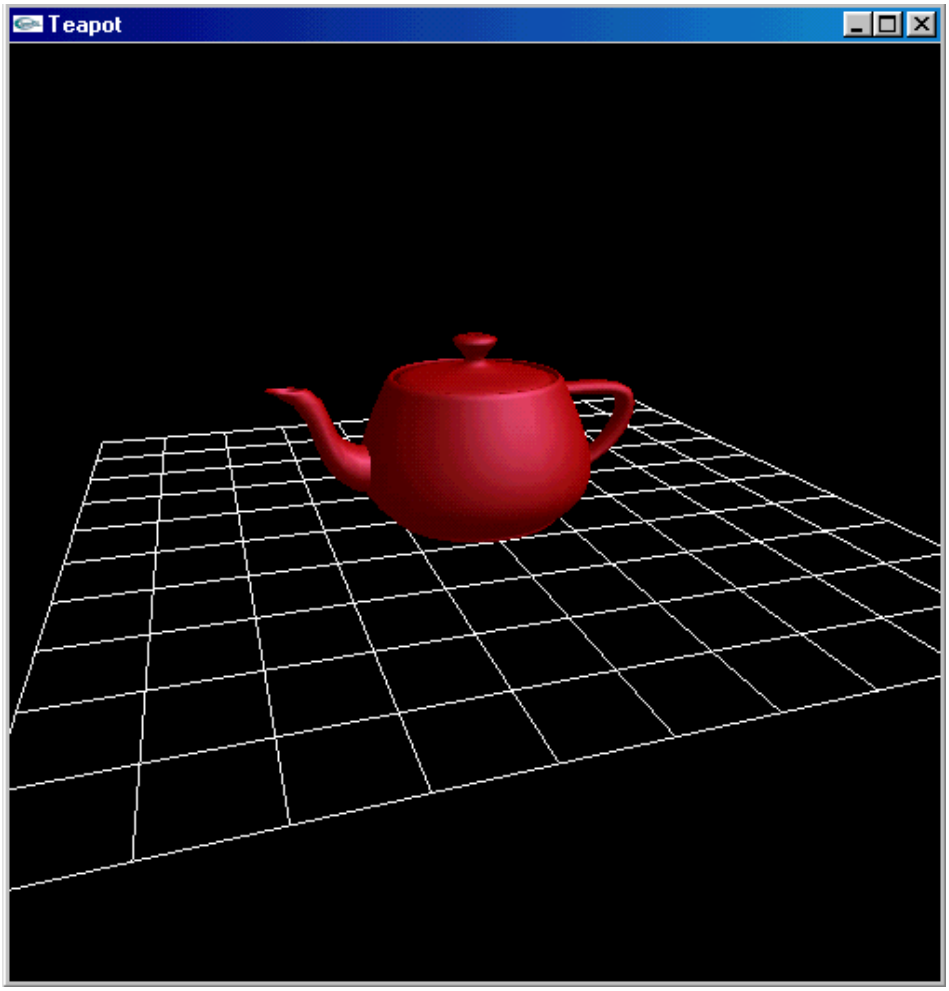
Fonksiyon sayesinde, ışıklandırmada kullanım için, materyalin özelliğini belirler. *face* parametresi, GL_FRONT, GL_BACK veya GL_FRONT_AND_BACK değerlerini alarak nesnenin yüz materyaline, materyaller için tanımlanmış olan özelliklerin uygulanması anlamına gelmektedir. Bu parametreye uygulanacak değerler ise *param* parametresi tarafından belirlenir. *param* parametresi, bir vektörü veya diziyi işaret eden bir işaretçidir. Bu *pname* parametresi için değerler tablo 2-3' te gösterilmektedir.

Tablo 2-3 glMaterial*() *pname* Parametresi

Parametre	Varsayılı Değerler	Anlamı
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient renk
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Diffuse renk
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular renk
GL_SHININESS	(0.0)	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive renk

face parametresi için GL_FRONT olarak tanımlanmış ise, GL_FRONT değeri OpenGL' e poligonun sadece ön kısmında materyal özelliklerini kullanılacağı anlatılmaktadır GL_BACK olduğunda ise poligonun arka kısmında materyal özelliklerinin kullanılacağı veya GL_FRONT_AND_BACK kullanıldığında ise poligonun her iki

yüzeyinde bu materyal özelliklerinin geçerli olacağı bildirilmektedir. Program çalıştırıldığında resim 2-13 ‘ te çizim yapılmaktadır.



Resim 2-13

2.28 Blending

OpenGL’ de renk karışımı (color blending) çizeceğiniz nesnelere saydamlık katılmasına izin vermektedir. Bu saydamlık sayesinde, gerçek hayatta görmekte olduğunuz

su, çam, pencereler ve diğer nesneleri simule edebilirsiniz.

Şu ana kadar nesneler üzerinde kullandığımız renk verme fonksiyonu **glColor3*()** fonksiyonuydu. Fonksiyon sadece üç parametre almaktadır. Bu parametrelerin ne olduklarını biliyorsunuz. Nesnelere saydamlık katabilmek için başka bir renk fonksiyonu kullanacağız. Bu fonksiyon şimdiye kadar kullanılan üç parametreye ek olarak dördüncü bir parametre almaktadır. Bu parametreye alfa parametresi denmektedir. Kullanacağımız fonksiyon ise **glColor4*()** fonksiyonudur. Peki ya alfa değeri ne yapar ? Alfa değeri nesnenin yeni rengini, frame buffer' ında bulunan renk ile birleştirmektedir. Frame buffer olarak söylenen şey sistemde kullanılan tüm buffer' ların bütünüdür. Yani color buffer, depth buffer, stencil buffer ve accumulation buffer size tek bir frame buffer' ını verebilmek için birleşirler. Buffer' lar konusuyla pek ilgimiz olamayacak.

Blending işleminde, RGB bileşenleri fragmanın rengini ve alfa bileşeni ise o fragmanın katılık derecesini yani şeffaf olmama derecesini söyler. Saydam veya şeffaf olan cisimlerin şeffaf olmama derecesi saydam olmayan kilere göre daha düşüktür bu da demek oluyor ki bu cisimlerin alfa değerleri daha düşüktür.

2.29 Kaynak ve Hedef Faktörleri

Blending işlemi esnasında, gelen fragmanın renk değerleri, bu kaynak yada source olarak bilinir, hali hazırda saklı tutulan piksellerin renk değerleri, bu ise hedef yada destination olarak adlandırılmaktadır, ile bütünleştirilir. İlk olarak kaynak ve hedef faktörlerini nasıl hesaplayacağınızı belirlersiniz. Bu faktörler RGBA yani, kırmızı, yeşil, mavi ve alfa, dörtlüleridir. Bu dörtlüler kaynağın R, G, B ve A değeri ile hedefin R, G, B ve A değerleri ile çarpılır. Bunu matematiksel olarak gösterirsek kaynak ya da source' un RGBA değerleri (S_r, S_g, S_b, S_a) ve hedefin ya da destination' ın RGBA değerleri ise (D_r, D_g, D_b, D_a) olsun. Son olarak blending işlemi sonucunda ortaya çıkan değerler aşağıdaki gibidir.

$$(R_s * S_r + R_d * D_r, G_s * S_g + G_d * D_g, B_s * S_b + B_d * D_b, A_s * S_a + A_d * D_a)$$

Kaynak ve hedef RGBA değerleri sırasıyla s ve d alt simge karakteri ile belirtilmektedir.

Şimdi kullanacağımız fonksiyon, kaynak ve hedef blending faktörlerini oluşturmaktadır. Kullanacağımız fonksiyonun adı **glBlendFunc()**' tur ve bu fonksiyon iki sabiti destekler. İlki hesaplanması gereken kaynak faktör ve diğeri ise hedef faltördür. Ek olarak blending işlemini aktif hale getirmek için her zaman aşağıdaki tek satırlık kodu yazmalısınız.

```
glEnable(GL_BLEND);
```

Blending işlemini inaktif yapmak istiyorsanız bildiğiniz gibi **glDisable()** fonksiyonuna GL_BLEND parametresini aktarabilirsiniz. **glBlendFunc()** fonksiyonun prototipi aşağıdaki gibidir.

```
void glBlendFunc( GLenum sfactor, GLenum dfactor );
```

sfactor parametresi kaynak blending faktörünün, *dfactor* parametresi ise hedef blending faktörünün nasıl hesaplanacağını belirtirler. Bu parametreler için mümkün değerler tablo 2-4 gösterilmektedir. Blending faktörleri 0 ile 1 aralığında değerler alır. Tabloda source, destination ve constant (sabit) renklerin RGBA değerleri altsimge karakteri olan s, d, c harfleri ile gösterilmektedir.

Tablo 2-4 Source ve Destinatin Blending Faktörleri

Sabit	İlgili Faktör	Hesaplanan Blending Faktörü
GL_ZERO	source veya destination	(0, 0, 0, 0)
GL_ONE	source veya destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)
GL_SRC_ALPHA	source veya destination	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	source veya destination	(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)
GL_DST_ALPHA	source veya destination	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	source veya destination	(1, 1, 1, 1)-(A_d, A_d, A_d, A_d)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f = min(A_s, 1 - A_d)
GL_CONSTANT_COLOR	source veya destination	(R_c, G_c, B_c, A_c)
GL_ONE_MINUS_CONSTANT_COLOR	source veya destination	(1, 1, 1, 1) - (R_c, G_c, B_c, A_c)
GL_CONSTANT_ALPHA	source veya destination	(A_c, A_c, A_c, A_c)
GL_ONE_MINUS_CONSTANT_ALPHA	source veya destination	(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)

2.30 Basit bir Örnek

Yapacağımız örnek bir Windows uygulamasıdır. Bu uygulamayı GLUT yardımı ile de yapabilirdik, fakat biraz değişiklik olsun istedik ve biraz da Windows uygulamalarında alıştırma yapmaya çalıştık. Örnek 2-6'da alfa değeri 0.75 olan ve birbirinden farklı ve kesişmiş iki üçkeni göstermektedir. Blending aktif durumdadır, kaynak blending faktörü GL_SRC_ALPHA olarak ve hedef blending faktörü ise GL_ONE_MINUS_SRC_ALPHA olarak ayarlanmıştır.

Program çalıştığında, ilk olarak sarı üçken pencerenin sol tarafında çizilir ve daha sonra çiyan renkli üçken ekranın sağ tarafına çizilerek üçkenler kesişirler. Pencerede çizilecek olan ilk üçkeni 'l' veya 'r' tuşlarına basarak belirleyebilirsiniz. 'l' tuşuna bastığınızda çizilecek ilk üçken sarı renkte olan ve ekranın sol tarafına bulunan üçken olacaktır, 'r' tuşuna basarsanız ekranın sağ tarafına çizilecek olan çiyan renkli üçken ilk olarak çizilir.

Örnek 2-6

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>

float angle = 0.0;
HDC g_HDC;
BOOL fullScreen = FALSE;
BOOL keyPressed[256];

int left_first = GL_TRUE;

void init(void) {

      glClearColor(0.0, 0.0, 0.0, 0.0);
      glShadeModel(GL_FLAT);
      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

```
	glEnable(GL_BLEND);
	return;
} // end of void initialize(void)

void resize(int width, int height) {

	glViewport(0, 0, (GLsizei) width, (GLsizei) height);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	if (width <= height)
		gluOrtho2D(0.0,1.0, 0.0,
				(GLfloat)height / (GLfloat)width);
	else
		gluOrtho2D(0.0, (GLfloat)width / (GLfloat)height,
				 0.0, 1.0);
//	glMatrixMode(GL_MODELVIEW);
//	glLoadIdentity();
	return;
} // end of void resize(int, int)

void draw_leftTriangle(void) {

	glBegin(GL_TRIANGLES);
		glColor4f(1.0, 1.0, 0.0, 0.75);
		glVertex3f(0.1, 0.9, 0.0);
		glVertex3f(0.1, 0.1, 0.0);
		glVertex3f(0.7, 0.5, 0.0);
	glEnd();

	return;
} // end of void draw_leftTrianfle(void)

void draw_rightTriangle(void) {

	glBegin(GL_TRIANGLES);
		glColor4f(0.0, 1.0, 1.0, 0.75);
		glVertex3f(0.9, 0.9, 0.0);
```

```
            glVertex3f(0.3, 0.5, 0.0);
            glVertex3f(0.9, 0.1, 0.0);
      glEnd();

      return;
} // end of void draw_rightTriangle(void)

void display(void) {

      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      if (left_first) {
            draw_leftTriangle();
            draw_rightTriangle();
      }else {
            draw_rightTriangle();
            draw_leftTriangle();
      }

      SwapBuffers(g_HDC);
      return;
} // end of void display(void)

void SetupPixelFormat(HDC hDC) {

      int nPixelFormat;

      static PIXELFORMATDESCRIPTOR pfd = {
                  sizeof(PIXELFORMATDESCRIPTOR),
                  1,
                  PFD_DRAW_TO_WINDOW |
                  PFD_SUPPORT_OPENGL |
                  PFD_DOUBLEBUFFER,
                  PFD_TYPE_RGBA,
                  32,
                  0, 0, 0, 0, 0, 0,
                  0,
```

```
                0,
                0,
                0, 0, 0, 0,
                16,
                0,
                0,
                PFD_MAIN_PLANE,
                0,
                0, 0, 0
        };

        nPixelFormat = ChoosePixelFormat(hDC, &pfd);

        SetPixelFormat(hDC, nPixelFormat, &pfd);

        return;
} // end of void SetupPixelFormat(HDC)

LRESULT CALLBACK WndProc(HWND hWnd,
                                        UINT message,
                                        WPARAM wParam,
                                        LPARAM lParam)
{

        static HGLRC hRC;
        static HDC hDC;
        char string[] = "Simple Windows Application";
        int width, height;

        switch(message) {

                case WM_CREATE:
                        hDC = GetDC(hWnd);
                        g_HDC = hDC;
                        SetupPixelFormat(hDC);
                        hRC = wglCreateContext(hDC);
                        wglMakeCurrent(hDC, hRC);
```

```
                init();
                return 0;
                break;
          case WM_CLOSE:
                wglMakeCurrent(hDC, NULL);
                wglDeleteContext(hRC);
                PostQuitMessage(0);
                return 0;
                break;
          case WM_SIZE:
                height = HIWORD(lParam);
                width = LOWORD(lParam);
                if (height == 0) height = 1;
                resize(width, height);
                return 0;
                break;
          case WM_KEYDOWN:
                keyPressed[wParam] = TRUE;
                return 0;
                break;
          case WM_KEYUP:
                keyPressed[wParam] = FALSE;
                return 0;
                break;
          default:
                break;
     } // end of switch(message)

     return DefWindowProc(hWnd, message, wParam, lParam);

} // end of WndProc(HWND, UINT, WPARAM, LPARAM)

int WINAPI WinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine, int nShowCmd)
{
```

```
WNDCLASSEX windowClass;
HWND hWnd = NULL;
MSG msg;
BOOL done = FALSE;
DWORD dwExStyle;
DWORD dwStyle;
RECT windowRect;

int width = 200;
int height = 200;
int bits = 32;

windowRect.left = 0;
windowRect.right = width;
windowRect.top = 0;
windowRect.bottom = height;

windowClass.cbSize = sizeof(WNDCLASSEX);
windowClass.style = CS_HREDRAW | CS_VREDRAW;
windowClass.lpfnWndProc = WndProc;
windowClass.cbClsExtra = 0;
windowClass.cbWndExtra = 0;
windowClass.hInstance = hInstance;
windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);
windowClass.hbrBackground = NULL;
windowClass.lpszMenuName = NULL;
windowClass.lpszClassName = "MyClass";
windowClass.hIconSm = LoadIcon(NULL, IDI_WINLOGO);

if (!RegisterClassEx(&windowClass))
      return 0;

if (MessageBox(NULL,
      "Would You Like To Run In Fullscreen Mode?",
      "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDYES)
            fullScreen = TRUE;
```

```
if (fullScreen) {
      DEVMODE dmScreenSettings;
      memset(&dmScreenSettings, 0, sizeof(DEVMODE));
      dmScreenSettings.dmSize = sizeof(dmScreenSettings);
      dmScreenSettings.dmPelsWidth = width;
      dmScreenSettings.dmPelsHeight = height;
      dmScreenSettings.dmBitsPerPel = bits;
      dmScreenSettings.dmFields = DM_BITSPERPEL |
                                  DM_PELSWIDTH |
                                  DM_PELSHEIGHT;

      if (ChangeDisplaySettings(&dmScreenSettings,
            CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL) {

            MessageBox(NULL, "Display mode failed",
                        NULL, MB_OK);
            fullScreen = FALSE;
      }

} // end of if (fullScreen)

if (fullScreen) {
      dwExStyle = WS_EX_APPWINDOW;
      dwStyle = WS_POPUP;
      ShowCursor(FALSE);
} else {
      dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
      dwStyle = WS_OVERLAPPEDWINDOW;
}

AdjustWindowRectEx(&windowRect,
                           dwStyle, FALSE, dwExStyle);


hWnd = CreateWindowEx(0,
```

```
        "MyClass",
        "The OpenGL Window Application",
        dwStyle | WS_CLIPCHILDREN |
        WS_CLIPSIBLINGS,
        0, 0,
        windowRect.right - windowRect.left,
        windowRect.bottom - windowRect.top,
        NULL,
        NULL,
        hInstance,
        NULL);

if (!hWnd)
        return 0;

ShowWindow(hWnd, SW_SHOW);
UpdateWindow(hWnd);

while(!done) {

        PeekMessage(&msg, hWnd, 0, 0, PM_REMOVE);

        if (msg.message == WM_QUIT)
                done = TRUE;
        else {

                if (keyPressed[VK_ESCAPE])
                        done = TRUE;
                if(keyPressed['l'] || keyPressed['L'])
                        left_first = TRUE;
                if (keyPressed['r'] || keyPressed['R'])
                        left_first = FALSE;

                display();
                TranslateMessage(&msg);
                DispatchMessage(&msg);
```

```
            }

      } // end of while(!done)

      return (int) msg.wParam;

} // end of WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
```

Çizilecek olan üçkenlerin sırası, kesiştikleri bölgenin rengini etkiler. Soldaki üçken çizildiğinde, çiyan fragmanı (kaynak veya source) sarı fragman ile karıştırılır. Tabi bu sarı fragman önceden frame buffer'ın (hedef veya destination) içinde bulunmaktadır. Sağ taraftaki üçken ilk çizilirse, sarı çiya ile karıştırılıcaktır. Alfa faktörünün 0.75 olmasından dolayı, kaynak için blending faktör 0.75, hedef için ise 1.0 – 0.75 = 0.25 olacaktır.

2.31 Üç Boyutlu Nesnelerde Blending

Üç boyutlu nesneleri saydam olarak çizeceğinizde, çizdiğiniz polgonlara bağlı olarak, farklı görünümler elde edebilirsiniz. Çizim sırasını belirlerken, depth buffer' ını da düşünmeniz gerekmektedir. Depth buffer' ı pencere içinde bulunan nesnenin bir parçası ile bakış noktası arasındaki mesafeyi kontrol eder. Çizilecek olan nesne bakış noktasına yaklaştığında, bu nesne için çizilecek olan yeni renk nesne üzerinde yer alır ve bu durumda nesnenin derinlik değeri depth buffer' ı içerisinde saklı tutulur. Bu yöntem ile saklı yani gözükmeyen nesnelerin parçaları gereksiz yere çizilmez ve blending için kullanılmaz.

Aynı ekranda saydam olan bir cisim ile saydam olmayan bir cismi beraber çizmek istediğinizde, saydam olmayan bir nesnenin arkasında bulunan bir nesnenin saklı tutulan yüzeyini silebilmek için depth buffer' ını kullanırsınız. Eğer saydam olamayan bir cismin arkasında ister saydam olsun veya olmasın herhangi bir nesnenin parçası saklı tutuluyorsa, uzak olan nesnelerin elimine edebilmek için depth buffer' ına ihtiyacınız

olucaktır. Fakat saydam bir cisim bakış noktasına daha yakın olduğunda, bu cismi diğer saydam olmayan cisimler ile blend etmek zorundasınız. Bunu yapabilmek için depth buffer' ını read-only mofunda kullanmak gerekir. Çizilecek üç boyutlu nesnelerin çizim sırasıda önemlidir. O zaman ilk olarak çizilecek olan nesneler saydam olmayan cisimlerdir tabi bu sırada depth buffer'ı normal olarak işlemektedir. Bu depth buffer' ını read-only moduna getirerek, derinlik değerlerini saklayın. Saydam nesneler çizildiğinde, onların derinlik değerleri saydam olamayan nesneler tarafından üretilen değerler ile kıyaslanır ve böylece saydam olan nesneler çizilmez tabi bu nesneler saydam olmayan nesnelerin arkasında ise…Eğer saydam nesneler bakış noktasına yakın olduğunda, saydam olmayan nesneleri saklamazlar, çünkü depth buffer değerleri değişmemiştir. Bunun yerine bu nesneler saydam olmayan nesneler ile blend edilmiştir. Depth buffer' ını yazılabilir olup olmadığını kontrol edebilmek için, **glDepthMask()** fonksiyonu kullanılır.

```
void glDepthMask(GLboolean flag);
```

flag parametresine, GL_FALSE değerini atarsanız, buffer read-only modunda olacaktır, fakat GL_TRUE olarak atarsanız yazılabilir bir buffer olacaktır.

Örnek 2-7, bize bu anlattıklarımızın küçük bir uygulamasını gösterecek bir program kodunu anlatmaktadır. Ekran içerisinde bir grid üzerinde bulunan saydam ve kırmızı renkli bir küp ile onun biraz ötesinde bulunan mavi renkli bir küre çizilmiştir ve bu çizilen nesneler y ekseni etrafında dönmektedir.

Örnek 2-7

```
#include <stdlib.h>
#include <GL/glut.h>
#include <GL/glu.h>

GLfloat cube_ambient[] = {0.23, 0.0, 0.0, 1.0};
GLfloat cube_diffuse[] = {0.44, 0.0, 0.0, 1.0};
```

```
GLfloat cube_specular[] = {0.33, 0.33, 0.52, 1.0};
GLfloat cube_emission[] = {0.0, 0.0, 0.0, 0.0};
GLfloat cube_shininess = 50;

GLfloat sphere_ambient[] = {0.0, 0.0, 0.39, 1.0};
GLfloat sphere_diffuse[] = {0.0, 0.47, 0.5, 1.0};
GLfloat sphere_specular[] = {0.0, 0.64, 0.34, 1.0};
GLfloat sphere_emission[] = {0.0, 0.0, 0.2, 0.0};
GLfloat sphere_shininess = 38;

GLfloat rotate_y = 0.0;

void draw_net(GLfloat size, GLint LinesX, GLint LinesZ) {

      int xc, zc;

      glBegin(GL_LINES);
            for (xc = 0; xc < LinesX; xc++)
            {
                  glVertex3f(
                        -size / 2.0 + xc / (GLfloat)(LinesX-1)*size,
                        0.0,
                        size / 2.0);
                  glVertex3f(
                        -size / 2.0 + xc / (GLfloat)(LinesX-1)*size,
                        0.0,
                        size / -2.0);
            } // end of for (xc = 0; xc < LinesX; xc++)

            for (zc = 0; zc < LinesZ; zc++)
            {
                  glVertex3f(
                        size / 2.0,
                        0.0,
                        -size / 2.0 + zc / (GLfloat)(LinesZ-1)*size);
                  glVertex3f(
                        size / -2.0,
```

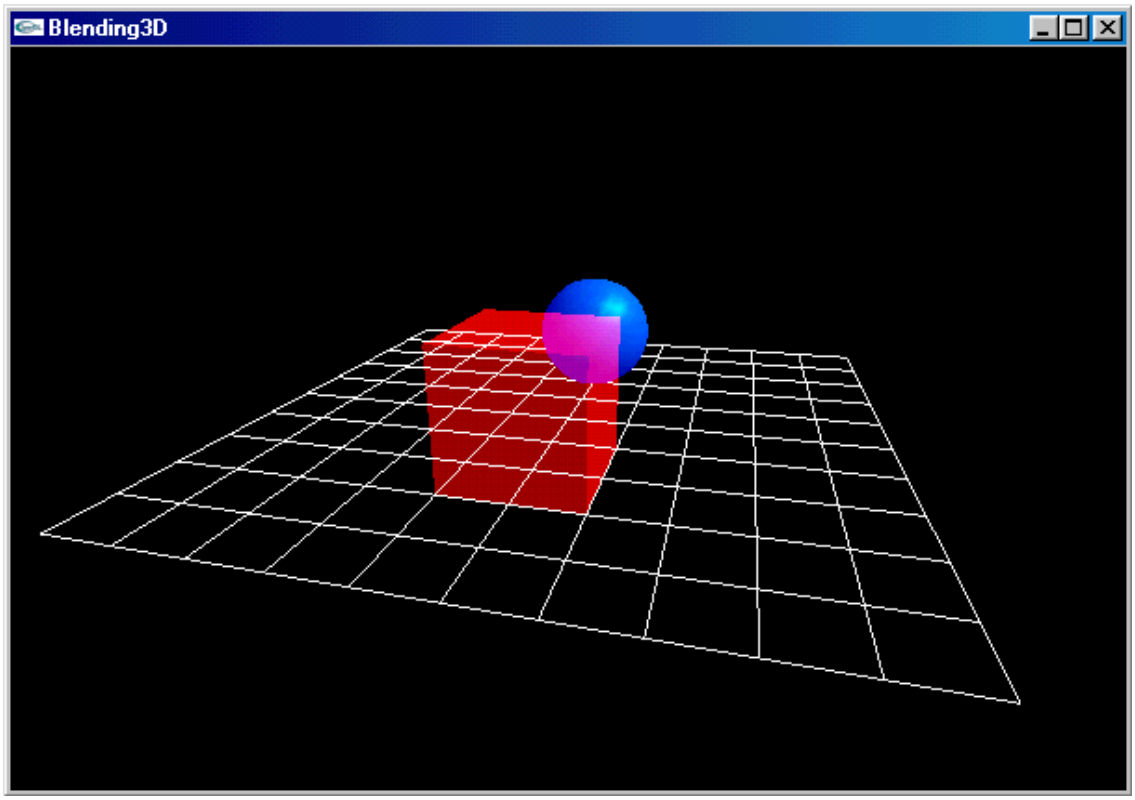
```
                        0.0,
                        -size / 2.0 + zc / (GLfloat)(LinesZ-1)*size);
            } // end of for (zc = 0; zc < LinesZ; zc++)

      glEnd();

      return;
} // end of void draw_net(GLfloat, GLint, GLint)

void init(void) {

      GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
      GLfloat light_ambient[] = {1.0, 1.0, 1.0, 1.0};
      GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};
      GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};

      glClearColor(0.0, 0.0, 0.0, 0.0);
      glShadeModel(GL_SMOOTH);

      glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
      glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
      glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

      glEnable(GL_LIGHTING);
      glEnable(GL_LIGHT0);
      glEnable(GL_DEPTH_TEST);

      glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

      return;
} // end of init(void)

void reshape(int w, int h) {

      glViewport(0, 0, (GLsizei)w, (GLsizei)h);
      glMatrixMode(GL_PROJECTION);
```

```
	glLoadIdentity();
	gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	gluLookAt(0.0, 2.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

	return;
} // end of void reshape(GLfloat, GLfloat)

void display(void) {

	glClear(GL_COLOR_BUFFER_BIT |
			GL_DEPTH_BUFFER_BIT);

	glPushMatrix();

		glRotatef(rotate_y, 0.0, 1.0, 0.0);

		glPushMatrix();
			glTranslatef(1.0, 0.5, 0.0);
			glMaterialfv(GL_FRONT, GL_AMBIENT,
					sphere_ambient);
			glMaterialfv(GL_FRONT, GL_DIFFUSE,
					sphere_diffuse);
			glMaterialfv(GL_FRONT, GL_SPECULAR,
					sphere_specular);
			glMaterialfv(GL_FRONT, GL_EMISSION,
					sphere_emission);
			glMaterialfv(GL_FRONT, GL_SHININESS,
					&sphere_shininess);
			glutSolidSphere(0.5, 16, 16);
		glPopMatrix();


		glPushMatrix();
			glTranslatef(-1.0, 0.5, 0.0);
			glMaterialfv(GL_FRONT, GL_AMBIENT,
```

```
                                    cube_ambient);
                glMaterialfv(GL_FRONT, GL_DIFFUSE,
                                    cube_diffuse);
                glMaterialfv(GL_FRONT, GL_SPECULAR,
                                    cube_specular);
                glMaterialfv(GL_FRONT, GL_EMISSION,
                                    cube_emission);
                glMaterialfv(GL_FRONT, GL_SHININESS,
                                    &cube_shininess);
                glEnable(GL_BLEND);
                glDepthMask(GL_FALSE);
                glBlendFunc(GL_SRC_ALPHA, GL_ONE);
                glutSolidCube(1.0);
                glDepthMask(GL_TRUE);
                glDisable(GL_BLEND);
          glPopMatrix();


          glDisable(GL_LIGHTING);
          glColor3f(1.0, 1.0, 1.0);
          draw_net(5, 11, 11);
          glEnable(GL_LIGHTING);

    glPopMatrix();

    glutSwapBuffers();

    return;
} // end of void display(void)

void rotate(void) {

    rotate_y += 0.5;

    if (rotate_y > 360.0)
          rotate_y -= 360.0;
```

```
	glutPostRedisplay();

	return;
} // end of rotate_teapot(void)

void keyboard(unsigned char key, int x, int y) {

	switch(key) {
		case 'R':
		case 'r':
			rotate_y = 0;
			glutPostRedisplay();
			break;
		case 27:
			exit(0);
	}

	return;
} // end of void keyboard(unsigned char, int, int)

int main(int argc, char** argv) {

	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
	glutInitWindowSize(600, 400);
	glutInitWindowPosition(100, 100);
	glutCreateWindow("Teapot");
	init();
	glutDisplayFunc(display);
	glutReshapeFunc(reshape);
	glutKeyboardFunc(keyboard);
	glutIdleFunc(rotate);
	glutMainLoop();

	return 0;
} // end of int main(int, char**)
```

Aşağıdaki resim 2-14, Örnek 2-7' deki kodu çalıştırdığınızda ortaya çıkacak sonucu göstermektedir.

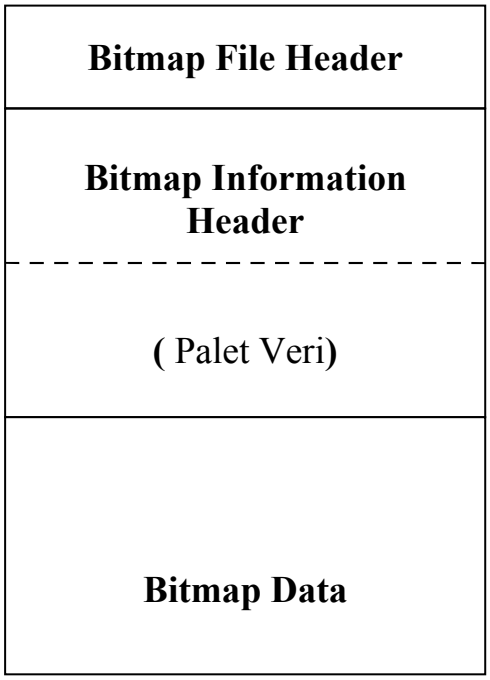


Resim 2-14

2.32 Bitmap Dosyaları

İleride açıklayacağımız texture mapping' de, kullancağımız resim dosyalarının adı bitmap resim dosyalarıdır. Bu dosyaların uzantıları bmp olarak bilinmektedir. Bitmap dosyalarında yada BMP dosyalarındaki en büyük şey, Windows işletim sistemini kullanan bir kişinin bu dosyaları yaratabilir, içeriğini değiştirebilir ve gözlenebilir olmasıdır. BMP dosyaları herhangi bir sıkıştırma şemalarını kullanmazlar bu sebepten dolayı boyutlarında küçük bir büyüme gözükebilir. Fakat sıkıştırma şemasından yoksun olmanın anlamı bu dosya formatının kolaylıkla okunabilir ve kullanılabilir olmasıdır.

Resim 2-15 bir BMP dosyasının yapısını göstermektedir. Bitmap file header, Bitmap information header ve bitmap data…



Resim 2-15

Bitmap dosya Bitmap File Header dosyasını yüklemek için kullanılan veri yapısı BITMAPFILEHEADER' dır ve bu yapı aşağıda gösterilmektedir.

```
typedef struct tagBITMAPFILEHEADER
{
      WORD  bfType;   // specifies the file type; must be BM (0x4D42)
      DWORD bfSize;   // specifies the size in bytes of BMP file
      WORD  bfReserved1;  // reserved; must be zero
      WORD  bfReserved2;  // reserved; must be zero
      DWORD bOffBits;  // specifies the ofset, in bytes, from
                  // BITMAPFILEHEADER stracture to the bitmap bits
} BITMAPFILEHEADER;
```

Bu yapı ile ilgili olarak, bir bmp dosyası yükleyeceğiniz zaman dosyanın uzantısı-

na dikkat etmeniz gerektiğidir. Bunu, dosyanın BITMAPFILEHEADER yapısının bir üyesine bakarak yani *bfType*' e bakarak dosyanın uzantısının bmp olup olmadığını anlayabilirsiniz. Bu üye bitmap dosyaları için her zaman 0x4D42 değerine eşittir. Bu sayıyı önceden Windows proglamlaması esnasında görmüştünüz. Bir bmp uzantılı dosyayı açarken bu sayı ile kontrol yapılmaktadır. *bfSize*, bize dosyanın boyutunu, *bOffBits* ise bize dosyanın başından itibaren Bitmap Data'nın nereden başladığını içeren bir ofset bilgisini gösterir.

Bir sonraki inceleyeceğimiz veri bölgesi, her BMP dosyasına özgü olan bilgileri içerir. Bu bölge, BMP dosyasının 8-bitlik ve bir palet verisine sahip olmasına göre bir veya iki parçaya ayrılmaktadır. Biz palet verisi ile ilgilenmeyeceğimiz için sadece BITMAPINFOHEADER yapısını okuyacağız.

```
typedef struct tagBITMAPINFOHEADER
{
      DWORD biSize;  // number of bytes required by the structure
      LONG  biWidth; // width of the bitmap, in pixels
      LONG  biHeight; // height of the bitmap, in pixels
      WORD  biPlanes; // number of color planes, must be 1
      WORD  biBitCount; // number of bits per pixel must be 1, 4, 8,
                        // 16, 24, 32
      DWORD biCompression; // type of compression
      DWORD biSizeImage; // size of image in bytes
      LONG  biXPelsPerMeter; // number of pixels perm eter in x-axis
      LONG  biYPelsPerMeter; // number of pixels perm eter in y-axis
      DWORD biClrUsed; // number of colors used by the bitmap
      DWORD biClrImportant; // number of colors that are important
} BITMAPINFOHEADER;
```

Bu yapının yanındaki açıklamaları okumak kendini açıklayacak kadar yeterlidir. Burada önemli olan bir BMP dosyasını nasıl belleğe yükleyeceğimizdir. BMP dosyasının Bitmap Data kısmında kırmızı ve mavi renk değerlerinin yerleri değişmiştir yani dosya bildiğiniz gibi kırmızı, yeşil ve mavi olarak sıralı değilde mavi, yeşil kırmızı ola-

rak okunmalı ve kırmızı ile mavi değerlerin yerlerinin değişmesi gerekmektedir. Bu işlem, dosya okunurken yapılmaıdır. Bir bitmap dosyası yaratılacaksa bu işlemi tersinden yapmak yani öncelikle mavi sonra yeşil ve en son olarak kırmızı renk değeri dosyaya yazılmalıdır çünkü bitmap dosya yapısına uymamız gerekir. Örnek 2-8' de 24-bitlik bir bitmap dosyasını yükleye ve bitmap dosyasını oluşturan program kodu ve onun başlık dosyası bulunmaktadır.

Örnek 2-8 Bitmap.h ve Bitmap.c

```
// Bitmap.h
#ifndef _BITMAP_H
#define _BITMAP_H
#include <windows.h>
typedef unsigned char *BITMAPIMAGE;
#define BITMAP_ID 0x4D42
#define FILE_NOT_OPENED_TO_WRITE -12
extern BITMAPIMAGE
Bitmap_LoadBitmapFile(const char*, const BITMAPINFOHEADER*);
extern int
Bitmap_WriteBitmapFile(const char*, LONG, LONG, BITMAPIMAGE);
#endif

// Bitmap.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Bitmap.h"
```

```
BITMAPIMAGE
```

```
Bitmap_LoadBitmapFile(const char* filename,
                      const BITMAPINFOHEADER* bitmapInfoHeader)
{

      FILE *filePtr;
      BITMAPFILEHEADER bitmapFileHeader;
      BITMAPIMAGE bitmapImage;
      DWORD imageIdx;
      unsigned char tempRGB;

      if ((filePtr = fopen(filename, "rb")) == NULL) {
            MessageBox(NULL, "File not found.", "Error", MB_OK);
            return NULL;
      }

      fread(&bitmapFileHeader,
              sizeof(BITMAPFILEHEADER), 1, filePtr);

      if (bitmapFileHeader.bfType != BITMAP_ID) {
            MessageBox(NULL,
                        "Invalid file extention.",
                          "Error", MB_OK);
            fclose(filePtr);
            return NULL;
      }

      fread((BITMAPINFOHEADER*)bitmapInfoHeader,
            sizeof(BITMAPINFOHEADER), 1, filePtr);

      fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);

      bitmapImage = (BITMAPIMAGE)
                     malloc(bitmapInfoHeader->biSizeImage);

      if (!bitmapImage) {
            MessageBox(NULL, "Out of memory", "Error", MB_OK);
            fclose(filePtr);
```

```
            return NULL;
        }

        fread((BITMAPIMAGE)bitmapImage,
                sizeof(*bitmapImage),
                bitmapInfoHeader->biSizeImage, filePtr);

        if (!bitmapImage) {
              MessageBox(NULL, "File not read.", "Error", MB_OK);
              free(bitmapImage);
              fclose(filePtr);
              return NULL;
        }

        for(imageIdx = 0;
            imageIdx < bitmapInfoHeader->biSizeImage;
              imageIdx+=3)
        {
              tempRGB = bitmapImage[imageIdx];
              bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
              bitmapImage[imageIdx + 2] = tempRGB;
        }

        fclose(filePtr);

        return bitmapImage;
}

int
Bitmap_WriteBitmapFile(const char* filename,
                                    LONG width, LONG height,
                                    BITMAPIMAGE imageData)
{

        FILE *filePtr;
        BITMAPFILEHEADER bitmapFileHeader;
        BITMAPINFOHEADER bitmapInfoHeader;
```

```
DWORD imageIdx;
unsigned char tempRGB;

if ((filePtr = fopen(filename, "wb")) == NULL)
      return FILE_NOT_OPENED_TO_WRITE;

bitmapFileHeader.bfSize = sizeof(BITMAPFILEHEADER);
bitmapFileHeader.bfType = BITMAP_ID;
bitmapFileHeader.bfReserved1 = 0;
bitmapFileHeader.bfReserved2 = 0;
bitmapFileHeader.bfOffBits = sizeof(BITMAPFILEHEADER) +
                             sizeof(BITMAPINFOHEADER);
bitmapInfoHeader.biSize = sizeof(BITMAPINFOHEADER);
bitmapInfoHeader.biPlanes = 1;
bitmapInfoHeader.biBitCount = 24;
bitmapInfoHeader.biCompression = BI_RGB;
bitmapInfoHeader.biSizeImage = width * height * 3;
bitmapInfoHeader.biXPelsPerMeter = 0;
bitmapInfoHeader.biYPelsPerMeter = 0;
bitmapInfoHeader.biClrImportant = 0;
bitmapInfoHeader.biWidth = width;
bitmapInfoHeader.biHeight = height;

for(imageIdx = 0;
    imageIdx < bitmapInfoHeader.biSizeImage;
      imageIdx += 3)
{
      tempRGB = imageData[imageIdx];
      imageData[imageIdx] = imageData[imageIdx + 2];
      imageData[imageIdx + 2] = tempRGB;
}

fwrite((BITMAPFILEHEADER*)&bitmapFileHeader,
         1, sizeof(BITMAPFILEHEADER), filePtr);

fwrite((BITMAPINFOHEADER*)&bitmapInfoHeader,
         1, sizeof(BITMAPINFOHEADER), filePtr);
```

```
        fwrite((BITMAPIMAGE) imageData,
                sizeof(*imageData),
                bitmapInfoHeader.biSizeImage, filePtr);

        fclose(filePtr);
        return 0;
}
```

2.33 Texture Mapping

Texture mapping konusu OpenGL' de çok geniş bir konudur, bu yüzden sadece gerekli olan kısımları inceleyeceğiz. Texture mapping, poligonlar üzerine resimleri yapıştırmanızı sağlamaktadır. Mesela, bir küre yarattınız ve bu küre üstüne 3 boyutlu olarak bir dünya resmi yağıştırabilirsiniz. Texture mapping' i tüm 3 boyutlu oyunlarda görebilirsiniz ve bu method oyulara daha çok gerçeklik katmaktadır.

Texture map'leri veriyi tutan kare dizilerinden oluşmaktadır. Her veri parçasına *texel* denmektedir. Bunlar veriyi taşıyan kare dizileri plmalarına karşın, texture map'leri karesel olmayan nesneler üzerine map edilebilirler, mesela silindirler ve küreler gibi…

Tasarımcılar genellikle, kendi grafik çalışmalarında iki boyutlu texture mapping yöntemini kullanırlar. Fakat tek boyutlu ve üç boyutlu texture kaplamalar da mevcuttur, fakat biz bunlar ile ilgilenmeyeceğiz. Konu olarak basit bir şekilde iki boyutlu texture mappingi inceleyeceğiz.

2.34 İki Boyutlu Texture

Bir dosyadan, texture mapping için kullanacağınız bir dosyayı belleğe yükledikten sonra, bunu texture map olarak tanıtmanız gerekmektedir. Texture map' in boyutları bu işi yapabilmeniz için hangi fonksiyonları kullanacağınızı belirtmektedir. Eğer 2 bo-

yutlu bir texture map işlemi yapmak istiyorsanız **glTexImage2D()** fonksiyonunu kullanın. Bir boyutlu texture map işlemi yapmak istiyorsanız **glTexImage1D()** fonksiyonunu, üç boyutlu olarak işlem yapmak istiyorsanız **glTexImage3D()** fonksiyonunu kullanabilirsiniz.

```
void glTexImage2D(GLenum target, GLint level, GLint intenalFormat,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum type, const GLvoid* texels);
```

target parametresi GL_TEXTURE_2D veya GL_PROXY_TEXTURE_2D olarak ayarlanabilir. Fakat bizim için bu parametreyi GL_TEXTURE_2D yapmak daha uygundur. *level* parametresi texture map' in çözünürlüğünü belirler. Sadece tek bir çözünürlük kullanmak istiyorsanız bu parametreyi 0 yapın. Şu an için bir çok çözünürlük modunun kullanımını tartışmayacağız. *internalFormat* paramtresi 1' den 4' e kadar veya size gösteremiyeceğimiz 38 sabit değerden birini kullanan sayıya eşittir. Bizim sadece kullanabileceklerimiz GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB ve GL_RGBA olmaktadır. *width* ve *height* parametreleri texture map' in genişlik ve boyutunu belirlerler. Bu değerler 2' nin üssü bir değer olmalıdırlar. *border* parametresi texture çevresinde bir sınırın olup olmadığını bildirmektedir. Sınır var ise bu değer 1, yok ise değer 0 olmalıdır. *format* parametresi, texture map'in verisinin formatını tanımlar ve bu format GL_COLOR_INDEX, GL_RGB, GL_RGBA, GL_RED, GL_GREEN,
GL_BLUE, GL_ALPHA, GL_LUMINANCE veya GL_LUMINANCE_ALPHA olabilir. *type* parametresi ise bu verinin tipini belirtir ve GL_BYTE,
GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT,
GL_UNSIGNED_INT, GL_FLOAT veya GL_UNSIGNED_FLOAT olabilir.
Son parametre olan *texel* parametresi bir işaretçidir. Bu işaretçi texture mapping için kullanacağınız resmi işaret eder. Mesela bir RGBA resmi yüklediniz ve onu gösteren işaretçi ise textureData olsun. Bu dizinin boyutu textureWidth ve textureHeight olsun. **glTexImage2D()** fonksiyonu ise aşağıdaki gibi kullanılabilir.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textureWidth, textureHeight,
             0, GL_RGBA, GL_UNSIGNED_BYTE, textureData);
```

2.35 Texture Nesneleri

Texture nesneleri, texture verilerini saklamanıza ve onları kullanmak için hazır durumda bekletmenizi sağlamaktadır. Bu nesneleri kullanarak, bir kerede bir çok texture' yi belleğe yükleyebilir ve çizim esnasında bunların herhangi birine referans verebilirsiniz.

Texture nesnelerini kullanamak için ilk yapacağınız şey bir texture ismi yaratmanızdır. Texture isimleri sıfırdan farklı işaretsiz bir tamsayılardır. Aynı texture ismini iki kere kullanmamak için **glGenTextures()** fonksiyonunu kullanın.

```
void glGenTextures(GLsizei n, GLuint *textureNames);
```

parametre *n*, kaç tane texture ismi yaratılacağını bildirmektedir ve bu isimle bir dizinin içine yerleştirilecek olan bir *textureNames* parametresinin gösterdiği dizi olacaktır. Örnek olarak vermek gerkirse, texture isimlerini saklayacağınız bir 3 elemanlı diziniz mevcutsa, fonksiyon aşağıdaki gibi kullanılacaktır.

```
unsigned int textureNames[3];
...
glGenTextures(3, textureNames);
```

Texture isimlerini yarattıktan sonra bu isimlerin texture verilerine verilmesi veya bağlanması gerekmektedir. Bunu **glBindTexture()** fonksiyonu ile gerçekleştirebilir.

```
void glBindTexture(GLenum target, GLuint textureName);
```

Bu fonksiyonu ilk olarak kullandığınızda, yeni bir texture nesnesi varsayılı değerlere sahip olarak yaratılır. OpenGL bu isim bağlama işleminden sonra texture nesnesinde-

ki varsayılı değerleri yeniden tanımlayacaktır. *target* parametresi bu fonksiyon için GL_TEXTURE_1D, GL_TEXTURE_2D veya GL_TEXTURE_3D olabilir. Bu isim bağlama işleminden sonra texture nesnesini o an kullanılabilecek texture durumuna getirmek için **glBindTexture()** fonksiyonunu kullanın. Mesela birkaç texture nesnesi yarattığınızı düşünelim ve bu nesnelerin texture verilerini ve texture özelliklerini bu nesnelere bağladınız. Ekranda poligonları çizeceğiniz zaman, **glBindTexture()** ile texture' ü belirleyerek, OpenGL' e belirlemiş olduğunuz texture' ü söyleyebilirsiniz. Aşağıdaki kod parçası, texture özellikleri ayarlandıktan sonra, ikinci texture nesnesini current texture olacağını, OpenGL' e anlatmaktadır.

```
unsigned int textureNames[3];
...
glGenTextures(3, textureNames);
...
glBindTexture(GL_TEXTURE_2D, textureNames[1]);
// texture verisi ve özelliklerini ayarla
glBindTexture(GL_TEXTURE_2D, textureNames[1]);
// nesneyi çiz
```

2.36 Texture Filitreleme

Texture resimleri, poligonlara uygulandıktan sonra, biçimi bozuk hale dönüşür ve sonuç olarak küçük bir piksel texel' in sadece küçük bir parçasını veya texel' leri gösterebilir. Resmin son hali için, OpenGL' e bu pikselleri ve texel' lerin nasıl hesaplanacağını texture filitreleme kullanarak anlatabilirsiniz.

Texture filitrelemede, magnification, bir pikselin bir texel parçasını göstermesi anlamını taşımaktadır ve minification ise bir pikselin texel' leri göstermesi anlamına gelir. OpenGL' e bu her iki durumu nasıl ele almak istediğinizi **glTexParameter()** fonksiyonunu çağırarak anlatabilirsiniz.

```
void glTexParameter(GLenum target, GLenum pname, GLint param);
```

target parametresi kullandığınız texture boyutuna bağlıdır ve GL_TEXTURE_1D, GL_TEXTURE_2D veya GL_TEXTURE_3D değerleri alabilir. Magnification filitrelemede *pname* parametresi GL_TEXTURE_MAG_FILTER değerini ve minification filitrelemede *pname* parametresi GL_TEXTURE_MIN_FILTER değerini almaktadır. *param* parametresinin alacağı değerler tablo 2-5' te gösterilmektedir.

Tablo 2-5 Texture Filitre Değerleri

Filitre	**Açıklama**
GL_NEAREST	Texture edilecek olan pikselin merkezine en yakın olan texel' i kullanır.
GL_LINEAR	Çizilecek olan pikselin merkezine en yakın olan texel'lerin ağırlıklı ortalamasını kullanır.
GL_NEAREST_MIPMAP_NEAREST	GL_NEAREST filitrelemeyi ve poligon çüzünürlüğüne en yakın imajı kullanır.
GL_NEAREST_MIPMAP_LINEAR	GL_LINEAR filitrelemeyi ve poligon çüzünürlüğüne en yakın imajı kullanır.
GL_LINEAR_MIPMAP_NEAREST	GL_NEAREST filitrelemeyi ve poligon çözünürlüğüne en yakın olan iki mipmap' ler arasıdaki ağırlıklı ortalamayı kullanır.
GL_LINEAR_MIPMAP_LINEAR	GL_LINEAR filitrelemeyi ve poligon çözünürlüğüne en yakın olan iki mipmap' ler arasıdaki ağırlıklı ortalamayı kullanır.

2.37 Texture Fonksiyonları

OpenGL, texture fonksiyonları aracılığı ile texture map renklerinin davranışını belirler. Her texture için, **glTexEnvi()** fonksiyonu ile dört tane texture fonksiyonundan seçim yapabilirsiniz.

```
void glTexEnvi(GLenum target, GLenum pname, GLint param);
```

target parametresi GL_TEXTURE_ENV değerine eşit olmalıdır. *pname* parametresine ise GL_TEXTURE_ENV_MODE değerini aktararak, frame buffer' ındaki renkler ile texture' lerin nasıl birleştirileceğini belirleyebilirsiniz. *param* parametresi için kullanılacak değerler tablo 2-6' da belirtilmiştir.

Tablo 2-6

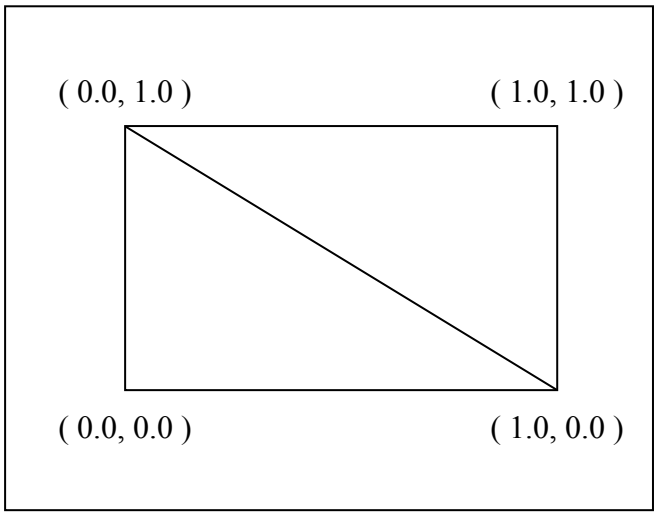
Mod	Açıklama
GL_BLEND	Texture rengi piksel rengi ile çarpılır ve sabit bir renk ile birleştirilir.
GL_DECAL	Texture, mevcut olan pikseller ile yer değiştirir.
GL_MODULATE	Texture rengi piksel rengi ile çarpılır.

glTexEnvi()' ini varsayılı değeri GL_MODULATE' tir. Aşağıdaki tek satırlık kod texture fonksiyonunu GL_DECAL' e ayarlamaktadır.

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

Ekranınızı çizdiğinizde, her verteksin texture koordinatlarını belirtmeniz gerekir.

Texture koordinatları, bir nesneye ait olan texture map' in her bir texel' i nereye konumlanacağını belirtmek için kullanılır. Bir texture koordinatı olan (0, 0) texture ' un sol alt köşesini, (1, 1) ise sağ üst köşesini belirtir. İki boyutlu texture' lerde bu koordinatlar (s, f) formundadır.Burada s ve f değerleri 0' dan 1' e doğru değişir. Resim 2-16 poligonun her verteksi için texture koordinatlarını göstermiştir. Bu koordinatlar her verteks render edilmeden önce, kendileri ile ilişkilendirilmiş olan texture koordinatları belirlenmelidir.



Resim 2-16

Texture koordinatlarını **glTexCoord*()** fonksiyonu ile belirlersiniz. Biz konu içerisinde iki boyutlu texture mapping yapacağımızdan dolayı bu fonksiyonun aşağıdaki iki boyutlu olanını kullanacağız.

```
void glTexCoord2f(GLfloat s, GLfloat t);
```

Aşağıda bu fonksiyonun basit bir kullanımını göstermektedir.

```
glBegin(GL_QUADS);
      glTexCoord2f(0.0, 0.0); glVertex3f(-0.5, 0.5, 0.5);
      glTexCoord2f(1.0, 0.0); glVertex3f(0.5, 0.5, 0.5);
      glTexCoord2f(1.0, 1.0); glVertex3f(0.5, 0.5, -0.5);
```

```
            glTexCoord2f(0.0, 1.0); glVertex3f(-0.5, 0.5, -0.5);
      glEnd();
```

Texture mapping konusu bu kadar… Örnek ise Windows platformunda geçekleştirilmiş olan dönen küpün bir benzeri vardır fakat bu örnek GLUT kütüphanesi fonksiyonları kullanılarak yapılmıştır. Bu kübün her bir yüzeyine bir bitmap dosyasından yüklenmiş resim yapıştırılmıştır. Resim ise satranç tahtasına benzeyen bir resim kübün her yüzeyi texture kaplama yapılmıştır. Kübü, farenizin sol tuşuna basık tutarak döndürebilirsiniz. Fareyi hareket halinde iken serbest bıraktığınızda küb sabit bir hızda dönmeye devam edecektir. Örnek 2-9' u inceleyiniz.

Örnek 2-9

```
#include <windows.h>
#include <stdlib.h>
#include <GL/glut.h>

#include "Bitmap.h"
#include "HiResTimer.h"

#define BITMAP_ID 0x4D42

BITMAPINFOHEADER bitmapInfoHeader;
BITMAPIMAGE bitmapData = NULL;

unsigned int texture;

int   last_x, last_y;
GLfloat rotationX = 0.0,
            rotationY = 0.0,
            timeElapsed = 0.0;

GLfloat dx = 0.0, dy = 0.0;

HiResTimer timer = NULL;
```

```
void DrawTextureCube(GLfloat xPos,
                     GLfloat yPos,
                     GLfloat zPos)
{

      glPushMatrix();
            glTranslatef(xPos, yPos, zPos);
            glBegin(GL_QUADS);  // top face
                  glTexCoord2f(0.0, 0.0);
                  glVertex3f(-0.5, 0.5, 0.5);
                  glTexCoord2f(1.0, 0.0);
                  glVertex3f(0.5, 0.5, 0.5);
                  glTexCoord2f(1.0, 1.0);
                  glVertex3f(0.5, 0.5, -0.5);
                  glTexCoord2f(0.0, 1.0);
                  glVertex3f(-0.5, 0.5, -0.5);
            glEnd();

            glBegin(GL_QUADS);  // front face
                  glTexCoord2f(0.0, 0.0);
                  glVertex3f(0.5, -0.5, 0.5);
                  glTexCoord2f(1.0, 0.0);
                  glVertex3f(0.5, 0.5, 0.5);
                  glTexCoord2f(1.0, 1.0);
                  glVertex3f(-0.5, 0.5, 0.5);
                  glTexCoord2f(0.0, 1.0);
                  glVertex3f(-0.5, -0.5, 0.5);
            glEnd();

            glBegin(GL_QUADS);  // right face
                  glTexCoord2f(0.0, 0.0);
                  glVertex3f(0.5, 0.5, -0.5);
                  glTexCoord2f(1.0, 0.0);
                  glVertex3f(0.5, 0.5, 0.5);
                  glTexCoord2f(1.0, 1.0);
                  glVertex3f(0.5, -0.5, 0.5);
```

```
            glTexCoord2f(0.0, 1.0);
            glVertex3f(0.5, -0.5, -0.5);
        glEnd();

        glBegin(GL_QUADS);  // left face
            glTexCoord2f(0.0, 0.0);
            glVertex3f(-0.5, -0.5, 0.5);
            glTexCoord2f(1.0, 0.0);
            glVertex3f(-0.5, 0.5, 0.5);
            glTexCoord2f(1.0, 1.0);
            glVertex3f(-0.5, 0.5, -0.5);
            glTexCoord2f(0.0, 1.0);
            glVertex3f(-0.5, -0.5, -0.5);
        glEnd();

        glBegin(GL_QUADS);  // bottom face
            glTexCoord2f(0.0, 0.0);
            glVertex3f(0.5, -0.5, 0.5);
            glTexCoord2f(1.0, 0.0);
            glVertex3f(-0.5,-0.5, 0.5);
            glTexCoord2f(1.0, 1.0);
            glVertex3f(-0.5, -0.5, -0.5);
            glTexCoord2f(0.0, 1.0);
            glVertex3f(0.5, -0.5, -0.5);
        glEnd();

        glBegin(GL_QUADS);  // back face
            glTexCoord2f(0.0, 0.0);
            glVertex3f(0.5, 0.5, -0.5);
            glTexCoord2f(1.0, 0.0);
            glVertex3f(0.5, -0.5, -0.5);
            glTexCoord2f(1.0, 1.0);
            glVertex3f(-0.5, -0.5, -0.5);
            glTexCoord2f(0.0, 1.0);
            glVertex3f(-0.5, 0.5, -0.5);
        glEnd();
    glPopMatrix();
```

```
      return;
} // end of void DrawTextureCube(GLfloat, GLfloat, GLfloat)

void init(void) {

      glClearColor(0.0, 0.0, 0.0, 0.0);
      glShadeModel(GL_SMOOTH);
      glEnable(GL_DEPTH_TEST);
      glEnable(GL_CULL_FACE);
      glFrontFace(GL_CCW);
      glEnable(GL_TEXTURE_2D);

      bitmapData =
            Bitmap_LoadBitmapFile("Checkerboard_64.bmp",
                                  &bitmapInfoHeader);

      glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
      glGenTextures(1, &texture);
      glBindTexture(GL_TEXTURE_2D, texture);

      glTexParameteri(GL_TEXTURE_2D,
                      GL_TEXTURE_MAG_FILTER,
                      GL_NEAREST);
      glTexParameteri(GL_TEXTURE_2D,
                      GL_TEXTURE_MIN_FILTER,
                      GL_NEAREST);

      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
                   bitmapInfoHeader.biWidth,
                   bitmapInfoHeader.biHeight, 0,
                   GL_RGB, GL_UNSIGNED_BYTE, bitmapData);

      return;
} // end of void initialize(void)

void reshape(int width, int height) {
```

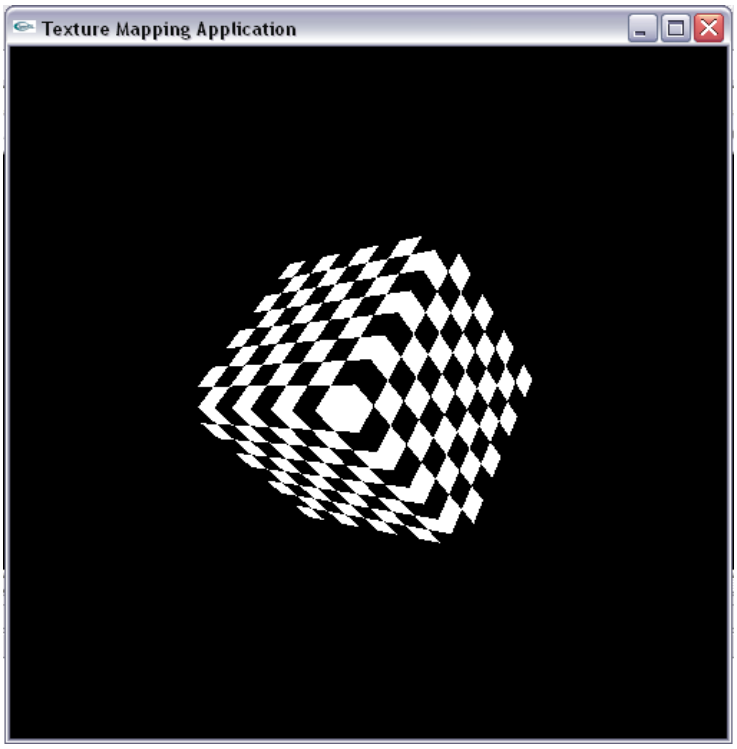
```
	glViewport(0, 0, width, height);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(60.0,
		(GLfloat)width/(GLfloat)height,
		1.0, 20.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

	return;
} // end of void resize(int, int)
void display(void) {

	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	glPushMatrix();
		glEnable(GL_TEXTURE_2D);
		glTexEnvf(GL_TEXTURE_ENV,
			GL_TEXTURE_ENV_MODE, GL_REPLACE);
		glBindTexture(GL_TEXTURE_2D, texture);
		glTranslatef(0.0, 0.0, -3.0);
		glRotatef(rotationY, 0.0, 1.0, 0.0);
		glRotatef(rotationX, 1.0, 0.0, 0.0);
		DrawTextureCube(0.0, 0.0, 0.0);
		glDisable(GL_TEXTURE_2D);
	glPopMatrix();
	glFlush();
	glutSwapBuffers();
	return;
} // end of void display(void)

void spin(void) {

	rotationX += dx;
	rotationY += dy;
```

```
      if (rotationX > 360.0)
            rotationX -= 360.0;

      if (rotationY > 360.0)
        rotationY -= 360.0;

      glutPostRedisplay();
      return;
} // end of void spin(void)

void mouse(int button, int button_state,
           int x, int y )
{
    if ( button == GLUT_LEFT_BUTTON &&
            button_state == GLUT_DOWN ) {
                  glutIdleFunc(NULL);
                  if (last_x == x)
                        dx = 0.0;
                  if (last_y == y)
                        dy = 0.0;
            last_x = x;
            last_y = y;
      } // end of if statement
      else
            glutIdleFunc(spin);


      return;
} // end of void mouse(int, int, int, int)


void motion(int x, int y ) {

      timeElapsed = HiResTimer_GetElapsed_Seconds(timer);
      rotationX += (float) (y - last_y);
      rotationY += (float) (x - last_x);
```

```
      dx = (float)(y - last_y)/(1000*timeElapsed);
      dy = (float)(x - last_x)/(1000*timeElapsed);

      if (rotationX > 360.0)
            rotationX -= 360.0;

      if (rotationY > 360.0)
        rotationY -= 360.0;

      last_x = x;
      last_y = y;

      glutPostRedisplay();

      return;
} // end of void motion(int, int)

void keyboard(unsigned char key, int x, int y) {

    switch(key) {
        case 27:
        case 'q':
            exit(0);
        default:
            break;
      } // end of switch(key)

  return;
} // end of void keyboard(unsigned char, int, int)

int on_exit(void) {

      free(bitmapData);
      bitmapData = NULL;
      free(timer);
      timer = NULL;
```

```
	return 0;
} // end of void on_exit(void)

int main(int argc, char** argv) {

	timer = HiResTimer_Create();

	if (!HiResTime_Init(timer)) {
		MessageBox(NULL, "Timer initialization failed",
				0, 0);
		free(timer);
		timer = NULL;
		exit(-1);
	} // end of if (!HiResTime_Init(timer))

	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
	glutInitWindowSize(500, 500);
	glutInitWindowPosition(100, 100);
	glutCreateWindow("Texture Mapping Application");
	init();
	glutDisplayFunc(display);
	glutReshapeFunc(reshape);
	glutKeyboardFunc(keyboard);
	glutMotionFunc( motion );
	glutMouseFunc( mouse );

	_onexit(on_exit);

	glutMainLoop();

	return 0;
} // end of int main(int, char**)
```

Resim 2-17, bu kodu çalıştıracağınız zaman, texture mapping ile kaplanmış olan kübü göstermektedir.



Resim 2-17

Kodu inceledikten sonra, kod içerisinde birkaç bilinmedik fonksiyon görüceksiniz. Bunlardan biri **glPixelStorei()** fonksiyonudur. Bu fonksiyonu açıklamadan önce birkaç söylenecek şey var. Geliştirmiş olduğunuz bir projeyi bir makineden diğer bir makineye taşıyorken, projeniz daha yavaş çalışır. Bununla ilgili bir çok faktör olmasına rağmen, bu faktörlerden ilgineceğimiz sadece biri olan hafızadaki alignment' tır. Bazı makineler-
de veri aktarımı daha hızlı yapılsın diye, veri 2, 4 veya 8 byte'lık sınırlar çerçevesinde memoride doğru bir pozisyonda konumlanır. Bu durumda, **glPixelStorei()** fonksiyonunu kullanarak, veri pozisyonunu (data alignment) kontrol edebilirsiniz.

```
void glPixelStorei(GLenum pname, Type param);
```

Fonksiyonu kullanmak için, *pname* parametresi GL_PACK_ALIGNMENT veya GL_UNPACK_ALIGNMENT olabilir. *param* parametresi 1, 2, 4 veya 8 olabilir. *pname* parametresini GL_PACK_ALIGNMENT olarak belirlediğinizde, OpenGL' e ve-

rinin nasıl paketleneceğini anlatırsınız. GL_UNPACK_ALIGNMENT parametresini kullandığımızda, OpenGL' e verinin hafızada nasıl paketlenmeyeceği anlatılır.

İlk olarak gördüğünüz diğer fonksiyonlar zaman ile ilgili olan fonksiyonlardır. HiResTimer.h başlık dosyası bu fonksiyonların tanımlamasını yapmaktadır. Zaman ile ilgili fonksiyonları niçin kullandımızı ve bu fonksiyonların ne işe yaradığını anlatalım.

Zaman, dünyada bir niteliktir, kendi oyun dünyanızda ise bu zaman kavramı bir önem taşımaktadır. Zamanı dikkatli bir şekilde izleyebilmek için bir timer kullnırsınız. Bu timer fiziksel hesaplamalara daha çok gerçeklik kazandırır ve hesaplamaları, çizim esnasındaki her çizilen çerçeveden (frame) bağımsız bir şekilde takipedebilirsiniz.

İki tip timer kullanabilirsiniz: Yüksek performansa sahip olan bir timer veya Windows' un mesaj olarak gönderdiği timer, WM_TIMER. Bu ikisi arasındaki en önemli fark hız ve doğruluktur. Biz yüksek performansa sahip olan timer' ı kullanıcaz çünkü hıza ve doğruluğa sahiptir. Eğer bilgisayarınız bunu desteklemiyorsa, bunun yerine WM_TIMER' ı kullanın. Yüksek perfomanstaki timer' ı kullanabilmek için makinenizde bir saniyede kaç clock atıldığını belirlemeniz gerekmektedir. Bundan sonra o anki zamanı öğrenip, uygulamanızın ne zaman başladığını gösteren bir değişkenin içine bu zamanı saklamanız gerekir. Bu her bir taslağı gerçekleştirmek için **QueryPerformanceFrequency()** ve **QueryPerformanceCounter()** fonksiyonlarını kullanmalısınız.

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER *lpFrequency);
BOOL QueryPerformanceCounter(LARGE_INTEGER *lpPerformanceCount);
```

QueryPerformanceFrequency() fonksiyonu bir saniyede kaç clock atıldığını göstermektedir. Clock sayısını *lpFrequency* işaretçisi aracılığı ile elde ederiz. Donanımınız bu fonksiyonu desteklemiyorsa, bu işaretçinin göstereceği değer sıfır olabilir ve fonksiyondan dönen değer sıfır' dır. Eğer donanımınız bu fonksiyonu destekliyorsa fonksiyondan dönen değer sıfırdan farklı olacaktır. **QueryPerformanceCounter()** ise o anki

timer' ın current değerini dönderir ve dönen değer *lpPerformanceCount* işaretçisi tarafından gösterilmektedir. Bu fonksiyonun donanım tarafında desteklenip desteklenmemesi gibi durumunda fonksiyonun izlediği tutum, **QueryPerformanceFrequency()** fonksiyonunun izlediği durumdan farksız olacaktır. Her iki fonksiyon da aynı paramere tiplerini almaktadırlar. Bu parametre tipi 64-bitlik integer olarak ifade edilir ve ismi LARGE_INTEGER' dır.

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG  HighPart;
    };
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

LowPart alanı 32-bitlik low-order parçasını ve HighPart ise 32-bitlik high-order parçasını belirtmektedir. QuadPart ise 64-bitlik işaretli bir tam sayıyı gösterir. Dikkat edeceğiniz gibi bu veri yapısı bir union' dur.

Örnek 2-10, HiResTimer.h başlık dosyasını ve HiResTimer.c ise bu başlık dosyasındaki fonksiyonların gerçekleştirimlerini göstermektedir.

Örnek 2-10

```
// HiResTimer.h

#ifndef _HIRESTIMER_H
#define _HIRESTIMER_H

#define WIN32_LEAN_MEAN

#include <windows.h>

struct _HIRESTIMER;
typedef struct _HIRESTIMER* HiResTimer;
```

```
extern HiResTimer HiResTimer_Create(void);
extern void HiResTimer_Destroy(HiResTimer*);
extern BOOL HiResTime_Init(HiResTimer);
extern float HiResTimer_GetElapsed_Seconds(HiResTimer);
extern float HiResTimer_GetFPS(HiResTimer, unsigned int);
extern float HiresTimer_LockFPS(HiResTimer, unsigned char);

#endif
// HiResTimer.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include "HiResTimer.h"

struct _HIRESTIMER {
       LARGE_INTEGER m_StartTime;
       LARGE_INTEGER m_ticksPerSecond;
       LARGE_INTEGER s_lastTime_GetElapsed_Seconds ;
       LARGE_INTEGER s_lastTime_GetFPS;
       LARGE_INTEGER s_lastTime_LockFPS;
};

HiResTimer HiResTimer_Create(void) {

      HiResTimer _this;
      _this = (HiResTimer)
                  malloc(sizeof(struct _HIRESTIMER));

      if (!_this) {
            MessageBox(NULL, "\tOut of memory", "Error", 0);
            return NULL;
      } // end of if (!_this)

      return _this;
} /* end of HiResTimer HiResTimer_Create(LARGE_INTEGER,
                                          LARGE_INTEGER)
```

```
  */

void HiResTimer_Destroy(HiResTimer *_this) {

      free(*_this);
      *_this = NULL;

      return;
} // end of void HiResTimer_Destroy(HiResTimer*)

BOOL HiResTime_Init(HiResTimer _this) {

      if (!QueryPerformanceFrequency(
            &(_this->m_ticksPerSecond)))
                  return FALSE;
      else {
            QueryPerformanceCounter(&(_this->m_StartTime));
            _this->s_lastTime_GetElapsed_Seconds =
                                          _this->m_StartTime;
            _this->s_lastTime_GetFPS = _this->m_StartTime;
            _this->s_lastTime_LockFPS = _this->m_StartTime;
            return TRUE;
      }
} // end of BOOL HiResTime_Init(HiResTimer)

float HiResTimer_GetElapsed_Seconds(HiResTimer _this) {

      LARGE_INTEGER currentTime;
      float seconds;

      QueryPerformanceCounter(&currentTime);
      seconds = ((float)currentTime.QuadPart -
      (float)_this->s_lastTime_GetElapsed_Seconds.QuadPart) /
      (float)_this->m_ticksPerSecond.QuadPart;

      _this->s_lastTime_GetElapsed_Seconds = currentTime;
      return seconds;
```

```
} // end of float HiResTimer_GetElapsed_Seconds(HiResTimer)

float HiResTimer_GetFPS(HiResTimer _this,
                        unsigned int elapsedFrames)
{

      LARGE_INTEGER currentTime;
      float fps;

      QueryPerformanceCounter(&currentTime);

      fps = (float)elapsedFrames*
            (float)_this->m_ticksPerSecond.QuadPart /
            ((float)currentTime.QuadPart -
            (float)_this->s_lastTime_GetFPS.QuadPart);

      _this->s_lastTime_GetFPS = currentTime;

      return fps;
} /* end of float HiResTimer_GetFPS(HiResTimer,
                                    unsigned int)

  */

float HiresTimer_LockFPS(HiResTimer _this,
                         unsigned char targetFPS)
{

      LARGE_INTEGER currentTime;
      float fps;

      if (targetFPS == 0) targetFPS = 1;

      do {

            QueryPerformanceCounter(&currentTime);
            fps = (float) _this->m_ticksPerSecond.QuadPart /
```

```
                    ((float)(currentTime.QuadPart -
                    _this->s_lastTime_LockFPS.QuadPart));

        }while(fps > (float)targetFPS);

        _this->s_lastTime_LockFPS = currentTime;

        return fps;
} /* end of float HiresTimer_LockFPS(HiResTimer,
                                        unsigned char)
  */
```

HiResTimer_Create() fonksiyonu sizin için bir timer yaratır ve döndüreceği değer HiResTimer olan bir işaretçi dönderecektir. Bu işaretçi yaratılan timer' ı göstermektedir**. HiResTimer_Destroy()** ise yarattığınız timer' ı yok eder, alacağı parametre ise bu yok edilecek olan timer' ın işaretçisidir. Bu işaretçinin gösterdiği timer' ın içi boşaltılıp işaretçiye NULL değeri atanmaktadır.

HiResTime_Init() fonksiyonu timer' ı ilklemektedir. Bu ilkleme işleminde, bir saniyede atılan clock sayısı ve o anki başlangıç clock değeri bulunur. **HiResTimer_GetElapsed_Seconds()** fonksiyonu, bize fonksiyonun çağrılmasından bu yana nekadar saniyenin geçtiğini döndürmektedir. Dönüş değeri float tipindedir. **HiResTimer_GetFPS()**, *elapsedFrames* parametresi üzerinden ortalama nekadar frame' in gösterildiğini söylemektedir. **HiresTimer_LockFPS()** fonksiyonu ise istenilen frame sayısı kadar frame oranını kilitleyerek bekletir. Bu istenilen frame sayısı *targetFPS* parametresi ile fonksiyona aktarılmaktadır.