KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

BIL 409 BİLGİSAYAR GRAFİKLERİ LABORATUARI

Ders Sorumlusu
Öğr.Gör. Ömer ÇAKIR

Deney Sorumluları
Araş.Gör. Çağatay M. YILMAZ
Araş.Gör. Beste ÜSTÜBİOĞLU
Araş.Gör. Mustafa YAZICI
Araş.Gör. Yusuf ÖZEN
Araş.Gör. M. Cemil AYDOĞDU
Araş.Gör. Seda EFENDİOĞLU
Araş.Gör. Bahar HATİPOĞLU
Araş.Gör. Hüseyin ÖZKAYA
Araş.Gör. M. Talha KAKIZ

2015-2016 Güz Dönemi

İÇİNDEKİLER

Önsöz

Bu ders kapsamında yapılacak deneylerle Bilgisayar Grafikleri-I dersinde anlatılan konularının pratik uygulamalarla pekiştirilmesi amaçlanmaktadır.

Laboratuarda, dersin Işın İzleme (Ray Tracing), DirectX 11 ve MAYA ana başlıkları altında anlatılan konularına ilişkin "Görünmeyen Yüzey ve Arkayüz Kaldırma", "DirectX ile FPS Oyunu", "Pürüzlü Yüzey Üretimi", "Maya ile 3D Modelleme", "MAYA ile Animasyon" deneyleri yanı sıra "OpenGL Uygulamaları", "WebGl Uygulamaları", "Yüzey Doldurma Teknikleri" ve "Ters Perspektif Dönüşüm ile Doku Kaplama" gibi değişik bilgisayar grafikleri konularına ait uygulamalar yapılacaktır.

Öğr.Gör. Ömer ÇAKIR

OpenGL Uygulamaları

1. Giriş

OpenGL, en yaygın kullanılan grafik programlama kütüphanesidir. Hızlı ve basit bir şekilde etkileşimli, 2B-3B bilgisayar grafik programları yapmanıza olanak sağlar. Kullanım alanı çok yaygındır ve bilgisayar grafiklerinin hemen hemen tüm alanlarında yaygın olarak kullanılır. Bazı kullanım alanları: araştırma, bilimsel görselleştirme, eğlence ve görüntü efektleri, bilgisayar destekli tasarım, etkileşimli oyunlar...

OpenGL, donanım-bağımsız bir arayüzdür. Görüntüde bulunan nesneleri tanımlamak ve bu nesneler üzerinde gerek duyulan işlemleri gerçekleştirmek için gerekli komutları içerir. OpenGL 'in donanım-bağımsız olmasının nedeni, pencere işlemlerini (ekranda bir pencere oluşturmak gibi) yapan ya da kullanıcıdan girdi alan herhangi bir komutunun bulunmamasıdır. Belirtilen bu işleri gerçekleştirmek için varolan işletim sisteminin mevcut özellikleri kullanılır. Ancak işletim sisteminde pencere işlemlerini gerçekleştirmek karmaşık işlemler içerdiğinden tüm bu işlevleri barındıran ve işletim sistemlerine özel olarak yazılmış GLUT (Graphic Library Utility) kütüphaneleri bulunmaktadır.

OpenGL, 3D nesneleri tanımlamak için yüksek-seviyede komutlar içermez. Bunun yerine; nokta, doğru ve poligon gibi alt-seviye geometrik primitif (ilkel) nesneleri içerir ve bu primitif nesneleri kullanarak karmaşık grafik nesneleri tanımlamamıza olanak sağlar.

2. Programlama Dilleri, İşletim Sistemleri ve Pencere Sistemleri Seviyesi Destek

2.1. Programlama Dilleri

Bir çok uygulama geliştiricisi, OpenGL kütüphanesini üst seviye dillerde kullanmak için bu dillere has uygulama programlama arayüzleri geliştirmişlerdir. Bu dillerden bazıları şunlardır: Ada, Common Lisp, C#, Deplhi, Fortran, Haskell, Java, Perl, Pike, Python, Ruby, Visual Basic...

Tartışma Sorusu-1 : OpenGL kütüphanesi neden herhangi bir programlama dili ile kullanılır? Buna neden ihtiyaç duyulmuştur? Bu diller ile kullanılmasaydı OpenGL ile uygulama geliştirme bazında neler yapılamazdı? Tartışınız.

2.2. İşletim Sistemleri ve Pencere Sistemleri

OpenGL, yaygın olarak kullanılan tüm işletim sistemleri ve pencere sistemlerince desteklenir. Ağ protokolleri ve topolojilerinden tam bağımsızlık sağlar. Tüm OpenGL uygulamaların işletim sistemi ve pencere sistemine bakılmaksızın herhangi bir OpenGL UPA (Uygulama Programlama Arayüzü- API) uyumlu donanımda aynı görsel sonucu üretir. Desteklenen bazı işletim sistemleri ve pencere sistemleri aşağıda listenmiştir:

- Microsoft Windows
- Apple Mac OS
- Linux - Debian, RedHat, SuSE, Caldera
- X Pencere Sistemi (X Window Systems - daha çok GNU/Linux ve Unix benzeri işletim sistemlerinde kullanılan grafik arayüz altyapısıdır.

3. OpenGL Tabanlı Bazı Uygulama Geliştirme Arayüzleri

3.1. OpenGL ES (OpenGL for Embedded Systems /Gömülü Sistemler için OpenGL)

OpenGL ES taşınabilir (mobil) cihazlar, PDA'lar, video oyun konsolları gibi gömülü sistemler için geliştirilmiş 2B/3B uygulama geliştirme arayüzü ve grafik işleme dilidir. Cihazlardaki süzgeçlerin (filtreler) verimli çalışması için telefon GPU'su ile bereber kullanılır. Glut ve Glu gibi kütüphaneler içermez. Telif ücreti gerektirmez ve platformalar arası çalışabilir. Günümüzde çoğu modern cihaz üzerinde bulunur ve bir çok uygulamada kullanılmıştır. Örneğin, mobil cihazlar için geliştirilen fotoğraf paylaşma uygulaması olan Instagram'da OpenGL ES kullanılmıştır.

OpenGL destekli bazı cihazlar: Samsung taşınabilir telefonlar, BlackBerry OS 7.0 ve sonrası BlackBerry cihazları, Aplple (iPad, iPhone vs.), Google Native Client...

3.2. WebGL

Web sayfaları üzerinde 3 boyutlu grafikler oluşturmak için kullanılan platforma bağımsız ve ücretsiz bir uygulama geliştirme arayüzüdür. HTML 5'in web üzerinde yaygınlaşmasıyla birlikte kullanımı artmıştır. Güncel internet tarayıcılarının çoğu tarafından desteklenmektedir.

4. OpenGL

4.1. Kullanım Avantajları

OpenGL kullanarak grafikler oluşturmanın avantajları aşağıda sıralanmıştır.

- Platform bağımsızdır (Windows, Linux, Mac) ve tüm OpenGL UPA uyumlu donanımlar üzerinde çalışır.
- Çok çeşitli sistemler üzerinde koşulabilir. (Kişisel bilgisayarlar, iş istasyonları, süper bilgisayarlar, gömülü sistemler vs.)
- Sistem kaynaklarını optimum şekilde kullanır.
- Bir çok programlama dili tarafından çağırılarak kullanılabilir.
- Kolay anlaşılır, hızlı öğrenilir.
- İçerdiği işlevlerin belgelendirmesi çok iyi yapılmıştır ve ücretsiz bol miktarda eğitici dokümana sahiptir.
- IBM, Sony, Google, Intel, Samsung'un da içinde bulunduğu şirketler tarafından grafik alanında açık standartları oluşturması amacıyla desteklenir ve finanse edilir.

4.2. Open GL Utility (GLUT)

OpenGL platformdan bağımsız olduğu için bazı işlemler bu kitaplık ile yapılamaz. Örneğin kullanıcıdan klavye veya fare ile veri almak, bir pencere çizdirmek gibi işler hep kullanılan pencere yöneticisi ve işletim sistemine bağlıdır. Bu yüzden bir an için OpenGL'in platform bağımlı olduğu düşünülebilir. Çünkü çalışma penceresini her pencere yöneticisinde (her ortamda) farklı çizdirecek bir canlandırma programı yazmak demek her bilgisayarda çalışacak ayrı pencere açma kodu yazmak demektir. Bu ise OpenGL'in doğasına aykırıdır. Bu gibi sorunları aşmak için OpenGL Araç Kiti (GLUT - OpenGL Utility Toolkit) kullanılmaktadır. Bu yüzden bu deneyde GLUT kitaplığı kullanılarak klavye ve fare için işletim sisteminden bağımsız giriş/çıkış işlemleri yapılması sağlanmıştır.

Aşağıda sık kullanılan bazı pencere işlevleri listelenmiştir :

- **glutInit()** işlevi GLUT'ı ilkler, diğer GLUT rutinlerinden önce bu komutun yazılması zorunludur.
- **glutInitDisplayMode()** işlevi renk modunu belirlemektedir.
- **glutInitWindowPosition()** işlevi ekranın sol-üst köşesini baza alarak grafik penceresinin ekrandaki yerini belirler.
- **glutInitWindowSize()** işlevi pencerenizin büyüklüğünü ayarlar.
- **glutCreateWindow()** işlevi OpenGL conteksli bir pencere oluşturur.
- **glutDisplayFunc()** işlevi, çizim penceresinin içeriğinin yeniden gösterileceği durumlarda çalıştırılacak fonksiyonu (çizim işlemlerinin yapıldığı fonksiyon) çağırır. Parametre olarak bu fonksiyonun adını alır.
- **glutKeyboardFunc()** ve **glutMouseFunc()** işlevleri klavyenin veya farenin herhangi bir tuşuna basıldığında çalıştırılacak fonksiyonu çağırır.
- **glutReshapeFunc()** işlevi, pencere büyüklüğünün değişeceği durumlarda çalıştırılacak fonksiyonu çağırır.

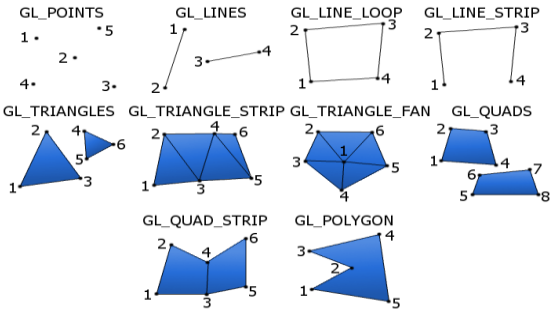
4.3. OpenGL Söz dizimi

OpenGL'de her komutunun önüne gl ön eki getirilmektedir (örneğin glBegin()) Aynı şekilde, tanımlanmış OpenGL sabitlerinin önüne GL ön eki getirilir (örneğin GL_ POLYGON). Ayrıca komut bildirindeki bazı ekler de bu komutlara birer anlam katmak için kullanılır. Örneğin glColor3f() komutunu incelersek, Color eki renk ile ilgili bir komut olduğunu, 3 eki 3 tane parametre aldığını ve f eki ise aldığı parametrelerin kayan noktalı sayı (float) tipinde olduğu anlaşılır.

4.4.OpenGL İlkel (Primitif) Geometrik Nesneleri

İlkel geometrik nesneler, OpenGL'in çizebildiği basit nokta, çizgi, poligon gibi nesnelerdir. (Şekil 1'de OpenGL ile çizilebilen ilkel geometrik nesneler gösterilmiştir.) Bu geometrik nesneler koordinat bilgileri ile tanımlanırlar ve bu koordinat bilgilerine köşe (vertex) denmektedir. OpenGL bu köşe bilgileri ile ilkel olan geometrik şekilleri çizebilmektedir. Fakat çizilecek olan nesnenin nokta, çizgi veya poligon olup olmadığını OpenGL'e bildirmek gerekir. Bu bildirim glBegin fonksiyonu tarafından gerçekleştirilir. Ardından köşe bilgileri aktarılıp nesnenin çizimini ve çizme modunun bittiğini göstermek için glEnd fonksiyonu kullanılır. Aşağıdaki glBegin, glEnd fonksiyonları ve köşe değerleri ile bir poligon nesnesinin çizimi gösterimektedir:

```
glBegin(GL_POLYGON);                    //Poligon çizmeye başla komutu.
        glVertex2f(0.25, 0.25);         //1. köşenin x ve y bileşenleri
        glVertex2f(0.75, 0.25);         //2. köşenin x ve y bileşenleri
        glVertex2f(0.50, 0.75);         //3. köşenin x ve y bileşenleri
glEnd();                                //Poligon çizmeyi bitir komutu.
```

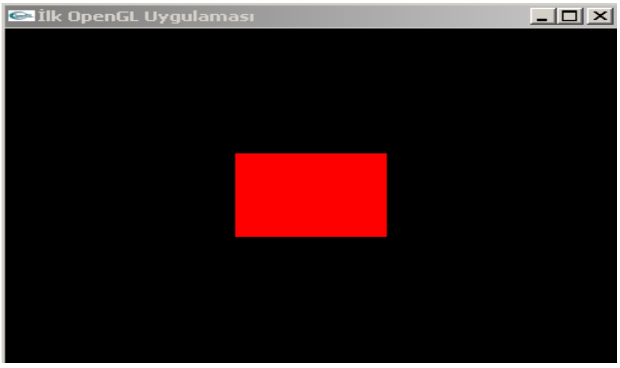


Şekil 1. İlkel (Primitif) Geometrik Nesneler

4.5. İlk OpenGL Uygulaması

Programda ilk olarak bir pencere oluşturulmakta daha sonra da display fonksiyonu ayarlanmaktadır. Display fonksiyonu içerisinde de her defasında çizilecek olan grafik çizilmektedir. Bu hali ile verilen kod basit bir OpenGL programının iskeletini oluşturmaktadır. Program çalıştırıldığında elde edilen ekran görüntüsü Şekil 2'de verilmiştir.

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

void ayarlar(void){
    glClearColor(0.0,0.0,0.0,0.0);
    glOrtho(-2.0, 2.0, -2.0, 2.0, -1.0, 1.0);       //Koordinat sistemini ayarla
}

void display(void){
    glClear(GL_COLOR_BUFFER_BIT);            // Renk bufferını  temizle
    glColor3f(1.0, 0.0, 0.0);                //Renk değeri ata
    glBegin(GL_POLYGON);                            //Poligon çizmeye başla
    glVertex2f(-0.5, -0.5);                  //Köşe değerleri
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
    glEnd();                                        //Poligon çizimi bitir
    glFlush();                                      //Çizim komutlarını çalıştır
 }

int main(int argc, char **argv){
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
    glutInitWindowPosition(0,0);
    glutInitWindowSize(500,400);
    glutCreateWindow("OpenGL Uygulamaları-I");
    ayarlar();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```



Şekil 2. İlk OpenGL Uygulaması - Dörtgen Çizimi

Tartışma Sorusu-2 : main fonksiyonu içerisinde pencere oluşturmak için kullanılan Glut kütüphanesine ait pencere fonksiyonlarına gönderilen parametreleri değiştirerek ortaya çıkan farkları inceleyiniz. glOrtho fonksiyonu ile koordinat sistemin nasıl değiştirildiğini gönderilen parametreleri değiştirerek gözlemleyiniz. Çizim nesnelerinin koordinat sisteminde nasıl yerleştirildiğini inceleyiniz.

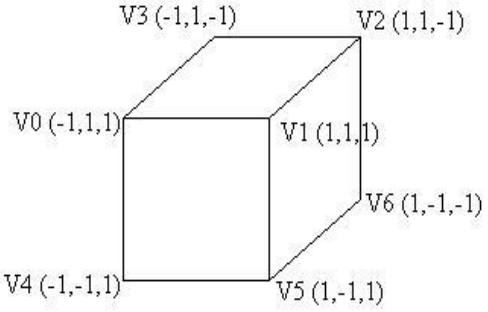
4.6. OpenGL Koordinat Sistemleri ve Dönüşümleri

OpenGL'de 3D grafik işlemlerinde birçok farklı koordinat sistemi kullanılır. Bunlar aşağıdaki gibidir ve bir uzaydan diğerine geçiş için dönüşüm matrisleri kullanılır.

- Nesne Uzayı (Object Space)
- Dünya Uzayı (World Space)
- Kamera Uzayı (Camera Space /Eye Space/View Space)
- Ekran Uzayı (Screen Space/Clip Space)

Nesne Uzayı: Geometrik nesneleri oluştururken nesnenin orjinine göre köşe değerlerinin hesaplanmasında kullanılan koordinat sistemidir. Örneğin, sekiz köşesi olan bir küpü geometrik olarak modellemek için köşe koordinat bilgileri kullanılabilir.

Vertex	Coordinates
0	-1,1,1
1	1,1,1
2	1,1,-1
3	-1,1,-1
4	-1,-1,1
5	1,-1,1
6	1,-1,-1
7	-1,-1,-1



Şekil 3. Modellenmiş Küp ve Köşe Değerleri

Dünya Uzayı : Nesne uzayında modellenen geometrik cisimlerin dünya koordinat sisteminde bir konuma yerleştirilmesi için kullanılan uzaydır. Geometrik nesneler model matrisler kullanılarak bu uzaya taşınır.

Örneğin, başlangıçta (0,0,0) konumundaki yukarıdaki küpü dünya uzayında (5, 0, 0) koordinatlarına taşımak istiyorsak küpün tüm köşe değerleri aşağıdaki gibi bir model matris ile çarpılmalıdır. Aşağıda, (-1, -1, 1) köşesinin +x yönünde 5 birim taşındığında dünya uzayındaki yeni koordinatları hesaplanmıştır.

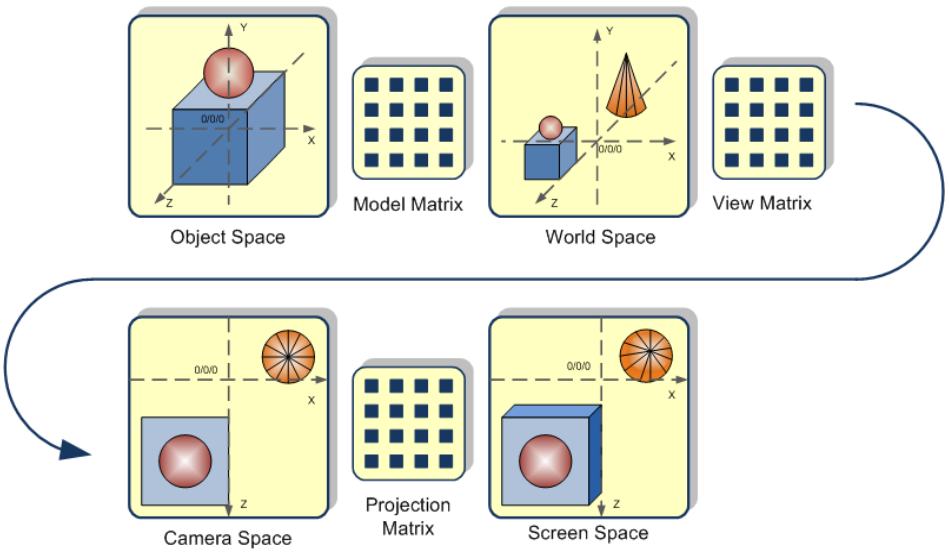
$$\begin{bmatrix} 1 & 0 & 0 & \mathbf{5} \\ 0 & 1 & 0 & \mathbf{0} \\ 0 & 0 & 1 & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \mathbf{-1} \\ \mathbf{-1} \\ \mathbf{1} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{4} \\ \mathbf{-1} \\ \mathbf{1} \\ 1 \end{bmatrix}$$

Model Dönüşüm Matrisi | Nesne Uzayı Köşe Koor. | Dünya Uzayı Koordinatları

Kamera Uzayı: OpenGL kütüphanesi ile uzayda istenilen bir noktaya kamerayı koymak ve bu noktadan istenilen bir yöne istenilen açıyla bakmak için kullanılır. Dünya uzayından kamera uzayına dönüşüm için view matrisler kullanılır.

Projection Space: 3 boyutlu kamera uzayı üzerindeki görüntülerin 2 boyutlu ekranda görüntülenecek biçime dönüştürülmesi için kullanılır. Günümüzde, 3 boyutlu holografik ekranlara sahip olmadığımızdan bu dönüşüm gereklidir. Dönüşüm için projection matrisleri kullanılır.

Aşağıda, OpenGL'de kullanılan tüm koordinat sistemleri ve dönüşüm işlemleri gösterilmiştir.



Şekil 4. OpenGL Koordinat Sistemleri ve Dönüşümler

4.7. OpenGL ile Dönüşüm (Transformation) İşlemleri

4.7.1. Taşıma (Translation)

Bu işlemin amacı bir şekli mevcut konumundan bozulmadan farklı bir konuma taşımaktır. glTanslatef() ve glTranslated() fonksiyonları bu işlemi gerçekleştirir. İki farklı şekilde kullanılabilir:

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

Örneğin, bir küpü koordinat sisteminin merkezinden (5, 5, 5) noktasına taşımak istenirse, ilk olarak modelview matrisini yüklenmeli ve ilklenmelidir. Daha sonra glTranslatef() fonksiyonu ile taşınmalıdır. Aşağıdaki kod yapacağımız işlemi gerçekleştirir:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glTranslatef(5.0, 5.0, 5.0);
KupCiz();                        //küp çizecek fonksiyon
```

Tartışma Sorusu-3 : glMatrixMode(GL_PROJECTION) ve glLoadIdentity işlevleri neden kullanılmaya ihtiyaç duyulmuştur? Modelview matrisi nedir? Araştırınız ve tartışınız.

4.7.2. Döndürme (Rotating)

OpenGL'de döndürme işlemi glRotate*() fonksiyonu ile gerçekleştirilmektedir. İki farklı şekilde kullanılabilir.

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

Örneğin, çizilen bir modeli y ekseni etrafında saat yönüne, 135 derece döndürmek istenirse, aşağıdaki kod bu işlemi gerçekleştirir. Burada y argümanın aldığı 1.0 değeri, y ekseni yöndeki birim vektörü belirtmektedir. İstenilen eksene göre döndürme işlemini yapmak için sadece birim vektörü belirtmek gerekir.

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

4.7.3. Ölçeklendirme (Scaling)

Modelin boyutundaki ayarlamaları yapmak için ölçeklendirme işlemi kullanılmaktadır. Nesnenin boyutları eksenlere göre büyültülüp küçültülebilir. OpenGL'de ölçeklendirme işlemi **glScale***() fonksiyonu ile gerçekleştirilir. İki farklı şekilde kullanılabilir.

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

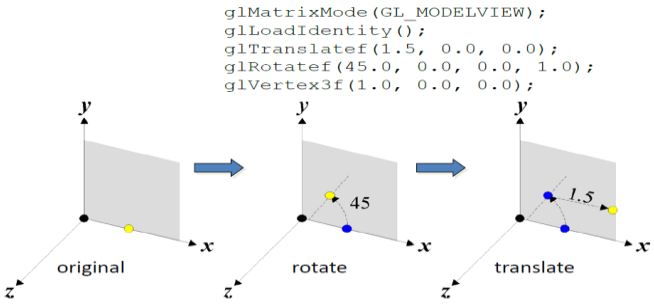
x, y, z parametrelerine geçirilen değerler her bir eksene göre ölçeklendirme değerini belirler. Örneğin, çizilen bir modelin derinlik ve yüksekliğini değiştirmeden, x ekseni üzerindeki genişliğini 2 katına çıkartılmak isteniyosa, aşağıdaki kod bu işlemi gerçekleştirir.

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

4.8.OpenGL Dönüşüm İşlem Önceliği

OpenGL'de dönüşümlerin uygulanma sırası dönüşüm fonksiyonlarının çağrılma sırası ile terstir. Yani çizim nesnesine yakın olan dönüşüm işlemi öncelikli olarak gerçekleştirilir.

Örneğin, aşağıdaki kod parçası çağrıldığında, öncelikle z ekseni üzerinde 45 derece döndürme işlemi yapılmış daha sonra ise x ekseni üzerinde 1.5 birimlik öteleme işlemi gerçekleştirilmiştir. (glVertex3f() çizim nesnesine en yakın dönüşüm en önce gerçekleştirilmiştir.)



Şekil 5. Dönüşüm Uygulanma Sırası

OpenGL'de farklı dönüşüm işlem sırası farklı sonuçlar üretir. Örneğin, aşağıdaki öteleme ve dönme işlemleri farklı sırada gerçekleştirilmiştir ve farklı sonuçlar üreteceklerdir.

```
// Example 1
Display(){
…
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -6.0);
glRotatef(45.0, 0.0, 1.0, 0.0);
glScalef(2.0, 2.0, 2.0);
DrawCube();
…}
= Trans * Rot * Scale * v
```

```
// Example II
Display(){
…
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(45.0, 0.0, 1.0, 0.0);
glTranslatef(0.0, 0.0, -6.0);
glScalef(2.0, 2.0, 2.0);
DrawCube();
…}
= Rot * Trans * Scale * v
```

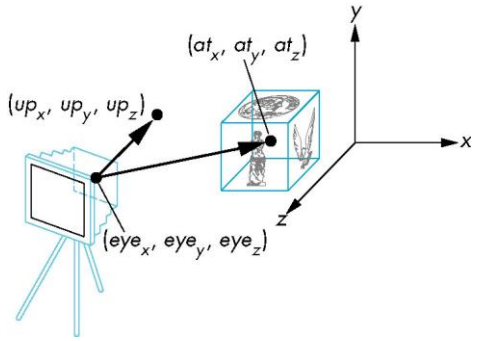
Şekil 6. Dönüşüm İşlem Sırası ve Farklı Çizim Sonuçları

4.9. OpenGL ve Kamera Görüntüsü

OpenGL kütüphanesi ile uzayda istenilen bir noktaya kamerayı koymak ve bu noktadan istenilen bir yöne istenilen açı ile bakmak mümkündür. Bu işlemin 3 öğesi bulunur:

1. Kameranın bulunduğu koordinatlar
2. Kameranın baktığı nokta
3. Kameranın bu eksen üzerindeki açısı

Bu durum kısaca aşağıdaki şekilde özetlenebilir.



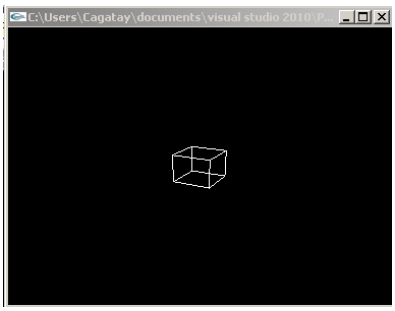
Şekil 7. OpenGL ile Kamera Konumu

Yukarıdaki şekilde de gösterildiği üzere kamera verilen eye_x, eye_y ve eye_z koordinatlarına yerleştirilmiş ve kameranın odak çizgisi verilen at_x, at_y ve at_z koordinatlarına yöneltilmiştir. Bu doğru üzerinde kamera istenildiği gibi döndürülebileceği için bu değeri

belirlemek için kameranın bu eksenle yaptığı normal vektörü de up_x , up_y, up_z değerleri ile belirlenmiştir.

Aşağıdaki örnekte, bir küp çizdirilmiş ve küpe x=3, y=3 ve z=6 kordinatlarında bulunan bir kamaredan bakılmıştır. Ekran görüntüsü Şekil 4'deki gibidir:

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

void init(void) {
 glClearColor(0.0, 0.0, 0.0, 0.0);
 glShadeModel(GL_FLAT);
}

void display(void) {
 glClear(GL_COLOR_BUFFER_BIT);
 glColor3f(1.0, 1.0, 1.0);
 glLoadIdentity();
   gluLookAt(3.0, 3.0, 6.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
 glutWireCube(1.0);
 glFlush();
}

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);
 glMatrixMode(GL_MODELVIEW);
}

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;
}
```



Şekil 8. OpenGL ile kamera görüntüsü

Tartışma Sorusu-4 : gluLookAt fonksiyonuna gönderilen parametreleri değiştirerek farklı açılarda çizim nesnesine bakınız. reshape fonksiyonun hangi nedenle kullanılmış olabileceğini tartışınız.

Tartışma Sorusu-5 : glTranslatef, glRotatef ve glScalef fonksiyonlarını işlevlerini

Tartışma Sorusu-6 : Daire, silindir, daire halkası gibi geometrik şekiller çizmek için kullanılabilecek yöntemleri tartışınız.

5. Deney Hazırlığı

Bu bölüm, deneye gelmeden önce her öğrenci tarafından yapılması gereken maddeleri içermektedir.

1. Deneye gelmeden önce Microsoft Visual C++ 2010 Express programı kurulmalıdır. *http://www.microsoft.com/visualstudio/tur/downloads#d-2010-express* adresinden veya bölümümüzün DreamSpark Premium sayfasından indirebilirsiniz.
2. Ek-1'de verilen adımlar takip edilerek Microsoft Visual C++ 2010 Express programına gerekli kütüphaneler eklenmelidir.
3. Deneyde verilen uygulama kodları Ek-1'de anlatıldığı gibi ide üzerinde çalıştırılmalıdır.
4. Deney föyü dikkatlice okunmalı ve deneye hazırlık soruları cevaplanmalıdır. Deney uygulama yönergesinde gerekli açıklamalar bulunmaktadır.

6. Deney Tasarım ve Uygulaması

1. OpenGL hakkında temel bilgiler soru-cevap şeklinde sorgulanır. Uygulama alanları ve güncel bazı OpenGL uygulamalarına örnekler verilir.
2. GLUT kütüphanesi nedir ve ne için kullanılır soruları cevaplanır.
3. "Kaynak Kodlar" klasörü içerisinde verilen uygulamalar çalıştırılır ve incelenir.
4. Basit bir dörtgen şekli çizen program incelenir. Kullanılan fonksiyonların işlevleri soru -cevap şeklinde sorgulanır.
5. OpenGL ile şekil değiştirme işlemlerinin (taşıma, döndürme ve ölçeklendirme) nasıl yapıldığı, işlem önceliği ve farklı çizim sonuçları tartışılır. OpenGL'de kullanılan koordinat uzayları ve dönüşümleri incelenir.
6. OpenGL ile kamera görüntüsü uygulaması incelenir ve gerekli tartışma soruları cevaplanır. Pareetreler değiştirilerek farklı açılardan kamera görüntüsü incelenir. İlgili fonksiyonların işlevleri soru-cevap şeklinde sorgulanır.
7. Örnek bir OpenGL programı öğrenciler tarafından gerçekleştirilir.

7. Deney Soruları

1. OpenGL nedir? Ne için kullanılır? Kullanım avantajları nelerdir? OpenGL kullanılarak geliştirilen uygulamaları araştırınız.

2. OpenGL ES veya WebGL ile geliştirilmiş güncel bir kaç örnek uygulama araştırınız.

3. GLUT nedir? Ne için kullanılır?

4. 2 boyutlu dörtgen şekli çizen OpenGL komutlarını açıklayınız.

5. Taşıma, döndürme ve ölçeklendirme işlemlerinin koordinat sisteminde nasıl gerçekleştirildiğini kağıt üzerinde basitçe çizerek anlatınız.
6. Dönüşüm işlemleri farklı sırada çağrıldığında farklı çizim sonuçları üretir. Örnek veriniz ve çizerek anlatınız.
7. OpenGL ile kamera görüntüsü uygulamasında kamera görüntüsü almayı sağlayan kodları açıklayınız.

8. Deney Raporu

Deney rapor şablonu bir sonraki sayfadadır. Deney raporu el yazısı ile şablon kapak sayfası olacak şekilde hazırlanacaktır. Raporlar, bir sonraki hafta deneyine kadar teslim edilebilir.

9. Kaynaklar

- An Interactive Introduction to OpenGL Programming - Dave Shreiner,Ed Angel, Vicki Shreiner
- Addison Wesley, "OpenGL Programming Guide", 6th Edition, 2008.
- http://www.opengl.org/
- http://www.khronos.org/
- http://www.opengl.org/wiki/Language_bindings
- http://www.opengl.org/documentation/implementations/#os
- http://www.bilgisayarkavramlari.com/
- http://www.lighthouse3d.com/opengl/glut/
- http://nehe.gamedev.net/

KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

2015-2016 GÜZ DÖNEMİ
OPENGL UYGULAMALARI DENEY RAPORU

GRUP		13:00-15:00		15:00-17:00	
NUMARA		AD SOYAD			

1. Deneyde Yapılanlar

Deneyde yapılanları yazınız.

2. Deney Uygulaması

Deneyde geliştirmeniz istenen OpenGL uygulama kodlarını yazınız.

3. Deney Soruları

Deney sorularını cevaplayınız.

4. Kazanımlar

Deneyden kazanımlarınız ve varsa eksiklikler.

Not: Deney raporu el yazısı ile bu şablon kapak sayfası olacak şekilde hazırlanacaktır. Raporlar, bir sonraki hafta deneyine kadar teslim edilebilir.

WebGL Uygulamaları

1. WebGL Nedir?

WebGL (OpenGL ES 2.0 for the Web) serbest kullanım lisanslına sahip çapraz platform desteğine sahip OpenGL ES 2.0 temelleri üzerine inşa edilmiş aşağı seviye 3D grafik uygulama programlama arayüzüdür (API) [1]. Bilinmesi gereken bazı özellikleri [2, 3] şu şekildedir.

- OpenGL Shading Language (GLSL) programlama dilini kullanan shader temelli API'dir.
- Web sayfalarında `<canvas>` elementini kullanarak etkileşimli 2D ve 3D grafiklerinin görselleştirmesine (rendering) olanak sağlar.
- Başlıca bilinen modern tarayıcı sağlayıcıları WebGL çalışma grubunda yer almaktadır ve tarayıcıların çoğu tarafından WebGL desteği sunmaktadır. Tarayıcı üzerinde implemente edilerek eklenti gerektirmez (plugin free), işletim ve pencere sistemi seviyesinde (operating / window system independence) bağımsızlık sağlar.

Şekil 1. WebGL Destekli Bazı Tarayıcılar

- Uygulamalar uzak sunucu üzerinde saklanabilir.
- Farklı web uygulamalarına kolayca bütünleştirilebilir, CSS ve JQuery gibi standart web paketleri ile birlikte kullanılabilir, web üzerinde taşınabilirliği arttırır.
- Masaüstü ve taşınabilir cihazlar üzerinde birlikte çalışabilir.
- WebGL her geçen gün daha fazla modern GPU özelliklerinden faydalanmakta ve geliştirilmektedir. Hızlı gelişen bir platformdur.

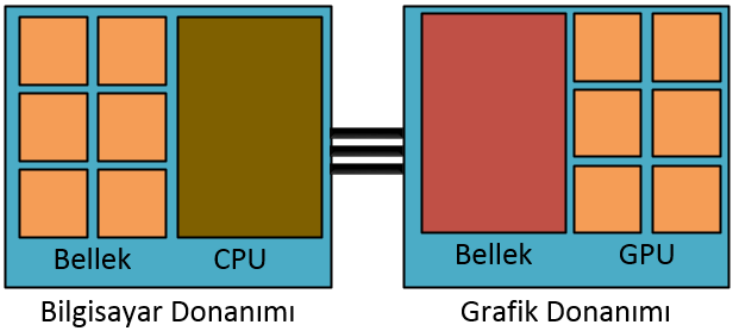
Deney Hazırlığı: WebGL 3D grafik API'si hakkında bilgi sahibi olunuz. Örnek uygulamaları araştırıp, inceleyiniz. WebGL'in eksiklikleri nelerdir, geleceğe yönelik midir, araştırınız.

Deney Sorusu: WebGL OpenGL ES temelli olması nedeni ile fonksiyonellik bakımından kısıtlamalara (ne gibi kısıtlamalar?) sahiptir. Bu eksikliği gidermek için kısıtlamaya sahip olmayan OpenGL üstüne yapılandırılabilir mi? Araştırınız. Münakaşa ediniz.

WebGL idealde makinenin grafiksel hesaplama ve bellek donanımının Merkezi İşlem Biriminden (MİB/CPU) ayrıldığı donanım mimarileri için tasarlanmıştır. Grafik İşlem Birimi (GİB/GPU) programın birçok kopyasını eş zamanlı koşturabilecek şekilde tasarlanmıştır. Normal bir CPU'da koşan programlardan farklı olarak küçük ve basit olması gerekmektedir. Bu tarz bir mimaride ortaya çıkabilecek en büyük sorun 3D grafiksel uygulamalar için CPU ile GPU arasında ortaya çıkacak haberleşme problemidir (önemli) [4, 5].

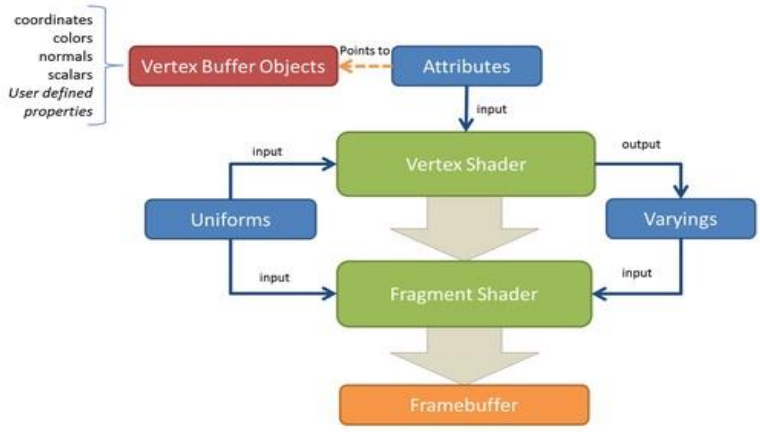
WebGL'in temellendiği fikirde burada ortaya çıkmaktadır. Amaç, iki birim arasında gerçekleşen haberleşme yükünü azaltmaktır. Bu işlem şu şekilde gerçekleştirilir. Komutları ve grafik kaynaklarını (çizim nesnesi köşe nokta bilgileri vd. veriler) sürekli ve parçalı olarak CPU GPU arasında göndermek yerine gerekli tüm grafiksel veriler GPU üzerine bir defada kopyalanır. Kümeler halinde gerçekleştirilen bu işlem ile haberleşme azaltılır ve CPU bağımsız grafik işlemleri gerçekleştirilir [4, 5].

WebGL aşağıda görselde verilen programlanabilir rendering (gerçekleme/görselleştirme) iş akışını [6] kullanmaktadır.



Şekil 2. Donanım

Kısım 2.2'de verilen uygulama kodlarının açıklanmasında gerekli bazı detaylar verilmiştir. Ayrıca, rendering iş akışının bir kısmının nasıl çalıştığını anlamak için Udacity tarafından hazırlanan tanıtıcı uygulamayı inceleyebilirsiniz.



Şekil 3. WebGL Rendering Pipeline (İşlem Akış Hattı) [6]

Deney Hazırlığı/Sorusu: WebGL ile kullanılan yukarıdaki rendering pipeline, sabit işlevselliğe sahip pipeline'a (fixed functionality pipeline) göre ne gibi avantajlara sahip olabilir? Araştırınız. Münakaşa ediniz.

2. Deney Uygulamaları

2.1. Canvas API ile Web Üzerinde 2D Çizim Uygulaması

Daha önce açıklandığı gibi WebGL uygulamaları 3D grafikler oluşturmak için `<canvas>` elementi, Javascript betik dilini ve GLSL shading dilini kullanmaktadır. `<canvas>` elementi web sayfalarında çizim işlemlerinin gerçekleştirileceği alanın belirlenmesi ve erişimi için kullanılır. HTML5 ile tanıtılan `<canvas>` elementi ve Canvas API'si kullanılarak WebGL kullanılmadan 2D çizimlerde gerçekleştirilebilir [2]. API ile ilgili daha detaylı bilgiye [7] ve tanıtıcı uygulamalara [8, 9, 10] bağlantıları kullanarak erişilebilirsiniz.

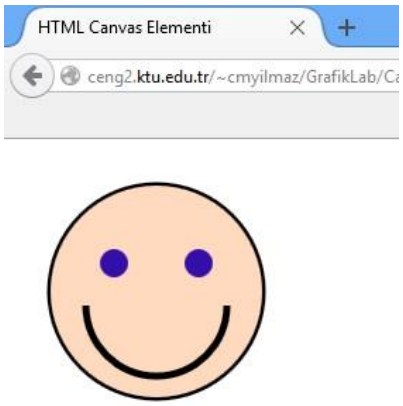
Ekstra: Canvas yerine SVG (Scalable Vector Graphics)'de kullanılabilir [11][12].

Deneyde, web sayfaları üzerinde grafik programlamanın temellerini kavramak amacıyla öncelikle Canvas API ile gerçekleştirilen uygulama incelenecektir (Şekil 4). Uygulama kodlarının bir kısmı aşağıdaki gibidir. Uygulamanın tamamına (.html dosyası) ve kaynak kodlara (web sayfasının kaynak kodlarını tarayıcınız ile görüntüleyerek) bağlantıyı kullanarak erişebilirsiniz. Herhangi bir metin editörü ile açarak html dosyası üzerinde gerekli değişiklikleri yapabilirsiniz.

Verilen uygulamada çizim için gerekli temel adımlara kısaca bakacak olursak;

1. HTML dosyasına `<canvas>` elementinin eklenmesi: `<canvas>` elementi tanımlanmış ve boyut attributeları kullanılarak tarayıcı üzerinde 600x500 boyutunda çizim alanını oluşturulmuştur. Javascript ile `<canvas>` elementine erişilmek için id tanımlayıcısı atanmıştır (Satır 7).
2. Javascript ile `<canvas>` elementine erişilmiştir (Satır 11).
3. 2D grafik çizimleri için rendering içeriğinin edinilmiştir (Satır 12).
4. Desteklenen Canvas API metotları ile 2D çizim işlemleri gerçekleştirilmiştir (Satır 15 ve sonrası).

```
1   <html>
2   <head>
3   <title>İlk Canvas API Uygulaması</title>
4   <meta charset="utf-8" />
5   </head>
6   <body>
7   <canvas id="canvascik" width="600" height="500">
8     Tarayıcı <canvas> elementini desteklemiyor!
9   </canvas>
10  <script type="text/javascript">
11  var canvas = document.getElementById("canvascik");
12  var icrk = canvas.getContext("2d");
13
14  // yüz
15  icrk.beginPath();
16  icrk.arc(100, 100, 75, 0, 2 * Math.PI, false);
17  icrk.lineWidth = 5;
18  icrk.stroke();
19
20  …
21
22 </script>
23 </body>
24 </html>
```



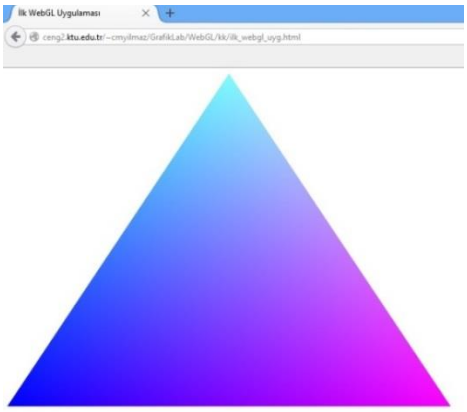
Şekil 4. İlk Canvas API uygulaması

Deney Hazırlığı: Ekte de verilen uygulama kodlarını ile bağlantıda verilen Canvas API'sini inceleyiniz. Çalıştırınız. (`<canvas>` destekli bir web tarayıcısı ile açmanız yeterli). Basit 2D çizimler gerçekleştiriniz. Yeterli seviyede HTML, Javascript bilgisine sahip olunuz.

Deneyin Yapılışı: Basit Canvas API metotları yardımıyla basit çizim işlemlerini gerçekleştirmeniz istenir.

2.2. İlk WebGL Uygulaması

Bu bölümde, ek Javascript kütüphaneleri kullanılmadan ilk WebGL uygulaması geliştirilecektir. Uygulama web sayfası üzerinde `<canvas>` elementi ile belirtilen alan üzerine geçişli renklere sahip aşağıdaki şekli çizdirmektedir. Uygulama ve kaynak kodlara bağlantıyı kullanarak erişebilirsiniz. Herhangi bir metin editörü ile açarak html dosyası üzerinde gerekli değişiklikleri yapabilirsiniz.



Şekil 5. İlk WebGL Uygulaması

Deney Hazırlığı: Aşağıdaki anlatımlardan ve kaynak kodlardan faydalanarak WebGL temelinde geliştirme yapma (özellikle vertex ve fragment shaderlar) deneyiminizi mümkün olduğunca arttırınız.

Deneyin Yapılışı: Deney sırasında örnek kodlar üzerinde değişiklikler yapmanız ve farklı çıktılar üretmeniz istenecektir. Kaynak kodlar üzerinde ve WebGL işleyişi hakkında bilginiz sorgulanacaktır.

WebGL' in temellerinin kavranması için önemli görülen açıklamalar aşağıda detaylıca anlatılmıştır.

Öncelikle `<canvas>` elementi ve rendering içeriğine erişimden bahsedilecek olursa; `getElementById()` metodu `<canvas>` elementinin id tanımlayıcısını parametre olarak alarak çizim nesnesine erişim sağlar. Rendering içeriğinin edinilmesi ise `getContext()` metodu ile gerçekleştirilir. Bu metot, HTML `<canvas>` elementine WebGL ile çizim yapılabilmesi için geriye bir `WebGLRenderingContext` nesnesi (OpenGL ES 2.0 rendering içerik nesnesi) döndürür. `viewport()` metodu ile çizim yapılan WebGL içeriğinin render edilecek çözünürlüğü belirlenir.

Not: `<canvas>` elementinin stil boyutlarının değişmesi web sayfası üzerinde görünen görünüm boyutunu değiştirir ancak rendering çözünürlüğünü değiştirmez.

```
<script type="text/javascript">
…
var canvas = document.getElementById('canvas_element_id'); // JS
var gl = canvas.getContext('experimental-webgl'); // JS
gl.viewport(0, 0, gl.drawingBufferWidth, gl.drawingBufferHeight); //GLSL
…
</script>
```

Değinilecek diğer bir önemli konu 3D nesnelere (meshler) ait vertex (köşe) verilerinin nasıl yükleneceğidir. Bu işlemler Vertex Buffer Object (VBO) nesneleri ile gerçekleştirilir. VBO'lar vertex verilerini (pozisyon, normal vektörleri, renk bilgileri, vb.) rendering için GPU üzerine aktarırlar. Uygulamada, `createBuffer()` bir `WebGLBuffer` bellek nesnesi oluşturulmuş, `gl.bindBuffer()` ile kullanılacak buffer (bellek) atanmıştır. `bufferData()` metodu ile ise çizimde kullanılacak verilerin belleğe atanması sağlanır. Aşağıda bir üçgene ait 3 köşe değeri (2D x, y eksen bileşenleri içeren) belleğe yüklenmiştir.

```
buffer = gl.createBuffer();
if(!buffer){
   console.log('Buffer oluşturulamadı.'); return;
}
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(
gl.ARRAY_BUFFER,
   new Float32Array([
     -1.0, -1.0,  /*Köşe 1*/
      1.0, -1.0,  /*Köşe 2*/
      0.0,  1.0]  /*Köşe 3*/
       ),gl.STATIC_DRAW
);
```

WebGL ile çizim işlemleri GLSL shading dili ile shaderlar kullanılarak gerçekleştirilir. Shaderlar programlanabilir GPU'ların ortaya çıkması ile kullanılmaya başlamıştır ve vertex ve piksellere hükmetmek (programlanabilir hale getirmek) üzere yazılan küçük program parçalarıdır. Shaderlar ayrıca giriş kısmında da anlatılan CPU ve GPU arasındaki iletişimi minimize etmek içinde kullanılır ve GPU üzerinde yoğun paralellikte koşan küçük program parçacıklarının yazılmasına olanak sağlar. Bu nedenle shaderlar daha detaylı incelenecektir.

2.2.1.Vertex Shaderlar

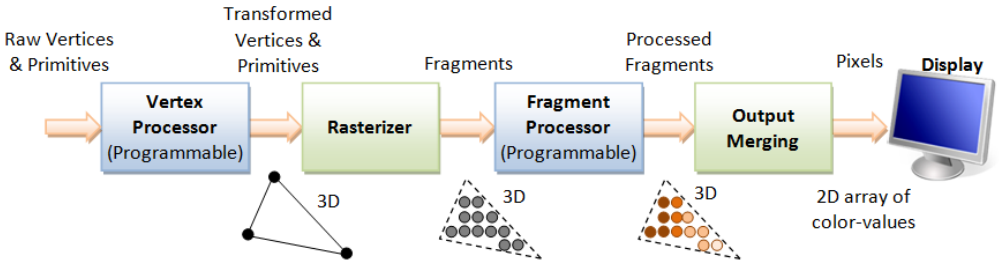
Vertex shaderlar vertex (poligonların köşe noktaları şeklinde düşünülebilir) attributelarının tanımlanması ve işlenmesini sağlarlar (mesh koordinatlarının setlenmesi şeklinde düşünebilirsiniz). Bu işlemler rendering sürecinde çalıştırılacak olan main() fonksiyonları içerisinde gerçekleştirilir. Vertex shaderlar, WebGL rendering pipelinenının ilk aşamasını oluşturur ve her bir vertex üzerinde bir dizi matematiksel işlem gerçekleştirirler. Örneğin, vertex konumlarının (positions) ekran konumlarına dönüşüm (rasterizerin kullanması için), texturing için gerekli koordinatların üretilmesi, vertexlerin ışıklandırma ortamındaki renk değerinin hesaplanması işlemleri gibi. Vertex shader üzerinde yapılan işlem çoğunlukla aşağıdaki matris çarpımı şeklindedir ve aşağıdaki şekilde gerçekleştirilirler [13 ,14].

```
gl_Position = PROJECTION_MATRIX * VIEW_MATRIX * MODEL_MATRIX * VERTEX_POSITION
```

Burada, `VERTEX_POSITION (x, y, z, 1)` mevcut konum değerlerini içeren 1x4 vektör, `VIEW_MATRIX` dünya uzayındaki kameranın görebilme uzayını, `MODEL_MATRIX` nesne uzay koordinatlarını dünya uzayı koordinatlarına çeviren 4x4 matrisi, `PROJECTION_MATRIX` kamera lensini ifade etmektedir [111].

2.2.2. Fragment Shaderlar

Rasterization sonrasında ilkel geometrik şeklin (üçgen vb.) kapladığı alan piksel boyutundaki fragmentlara (piksellere değil) parçalanır. Her bir fragment konum, derinlik değeri bilgisi ve renk, texture koordinatları gibi interpolated parametreleri içerebilir. Diğer bir deyişle zaman içindeki ışıklandırma, transformasyon vb. değişimler sonucunda oluşacak olası yeni pikselin hesaplanması fragmentlar ile gerçekleştirilir. İş akışında vertex shaderların çıktısını alır ve ilgili renk, derinlik bilgilerini atarlar. Bu işlemlerden sonra fragmentlar ekran üzerinde görüntülenmek için frame buffera gönderilir [13, 14].



Şekil 6. 3D Grafik Rendering İş Akışı [15]

Vertex ve fragment shader kullanılarak gerçekleştirilen basit bir 3D grafik rendering pipeline yukarıdaki gibidir. Görselde ifade edilen işlemler kısaca şu şekildedir:

- Vertexlerin İşlenmesi**:** Modeli oluşturan her bir bağımsız vertexler üzerinde işlemler vertex shaderlar ile programlanır.
- Rasterization**:** Her bir primitif geometrik şeklin (örn. üçgen) fragmentlara ayrılması işlemidir. Basit olarak vertexlerin kapladığı alandaki piksel değerlerinin belirlenmesidir.
- Fragmentlerin İşlenmesi: Birbirinden bağımsız her bir fragmentin ortama göre ilgili renk vb. değerlerinin fragment shaderlarla hesaplanmasıdır.
- Çıktıların Birleştirilmesi: Tüm ilkel çizim nesnelerine ait 3D uzaydaki fragmentler kullanıcı ekranında 2D renk-piksel bilgisine dönüştürülür.

Uygulamaya dönecek olursak, vertex shaderlar kısaca şu şekilde çalışmaktadır:

Header içinde tanımlanan `a_position` değişkeni bir attributedur ve bu shader fonksiyonuna gönderilen parametreleri ifade etmektedir. Örneğin, vertexlerin pozisyon bilgileri, renkleri ve texture koordinatları vb. bilgiler vertex shader bu attributeler ile gönderilir. Uygulamada, belleğe atanan ve her bir 2D (`vec2` tip değişkeni kullanılır) köşe koordinatları (`[-1.0, -1.0]` vd.) bu fonksiyona `a_position` attribute bilgisi kullanılarak gelecektir. Gelen her bir köşe değeri `gl_Position` değişkene satır 4'deki gibi atanır. `gl_Position vec4` formatındadır ve 4 bileşene ihtiyaç duyduğundan atama bu şekilde (kullanılmayan son iki konum bileşenine varsayılan değerler verilerek) gerçekleştirilmiştir.

GPU üzerindeki işlemlerin tümü her bir vertex için bu özel tanımlı `gl_Position` değişkeni ile erişilerek gerçekleştirilir.

```
1  <script id="2d-vertex-shader" type="x-shader/x-vertex">
2   attribute vec2 a_position;
3     void main() {
4       gl_Position = vec4(a_position, 0, 1);
5        }
6  </script>
```

Uygulamada fragment shader programı kısaca şu şekilde çalışmaktadır:

Vertex shaderdaki gibi bir main fonksiyonu içerir ve rasterization sonrası hesaplanan herbir fragment için bu metot koşulur. Herbir fragmentin renk değeri ilgili fragmentin x ve y bileşenleri kullanarak satır 3'deki gibi hesaplanır. Burada, `gl_FragColor` özel değişkeni ayrı her bir fragmentin sahip olacağı renk değerini ifade etmektedir. `gl_FragCoord.x` ve `gl_FragCoord.y` değişkenleri ile her bir fragmentin koordinat bilgileri atanmış ve sırası ile çizim alanının yükseklik ve genişlik değerlerine bölünerek geçişli *vec4* formatındaki renk değerleri atanmıştır [16].

```
1  <script id="2d-fragment-shader" type="x-shader/x-fragment">
2  void main(){
3     gl_FragColor = vec4(gl_FragCoord.x / 640.0, gl_FragCoord.y / 480.0, 1.0, 1.0);
4  }
5  </script>
```

Shaderların kullanımındaki diğer bir önemli noktada rendering aşamasında kullanılacak shader programlarının oluşturulmasıdır. Vertex ve fragment shaderlar için aşağıdaki şekilde oluşturulur. Gerekli açıklamalar yorum satırları ile verilmiştir.

```
// -- getElementById ve the id attribute shader kaynak kodunu edin
shaderScript = document.getElementById("2d-vertex-shader");
shaderSource = shaderScript.text;
// -- boş bir shader nesnesi oluştur
vertexShader = gl.createShader(gl.VERTEX_SHADER);
// -- shader nesnesine shader kaynak kodunu ekle
gl.shaderSource(vertexShader, shaderSource);
// -- shader kodunu derle
gl.compileShader(vertexShader);
...
// -- program nesnesi oluştur
program = gl.createProgram();
// -- vertex ve fragment shaderları programa yükle
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
// -- shaderları program nesnesine bağla
gl.linkProgram(program);
// -- mevcut rendering aşamasında bu programı kullan
gl.useProgram(program);
```

Yukarıda sırası ile rendering içeriğine erişilmiş, `WebGLBuffer` bellek nesnesine bir üçgenin köşe noktaları yerleştirilmiş, vertex ve fragment shader program parçaları tanımlanmış ve rendering aşamasında kullanılacak program oluşturulmuştur.
Son olarak `render()` fonksiyonu açıklanacak olursa;

`window.requestAnimationFrame()` metodu ile tarayıcıya bir animasyon yürütüleceği bildirilir ve bir sonraki çizimden önce çağrılacak fonksiyon adı parametre olarak verilir. `clearColor()` metodu renk belleğini `red, green, blue ve alpha` ile belirtilen değerlere setler (temizler). `vertexAttribPointer(attrib, index, gl.FLOAT, false, stride, offset)` WebGL' in veriyi nasıl yorumlayacağını tanımlar. Bu kısmı detaylandırmak istersek, insan tarafından okunurluğu arttırılmak için aşağıdaki gibi düzenlenen diziyi vertex shader nasıl yorumlayacak sorusunu sormamız gerekir [16].

```
new Float32Array[
        -1.0, -1.0,  /*Köşe 1*/
         1.0, -1.0,  /*Köşe 2*/
         0.0,  1.0 ] /*Köşe 3*/
```

Sorunun cevabı olan işlem şu şekilde gerçekleştirilir:
`gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);` çağrısını yaptığımızda, shader peşi sıra okunan `Float` türündeki her iki değeri bir vertex kabul edecektir ve vertex shaderlar bu değerlere `a_Position` attribute değişkeniyle GPU üzerinde erişeceklerdir [16].

```
function render() {
    window.requestAnimationFrame(render, canvas);

    gl.clearColor(1.0, 1.0, 1.0, 0.2);
    gl.clear(gl.COLOR_BUFFER_BIT);
    positionLocation = gl.getAttribLocation(program, "a_position");
    gl.enableVertexAttribArray(positionLocation);
    gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);
    gl.drawArrays(gl.TRIANGLES, 0, 3);
}
```

`render()` fonksiyonunda son olarak `drawArray()` metodu `gl.TRIANGLES` türünde rendering yapılacağını ve bu işlemin buffer içindeki 3 adet vertex kullanılarak gerçekleştirileceği bildirilmektedir.

2.3. WebGL ile 3D Nesne Üretimi

Bu kısımda, ekte kaynak kodları verilen (indirin ve index.html dosyası ile çalıştırın) ve buradan çalışabilir haline erişebileceğiniz 3D küp uygulaması incelenecektir. Uygulamada 3D bir küp nesnesi üretilmiş ve 3D nesnenin çeşitli eksenlerde öteleme ve döndürülmesi gerçekleştirilmiştir. İlk WebGL uygulamasında temelleri kavramak amacıyla hazır herhangi bir kütüphane kullanılmadan tüm metotlar detaylı şekilde incelenmişti. Bu uygulamada shader program hazırlama vb. kısımlarda ek kütüphaneler kullanılarak kod karmaşıklığı, maliyeti azaltılmıştır. Uygulamanın değinilmesi gereken bazı noktalar şu şekildedir.
Kullanılan ve harici olarak eklenen Javascript dosyaları şunlardır: `sylvester.js` ve `glUtils.js` JavaScript ve WebGL ile ilgili vektör, matris üzerindeki işlemleri kolaylaştırır, `webgl-demo.js` ise uygulama kodlarının yazılacağı ana .js dosyasıdır. Ayrıca, uygulamada WebGL ile 3d nesnelerin üretimi hakkında daha detaylı bilgiye MDN üzerinden erişebilirsiniz.

Deney Hazırlığı: Vertex ve fragment shader içeriklerini, 3D nesne üretimi ve transformasyon işlemlerinin nasıl gerçekleştiğini kod üzerinde inceleyiniz.

Deneyin Yapılışı: Örnek kodlar üzerinde değişiklikler yapmanız ve farklı grafiksel çıktılar üretmeniz istenebilir.

Deney Sorusu: Dikkat ederseniz WebGL ile geliştirme yapmak hala zor ve karmaşık gibi görünüyor, çözüm önerileriniz nelerdir, grup üyeleri arasında Münakaşa ediniz.

2.4. Three.js ile WebGL Uygulamaları

Three.js 3D bilgisayar grafiklerinin web tarayıcısı üzerinde oluşturulması ve görüntülenmesine olanak sağlayan bir JavaScript kütüphanesidir. WebGL'in birçok detayından soyutlayarak ile hızlı ve kolay uygulama geliştirmenize olanak sağlar (`Babylon.js` gibi). Deneyin bu kısmında Three.js ile geliştirilen uygulamalar incelenecektir. Kütüphanenin GitHub üzerindeki kaynak kodları inceleyecek olursanız saf WebGL üzerine inşa edildiğini görülebilirsiniz.

Deney Hazırlığı: Gerekli uygulama ve kaynak kodlarını `http://threejs.org/` adresinde download bağlantısına klikleyerek indiriniz. Örnek uygulama kodları `../mrdoob-three.js-d6384d2/examples` klasöründe yer almaktadır. `../examples/index.hml` dosyasını veya `http://threejs.org/examples/` bağlantısını kullanarak uygulamaları inceleyiniz. Html dosyalarını tarayıcı ile çalıştırarak da örnekleri inceleyebilirsiniz.

2.4.1. Three.js ile İlk Uygulama

Bu bölümde `../examples` klasöründe yer alan `webgl_geometry_cube.html` uygulaması incelenecektir. Uygulama kodları ve gerekli açıklamalar şu şekildedir: Öncelikle, WebGL etkin `<canvas>` elementi üzerinde grafik içeriğinin çizim için gerekli renderer aşağıdaki gibi hazırlanır.

```
var renderer = new THREE.WebGLRenderer();
renderer.setPixelRatio( window.devicePixelRatio );
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

`PerspectiveCamera( fov, aspect, near, far )` metodu ile bakış için gerekli kamera görüş bölgesi hazırlanır. `fov` parametresi ile dikey görüş alanı, aspect ile yatay/dikey görüş alan oranı ayarlanır. `near` ve `far` parametreleri ile clipping plane belirlenir. Yani verilen değerler arasındaki nesneler render edilir, görüş alanı dışındaki gereksiz nesnelerin render edilmesi engellenir (önemlidir).

```
var camera = new THREE.PerspectiveCamera( 70,
                                window.innerWidth/window.innerHeight, 1, 1000 );
```

Kamera duruş pozisyonu setlenir: `camera.position.z = 400;`

Three.js görüntü listesi şeklinde bir yapı kullanır. Yöntemde çizim nesneleri bir listede tutulur ve ekrana çizilir. Bunun için öncelikle bir `Three.Scene` nesnesi oluşturulur. `Scene` nesnesi üzerine çizim nesneleri, ışık kaynakları ve kameralar yerleştirilir ve Three.js ile ekran üzerinde neyin ve nerenin render edileceğini belirlenir:

```
var scene = new THREE.Scene();
```

3D bir nesnelerin çizimine gelecek olursak, bu işlemler meshler vasıtasıyla gerçekleştirilir. Her mesh kendi geometri ve materyal bilgisine sahip olur. Geometri, çizim nesneleri için gerekli vertex kümelerini ifade eder. Materyal ise nesnelerin boyama bilgisini ifade eder. Dikkat ederseniz çizim işlemleri Three.js ile aşağıdaki görüleceği üzere kolaylaşır.

```
// - verilen boyutlarda bir box nesnesi oluştur
var geometry = new THREE.BoxGeometry( 200, 200, 200 );
// - texture yükle
var texture = THREE.ImageUtils.loadTexture( 'textures/crate.gif' );
texture.anisotropy = renderer.getMaxAnisotropy();
// - nesneyi kaplamak için yüklenen texture ile materyal oluştur
var material = new THREE.MeshBasicMaterial( { map: texture } );
// - mesh nesnesini oluştur
mesh = new THREE.Mesh( geometry, material );
// - nesneyi render edilecek Scene nesnesine ekle
scene.add( mesh );
```

`onWindowResize()` metodu ile yeni tarayıcı ekran genişlik ve yükseklik bilgileri ile kamera ve renderer parametreleri güncellenir.

```
function onWindowResize(){
	camera.aspect = window.innerWidth / window.innerHeight;
	camera.updateProjectionMatrix();
	renderer.setSize( window.innerWidth, window.innerHeight );
}
```

`animate ()` *fonksiyonundaki* `requestAnimationFrame()` metodu ile tarayıcıya bir animasyon yürütüleceği bildirilir ve bir sonraki çizimden önce çağrılacak fonksiyon adı parametre olarak verilir. Mesh nesnesinin rotation (dönüş) değeri arttırılarak, her çizimde farklı bir açıda çizim yapılır ve nesne sürekli dönecek şekilde anime edilir.

```
function animate(){
	requestAnimationFrame( animate );
	mesh.rotation.x += 0.005;
	mesh.rotation.y += 0.01;
	renderer.render( scene, camera );
}
```

Uygulamada aşağıdaki değişiklikleri yaparak farklı bir animasyon elde edebilirsiniz.

```
<script>
	var startTime = Date.now();
	var camera, scene, renderer;
	...
</script>
function animate(){
	...
	mesh.rotation.y += 0.01;
	var dtime = Date.now() - startTime;
	mesh.scale.x = 1.0 + 0.3 * Math.sin(dtime/500);
	mesh.scale.y = 1.0 + 0.3 * Math.sin(dtime/500);
	mesh.scale.z = 1.0 + 0.3 * Math.sin(dtime/500);
	renderer.render( scene, camera );
}
```

Deney Hazırlığı: Farklı geometri ve materyale sahip 3D nesnelerin üretimini `../examples/webgl_materials.html` ve `../examples/ webgl_geometries.html` uygulamaları ile inceleyiniz.

Deneyin Yapılışı: `Three.js` kütüphanesi ile verilen örnek uygulama kodlarından faydalanarak içeriği sizce belirlenecek bir uygulama geliştirmeniz istenir.

3. Deney Hazırlığı

Deneye hazırlık için yapılması gerekenler ve kaynak kodlar föy içerisinde açıklamalar ve bağlantılarla verilmiştir. Dikkatlice okuyunuz ve deneye hazır geliniz.

Bilinmesi gereken bazı diğer hususlar şu şekildedir: Uygulama kodları Notepad++, Sublime Text, Gedit benzeri basit bir metin düzenleyicisi ile görüntüleyebilir veya düzenleyebilirsiniz. WebGL vertex ve fragment shaderlarını Mozilla Firefox Shader Editör ile görüntüleyebilir ve düzenleyebilirsiniz.

Hazırlık ile ilgili sorularınızı cmylmz@ceng.ktu.edu.tr adresi elektronik posta ile iletebilirsiniz.

4. Deney Tasarım ve Uygulanışı

Hazırlık soruları ile deneye hazırlık seviyesi ölçülecek.

Öğrencilerin deney sorularını cevaplandırması istenecek.

Deneyin yapılışı başlıklarında belirtilen bilgi ölçme ve uygulama geliştirme işlemleri gerçekleştirilecek.

5. Deney Raporu

Rapor adreste yer alan doküman kapak dosyası olacak şekilde hazırlanacaktır. Rapor içeriği deney sırasında bildirilecektir. Rapor grup olarak hazırlanacak ve bir hafta içerisinde teslim edilecektir. Kopya raporları hazırlayan gruplar gerekli sorumluluğu üzerlerine almış sayılırlar.

6. Kaynaklar

1. https://www.khronos.org/webgl/ OpenGL ES 2.0 for the Web, 2015.
2. WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL, Matsuda, K., Temmuz 2015.
3. http://www.siggraph.org/content/siggraph-university-introduction-opengl-programming An Intr. to OpenGL Programming, 2015.
4. https://www.khronos.org/opengles/2_X/ The Standard for Embedded Accelerated 3D Graphics, 2015.
5. http://my2iu.blogspot.in/2011/11/webgl-pre-tutorial-part-1.html WebGL Pre-Tutorials, 2015.
6. https://www.safaribooksonline.com/library/view/webgl-beginners-guide/9781849691727/ch02s02.html Overview of WebGL's rendering pipeline, 2015.
7. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API Canvas API, 2015.
8. https://developer.mozilla.org/en-US/demos/tag/tech%3Acanvas Canvas API Demos, 2015.
9. http://code.tutsplus.com/articles/21-ridiculously-impressive-html5-canvas-experiments--net-14210 21 Ridiculously Impressive HTML5 Canvas Experiments, 2015.
10. http://kripken.github.io/webgl-worker/playcanvas/simple-worker.html Simle Worker Canvas Example, 2015.
11. http://www.sitepoint.com/7-reasons-to-consider-svgs-instead-of-canvas/ Reasons to Consider SVGs Instead of Canvas, 2015.
12. http://www.svgopen.org/2010/papers/58-WebGL__SVG/ Embedding WebGL documents in SVG, 2015.

13. http://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html WebGL Fundamentals, 2015.
14. http://www.html5rocks.com/en/tutorials/webgl/shaders/ An Introduction to Shaders, 2015.
15. http://www3.ntu.edu.sg/home/ehchua/programming/opengl/cg_basicstheory.html 3D Graphics with OpenGL, 2015.
16. https://blog.mayflower.de/4584-Playing-around-with-pixel-shaders-in-WebGL.html Playing around with frag. shaders in WebGL, 2015.

Yüzey Doldurma Teknikleri

1. Giriş

Bu deneyde dolu alan tarama dönüşümünün nasıl yapıldığı anlatılacaktır. Dolu alan tarama dönüşümü poligon içindeki piksellerin bulunup, bu piksellere karşılık gelen doğru parlaklık değerlerinin atanması anlamına gelir. Tarama dönüşümü, katı cisim üretimi için yüzeylerin boyanmasında kullanılmaktadır.

2. Sıralı Kenar Liste Yöntemi (Scan Line Algorithm)

Bu yöntem, poligonun kenarları ile tarama satırlarının kesişim noktalarının kullanılmasına dayanır. Kesişim noktaları bulunup bu noktalar üzerinde sıralama işlemleri yapılarak, belli ölçütler ışığında doldurulması gereken pikseller belirlenir.Yöntemin etkinliği sıralama yönteminin etkinliğiyle doğru orantılıdır.

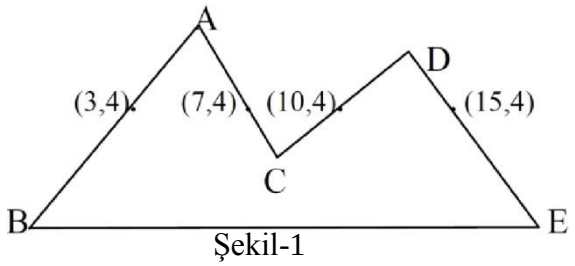
Basit şekilde bu yöntem aşağıdaki adımlardan oluşur:

1. Tüm poligon-tarama satırı kesişimleri bulunur. Poligon, bilinen DDA (Digital Differential Analyzer) veya Breshenam çizgi çizme yöntemleriyle oluşturulacağı için kesişim noktaları, çizgiler çizilirken kolaylıkla elde edilebilir. Noktalar bir listede tutulur. Her bir liste elemanı (x,y) şeklinde bir noktayı işaret edecektir.

2. Bütün (x,y) noktaları, y değerlerinin azalış veya artışına göre sıralandıktan sonra, aynı y değerine sahip noktalar da x'in artan sırasına göre sıralanır.

3. Sıralamadan sonra, listeden nokta çiftleri seçilerek doldurulacak pikseller aşağıdaki kurala göre belirlenir.

Kural: (x_1,y_1) ve (x_2,y_2) nokta çifti için, x tamsayı olmak üzere, x birer arttırılarak $x_1 \leq x + 1/2 \leq x_2$ koşulunu sağlayan (x,y_1) noktaları doldurulur. $(y_1=y_2)$

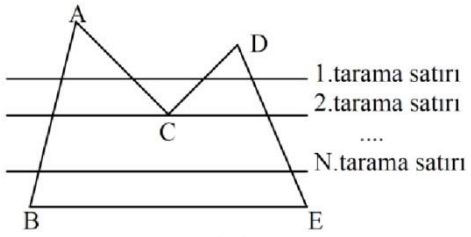


Şekil-1

Yandaki şekilde (3,4)-(7,4) ve (10,4)-(15,4) çiftleri alındığında, yukarıdaki kurala göre doldurulacak noktaların ilk çift için (3,4), (4,4), (5,4), (6,4) ikinci çift için ise (10,4), (11,4), (12,4), (13,4), (14,4) olduğu görülmektedir.

Buna benzer bir yöntem de poligonun bulunduğu düzlem üzerinde, poligon sınırları dışından başlayıp soldan sağa doğru tarama satırları geçirilerek yüzeyin doldurulmasıdır. Soldan sağa doğru ilerlerken tek sayıda olan kesişimlerden sonraki pikseller doldurulur, kesişim sayısı çift olduğu anda doldurma işlemi kesilip, sağa doğru ilerlemeye devam edilir.

Şekil-2'de her bir tarama satırı için doldurulacak pikseller belirlenirken bazı problemlerle karşılaşılabilir. Örneğin ikinci tarama satırında soldan sağa doğru ilerlerken C noktasına rastlandığında doldurma işlemi durdurulacak ve bir sonraki DE kenarıyla kesişme noktasından itibaren yüzey tekrar doldurulmaya başlanacaktır. Yani C noktasından DE kenarına kadar olan pikseller doldurulmamış olacaktır.



Şekil–2

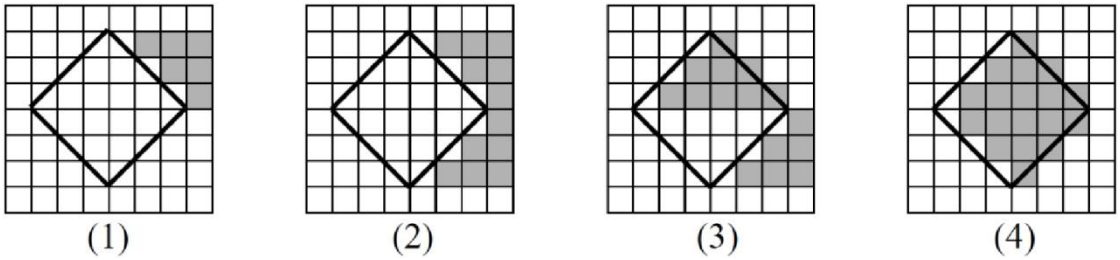
C noktası bir yerel minimum noktasıdır. Yerel minimum noktayı oluşturan kenarlar X-Y düzleminde Y ekseni boyunca artan değerlere sahiptir. Yukarıdaki probleme benzer bir problem D noktası için de vardır ve D noktası bir yerel maksimum noktasıdır. Bu durumda çözüm olarak tarama satırı boyunca yerel maksimum ve yerel minimum noktalar ihmal edilebilir. Kenar liste yönteminde de ikililer şeklinde değerlendirilme olacağından yerel minimum ve yerel maksimum noktaları listeye ikişer kere alınarak problem ortadan kaldırılmış olur.

Bu yöntem, her piksel bir kere adreslendiği için, etkin bir yöntemdir. Hızlı olduğu için gerçek zamanlı uygulamalara uygundur. Üstünlüklerine rağmen bu yöntem sıralama ön işlemine ek olarak yatay çizgiler bulunduran poligonların yatay kenarlarının ayrıca ele alınmasını gerektirir.

3. Kenar Doldurma Yöntemi

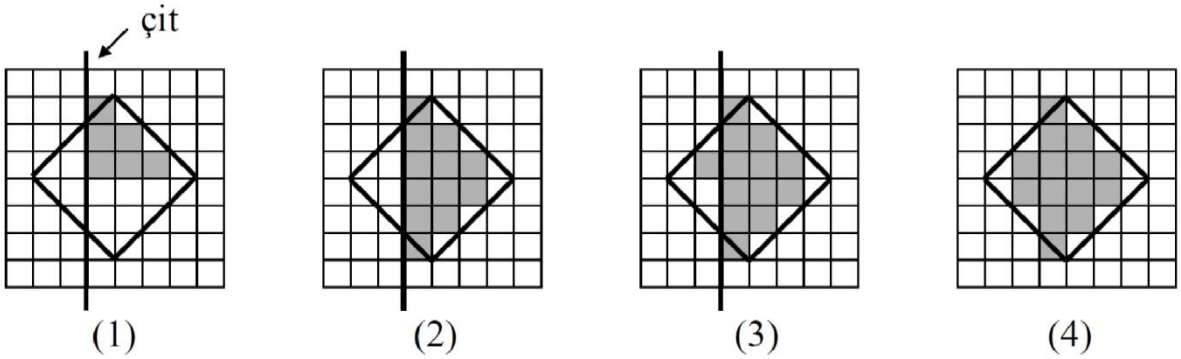
Kenar doldurma tekniğinde kesişim noktalarının tutulduğu listede sıralama ve listeyi düzenleme gibi işlemler yapmadan sadece kenarlar kullanılarak yüzey doldurma işlemi gerçekleştirilir.

Bir kenar seçilir ve o kenarın sağındaki tüm pikseller doldurulur. Eğer sağa doğru ilerlerken rastlanan piksel doldurulmuşsa, o piksel zemin rengine çevrilir. Tüm kenarlara bu işlem uygulanır ve sonuçta doldurulmuş yüzey elde edilir. Şekil-3'te bu uygulamanın adımları görülmektedir.



Şekil-3

Bu yöntemin sakıncası, poligon içindeki ve dışındaki piksellere birçok kere erişilmesidir. Büyük poligonlar için etkinliği azalan bir yöntemdir. Adreslenen piksel sayısını sadece poligon içindeki piksellerle sınırlamak için bir çit (fence) kullanılabilir. Poligonun herhangi bir noktasından bir çit seçilir. Kenarlardan çite doğru tarama işlemine başlanır. Yukarıdaki gibi, zemin renginde olan pikseller doldurulur, doldurulmuş pikseller ise zemin rengine çevrilir. Tüm kenarlar bitince poligon doldurulmuş olur. Aynı şekil için bu yöntemin adımları aşağıdaki gibi olacaktır.

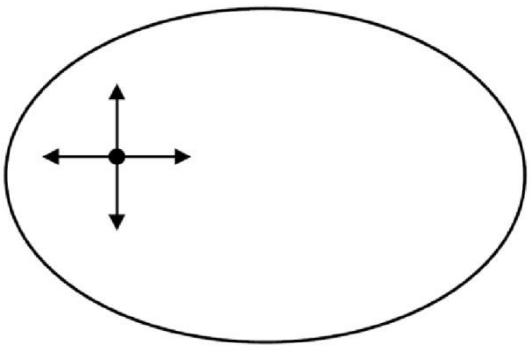


Şekil-4

4. Çekirdek Doldurma Yöntemi (Flood Fill / Seed Fill Algorithm)

Sınırları tanımlı bir yüzey için geliştirilebilecek yöntemlerden biri de çekirdek doldurma yöntemidir. Bu yöntemde yığın, kuyruk veri yapıları veya özyinelemeli (recursive) fonksiyonlar kullanılabilir.

Örneğin, özyinelemeli fonksiyonlar kullanılarak uygulama geliştirilmek istenirse, öncelikle doldurulacak alan sınırları içerisinde bir piksel (seed point / boyama işleminin başlanacağı piksel) seçilir. Daha sonra, aşağıda sözde kodu verilen yapıdaki bir boya(x,y,renk) fonksiyonu piksel koordinat bilgileri ve boyanacak renk bilgisi parametreleri ile çağrılır. Bu fonksiyon, öncelikle pikselin kenar pikseli olup olmadığını kontrol eder (boyanacak şeklin sınırlarının aşılıp aşılmadığının kontrolü) ve kenar pikseli değilse o piksel ilgili renge boyar. Daha sonra, boyanan piksele komşu pikseller özyinelemeli boya fonksiyonuna parametre olarak gönderilir. İşlem yüzey doldurulana kadar özyinelemeli olarak devam eder.



Şekil-5

```
Fonksiyon  boya(...)
      Eğer seçilen piksel sınır değeri değilse ve doldurulmamışsa
      Başla
            Piksel(x,y)=renk;
            boya(x+1,y,renk);
            boya(x,y+1,renk);
            boya(x-1,y,renk);
            boya(x,y-1,renk);
      Son
```

Özyinelemeli boya fonksiyonu sözde kodu

Çekirdek doldurma yöntemi yığın veri yapısı kullanılarak geliştirilmek istenirse, öncelikle başlangıç pikseli seçilir ve sonra aşağıdaki adımlarla yüzey doldurma işlemi gerçekleştirilir :

1. Seçilen piksel yığına itilir.
2. Yığından bir piksel çekilir. Piksel gereken renge boyanır.
3. Pikselin sağ, sol, üst ve alt komşularına bakılır. Herhangi bir komşu, sınır değeri değilse ve doldurulmuş değilse yığına itilir.
4. Yığındaki elemanlar bitinceye kadar 2. adıma gidilir.

Çekirdek doldurma yönteminin sakıncalarından biri, bir pikselin kendisini oluşturan pikseli de test etmesidir. Birkaç ek düzeltmeyle bu problem giderilebilse de, bu teknikte yığın boyutunun büyük olması kaçınılmazdır. Ayrıca bir piksele erişim sayısı 3-5 arasında olduğu için diğer yöntemlere göre daha yavaştır. Buna rağmen en büyük tercih sebebi, rasgele seçilen her yüzeyde iyi sonuç vermesidir.

5. Deney Hazırlığı

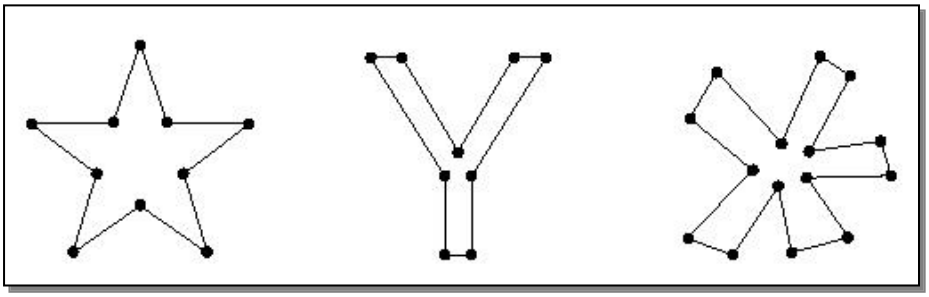
1. Deneyde DDA ve Breshenam doğru çizme yöntemlerinden de bahsedilecektir. Bu yöntemler hakkında bilgi sahibi olunuz.
2. Ekte verilen sıralı kenar liste yöntemini gerçekleyen uygulamayı (ScanLine klasörü içerisinde) çalıştırınız ve kaynak kodları inceleyiniz. (Uygulama OpenGL ile geliştirilmiştir, OpenGL'in yapılandırılması ile ilgili yönlendirmeler "IDE Kurulumu ve OpenGL Yapılandırması" adlı dökümanda verilmiştir.)
3. Ekte verilen ve çekirdek doldurma yöntemini kuyruk veri yapısı ile gerçekleyen FloodFill adlı uygulamayı çalıştırıp kodlarını inceleyiniz. Bu uygulamada, yukarıda belirtilen algoritmanın performansını yavaşlatan sakıncalarda kaçınma amacı ile farklı bir kuyruğa ekleme sıralaması kullanılmıştır. İnceleyiniz. (Uygulamayı çalıştırmak için gerekli bilgi BeniOku dosyası içerisinde verilmiştir.)
4. Çekirdek doldurma yönteminin etkinliğini arttırmak için neler yapılabilir, tartışınız.
5. Bildiğiniz farklı bir yöntem varsa o yöntemle, ya da buradaki yöntemlerin herhangi biriyle basit bir uygulama geliştiriniz.

6. Deney Tasarımı ve Uygulaması

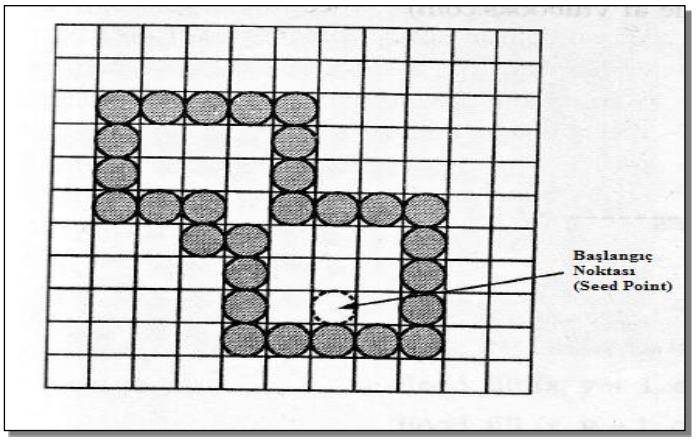
Deneyde, yüzey doldurma tekniklerinin işleyişleri tartışılacak, anlatılan yöntemlerle gerçekleştirilmiş uygulamalar incelenecektir. Bu yöntemlerde karşılaşılan ve karşılaşılabilecek problemler tartışılacaktır. Yöntemlerin hız ve bellek kullanımı yönünden üstünlükleri ve sakıncaları değerlendirilecektir.

7. Deney Soruları

1. Yüzey doldurma kavramını açıklayınız. Kullanım alanları nelerdir, araştırınız. Yukarıda bahsedilen 3 tekniğin işleyişini açıklayınız.
2. DDA ve Breshenam çizgi çizme yöntemlerini çizerek anlatınız ve bu iki algoritmayı performans ve doğruluk açısından karşılaştırınız.
3. Aşağıdaki konkav poligonlar sıralı kenar liste yöntemi ile doldurulmaya çalışılırsa nasıl sorunlarla karşılaşılır? Algoritma üzerinde nasıl düzeltmeler yaparak bu sorunların üzerinden gelirsiniz?



4. Aşağıdaki şekildeki kapalı alan 4 komşulu (four connected region) çekirdek doldurma yöntemi ile doldurulmaya çalışıldığında herhangi bir sorun ile karşılaşılır mı? Karşılaşılırsa, bu sorun çekirdek doldurma algoritması üzerinde nasıl bir iyileştirme yapılarak aşılabilir? Açıklayınız.



5. Yüzey doldurma tekniklerini hız, bellek gereksinimi gibi performansı etkileyen kriterlere göre karşılaştırınız. Bu algoritmaların, işleyişleri sırasında hangi durumlarda hata verebileceğini açıklayınız.

8. Deney Raporu

Deney rapor şablonu bir sonraki sayfadadır. Deney raporu el yazısı ile şablon kapak sayfası olacak şekilde hazırlanacaktır. Raporlar, bir sonraki hafta deneyine kadar teslim edilebilir.

KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

2015-2016 GÜZ DÖNEMİ
YÜZEY DOLDURMA TEKNİKLERİ DENEY RAPORU

GRUP		13:00-15:00		15:00-17:00	
NUMARA		AD SOYAD			

1. Deneyde Yapılanlar

Deneyde geliştirmeniz istenen kendi yüzey doldurma yönteminin sözde kodunu (pseudocode) yazınız, algoritmanın işleyişini bir poligon üzerinde kısaca açıklayınız.

2. Deney Soruları

Bu bölüme, ayrı ayrı Deney Sorularının cevapları verilecektir.

3. Kazanımlar

Deneyden kazanımlarınız ve varsa eksiklikler.

Not: Deney raporu el yazısı ile bu şablon kapak sayfası olacak şekilde hazırlanacaktır. Raporlar, bir sonraki hafta deneyine kadar teslim edilebilir.

MAYA ile 3D Modelleme

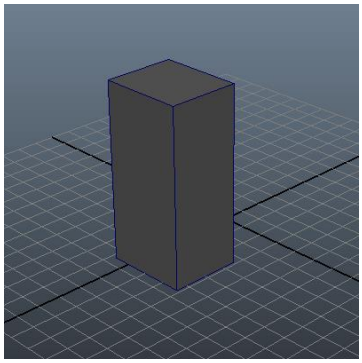
1. Giriş

3D oyunlar ve animasyonlar Bilgisayar Grafiklerinin günümüzde en yaygın uygulama alanları olarak göze çarpmaktadır. Her ikisinin temel yapıtaşı olan karakterlerin, gerçeğine yakın modellenmesi önemli bir aşamadır ve buna yönelik onlarca yazılım geliştirilmiştir. Bunlar içinde belki de en yaygın kullanılanı MAYA'dır.

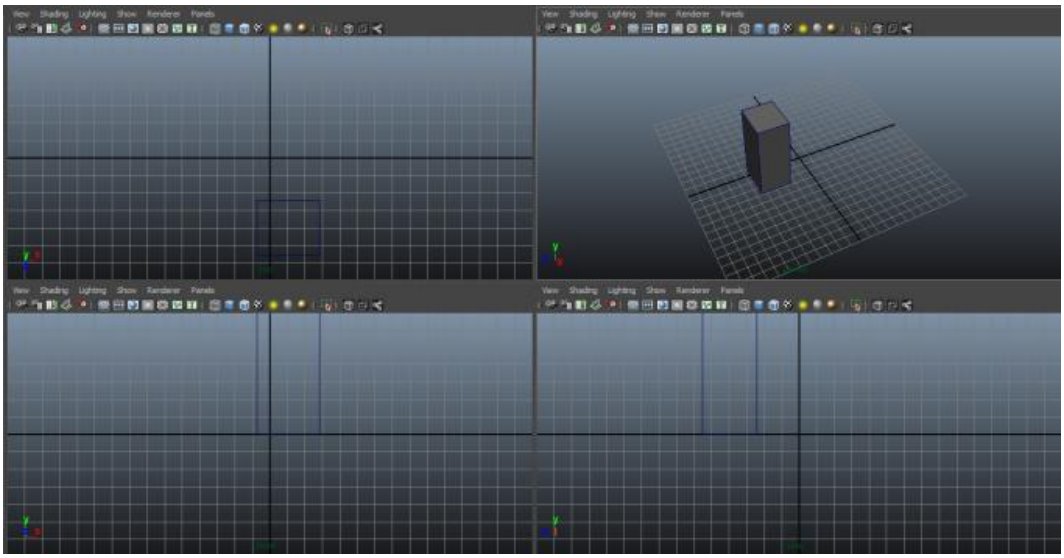
3D modelleme için Poligonal ve NURBS olmak üzere iki ana yöntem vardır. Bu deneyde MAYA ile poligonal modelleme anlatılacaktır.

2. MAYA Ortamının Tanıtımı

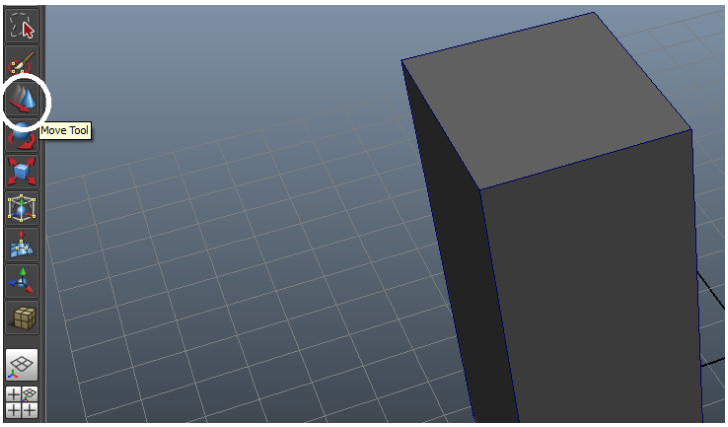
3D modellemeye başlamadan önce MAYA ortamının, sık kullanılan kısayol tuşları ve menüler ile tanıtımında fayda vardır. Şekildeki dikdörtgen prizma Polygons→PolygonCube (soldan 2.) tıklanıp mouse ile çizilmiş olsun.



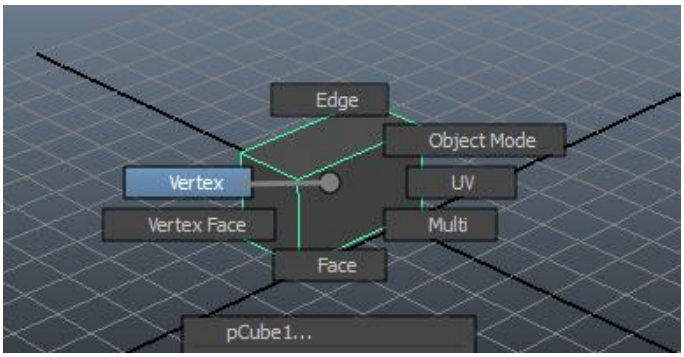
- Bu cisme değişik açılardan bakmak için klavyenin Alt tuşu ve farenin sol butonuna (Alt+Sol_Fare) basılı olarak hareket ettirilir.
- Cisme yaklaşıp cisimden uzaklaşmak için farenin ortasındaki tekerlek (scroll wheel) kullanılır. Sağa-sola, yukarı-aşağı hareket etmek için Alt+scroll tuşları basılı olarak fare hareket ettirilir.
- Cisme önden (front view), yandan (side view), üstten (top view) ve perspektif (persp view) olarak 4 farklı pencereden bakmak mümkündür. Pencereler arası geçiş için fare pencerenin üzerine getirilip space tuşuna basılır. Böylece örneğin yandan görünüş aktifken tekrar diğer görünüşlerden birine geçiş yapmak için tekrar space tuşuna basılır. 4 farklı pencere aşağıdaki şekilde gösterilmiştir:



- Cismi hareket ettirmek için cisim seçili iken move tooluna tıklanır ve fare ile oklardan tutup çekilerek istenilen eksende hareket ettirilir.



- Move toolunun altında rotate ve scale toolları vardır. Cisim seçili iken rotate tooluna tıklandığında beliren çemberlerin herbiri için fare ile tutulup çekildiğinde bir eksende dönme işlemi yapılır. Scale ile de ortaya çıkan küpler çekilerek ölçekleme (büyültme-küçültme) yapılır.
- Move, rotate ve scale toolları sırasıyla W, E ve R kısayol tuşlarına basılarak da kullanılabilir.
- Shift tuşu ile birden fazla cisim seçilebilir.
- Yapılan işlemler Z tuşu ile geri alınabilir.
- Cismin poligonal (wireframe) görünümü ile boyanmış (shaded) görünümü arasında geçişler yapmak için sırasıyla 4 ve 5 tuşlarına basılır.
- Cisim üzerinde farenin sağ butonuna tıklanırsa aşağıdaki gibi bir menü çıkar. Deney boyunca Edge, Vertex, Face ve Object Mode sıkça kullanılacaktır. Bunlardan Edge seçildikten sonra o cismin herhangi bir kenarına tıklandığında o kenar seçili olur; Vertex için köşesi; Face için (üçgen veya dikdörtgen) yüzeyi ve son olarak Object Mode için de cismin tamamı seçili olur.



3. 3D Modellemeye Giriş

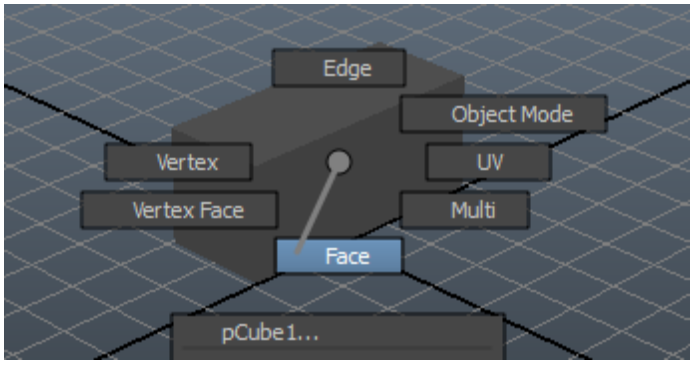
Herhangi bir cismi poligonal modellerken genellikle küp gibi basit bir şekil çizip bu şekil üzerinde değişiklikler yapılarak model oluşturulur. Değişiklik daha çok modelin karmaşıklığına bağlı olarak yeni poligonlar üretmek şeklinde gerçekleşir. Bunun için şu toollar kullanılacaktır:

- Extrude Tool
- Split Polygon Tool
- Insert Edge Loop Tool

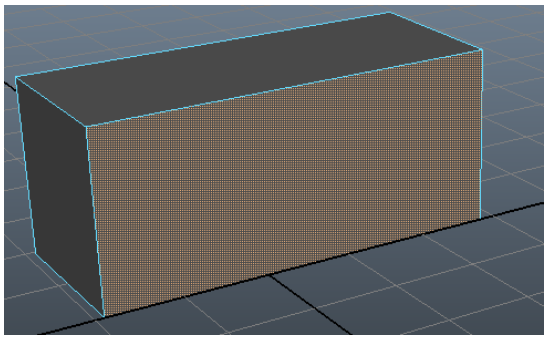
3.1. Extrude Tool

Bu bölümde Extrude tool ile çok basit bir insan modelinin çizimi anlatılacaktır. Split Polygon ve Insert Edge Loop toolları insan yüzünün modelleneceği 4. Bölümde anlatılacaktır.

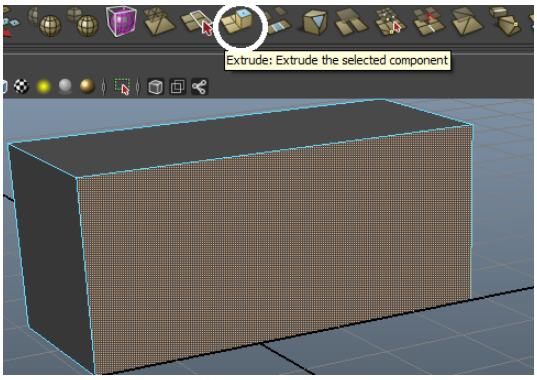
Extrude (çekme), adından da anlaşılacağı gibi işaretlenen face, edge veya vertexin yenisini oluşturup istenilen doğrultuda çekme (uzatma) işlemidir. Bunun için öncelikle cismin, farenin sağ butonuna tıklanıp belirlenen Edge, Vertex, Face ve Object Mode'larından birine göre ilgili bölgesinin farenin sol butonu ile seçilmiş olması gerekir. Dikdörtgen prizmaya ait bir yüzeyin (face) extrude edilmesi aşama aşama aşağıda gösterilmiştir:



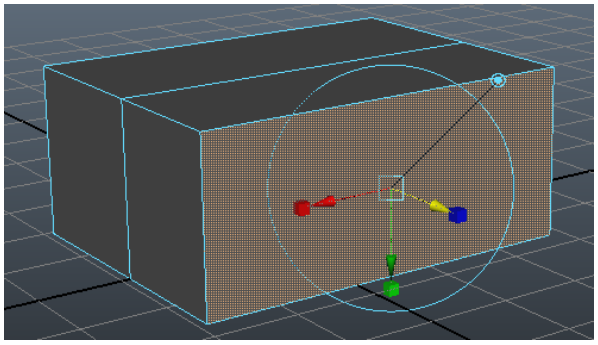
(Farenin sağ butonuna tıklanıp face seçilir)



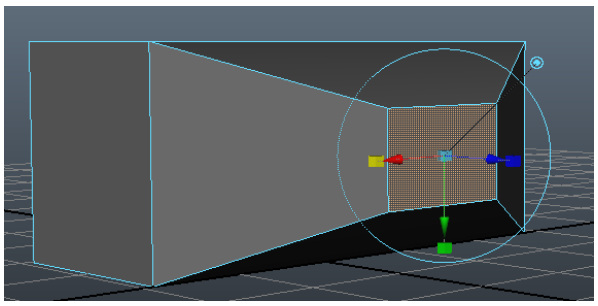
(Sol buton ile extrude edilecek face seçilir)



(Polygon→Extrude'a tıklanır)

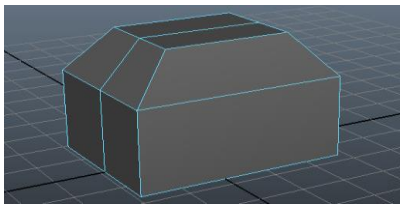


(Oklardan biri çekilerek ilgili yönde yeni face üretilir)

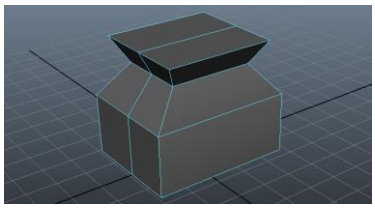


Extrude işleminde oklardan biri ile istenilen yönde yeni face üretilirken aynı zamanda yukarıdaki gibi küplerle ölçekleme de yapılabilir.

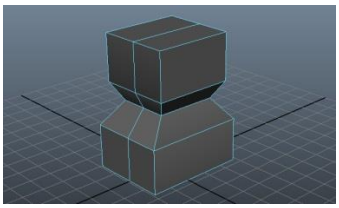
Sadece extrude işlemleri ile basit bir insan modeli çizimi aşağıda aşama aşama gösterilmiştir:



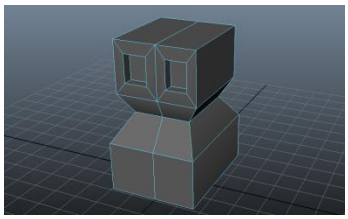
(a)



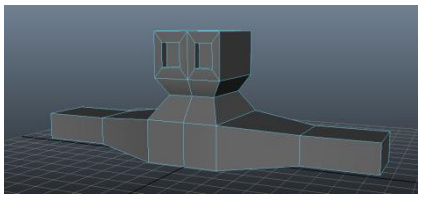
(b)



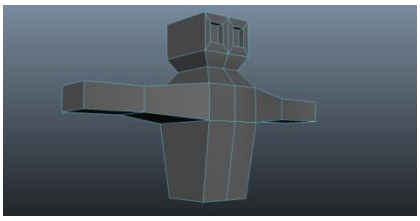
(c)



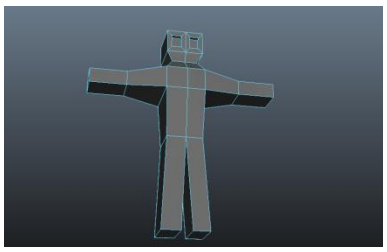
(d)



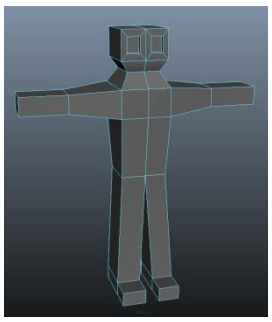
(e)



(f)



(g)



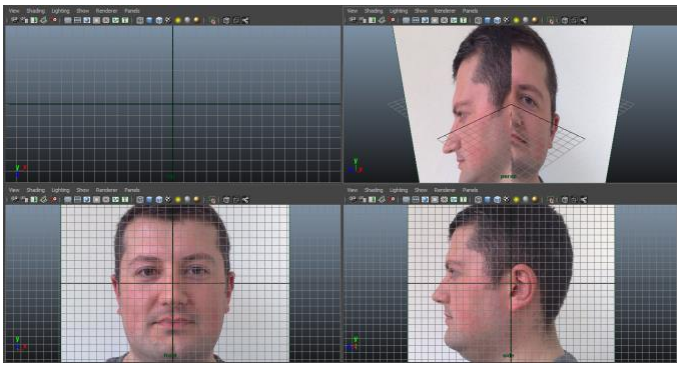
(h)

Gözler çizilirken komşu iki face seçildikten sonra içe doğru extrude yapılırsa gözlerin şaşı gibi biribirine yakın çizildiği görülür. Herbir face içinde ayrı ayrı extrude yapabilmek için Edit Mesh → Keep Faces Together seçimi kaldırılır.

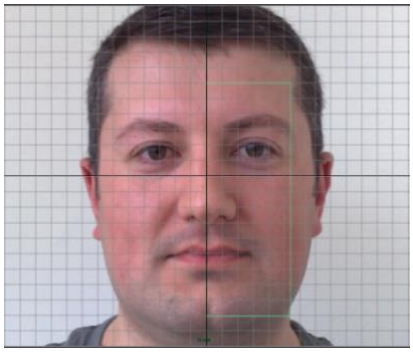
4. Split Polygon ve Insert Edge Loop Tool

Bu bölümde, insan yüzü modelleme örneği üzerinde Split Polygon Tool ve Insert Edge Loop Tool kullanımı anlatılacaktır.

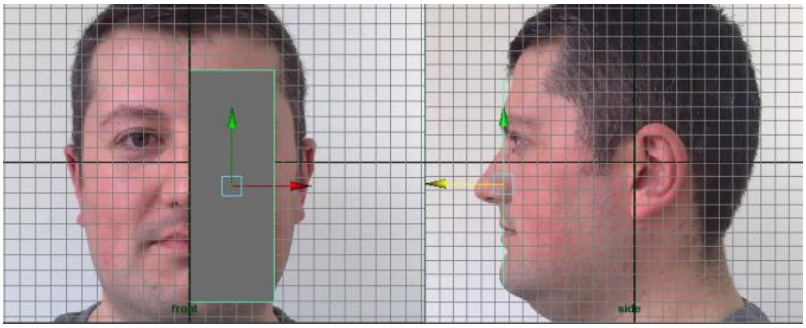
Öncelikle View→Image Plane→Import Image ile önden (front.jpg) ve yandan (side.jpg) çekilmiş resimler sırasıyla front view (sol alt köşede) ve side view (sağ alt) pencerelerine şekildeki gibi yüklenir:



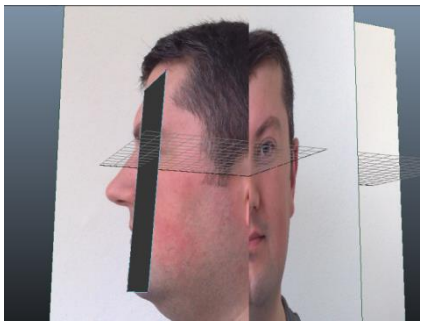
Eğer yüklenen resim şekilde koyu çizgilerle gösterilen merkezden sağa/sola veya yukarı/aşağı kaymışsa merkeze getirmek için Öncelikle View→Image Plane Attributes→imagePlane1 (veya 2) ‘e tıklayınca sağda açılan panelde Placement Extras→Center ‘da front view için X; side view için de Z değeri değiştirilir. Front view penceresi aktif iken Polygons→Plane (soldan 5.) seçilir ve ilk yüzey aşağıdaki gibi çizilir:

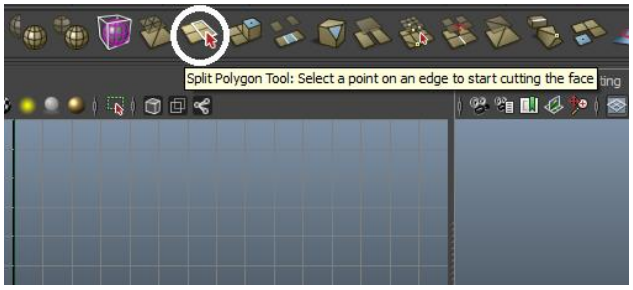


Plane tam olarak yüklenen resmin üstünde olduğundan sadece kenarları görünüyor. Side view aktif yapılır, Move tooluna tıklanırsa z ekseni boyunca move yapıldığında tamamı görünür:

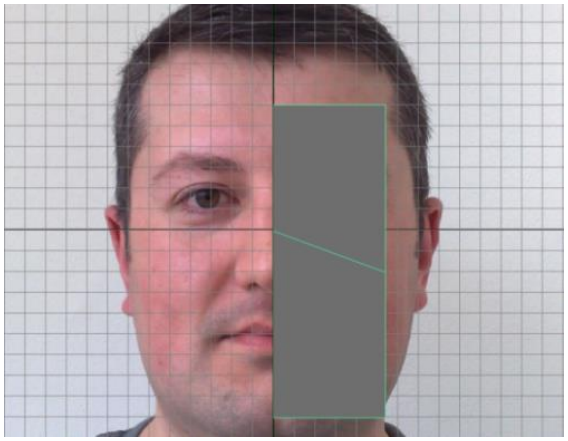


Farenin sağ butonu ile Plane vertex modunda seçilip side viewde plane’e ait yukarıdaki iki köşe noktası resimde alın bölgesine; aşağıdaki iki köşe noktası da çene bölgesine gelecek şekilde move yapılır:

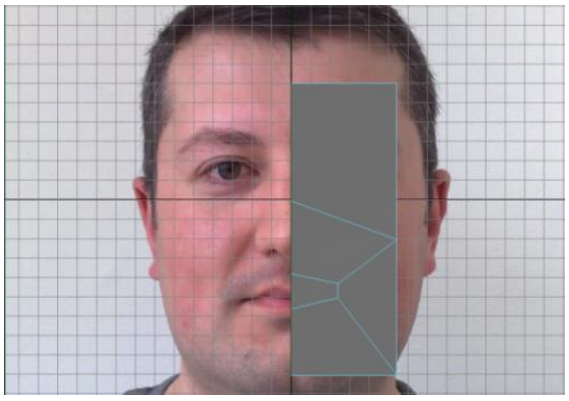




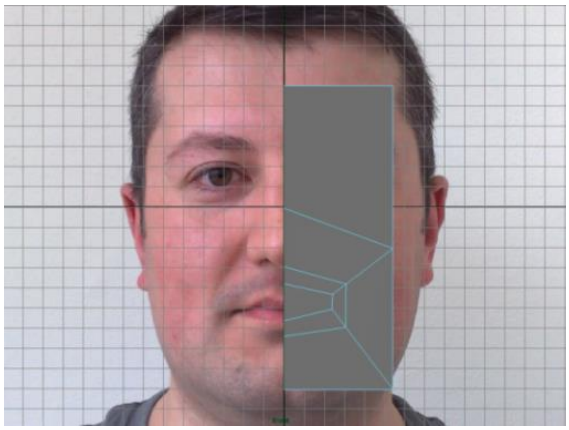
Şimdi şekilde gösterilen Split Polygon tool kullanılarak plane iki parçaya bölünecektir. Bunun için front view penceresinde önce plane seçili iken Split Polygon toola tıklanır. Sonra burnun ve yanağın üstünden geçen kenarlarda istenilen iki noktaya (burnun ortası ve elmacık kemiği civarı) tıklanıp en son Enter tuşuna basılarak plane iki parçaya ayrılmış olur:



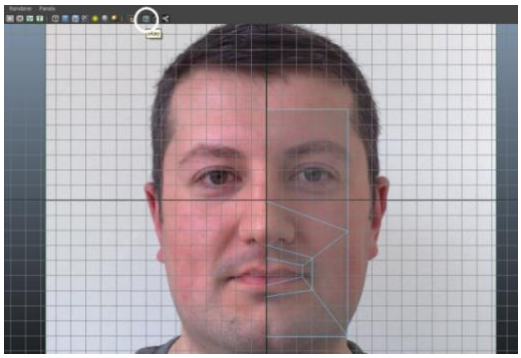
Basit bir ağız yapmak üzere alt plane, extrude tool ile çizilen modelin gözleri gibi içe ve sola doğru extrude yapılır:



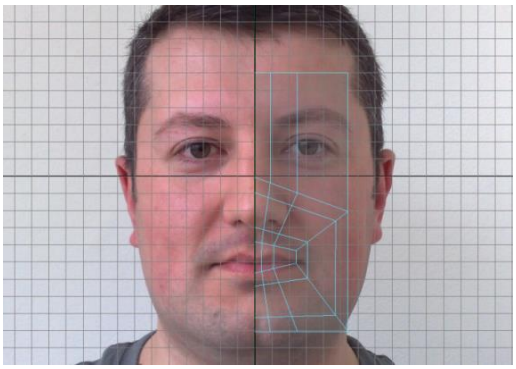
Burnu yapmadan önce modeldeki poligon sayısını bir miktar arttırmakta fayda var. Öncelikle Edit Mesh→Insert Edge Loop seçilip burnun üzerinden geçen kenara tıklanarak yeni kenarlar elde edilir:



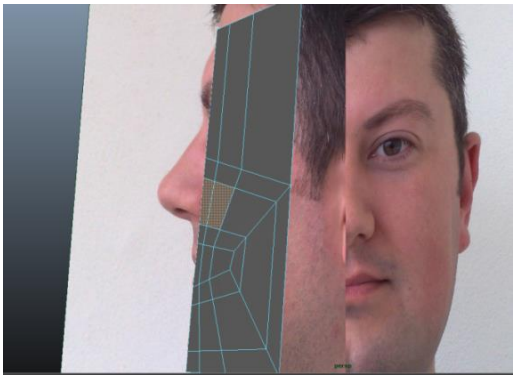
Burun deliğinin solundan, sağından ve üstünden olmak üzere üç tane daha Edge Loop eklemek istiyoruz. Bunun için burun deliklerini görmek gerekiyor. Bunu Xray modu ile yapabiliriz:



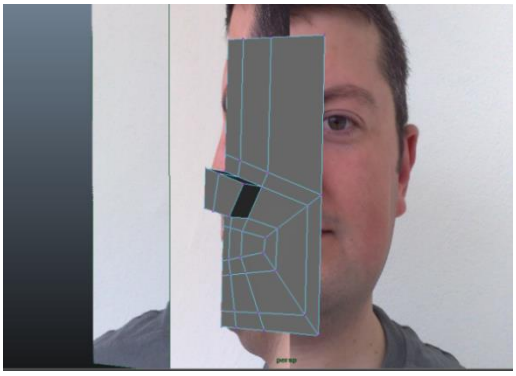
Bahsedilen Edge Looplar eklendiğinde:



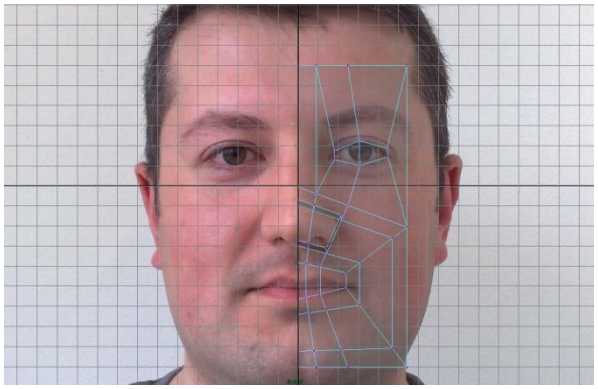
Perspektif pencere aktif iken aşağıdaki iki face seçilir:



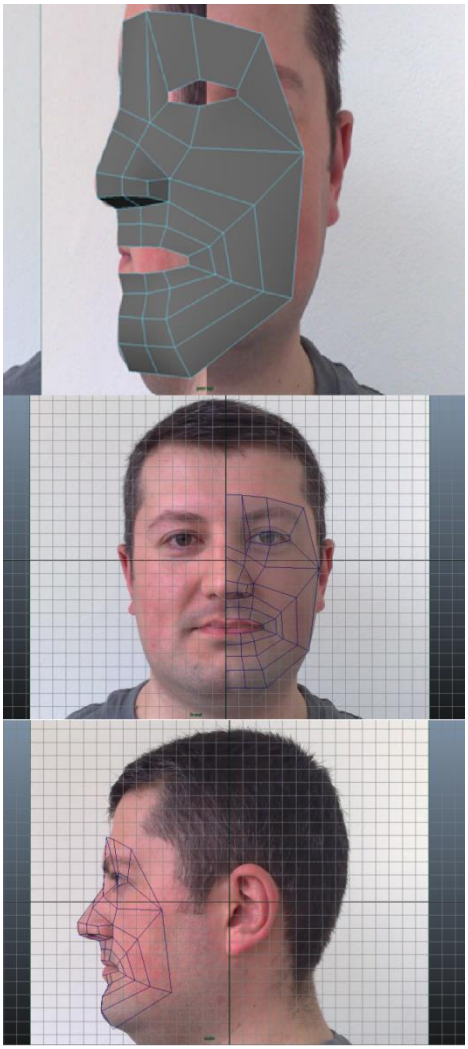
Sonra extrude yapılır:



Gözleri çizmek için 3. bölümdekinin benzeri biçimde extrude yapılacaktır. Yalnız bu sefer Keep Faces Together seçili olmalıdır. Front ve side viewlerdan bakarak gerekli move işlemleri yapılarak göz tam yerine kaydırılır:



Gözün doğru konuma getirilmesi için yapılan işlemler ağız, burun, çene ve yanaklar için de yapılmalıdır. Gerekli düzenlemeler yapıldıktan sonra modelin son hali aşağıdaki gibi olmalıdır:



5. Deney Hazırlığı

- http://www.autodesk.com/education adresinden üyelik yaptırarak MAYA 2016 öğrenci versiyonunu "Free Software" linki ile indirip lisanslı kullanabilirsiniz.
- Deneyin sorumlusundan deneyde anlatılan konularla ilgili video tutorialları temin ediniz.
- Plane, Torus, Sphere gibi değişik şekillller çizip MAYA Ortamının Tanıtımı bölümünde anlatılan işlemleri bu şekillere uygulayınız.
- Herbir deney grubu kendisi ile ilgili aşağıda ismi verilen şekli çizip **.mb** dosyası olarak deneye getirsin:

A1 ve B1	Masa ve sandalye çiziniz
A2 ve B2	Bilgisayar Kasası, Monitor ve Klavye çiziniz
A3 ve B3	Araba çiziniz.
A4 ve B4	Gemi çiziniz.
A5 ve B5	Uçak çiziniz.
A6 ve B6	Ağaç çiziniz.
A7 ve B7	Gözlük çiziniz.
A8 ve B8	Kol saati çiziniz.
A9 ve B9	Ev çiziniz.

- Ömer Hoca'nın Bilgisayar Grafikleri Laboratuarı sayfasında yer alan önden ve yandan resimlerini, kişisel bilgisayarlarınızda front ve side viewlardaki imagePlane'lere yükleyiniz ve yüz modelini çizmeye hazır bir şekilde kişisel bilgisayarlarınızla birlikte deneye geliniz.

6. Deney Tasarımı ve Uygulaması

- Extrude tool kullanarak deneyde anlatılan basit insan şeklini çiziniz.
- Extrude tool ile Move tool arasında ne farklar vardır?

7. Deney Raporu

Extrude tool ile birlikte Split Polygon Tool ve Insert Edge Loop Tool kullanarak grubunuzdan bir öğrencinin yüzünü modelleyiniz.

MAYA ile Animasyon

1.Giriş

Bilgisayar Grafiklerinin yaygın uygulama alanlarından biri de 3D animasyonlardır. Patlama gibi özel efektler, yüklü miktarda paralar harcanmaksızın animasyon yöntemleri ile gerçeklenebilmektedir.

Bu deneyde MAYA'da 3D animasyon geliştirme yöntemlerinden bahsedilecektir. MAYA ortamı 3D Modelleme deney föyünde tanıtılmıştır. Bu deneyde doğrudan animasyon konusu anlatılacaktır.

2. Keyframe ve Graph Editor

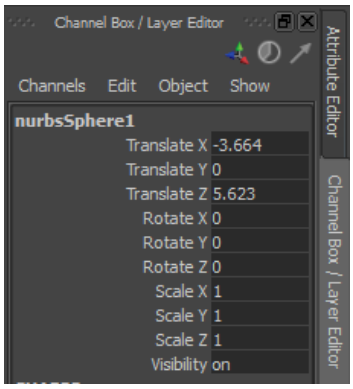
Geliştireceğimiz animasyonu bir video olarak düşünelim. Bilindiği gibi video resimlerden oluşmaktadır. Bu resimlere frame denilir. Animasyon geliştirirken ara ara bazı frameler setlenip o framelerde cisimler üzerinde değişiklikler yapılır. İşte bu framelere **keyframe** denilir.

Bir keyframeden diğerine geçerken cismin bir konumdan diğerine gittiği varsayılsın. Bu hareketin aradaki frameleri nasıl etkilediğini belirlemek için **Graph Editor** kullanılır. Örneğin havaya atılan bir topun animasyonu yapılsın. Topun ilk konumu ve havada en yüksek noktaya ulaştığı noktalar keyframe olarak setlensin. Bu iki nokta arasında topun hızının nasıl değişeceği Grap Editorde kullanılan eğri (Curve) ile belirlenir.

2.1. Topun Zıplaması Animasyonu

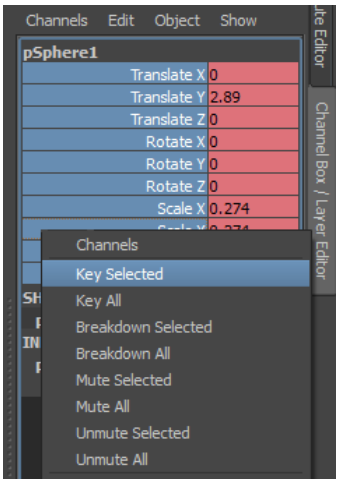
Keyframe setlemesi ve hareketin Graph Editor ile belirlenmesi zıplayan bir topun animasyonu ile anlatılacaktır.

Öncelikle bir küre (sphere) çizelim. Bu küreyi top olarak düşünürsek topun zıplaması animasyonu yapılırken onun üzerinde öteleme (translation), dönme (rotation) ve ölçekleme (scaling) gibi işlemleri gerçekleştirme için **Channel Box** menüsü kullanılacaktır. Channel Box'ı görüntülemek için Display→UI Elements→Channel Box 'a tıklanır.

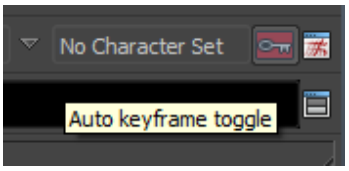


MAYA penceresinin aşağısına yakın **Time Slider** kısmında animasyon yaparken setleyeceğimiz frameler görülmektedir. Time sliderda default olarak 24 frame olduğu görülmektedir. Time Sliderın hemen altında 1...24 şeklinde **Range Slider** barı görülüyor. Onun sağında da 24.00 ve 48.00 yazıyor. 24 sayısı yukarıda da söylendiği gibi time sliderda gösterilecek frame sayısını temsil ediyor. Ama üretilecek animasyon belki de yüzlerde frameden oluşacak. İşte 48 de toplam frame sayısını temsil ediyor. Aynı anda time sliderda gösterilecek frame sayısı 24'e setlenmiş durumda. Bu sayıyı değiştirmek için 48.00'ın solunda yazan 24.00 değiştirilebileceği gibi range slider mouse ile tutulup çekilebilir.

Yukarıda da bahsedildiği gibi küre üzerindeki öteleme, dönme ve ölçekleme işlemleri için Channel Box kullanılacaktır. Channel Box'taki değişimlerin key frame olarak setlenebilmesi için sağ mouse butonuna tıklanır ve seçilen Translate, Rotate ve Scale özellikleri sol butonla **Key Selected** yapılır.

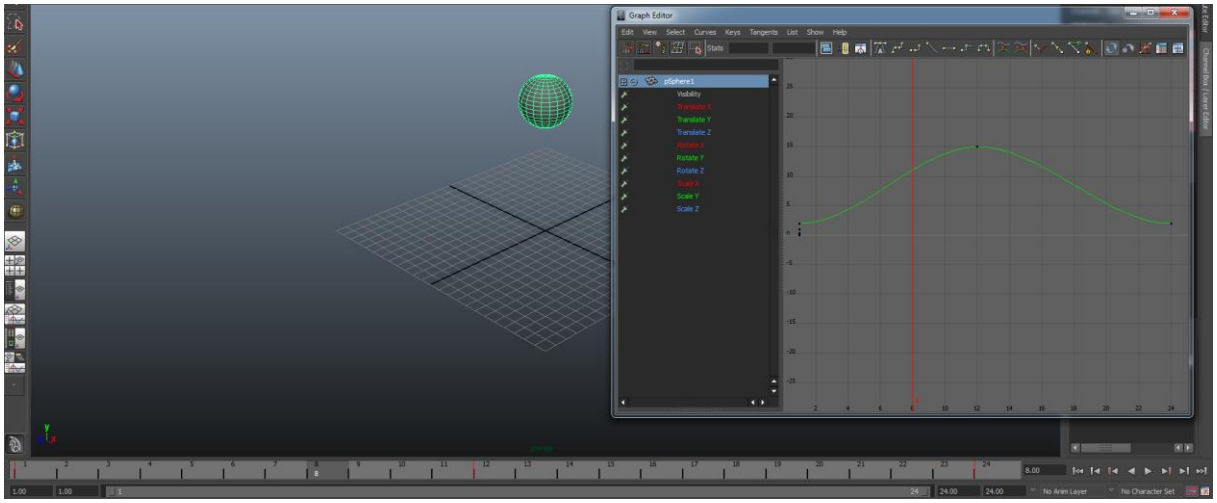


Ayrıca cisim üzerinde yapılacak herbir koordinat değişiminin otomatik olarak key frame olarak setlenmesi için sağ alt köşedeki anahtar sembolüne tıklanır.



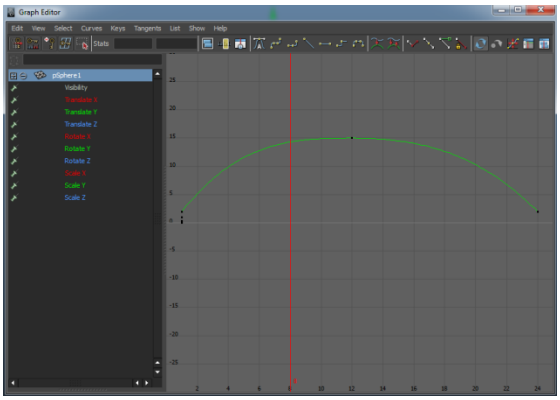
Böylece animasyon için gerekli hazırlıklar tamamlanmış olur. Geliştirilecek animasyonda hem range slider hem de toplam frame sayısı 24 olsun (24.00 24.00). Time sliderda herhangi bir frame mesela 12. frame'e tıklanıp Channel Box'tan translation değeri mesela 15 olarak değiştirildikten sonra son frame olan 24. frame'e tıklanıp translation tekrar 0 yapılarak zıplama animasyonu gerçekleştirilmiş olur. Animasyonu izleyebilmek için Play butonuna tıklanır.

Bilindiği gibi animayon için setlenen key frameler arasında cismin nasıl hareket edeceğine Graph Editor'deki eğrilerle karar verilir. Window→Animation Editors→Graph Editor ile Graph Editor açılabilir. Time slider mouse ile ilerletilirken hem top hareket eder hem de Graph Editor'deki eğri üzerinde düşey bir çizgi ile o anda eğrinin neresinde olunduğu görülür.

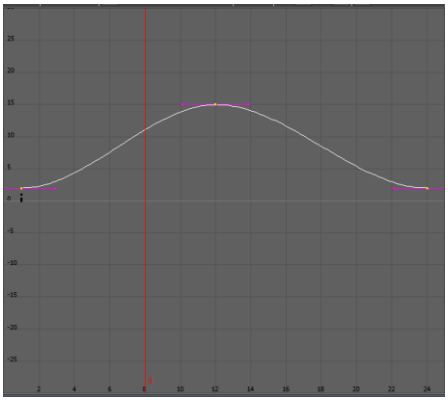


Yukarıdaki ekran görüntüsü Time Slider'da 8. framede iken alınmıştır. Dikkat edilirse Graph Editor'deki düşey çizgi 8. Frame üzerindedir. Dolayısıyla yatay eksen frameleri temsil etmektedir. Ayrıca eğrinin tepe noktası da 15'i göstermektedir. Çünkü cismin Y ekseni boyunca yükseldiği tepe noktası değeri 15'tir. Bu noktaya çıkarken hızının nasıl değiştiğine eğrinin karakteristiğine görer karar verilmektedir. Burada hızın değişimini **eğrinin eğimi** temsil etmektedir. Dolayısıyla 0..4 arası framelerde eğim düşük olduğundan hız düşük, 4..8 arası hızda doğrusal yakın bir artış var ve 8..12 arası yine eğim giderek azalmakta ve tepe noktasında eğim sıfır olduğundan top durmaktadır.

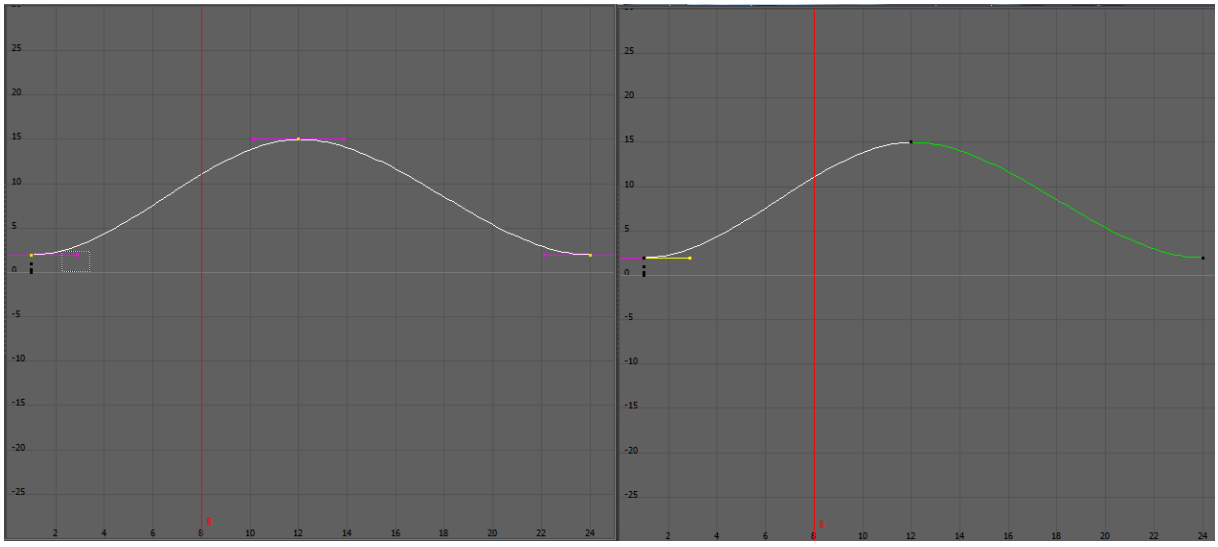
Aslında yukarıda anlatılan hız değişimi gerçek bir zıplama hareketini temsil etmemektedir. Mesela 0..4 arası eğim yüksek olmalı çünkü top yerden zıplarken yüksek hızla zıplamalıdır. Sonra da eğim sürekli azalmalıdır. Yani eğri aşağıdaki gibi olmalıdır:



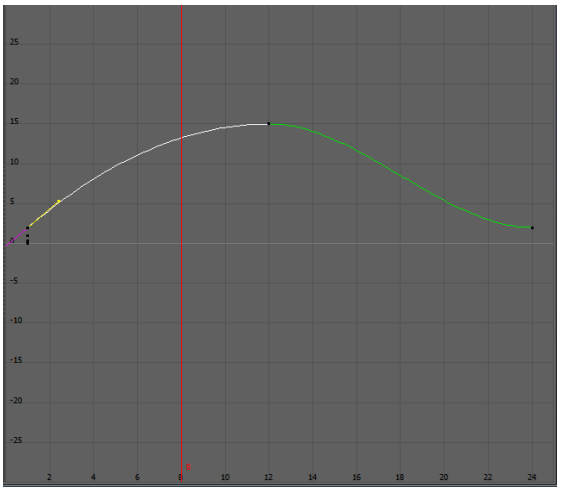
Eğriyi yukarıdaki gibi değiştirmek için öncelikle mouse ile eğriye tıklanır:



Eğri üzerinde 3 tane yatay çizgi ortaya çıktığı görülür. Bu çizgilere **tangent** denilir ve eğri üzerindeki değişiklikler de bu çizgiler yardımıyla yapılır. Bunun için ilgili tangent yine mouse ile seçilir:



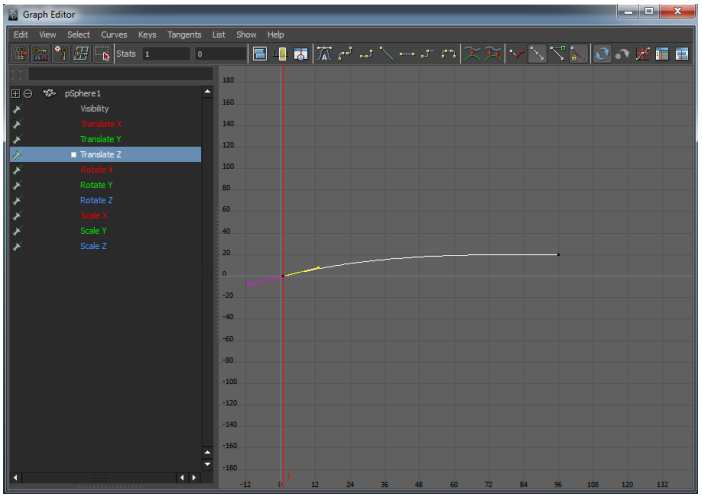
Sonra klavyeden W tuşuna bir kez basılır ve mouseun orta tuşuna (scroll wheel, tekerlek) **basılı** tutularak eğri üzerinde istenilen değişiklik yaplır:



Animasyonda top sadece birkez zıplamaktadır. Birkaç kez mesela 4 kez zıplayıp durması için frame sayısını 24x4=96 yapalım:

Yeni keyframeler setleyerek topun yükseksiklerini 36. framede 12, 48. framede 2, 60. framede 8 72. Framede 2, 84. framede 3 ve 96. Framede yine 2 yapalım. 0 yerine 2 yapılmasının nedeni kürenin yarıçapının 2 olmasıdır. Yani top zıplarken grid'in üzerinde olsun, içine girmesin diye böyle setlenmiştir.

Dikkat edilirse topun hareketi hala tek ersen (Y) boyuncadır. Hem zıplayıp hem de ileri hareket etmesi için time sliderda 96. frame gidip Channel Box'ta bu sefer Translate Z key selected yapılır ve değer olarak mesela 20'ye setlenir. Grap Editor açılıp orda da Translate Z seçilir ve eğri üzerinde gerekli değişiklikler yaplırsa Z ekseni boyunca hareket de sağlanmış olur.



Animasyonun daha da gerçekçi olması için top yere çarptığında deforme edilebilir. Bunun için Scale değerleri Y için 0.7, X ve Z için de 1.4 yapılabilir. Yalnız bu işlemler için çarpma anlarına yakın (2 frame kadar) yeni keyframeler setlenmelidir.

Son olarak Graph Editorle ilgili birkaç kısa yol tuşu hakkında bilgi verelim :

- **Scroll Wheel** : Scroll wheel tekerleği döndürülerek Graph Editor üzerinde zoom in/out yaptırır.
- **Alt+Scroll Wheel** : Graph editordeki eğrileri sağa sola yukarı aşağıya kaydırır. Burada scroll wheel tuşu **basılı** olmalıdır.
- **Alt+Shift+Scroll Wheel** : Aynı anda tek eksende (X veya Y) kaydırma yaptırır.
- **Alt+Shift+Sağ Buton** : Aynı anda tek eksende grid aralıklarını açar/daraltır.

3. Dynamic Tool

Genel anlamda dinamik fizik biliminin nesne hareketlerinde tanımlanan bir parçasıdır. Maya'da Dynamics animasyonları fiziksel güçleri simule etmek için çeşitli fizik yasalarını dikkate alır. Kullanıcı nesnenin hangi fiziksel kuvvetler karşısından etki görmesini istiyorsa bunları ayarlar geriye kalan kısımları her bir frame'de setlemek yerine yazılımın yapmasını bekler.

Dynamics animasyon bileşeni kullanıcının geleneksel keyframe animasyonlarda yapılması çok zor ve çok uğraştırıcı olan gerçekçi sahneleri gerçekleştirmesine imkân tanır. Örneğin; okyanus dalgalanması, dalgalanan bayrak ya da patlama efektleri bunlardan bazılarıdır. Dynamics bileşeni kendi içinde çeşitli araçlara sahiptir.

Fields: Field'ları kullanarak simülasyonlarda gerçekliğe ve doğallığa yaklaşılır. Örneğin; particle hareketleri çeşitli field'lar ile birlikte kullanılarak sağlanabilir. Field'lar particle, nParticle, nCloth, soft body, rigid body, akışkanlar, bulutlar ya da saçlar gibi nesnelere etki eder ve bunların hareketini sağlarlar. Poligonal ve nurbs yüzeyler ya da cisimler ise rigid body içinde ele alınır. Maya'da kullanıma sunulan field çeşitleri aşağıdaki gibidir:

1) Air Field : Hava kuvveti sağlar.
2) Drag Field : Sürtünme ya da baskı kuvveti sağlar.
3) Gravity Field : Yerçekimi kuvveti sağlar.
4) Newton Field : Newton kuvveti etkisi yapar.
5) Radial Field : Cisimlere karşı itme ya da çekme kuvveti uygular.
6) Turbulence Field : Cisimlere ya da yüzeylere türbülans etkisi yapar.
7) Uniform Field : Cisimlere belirtilen yönde kuvvet etki eder.
8) Vortex Field : Bu kuvvet cisimleri dairesel ya da spiral olarak kendine çeker.
9) Volume Axis Field : Bir eksen boyunca cisimlere kuvvet uygular.
10) Volume Axis Curves : Bir eksen boyunca curve'ler yardımı ile kuvvet uygular.

Rigid Body: Rigid body polygonal ya da nurbs yüzeyleri sert bir yüzeye çevirmek için uygulanır. Geleneksel surface'lerden farklı olarak rigid body'ler animasyon süresince birbirleri ile hareket eder ve gerekli ayarlar yapılırsa sahnedeki kuvvetlerin etkisi altında kalabilir, particle'lar ile çarpışabilir ve üzerlerine constraint eklenebilirler.

Maya aktif ve pasif olmak üzere iki tip rigid body'e sahiptir. Aktif body olarak tanımlanan cisimler ortamın yerçekiminden, rüzgarından ya da çakışma gibi dinamikliğinden etkilenir. Key setlemeleri yapılmasına izin vermezler. Pasif body olarak tanımlanan cisimler ise active olarak setlenen cisimlere etki etmek için oluşturulur. Key setlemeleri yapılabilir. Rotate, translation ve move işlemlerine izin verirler. Ancak dinamik etkilere karşı

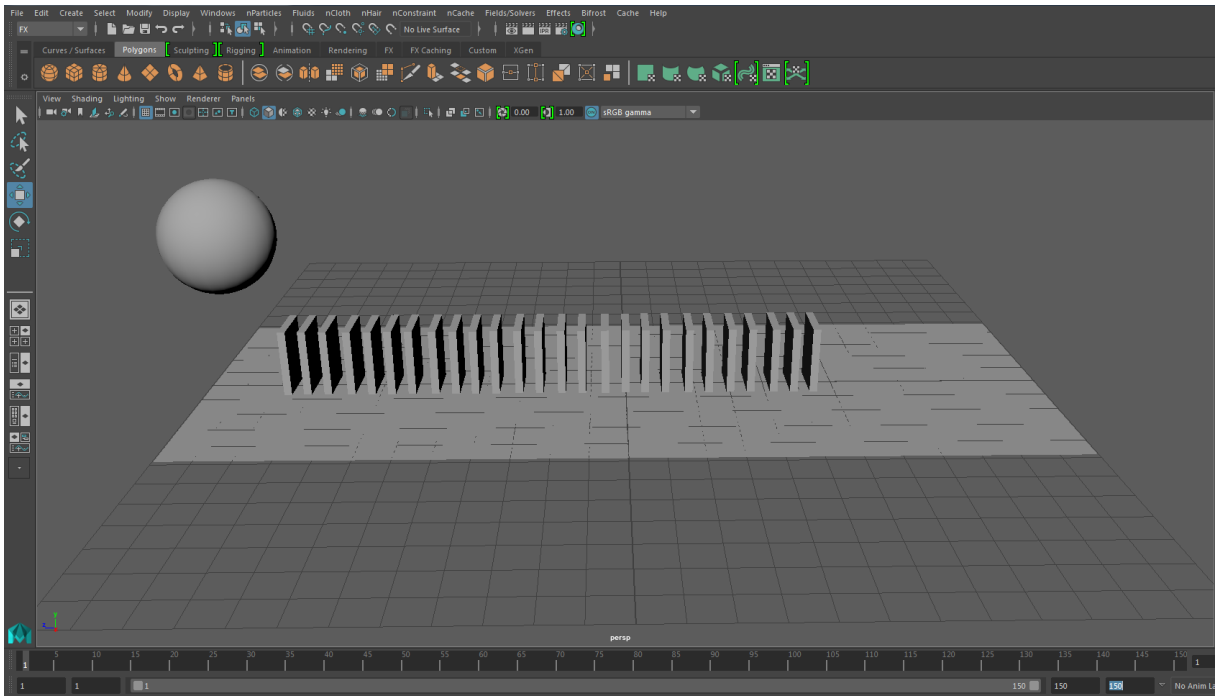
etkilenmezler. Bu iki body türüne örnek vermek gerekirse topun bir zeminden sekmesi örneği verilebilir. Top aktif body olarak tanımlanmalıdır. Çünkü yer çekiminden etkilenerek yere düşmeli ve daha sonra yere çarpmanın etkisi ile yerden yükselmelidir. Ancak zemin pasif olarak setlenmelidir. Çünkü top sektiğinde zemini yerinden oynatmamaktadır. Ve zeminde herhangi bir bozulma olmamaktadır. Rigid body'nin dinamik animasyonunun kontrolü rigid body solver denilen maya componenti ile yapılır. Ve ortamdaki kuvvetler ve çakışmalar tarafından da hareketler oluşturulur.

3.1. Domino Taşlarının Devrilmesi

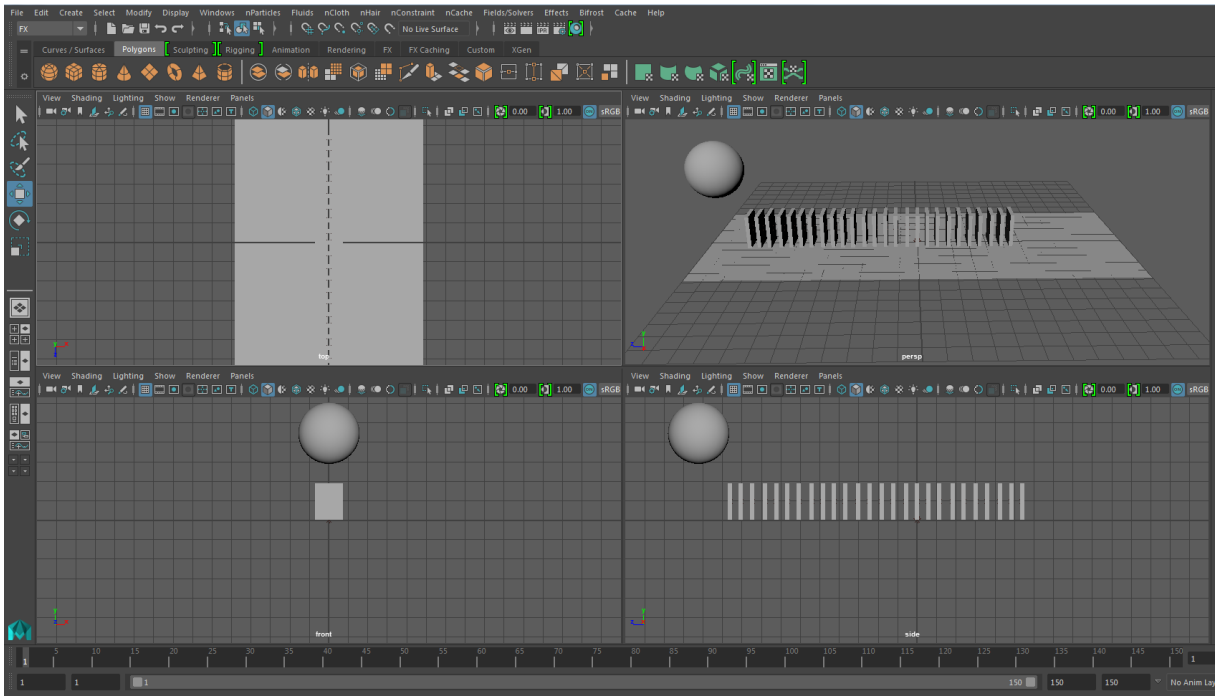
Domino taşlarının Dynamic Tool ile devrilmesi animasyonunda öncelikli olarak poligonlardan yararlanarak bir zemin, devrilecek olan taşlar ve bu taşları devirecek bir top oluşturulur. Bunun için;

- Zemin oluşturmada; Polygons→Polygon Plane,
- Domino taşlarını oluşturmada; Polygons→Polygon Cube,
- Topu oluşturmada; Polygons→Sphere seçilerek çizilir.

Maya ortamında bu poligonların oluşturulmuş hali aşağıda gösterilmektedir.



Domino taşlarının oluşturulmasında; bir Polygon Cube oluşturulduktan sonra diğer taşlar, oluşturulmuş Polygon Cube'den Copy-Paste yapılarak üretilir. Dikkat edilmesi gereken bir diğer önemli nokta ise oluşturulan topun yukarıdan bırakıldığında ilk domino taşına çarpacak biçimde yerleştirilmiş olmasıdır.



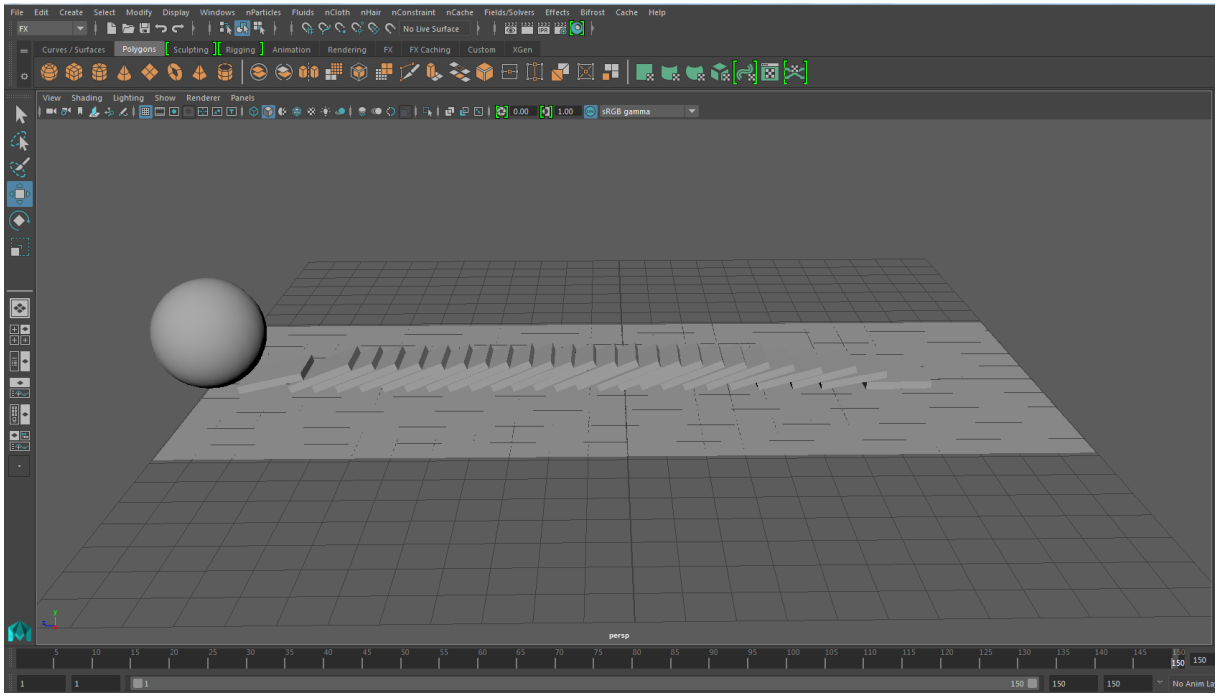
Poligonlar oluşturulduktan sonra bu poligonlara sırasıyla Dynamic özellikler kazandırılır. Bunun için;

•Zemin seçili iken; FX→Field/Solvers→Create Passive Rigid Body özelliği,

•Domino taşlarının hepsi seçili iken (Shift tuşuna basılı tutularak Mouse ile tıklanıp); FX→Field/Solvers→Create Active Rigid Body özelliği,

•Top seçili iken (Shift tuşuna basılı tutularak Mouse ile tıklanıp); FX→Field/Solvers→Create Active Rigid Body özelliği eklenir.

Topun yukardan aşağı düşmesi, domino taşlarının ise birbirine çarpıp düşebilmesi için top ve taşlar seçili iken hepsine FX→Field/Solvers→Gravity özelliği kazandırılır.



4. Deney Hazırlığı

- http://www.autodesk.com/education adresinden üyelik yaptırarak MAYA 2016 öğrenci versiyonunu "Free Software" linki ile indirip lisanslı kullanabilirsiniz.
- Deneyin sorumlusundan deneyde anlatılan konularla ilgili video tutorialları temin ediniz.
- "Maya ile 3D Modelleme" deney föyünden MAYA kullanımı hakkında bilgi edininiz.
- Topun zıplaması animasyonunu deformasyon ve dönme efektlerini de katarak yapınız.
- Graph Editordeki eğrilerde değişiklikler yaparak etkilerini gözlemleyiniz.
- Laboratuarın web sayfasına eğimli bir zeminde küpün yuvarlanması animasyonunun videosu konmuştur. Siz de bir benzerini yapıp **.mb** (Maya Binary) olarak getiriniz.
- Domino taşlarının devrilmesi simülasyonunu topa gravity özelliği verme yerine topa zıplama animasyonu vererek gerçekleyiniz.

5. Deney Tasarımı ve Uygulaması

- Laboratuarın web sayfasına basit bir bilardo animasyonunun videosu konmuştur. Siz de bunu animasyon ve simülasyon özelliklerini birlikte kullanarak yapınız.
  Not → Bilardo tahtası plane üzerinde, Split Polygon ve Extrude toolları ile değişiklikler yapılarak çizilmiştir.

Ters Perspektif Dönüşüm ile Doku Kaplama

1. Giriş

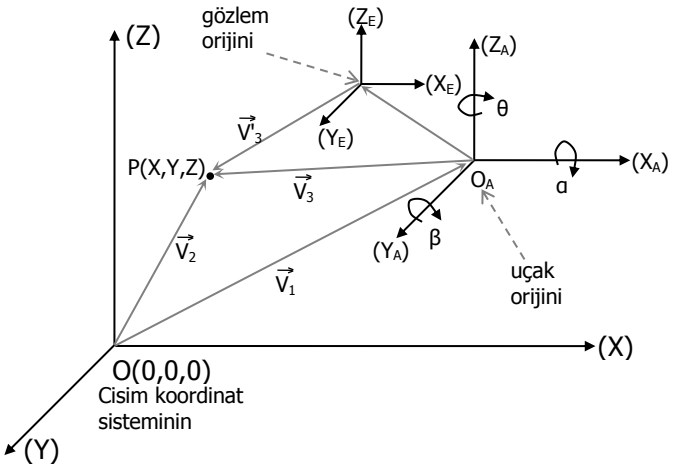
Bu deneyde, genel hatları ile paralel ve perspektif izdüşüm teknikleri, ters perspektif dönüşüm ile doku kaplama ve örtüşme (aliasing) problemi üzerinde durulacaktır. Deney sonunda öğrencilerin, gerekli matematiksel ve geometriksel bilgileri edinmeleri ve bu tür programlar geliştirebilmeleri amaçlanmaktadır.

2. Temel Grafiksel İşlemler

1.1. Öteleme (Translation)

Bilgisayar grafiklerinde nesneler farklı koordinat sistemlerine sahip olabilirler. Hesaplamaların doğru yapılabilmesi için bu koordinat sistemleri arasındaki dönüşümün yapılması gerekmektedir.

Bir noktanın cisim koordinat sisteminden gözlemci koordinat sistemine dönüşümü Şekil-1 yardımı ile açıklanabilir. (X,Y,Z) cisim koordinat sisteminde bir P(X,Y,Z) noktası tanımlansın. Bu nokta daha sonra, uçakla hareket eden ve uçağın önüne doğru pozitif Y_A, sağ kanadı boyunca pozitif X_A ve uçağın üstünden aşağıya doğru pozitif Z_A ile tanımlanmış eksenlere sahip (X_A, Y_A, Z_A) uçak sistemine çevrilir. Basitlik için pilotun hareket etmediği ve gözünün pozisyonunun uçak sistemi ile aynı olduğu varsayılabilir.



Şekil-1 Cisim ve gözlemci koordinat sistemleri.

Önce, manzaradaki her bir nokta O_A merkezli gözlemci koordinat sisteminde tanımlanmalıdır. Gözlemci ve cismin koordinat sistemlerinin eksenlerinin paralel ve aynı doğrultuda oldukları ve bir P(X,Y,Z) noktasının O_A (X_A, Y_A, Z_A) merkezinden gözlendiği varsayılsın. P noktasının O_A merkezine göre yeri $\vec{V}_3$ vektörü ile ifade edilirse:

$$\vec{V}_3 = \vec{V}_2 - \vec{V}_1 \qquad (1)$$

yazılabilir. $\vec{V}_1$ ve $\vec{V}_2$ vektörleri sırasıyla O_A ve P noktalarını cisim koordinat sisteminde tanımlayan vektörlerdir. Eğer $\vec{V}_1$ ve $\vec{V}_2$ vektörleri cisim sistemindeki koordinatlar cinsinden ifade edilirse, $\vec{V}_3$ vektörü için aşağıdaki bağıntı yazılabilir:

$$\vec{V}_1 = \begin{bmatrix} X_A \\ Y_A \\ Z_A \end{bmatrix} \qquad \vec{V}_2 = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad \vec{V}_3 = \begin{bmatrix} X_t \\ Y_t \\ Z_t \end{bmatrix} = \begin{bmatrix} X - X_A \\ Y - Y_A \\ Z - Z_A \end{bmatrix} \qquad (2)$$

Burada kullanılan fark alma işlemi öteleme olarak adlandırılmaktadır.

Öteleme koordinat sistemlerini çakıştırmanın haricinde cismi/gözlemciyi yer değiştirmede de kullanılmaktadır.

1.2. Dönme (Rotation)

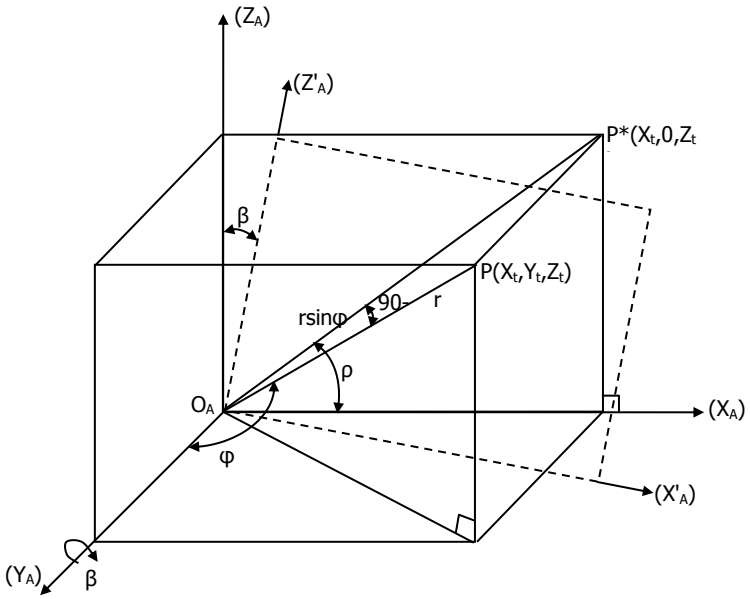
Denklem (2) ile verilen öteleme dönüşümü yalnızca her iki koordinat sistemindeki eksenlerin paralel ve aynı doğrultuda olması durumunda geçerlidir. Fakat gözlemci koordinat sisteminin 6-serbestlik derecesine sahip olması bir noktanın dönüşümünde ötelemenin yanı sıra dönme dönüşümünü de gerekli kılmaktadır. Gözlemci koordinat sisteminin eksenleri etrafındaki dönmeler α, β ve θ ile verilirse, o zaman Y_A ekseni etrafında saat ibreleri yönünde β açısı kadar dönme için aşağıdaki bağıntılar yazılabilir (Şekil-2).

P noktası polar koordinat sisteminde ifade edilirse aşağıdaki eşitlikler yazılabilir:

$$X_t = r\sin(\varphi)\cos(\rho)$$

$$Y_t = r\cos(\varphi)$$

$$Z_t = r\sin(\varphi)\sin(\rho)$$



Şekil-2 Y_A ekseni etrafında β kadarlık dönme.

$$
\begin{aligned}
X_t &= r\sin(\varphi)\cos(\rho+\beta) = r\sin(\varphi)\big(\cos(\rho)\cos(\beta)-\sin(\rho)\sin(\beta)\big) = r\sin(\varphi)\cos(\rho)\cos(\beta) - r\sin(\varphi)\sin(\rho)\sin(\beta) \\
&= x_t\cos(\beta) - Z_t\sin(\beta) \\
Y_t &= Y_t \\
Z_t &= r\sin(\varphi)\sin(\rho+\beta) = r\sin(\varphi)\big(\sin(\rho)\cos(\beta)+\cos(\rho)\sin(\beta)\big) = r\sin(\varphi)\sin(\rho)\cos(\beta) + r\sin(\varphi)\cos(\rho)\sin(\beta) \\
&= Z_t\cos(\beta) + X_t\sin(\beta)
\end{aligned}
$$

Buradan P noktası matrissel olarak aşağıdaki gibi ifade edilebilir :

$$
\begin{bmatrix} X_{t\beta} \\ Y_{t\beta} \\ Z_{t\beta} \end{bmatrix} = \begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \begin{bmatrix} X_t \\ Y_t \\ Z_t \end{bmatrix} \qquad (3)
$$

Gözlemci koordinat sisteminin X_A ve Z_A eksenleri etrafındaki rotasyonları için de denklem (3)'e benzer bağıntılar yazılabilir. Böylece cisim uzayındaki P noktası dönüşümden sonra P_T (X_T, Y_T, Z_T) olarak denklem (4) ile verilebilir.

$$
\begin{bmatrix} X_T \\ Y_T \\ Z_T \end{bmatrix} = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_t \\ Y_t \\ Z_t \end{bmatrix} \qquad (4)
$$

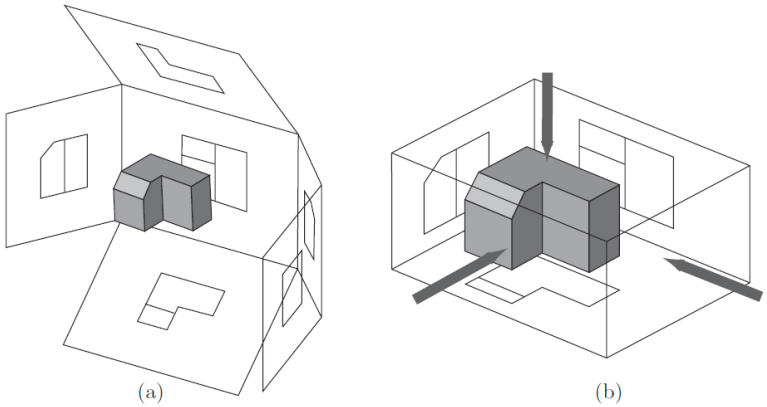
3. Paralel İzdüşüm Teknikleri

Paralel izdüşüm yöntemleri nesneleri 3B uzaydan 2B görüntü düzlemine paralel ışınlar boyunca izdüşürür. Diğer bir deyişle, ℓ doğrusundan ℓ' doğrusuna paralel izdüşüm, ℓ deki her bir P noktasının ℓ' de Φ(P) noktasına atandığı öyle bir Φ eşlemesidir ki her bir nokta ve görüntüsünü birleştiren doğrular birbirine paralel durmaktadır.

Paralel izdüşümler dik (orthographic), aksonometrik (axonometric) ve eğik (oblique) olmak üzere 3 ana sınıfa ayrılırlar.

2.1. Dik İzdüşüm

Paralel izdüşüm yöntemlerinden en basitidir. Ana prensibi, nesneyi çevreleyen bir kutu varsayımına dayanır. Bu kutunun 6 kenarına nesnenin dik izdüşümleri alınarak işlem gerçekleştirilir. Eğer nesne basit bir şekle sahipse 3 tane birbirine dik kenar kullanmak da yeterli olabilir. Eğer nesne karmaşık ve alışılmamış bir şekle sahipse bölgesel bakışlar kullanılabilir.



Şekil-3 Dik izdüşüm örnekleri (a) 6 kenara (b) 3 kenara.

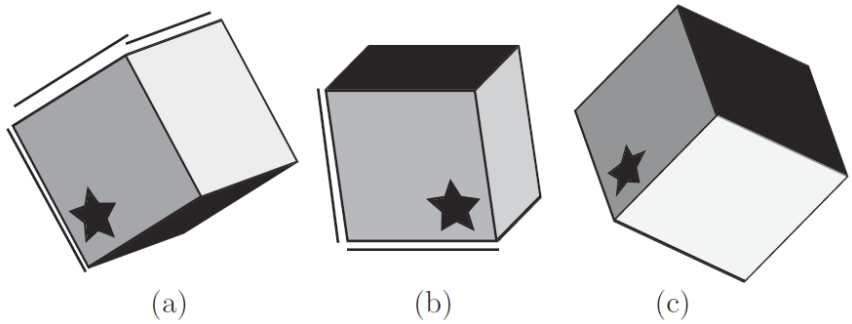
Eğer çevreleyen kutunun bir kenarı xy düzlemi ise izdüşüm işlemi sadece z bileşeninin silinmesiyle gerçekleştirilebilir Benzer şekilde, diğer eksenler silinerek farklı izdüşümler de elde edilebilir.

Dik izdüşümlerin en büyük avantajı açı ve uzunluk bilgisinin korunmasıdır. Bu yüzden teknik çizim yapanlar tarafından yaygın olarak kullanılmaktadır.

2.2. Aksonometrik İzdüşüm

Dik izdüşüm nesnenin sadece tek bir yüzünü göstermektedir. Bu yüzden 3 veya 6 izdüşüm gerçekleştirilir. Her bir izdüşüm detaylandırılabilir ve o yüz için şekli iyi temsil edebilir, fakat nesnenin geri kalanı hakkında bilgi vermez. Bundan dolayı dik izdüşümleri yorumlamak deneyim gerektirir. Bu sorunu çözmek için daha kolay anlaşılabilir ve şekli tek bir görüntü ile daha fazla yüzünü göstererek iyi temsil edebilecek izdüşümler düşünülmüştür. Bunun için, perpektif dönüşümden daha basit, izdüşüm ile gerçek dünya koordinatları arasında uyumu olan ve uzaktaki nesneleri küçük göstermeyen bir izdüşüm yöntemi geliştirilmiştir.

Çin izdüşümü olarak da adlandırılan bu yöntem ufuk noktasının sonsuzda olduğu bir perspektif izdüşüm yöntemi olarak düşünülebilir. Paralel doğrular perspektif izdüşümdeki gibi ufuk noktasında birleşmez, paralel devam eder.



Şekil-4 Aksonometrik izdüşüm örnekleri (a) trimetric (b) dimetric (c) isometric.

Bu izdüşüm yöntemi isometric, dimetric ve trimetric olmak üzere 3 sınıfa ayrılır. Isometric izdüşüm, en yaygın kullanılan aksonometrik izdüşüm olup temel kenarlar veya eksenlerin izdüşüm düzleminin normali ile eşit açı yaptığı biçimidir. Dimetric izdüşümde 3 temel nesne ekseninden 2'si izdüşüm düzleminin normali ile eşit açı yapmaktadır. Trimetric'te ise bütün açılar birbirinden farklı olmaktadır.

2.3. Eğik İzdüşüm

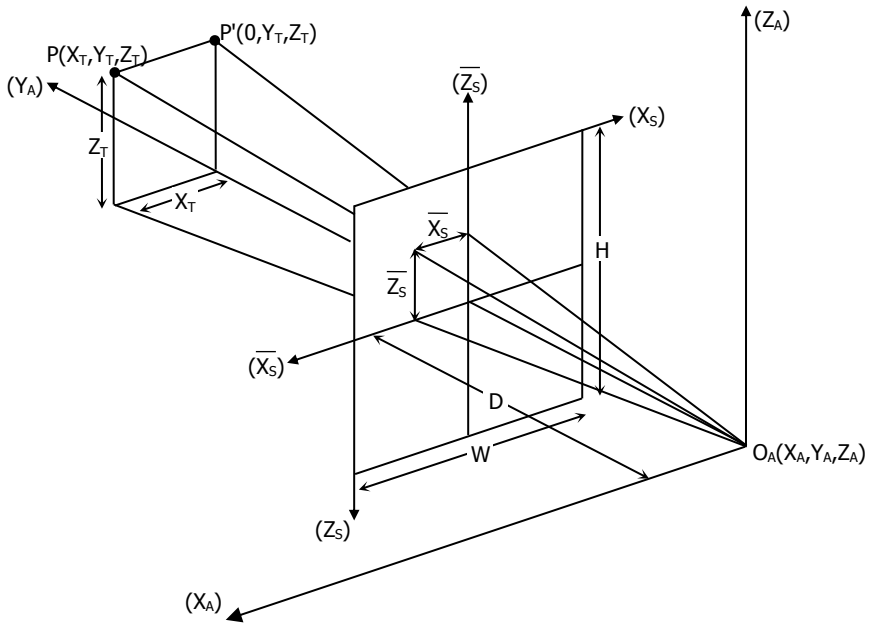
Eğik izdüşüm, izdüşüm ışınlarının görüntü düzlemine dik gelmediği paralel izdüşümün özel bir durumudur. Aksonometrik izdüşümde karşıdan bakıldığında derinlik bilgisi gözükmemekteydi. Eğik izdüşümde ışınlar eğik yollandığından görüntü düzlemine paralel duran bir cisim 3 boyut bilgisi ile (genişlik, yükseklik ve derinlik) görülmektedir. Eğer cisim görüntü düzlemine paralel durmazsa gerçek boyutları/ölçüleri hesaplamak için ek işlem yapılması gerekmektedir.

Şekil-5 Eğik izdüşüme bir örnek.

4. Perspektif İzdüşüm

İnsan gözü bir manzaraya baktığı zaman uzaktaki nesneleri yakındaki nesnelere göre daha küçük görür ve paralel doğruları ufukta birleşiyormuş gibi görür. Bu, perspektif olarak adlandırılır ve daha gerçekçi görüntülerin üretilmesini sağlar. Paralel izdüşüm yöntemlerinde ölçümlerin doğruluğunun korunmasına karşın bu özellikler bulunmamaktadır.

Perspektif izdüşüm şu şekilde gerçekleştirilir: Bir cismin tüm noktaları görüntü düzlemine izdüşürülür. İzdüşüm hatları görüntü düzlemini keserek gözlem noktasına ulaşır. Görüntü koordinat sisteminin merkezi genellikle görüntü düzleminin merkezi ile uyuşacak şekilde ve bakış noktasından bu merkeze gelen hat görüntü düzlemine dik olacak şekilde seçilir.



Şekil-6 Perspektif Projeksiyon

Gözlemci, gözlemci koordinat sisteminin merkezinde oturmakta ve görüntü düzleminden D kadar uzaklıkta bulunmaktadır (Şekil-6). Nesne noktalarına karşı düşen görüntü noktaları benzer üçgenler vasıtasıyla kolaylıkla belirlenebilir.

$$\vec{X}_S = D * \frac{X_T}{Y_T}$$
$$\vec{Z}_S = D * \frac{Z_T}{Y_T} \qquad (5)$$

Görüntü düzlemi olarak ekran kullanılırsa görüntü noktalarının kolayca hesaplanabilmesi için koordinatların pozitif olması gerekmektedir. Ayrıca $\vec{X}_S$ ekseninin sağa doğru $\vec{Z}_S$ ekseninin de ekranın altına doğru olması görüntü noktalarının koordinat değerlerinin raster tarama kuralına uygun olması için yararlıdır. Yeni düzenleme ile (5) denklemi aşağıdaki gibi yazılabilir.

$$\vec{X}_S = \left(-D * \frac{X_T}{Y_T} \right) + C_X$$
$$\vec{Z}_S = \left(-D * \frac{Z_T}{Y_T} \right) + C_Z \qquad (6)$$

5. Ters Perspektif Dönüşüm

Ters perspektif dönüşüm, normal perspektif dönüşümün ters yönde uygulanarak, ekran üzerindeki bir noktanın cisim koordinat sistemine dönüşümünü gerektirir. Şekil-1'deki P(X, Y, Z) noktası şu şekilde ifade edilebilir:

$$\vec{V}_2 = \vec{V}_1 + \vec{V}_3$$

$\vec{V}_1$ vektörü gözlemci koordinat sistem merkezinin cisim koordinat sistemine göre konumunu belirler. Bu nedenle herhangi bir rotasyona uğramaz. $\vec{V}_3$ vektörü ise P noktasının gözlemci koordinat sisteminin merkezine göre yeri olduğu için üç ayrı rotasyona tabidir.

$$\vec{V}_3^{\,r} = [\beta][\alpha][\theta]\vec{V}_3$$

Ters dönüşüm uygulanarak şu sonuç elde edilebilir.

$$\vec{V}_3 = [\beta]^{-1}[\alpha]^{-1}[\theta]^{-1}\vec{V}_3^{\,r}$$

O zaman $\vec{V}_2$ vektörü aşağıdaki şekilde yeniden yazılabilir:

$$\vec{V}_2 = \vec{V}_1 + [\beta]^{-1}[\alpha]^{-1}[\theta]^{-1}\vec{V}_3^{\,r}$$

$\vec{V}_2$ vektörü, $\vec{V}_3^{\,r}$ ve gözlemci koordinat merkezine göre görüntü pikselinin yeri P arasında aşağıdaki ilişki oluşturularak ekran koordinatları olarak ifade edilebilir.

$$\vec{V}_3^{\,r} = K * P^*, \quad P^* = \begin{bmatrix} \vec{X}_S \\ D \\ \vec{Z}_S \end{bmatrix}$$
$$\vec{V}_2 = \vec{V}_1 + [\beta]^{-1}[\alpha]^{-1}[\theta]^{-1} K * P^*$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_A \\ Y_A \\ Z_A \end{bmatrix} + K * [\beta]^{-1} [\alpha]^{-1} [\theta]^{-1} \begin{bmatrix} X_S \\ D \\ Z_S \end{bmatrix} \qquad (7)$$

Bu formülle 3 boyutlu dünya koordinatlarını 2 boyutlu ekran koordinatlarından elde edebilmek için bir bilinmeyenin lineer bağımlı yapılması veya doğrudan verilmesi gerekir. Diğer bir deyişle bu yöntem işlemsel olarak basit olmasına karşın x=5, $y^2+z^2=r^2$ gibi formülü bilinen yüzeyleri kaplayabilmektedir. Örnek olarak Z=0 yüzeyini (X-Y düzlemi) ele alalım. Her bir ekran koordinatı için (7) nolu denklemde Z=0 yazılarak K sabiti bulunur. Bu, ekran üzerindeki her bir noktanın cisim koordinat sistemindeki büyüklüğü arasındaki ilişkiyi ifade eder. Daha sonra K sabiti yerine yazılarak X ve Y değerleri hesaplanır. Böylece ekrandaki her bir pikselin dünyadaki konumu hesaplanır. Burada bulunan (X, Y, Z) değerleri bize ters perspektif dönüşümün sonucunu verir.

Ekran koordinatlarının Xs ve Zs düzleminde doğrusal olarak taranması yerine açısal tarama da gerçekleştirilebilir. Bu durumda (7) denklemi şu şekilde değişikliğe uğrar:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_A \\ Y_A \\ Z_A \end{bmatrix} + K * [\beta]^{-1} [\alpha]^{-1} [\theta]^{-1} \begin{bmatrix} \tan\lambda \\ 1 \\ \tan\gamma \end{bmatrix} \qquad (8)$$

Burada λ yatay eksendeki tarama açısını, γ da dikey eksendeki tarama açısını ifade etmektedir.

6. Doku Kaplama (Texture Mapping)

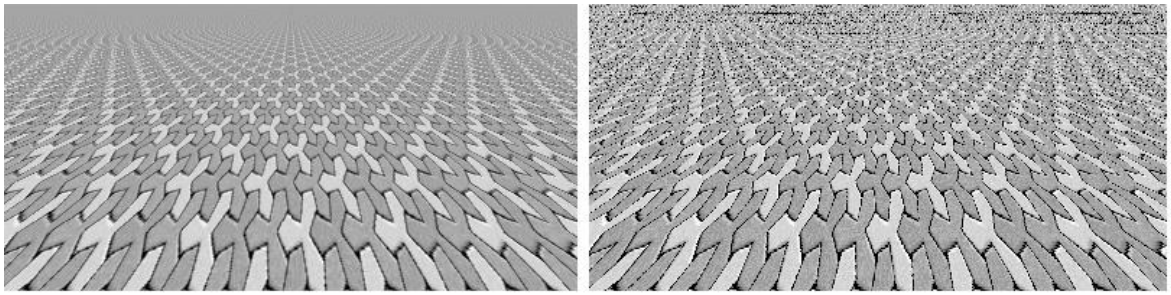
Bilgisayar grafiklerinde bir cismin yüzey ayrıntıları doku olarak adlandırılır. Tuğlalardan örülmüş büyük bir duvara yeterli uzaklıktan baktığımız zaman duvardaki her bir tuğlayı bir doku elemanı olarak düşünebiliriz. Ama bu duvara yakından bakıldığında tuğlalar artık ayrı cisimlerdir ve cisim üretme teknikleri ile üretilmelidir. Böyle yüzey ayrıntılarının cisim üretme teknikleriyle üretilmesi gerçek-zamanlı sistemler için fazla hesaplama gerektirdiğinden uygun değildir.

Doku, cisimlere doğal görünüm kazandırır. Doku, cisme yapıştırılmalı ve cisimle aynı dönüşümleri geçirmelidir. Dokunun dönüşümü, doku elemanlarının (çimenli ortamda otların veya ağaçta yaprakların) dönüşümünü gerektirir. Doku 2-boyutlu veri dizileridir. Bu veriler renk veya parlaklık bilgisi olabilir.

Doku kaplamayı maharetli bir iş yapan asıl konu, dokuların dörtgensel olmayan bölgelere de uygulanabilmesi olmuştur. Doku, farklı dönüşümlerin çokgenin görünüşü üzerindeki etkilerini karşılayacak şekilde bozulmaya uğrar. Boyu bir doğrultuda uzarken, diğer doğrultuda kısalabilir. Döndürüldüğü için orjinalinden farklı görünebilir. Dokunun büyüklüğüne, dörtgenin bozulmasına ve ekrandaki görüntüye bağlı olarak, piksellerin bazıları bir fragmandan fazlasına eşlenebilir, bazı fragmanlar da birden çok piksel tarafından örtülebilir.

Doku kaplama bir doku görüntüsünün tekrarlanması ile yapılabileceği gibi tek bir doku görüntüsünün bir yüzeye lineer bağıntılarla gerilmesi ile de yapılabilir.

Ters perspektif dönüşüm yöntemiyle doku üretiminde doku örüntüsünün karmaşıklığı hesaplama miktarını değiştirmez. Öncelikle ekrandaki her bir pikselin gerçek dünyadaki karşılığı bulunur. Sonra istenilen bir yaklaşımla doku bu yüzey üzerine kaplanır. Örneğin z=a yüzeyi kaplanacak ise x ve y değerlerinin doğrudan kullanılması mantıklı olur. Bu değerler (x, y), dokunun en ve boyuna göre mod işlemine tabi tutulur. Elde edilen doku konumlarından renk/parlaklık bilgisi alınarak ekrana basılmasıyla doku kaplama gerçekleştirilmiş olur. Fakat yukarıda da bahsedildiği üzere bir noktaya birden çok doku noktasından oluşan bir bölge karşılık geldiğinde örtüşme (aliasing) problemi ortaya çıkar (Şekil-7.a). Bunu önlemek için ilgili bölgedeki piksellerin ortalama renk/parlaklık bilgisi alınmalıdır. Burada ortaya çıkabilecek problem işlem karmaşıklığının artmasıdır. Mipmapping adı verilen yöntemle doku piramit yapıda değerlendirilerek işlem yükü hafifletilebilmektedir. Literatürde bundan başka birçok örtüşme önleme (anti-aliasing) yöntemi bulunmakta olup burada detaylandırılmayacaktır.



(a) (b)

Şekil-7 Doku kaplamada örtüşme problemi (a) örtüşme önlenmiş (b) örtüşme problemli görüntü

7. Deney Hazırlığı

- Dönmede kullanılan formüllerin çıkarılışını inceleyiniz.
- Quaternion'lar kavramını araştırınız ve dönmede kullanımını inceleyiniz.
- Paralel izdüşüm yöntemlerini kavrayınız.
- Perspektif/ters perspektif dönüşüm ve izdüşüm kavramlarını anlamaya çalışınız.
- Örtüşme yöntemlerinden mipmapping hakkında araştırma yapınız.
- Aşağıdaki sorunun çözümünü araştırınız :
  (x, y, z) cisim uzayındaki bir küre dilimi y ekseni etrafında -45 derece ve x ekseni etrafında +35 derece döndürüldükten sonra ortografik (dik) izdüşüm kullanılarak 32x32 piksellik bir (Px, Py) görüntü uzayında görüntülenmektedir. U, w düzlemindeki 64x64 piksellik basit bir ağ (grid) uygun bir dönüşüm ile bu küre dilimi üzerine yerleştirilmek isteniyor. 32x32 piksellik görüntü uzayına karşılık düşen cisim uzayı penceresinin $-1 \le x' \le 1,\ -1 \le y' \le 1$ olduğu varsayılmaktadır. Her piksel sol alt köşesinin koordinatları ile tanımlanmaktadır. Px=22 Py=22 pikselinin 64x64 piksellik u, w uzayındaki konumunu bulmak için:
  - (Px, Py) piksel değerleri ile (x', y', z') koordinatları arasındaki geçişi sağlayan bağıntıları bulunuz.
  - Transformasyon matrisini kullanarak (x, y, z) koorinatlarını hesaplayınız.
  - (Px, Py) pikseline ilişkin u, w koordinatlarını hesaplayınız.

8. Deney Tasarımı ve Uygulaması

- Rotasyon formüllerinin oluşumunu gösteriniz.
- Paralel izdüşüm türlerinin nasıl oluşturulduğunu ve nerelerde kullanıldığını kavrayınız.
- Dik (ortografik) izdüşüm ile yarım küre üreterek doku ile kaplayınız.
- Perspektif dönüşüm ile silindir yüzeyi kaplama kodlarını yazınız.
- Ters perspektif dönüşüm mantığı ile silindir yüzeyini kaplayınız.
- Z=0 yüzeyini ters perspektif dönüşüm ile kaplayacak kodları yazınız.
- Aynı yüzeyi açısal tarama mantığı ile kaplayınız.
- Bu yüzey için örtüşme önleme kodlarını geliştiriniz.
- Deneye hazırlık kısmındaki örnek soruda belirtildiği gibi dik izdüşüm ile ürettiğiniz yarım küreden bir küre dilimi seçerek bu küre dilimine dokuyu geriniz.

Görünmeyen Yüzey ve Arkayüz Kaldırma

1. Giriş

Bilgisayar grafiklerinin en önemli problemlerinden biri katı nesnelerin görünmeyen yüzeylerinin kaldırılmasıdır. Görünmeyen yüzeylerin kaldırılması, belli bir bakış noktasına göre görüntü düzlemindeki herhangi bir piksele karşılık gelen yüzeylerden en yakın olanını belirleme (diğerlerini kaldırma) işlemidir. Burada yüzeylerin birbirlerini kapatmamaları halinde bakış noktasından görülebilecekleri varsayılmaktadır. Arkayüz kavramı ise bunun tam tersidir. Yani herhangi bir yüzeyin önünde başka bir yüzey olmasa da o yüzey arkayüz veya başka bir deyişle bakış noktasına göre ters çevrilmiş bir yüzey olduğundan görülmesi imkansız olmasıdır. Ters çevrilmişten maksat o yüzeyin normalinin bakış noktasına doğru olmamasıdır. Bu konuda detaylı bilgi Arkayüz Kaldırma konusunda verilecektir.

Gerek görünmeyen yüzeylerin kaldırılması gerekse de arkayüz kaldırma için **Işın İzleme (Ray Tracing)** yöntemi kullanılacaktır. Işın izleme yönteminde bakış noktasından çıkan ve görüntü düzlemindeki piksellerin her birinden geçen ışınlar ile 3D cisimler arasında kesişim testleri yapılır ve ışık kaynakları da dikkate alınarak ekrana görüntüsü çizilecek cismin renk değeri hesaplanır. Herhangi bir ışın birkaç tane cisim ile kesişiyorsa bunlardan bakış noktasına en yakın olanının çizilip diğerlerini atılması işlemine görünmeyen yüzeylerin kaldırılması denir.

2. Vektörel İşlemlerle İlgili Temeller

Işın vektörel bir büyüklük olduğu için vektörel işlemler hakkında bilgiler vermekte fayda vardır:

Herhangi bir vektörün boyunu bulmak için (x,y,z) koordinatlarının karelerinin toplamının karekökü alınır. Örneğin **R=(0,6,8)** vektörünün boyu $|\mathbf{R}| = \sqrt{0^2 + 6^2 + 8^2} = 10$.

Herhangi bir vektörün boyunu 1 birim yapma işlemine "normalizasyon" denir ve işlem için (x,y,z) koordinatlarının her biri vektörün boyuna bölünür. Yukarıdaki **(0,6,8)** vektörü normalize edildiğinde $\left(\frac{0}{10}, \frac{6}{10}, \frac{8}{10}\right) = (\mathbf{0}, \mathbf{0.6}, \mathbf{0.8})$ bulunur. Normalize edilmiş vektörün boyu hesaplandığında $\sqrt{(0)^2 + (0.6)^2 + (0.8)^2} = 1$ olduğu görülür. Normalize edilmiş vektöre "birim vektör" denir. (Föy boyunca vektörel büyüklükler **koyu**, skaler büyüklükler normal font ile yazılacaktır).

Vektörler arasında vektörel ve skaler olmak üzere iki temel çarpım işlemi vardır. $\mathbf{R_1}$ ve $\mathbf{R_2}$ gibi iki vektörün sırasıyla vektörel ve skaler çarpımı aşağıdaki gibi yapılır:

$$R_1 \times R_2 = (R_{1y}R_{2z}-R_{1z}R_{2y},\ R_{1z}R_{2x}-R_{1x}R_{2z},\ R_{1x}R_{2y}-R_{1y}R_{2x})$$

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

Vektörel çarpım yüzey normalinin hesaplanmasında kullanılır. Yüzey normali yüzeye dik olan vektördür. İki vektörün skaler çarpımında (x,y,z) koordinatları çarpılıp toplanır veya vektörlerin boylarının aralarındaki açının kosinüsüyle çarpımı olarak da hesaplanabilir. Dolayısıyla skaler çarpımı yapılan vektörler birim vektör olursa bu vektörlerin skaler çarpımı aralarındaki açının kosinüsünü verir. Skaler çarpım Phong boyamada diffuse ve specular renk bileşenlerinin hesaplanmasında kullanılır. Vektörel çarpım vektörel; skaler çarpım skaler bir değer döndürür.

Köşe noktalarının koordinatları **V0,V1,V2** şeklinde verilen üçgenin yüzey normali **N** vektörü, **(V1-V0)** ve **(V2-V0)** vektörlerinin vektörel çarpımı ile şöyle hesaplanır:

V0=(0,40,120) V1=(30,-40,60) V2=(-30,-40,60)

(V1-V0)=(30,-80,-60) (V2-V0)=(-30,-80,-60)

N=(V1-V0)x(V2-V0)=(-80*-60 - -60*-80, -60*-30 - 30*-60, 30*-80 - -80*-30)

N = (0, 3600, -4800)

N vektörü normalize edilirse **(0,0.6,-0.8)** elde edilir.

3. Işının Tanımı ve Birincil Işınların Üretilmesi

Başlangıç noktası ve doğrultuya sahip vektörel bir büyüklük olan **R** ışını:

$$R = R_o + tR_d$$

olarak ifade edilir. Burada R_o ışının başlangıç noktası, R_d doğrultusudur. t ise ışının 3D uzayda R_d doğrultusu boyunca kaç birim gideceğini belirleyen skaler bir değerdir. Doğrultu vektörü R_d 'nin hesaplanabilmesi için 2 noktaya ihtiyaç vardır. Bunlar R_1 ve R_2 olarak alınırsa R_1'den R_2'ye doğru olan doğrultu vektörü $R_d=R_2-R_1$ ile hesaplanır. Işın izlemede doğrultu vektörleri birim vektör olmalıdır. Dolayısıyla R_d normalize edilerek boyu 1 birim yapılır.

Işın izlemede ilk adım başlangıç noktasından çıkıp 3D görüntü düzlemindeki piksellerin herbirinden geçecek olan birincil ışınların doğrultularının belirlenmesi işlemidir. Bunun için piksel koordinatlarından başlangıç noktasının koordinatı çıkarılır. Ardından normalize edilerek doğrultunun boyu 1 birim yapılır. Işınlar ile 3D ortamdaki cisimler arasında kesişim testleri yapılarak görüntü düzlemine hangi cismin şeklinin çizileceği belirlenir.

Şekil 1'den görüldüğü gibi ışınların 3D görüntü düzleminden geçtikleri pikseller ile en son ekranda üretilen görüntüdeki pikseller farklı koordinat sistemlerini kullanmaktadır. Örneğin bilgisayar ekranındaki 800x600 çözünürlükteki bir görüntünün (x,y) koordinatları sol üst köşede (0,0) sağ alt köşede de (799,599)'dur. Aynı çözünülürlükteki bir görüntü düzleminin sol üst köşesinin koordinatları (-400,300,500) olmalıdır (görüntü düzleminin bakış noktasına uzaklığı 500 birim alınmıştır). Dolayısıyla ekrandaki herhangi bir pikselin görüntü

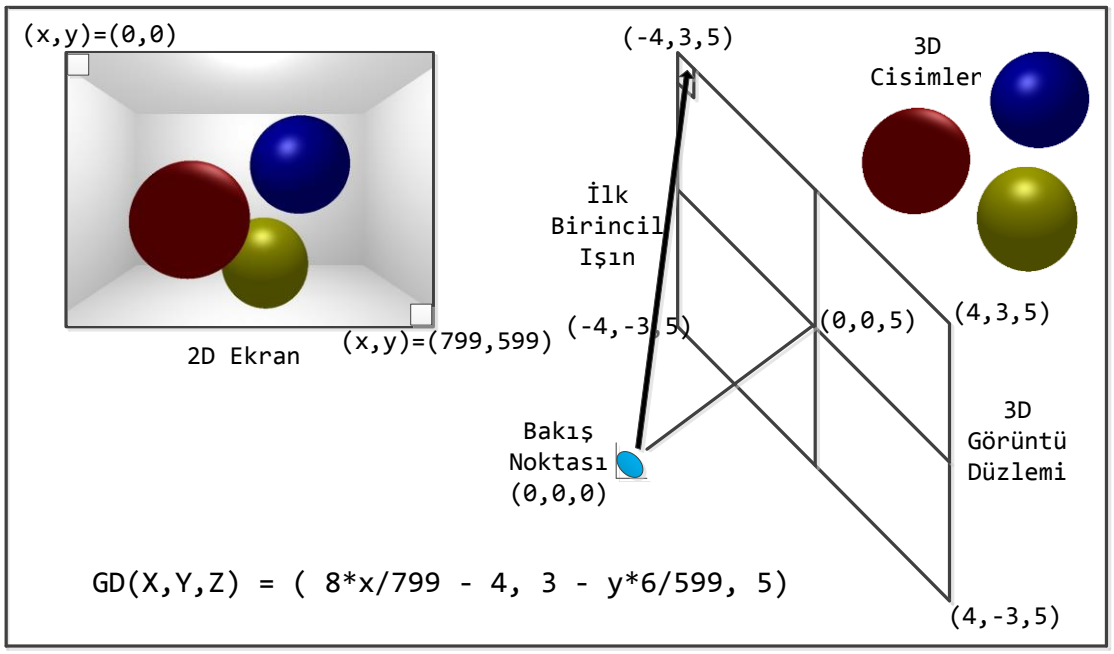
düzlemindeki karşılığını bulmak için bir dönüşüm yapmak gerekir. Ekrandaki 800x600 çözünürlüğünde bir görüntünün (x,y) koodinatlarının 3D Görüntü Düzlemi **GD(X,Y,Z)**'deki karşılığı aşağıdaki ifade ile bulunur:

GD(X,Y,Z) = (x - 399, 299 - y, 500)

Yukarıdaki dönüşümün pratikte kullanılması pek tercih edilmez. Çünkü çizilecek görüntünün çözünürlüğü değiştirilmek istenildiğinde sadece 399, 299 gibi değerleri değiştirmek yeterli olmaz. 3D cisimlerin koordinatlarını da değiştirmek gerekir. Dolayısıyla 3D cisimlerin koordinatlarından bağımsız dönüşüm imkanı sağlayan aşağıdaki ifade kullanılmalıdır:

GD(X,Y,Z) = (8*x/799 - 4, 3 - y*6/599, 5)

Burada görüntü düzlemi (8x6) boyutunda seçilmiştir. Üretilecek görüntünün çözünürlüğü 800x600'den farklı mesela 1024x768 olduğunda sadece 799'u 1023 ve 599'u 767 yapmak yeterlidir.



Şekil 1: 2D Ekran ve 3D Görüntü Düzlemi Arasındaki İlişki

4. Işın-Üçgen Kesişim Testi

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

- Işın ile üçgenin tanımladığı yüzey arasında kesişim testi.
- Işın yüzey ile kesişiyorsa kesişim noktasının üçgenin içinde olup olmadığını belirleme.

Birinci aşama için üçgenin tanımladığı yüzeyin denklemini çıkarmak gerekmedir. Bilindiği gibi yüzey denklemi Ax+By+Cz+D=0 ‘dır. Burada **(A,B,C)** yüzey normalidir. Yukarıda **V0,V1,V2** şeklinde verilen üçgenin yüzey denklemini çıkaralım:

Daha önce üçgenin normali **N=(0,0.6,-0.8)** olarak hesaplanmıştı. Dolayısıyla **N=(A,B,C)** biliniyor. Yüzeyin üzerinde olduğu için yüzey denklemini sağlayacağından üçgenin köşe noktalarından herhangi biri D ‘nin hesabı için kullanılabilir. Dolayısıyla Ax+By+Cz+D=0 ‘daki **(x,y,z)** yerine köşe noktalarından herhangi birinin mesela **V0** ‘ın **(x,y,z)** ‘sini yazıp sıfıra eşitlersek D ‘yi :

Ax + By + Cz + D = 0
0*0 + 0.6*40 + -0.8*120 + D = 0
D = 72

olarak buluruz. Dolayısıyla yüzey denklemi 0.6y - 0.8z + 72 = 0 ‘dır.

Işın yüzey ile kesişiyorsa üçgenin köşe noktalarında olduğu gibi ışının yüzey üzerindeki koordinatları da yüzey denklemini sağlamalıdır. Dolayısıyla şöyle yazabiliriz:

A($\mathbf{R_{0x}}$+t$\mathbf{R_{dx}}$)+B($\mathbf{R_{0y}}$+t$\mathbf{R_{dy}}$)+C($\mathbf{R_{0z}}$+t$\mathbf{R_{dz}}$)=0

Yukarıdaki denklem t ‘ye göre düzenlenirse:

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

t>0 ise ışın yüzey ile kesişiyor demektir. t<0 ise kesişmiyor, t=0 ise paralel demektir.

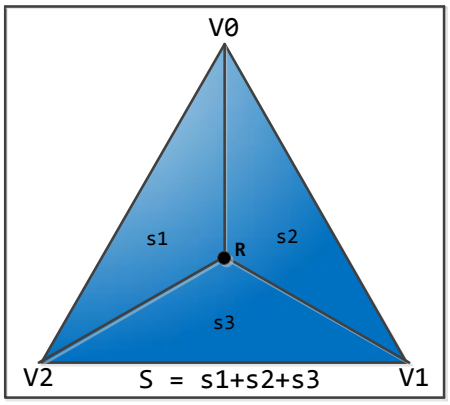
Yukarıda **V0,V1,V2** üçgeni için yüzey denklemi hesaplanmıştı. $\mathbf{R_o}$**=(0,0,0)** başlangıç noktasından **Rd=(0,0,1)** doğrultusu boyunca giden **R** ışınının bu yüzey ile kesişip kesişmediğini t değerini hesaplayarak belirleyebiliriz.

$$t = -\frac{N * R_o + D}{N * R_d} = -\frac{72}{-0.8} = 90$$

Işının yüzey üzerindeki koordinatları **R=**$\mathbf{R_o}$+t$\mathbf{R_d}$**=(0,0,0)**+90**(0,0,1)=(0,0,90)** olarak bulunur. Böylece kesişim testinin I. Aşaması tamamlanmış oldu.

II. aşamada **(0,0,90)** noktasının üçgenin içinde olup olmadığına karar verilmelidir. Bunun için değişik yöntemler denenebilir. Burada “alan testi” yöntemi kullanılacaktır. Buna göre Şekil 2‘den de görüldüğü gibi yukarıda hesaplanan kesişim noktasından üçgenin köşelerine doğrular çizerek 3 alt üçgen oluşturulur. Bu alt üçgenlerin alanları toplamı büyük üçgenin alanına eşitse kesişim noktası üçgenin içinde demektir.

V0,V1,V2 üçgeninin alanı 0.5*|**(V1-V0)x(V2-V0)**| ile hesaplanabilir. **R=(0,0,90)** noktası kullanılarak oluşturulan s1, s2 ve s3 alt üçgenlerin alanları ve **V0,V1,V2** üçgeninin S alanını hesaplandığında S=3000, s1=750, s2=750 ve s3=1500 çıkar. S=s1+s2+s3 olduğundan kesişim noktası üçgenin içindedir.



Şekil 2: Alan Testi

5. Görünmeyen Yüzeylerin Kaldırılması (Hidden Surface Removal)

Giriş bölümünde de bahsedildiği gibi görünmeyen yüzeylerin kaldırılmasından maksat aynı ışın doğrultusu boyunca kesişen cisimlerden en yakın olanın belirlenip diğerlerinin atılması (kaldırılması) dır. Bunu bir örnekle açıklayalım:

U0(0, 30, 40)	**U1(40, -30, 120)**	**U2 (-40, -30, 120)**
Y0(-50, 30, 124)	**Y1(50, 30, 124)**	**Y2 (0, -30, 44)**
Z0(-30, 0, 37)	**Z1(30, 40, 117)**	**Z2 (30, -40, 117)**

Sırasıyla kırmızı, yeşil ve mavi renklere sahip **U0,U1,U2** üçgeni **Y0,Y1,Y2** üçgeni ve **Z0,Z1,Z2** üçgeninin köşe noktalarının koordinatları yukarıda verilmiştir. Başlangıç noktası **R_o=(0,0,0)** 'dan çıkan ve görüntü düzleminde **(0,0,5)** noktasındaki pikselden geçen ışın ile bu üçgenler arasında kesişim testleri yapılırsa t_U=80, t_Y=84, t_Z=77 uzaklık değerleri hesaplanır. Uzaklıklar sıralandığında mavi renkli Z üçgeninin bakış noktasına daha yakın olduğu görülür. Dolayısıyla ışının geçtiği piksel mavi renge boyanır. Kırmızı renkli U ve yeşil renkli Y üçgenleri görüntü düzleminde **(0,0,5)** noktasındaki pikselden görünmeyen üçgenlerdir.

6. Arkayüz Kaldırma (Backface Culling)

Giriş bölümünde de bahsedildiği gibi arkayüz olan yüzey (veya üçgen) bakış noktası ile arasında başka yüzeyler olmasa bile ters durduğu için görünmeyen yüzeydir. Yüzeyin veya üçgenin ters durması ne demektir?

Y=0 yüzeyini düşünelim. Bu yüzeyin +Y ve –Y eksenlerine bakan iki farklı yüzü vardır. Y=0 yüzeyi üzerinde bir üçgen tanımlayalım:

U0(0, 0, 40) **U1(40, 0, -40)** **U2 (-40, 0, -40)**

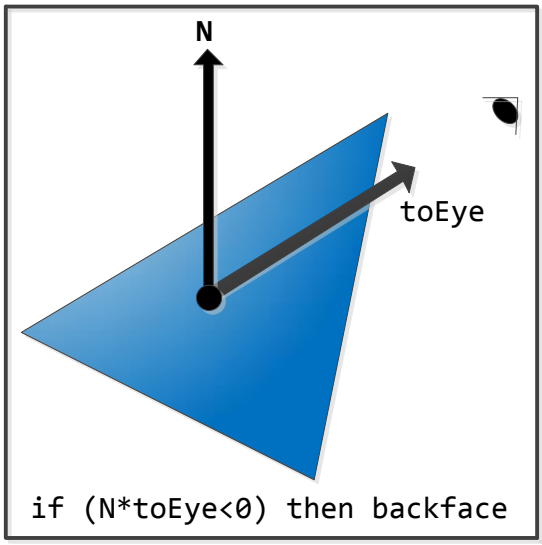
Bu üçgenin yüzey normali hesaplanırsa **(0, 6400, 0)** bulunur. Normalize edilirse **(0,1,0)** olur. Şimdi köşe noktalarının koordinatlarını farklı sırada yazalım:

U0(40, 0, -40) **U1(0, 0, 40)** **U2 (-40, 0, -40)**

Tekrar yüzey normali hesaplanırsa **(0, -6400, 0)** bulunur. Normalize edilirse bu sefer **(0,-1,0)** olur. İlk hesaplanan yüzey normali **(0,1,0)** +Y eksenine doğru şimdiki ise **(0,-1,0)** –Y

eksenine doğru çıktı. Dolayısıyla köşe noktalarının sırası değişince yüzey normalinin doğrultusu da değişmektedir. Köşe noktaları 3D kartezyen koordinat sisteminde Z ekseni (0,0,0) 'dan ileriye doğru pozitif artıyor ise (+Z) saat yönünde (ClockWise-CW); negatif olarak artıyor ise (-Z) saat yönünün tersi sırada (CounterClockWise-CCW) tanımlanmalıdır. Bu kurallara sırasıyla sol el ve sağ el kuralı denir. Sol el kuralında sol elin 4 uzun parmağı +X eksenini gösterirken +Y eksenini gösterecek şekilde katlandığında baş parmağın doğrultusu +Z eksenini gösterir. Aynı işlem sağ el ile yapıldığında baş parmak yine +Z 'i gösterir. DirectX 3D kartezyen koordinatlar için + eksenleri sol el; OpenGL de sağ el kuralına göre belirler.

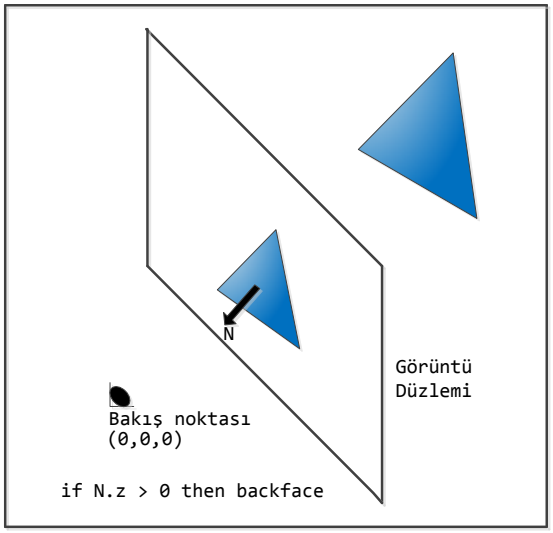
Yüzey normalinin doğrultusunun köşe noktalarının sırasına bağlı olarak değişmesi ile arkayüz olması arasında ne ilişki var? Bunu U üçgeni içindeki **(0,0,0)** noktasının diffuse katsayısını hesaplayarak açıklayalım:



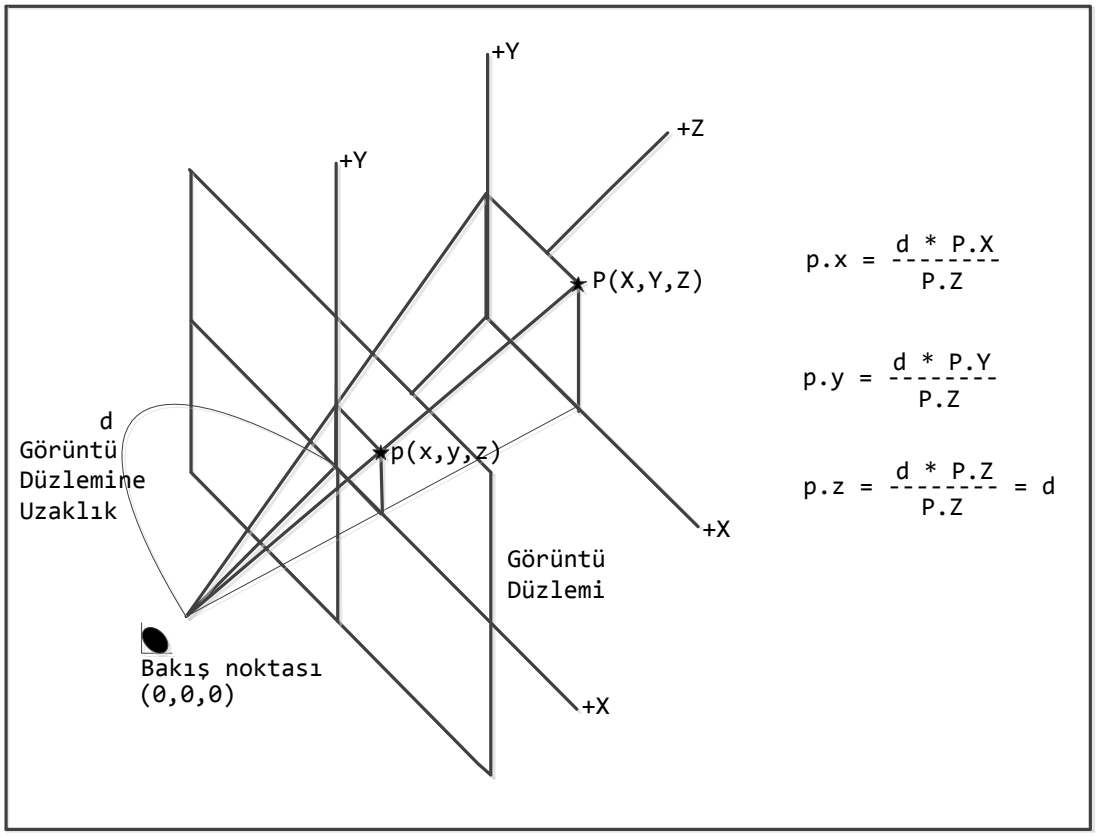
Şekil 3: Skaler çarpımla arkayüz kaldırma

Işık kaynağı **(0,60,80)** noktasında olsun. Işık kaynağına doğru olan vektör **(0,60,80)-(0,0,0)=(0,60,80)** normalize edilirse **(0,0.6,0.8)** bulunur. Diffuse kaysayıyı bulmak için bu vektör ile yüzey normali skaler çarpılırsa 0.6 bulunur. Aynı işlemler köşe noktalarının sırası değiştirilerek normali **(0,-1,0)** olan üçgen için yapılırsa bu sefer -0.6 bulunur. Renk değerinin negatif olması imkansız olduğundan -0.6 diffuse katsayı olarak kullanılamaz. Başka bir deyişle bu yüzey ışık kaynağı tarafından aydınlatılmıyor demektir. Demek ki yüzeyin ışık kaynağı tarafından aydınlatılıp/aydınlatılmadığına normalin doğrultusuna bağlı olarak karar veriyoruz. Benzeri şekilde yüzeyin bizim tarafımızdan görülüp görülmediğine de yine normaline göre karar veririz. Eğer bakış noktasına doğru olan vektör ile normalin skaler çarpımı negatif çıkıyorsa yüzeyin bakış noktasına göre görülmesi imkansız demektir. Yani arkayüzdür. Böylece arkayüz belirlemede kullanılan birinci yöntem açıklanmış oldu. Mesela yukarıda ikinci olarak tanımlanan U üçgeni bakış noktası **(0,60,80)** alındığında arkayüz olmaktadır. Bakış noktasına doğru olan vektör ile yüzey normalini skaler çarparak arkayüz kaldırma Şekil 3'te gösterilmiştir.

Arkayüz kaldırmada kullanılan diğer bir yöntemde üçgenin görüntü düzlemine izdüşümü alınır ve normali hesaplanır. Eğer normalin Z bileşeni >0 ise arkayüzdür. Yukarıdaki yöntemde skaler çarpma ile arkayüz belirlendiğinden "skaler çarpımla arkayüz kaldırma" olarak isimlendirildi. Burda da izdüşüm sonrası vektörel çarpımla normal hesaplandığından bu yöntem "vektörel çarpımla arkayüz kaldırma" olarak isimlendirilecektir. Vektörel çarpımla arkayüz kaldırma Şekil 4'te gösterilmiştir.



Şekil 4: Vektörel çarpımla arkayüz kaldırma



Şekil 5: Perspektif Dönüşüm ile İzdüşüm

Herhangi bir noktanın benzer üçgenler yardımıyla görüntü düzlemine perspektif dönüşüm ile izdüşümünün nasıl yapıldığı Şekil 5'te gösterilmiştir.

7. Deney Hazırlığı

- Görünmeyen Yüzeylerin Kaldırılması bölümündeki örnekte hesaplanan t değerlerini bulmak için gerekli ara işlemleri yazıp deneye getiriniz.
- Görünmeyen yüzey ile arkayüz arasında ne fark vardır? Karşımızda duran bir küpün görmediğimiz yüzeyleri görünmeyen yüzey midir yoksa arka yüz müdür?
- Vektörel çarpım ile arkayüz kaldırırken izdüşüm sonrası neden normalin X veya Y değil de Z bileşenine bakılır? Bu bileşenin neden sıfırdan küçük olması ile değil de büyük olması ile yüzey arkayüz olur?
- Önce hangisi kaldırılmalıdır? Görünmeyen yüzey mi arkayüz mü?
- Işın üçgen kesişim testinde kullanılan alternatif yöntemlerden biri de crossings testidir. Kaynak kodların olduğu klasördeki **CrossingTest.pdf** isimli belgeyi inceleyiniz.

8. Deney Tasarımı ve Uygulaması

V0(50, -30, 40) V1(0, 30, 120) V2(-50, -30, 40)

- Yukarıda köşe noktaları V0,V1,V2 olarak verilmiş üçgenin arkayüz olup olmadığını hem skaler çarpım hem de vektörel çarpım yöntemlerine göre belirleyiniz. Bakış noktası **(0,0,0)** 'dır. Yüzey üzerindeki nokta olarak köşe noktalarından herhangi birini alabilirisiniz. Hem skaler hem de vektörel çarpımda sonucun değeri değil işareti (pozitif mi negatif mi) önemli olduğundan yüzey normali ve bakış noktasına doğru olan vektörleri normalize etmenize gerek yoktur.
- Deneyde anlatılan arkayüz kaldırma, ışın-yüzey ve ışın-üçgen kesişim testlerinin kodlarını yazınız.
- Işın üçgen testinde noktanın üçgenin (poligonun) içinde olup olmadığının belirlenmesinde kullanılan en hızlı tekniklerden biri de crossings testidir. Bu yöntemde, noktadan herhangi bir doğrultuda ışın gönderilir. Eğer poligonla tek sayıda kesişirse poligonun içinde, çift sayıda kesişirse de dışında olduğuna karar verilir. Bunun için size verilen **CrossingTest.pdf** isimli belgede yalancı dilde yazılmı algoritmayı inceleyerek mantığını anlamaya çalışınız. İlgili algoritmayı kodlayarak programı test ediniz.

DirectX ile FPS Oyunu

1. Giriş

Oyunlar, Bilgisayar Grafiklerinin en popüler uygulama alanlarından biridir ve bilgisayarın dışında tabletler ve cep telefonları gibi farklı donanımları destekleyen oyunların da yaygınlaşmasıyla birlikte artan oranda popüler olmaya devam edeceği öngörülmektedir. 3 boyutlu (3D) oyun türleri arasında en yaygın olanlarından biri de oyun ortamının, oyuncunun gözlemlerine göre çizildiği FPS (First Person Shooter) tarzı oyunlardır.

Bu deneyde DirectX 11 API ile tek kullanıcılı basit bir FPS oyununun temel özelliklerinden bahsedilecektir.

2. 3D Cisme Yollanacak Işının Üretilmesi ve Kesişim Testi

Herhangi bir FPS oyununun en temel amaçlarından biri ateş edip rakip oyuncuyu vurmaktır. Ateş edilen rakip oyuncunun vurulup vurulmadığına karar vermek için ona doğru bir ışın yollanır ve bu ışın ile rakip oyuncuyu temsil eden modeli oluşturan üçgenler arasında kesişim testleri yapılır. Bu bölümde, mouse ile ekranda rakip oyuncuya denk gelen piksele tıklanması ile ateş edilip yollanacak mermiyi temsil edecek olan 3D ışının başlangıç noktası ve doğrultusunun nasıl üretildiği ve modelle nasıl keşim testi yapıldığı anlatılacaktır.

Ekranda rakibe ateş etmek amacıyla tıklanan noktaya ait 2D koordinatlardan 3D ortamdaki cisimle kesişim testinde kullanılacak ışının üretilmesi için bir takım transformasyonlar gerekmektedir.

Bilindiği gibi 3D ortamdaki herhangi bir noktanın ekran koordinatlarına dönüşümü için bu nokta sırasıyla `World`, `View` ve `Projection` matrisleri ile çarpılır. Dolayısıyla ekranda tıklanan 2D noktadan 3D ışını üretmek için matris çarpımlarının tersi yani ters matrisler ile işlemler yapmak gerekir.

Çizim yapılacak pencerenin `800x600` çözünürlükte olduğu varsayıldığında bu pencerenin sol üst köşesinin koordinatları `(0,0)` sağ alt köşesinin de `(799,599)` olur. Bu koordinatlardan pencerenin merkezi `(0,0)`'dan sağa doğru `+X`, sola doğru `-X`; yukarı doğru `+Y` ve aşağı doğru `-Y` olacak şekilde normalize edilmiş yani `(-1,+1)` arası değişen koordinatlara dönüşüm yapan DirectX kodu aşağıdaki gibidir:

```
D3DXVECTOR3 v;
v.x =  ( ( ( 2.0f * ptCursor.x ) / Width ) - 1 ) / pmatProj->_11;
v.y = -( ( ( 2.0f * ptCursor.y ) / Height ) - 1 ) / pmatProj->_22;
v.z = 1.0f;
```

Burada `ptCursor.x` ekranda tıklanan noktanın `(0,799)` arası değişen `x`; `ptCursor.y` de `(0,599)` arası değişen `y` koordinatlarıdır. `pmatProj` de `Projection` matrisidir. İşlemleri ters sırada yaptığımızdan `Projection` matrisinin ilgili diyagonal bileşeni ile çarpmak yerine ona bölüyoruz.

Yukarıdaki işlemler ışının üretilmesi için yeterli değildir. `v` vektörü ayrıca `World` ve `View` matrislerinin tersi olan `m` matrisi ile de aşağıda verilen koddaki gibi çarpılmalıdır :

```
// Get the inverse view matrix
const D3DXMATRIX matView = *g_Camera.GetViewMatrix();
const D3DXMATRIX matWorld = *g_Camera.GetWorldMatrix();
D3DXMATRIX mWorldView = matWorld * matView;
D3DXMATRIX m;
D3DXMatrixInverse( &m, NULL, &mWorldView );

// Transform the screen space pick ray into 3D space
vPickRayDir.x = v.x * m._11 + v.y * m._21 + v.z * m._31;
vPickRayDir.y = v.x * m._12 + v.y * m._22 + v.z * m._32;
vPickRayDir.z = v.x * m._13 + v.y * m._23 + v.z * m._33;
vPickRayOrig.x = m._41;
vPickRayOrig.y = m._42;
vPickRayOrig.z = m._43;
```

Böylece 2D ekranda tıklanan noktaya 3D uzayda karşılık gelen ışının başlangıç noktası `vPickRayOrig` ve doğrultusu `vPickRayDir` üretilmiş olur. Bu noktaya kadar verilen kod satırları "`Picking`" örnek programı içindeki `Pick()` isimli fonksiyondan alınmıştır. Yine bu fonksiyonda 3D ortamdaki cismin tam olarak hangi üçgenine tıklandığını belirlemek üzere Tomas Möller'in ışın-üçgen kesişim testi yöntemini kullanan `IntersectTriangle()` fonksiyonu çağrılmaktadır.

.obj formatındaki modeli temsil eden `verticesObjModel[]` dizisinden o anki üçgeni temsil eden `i` değişkeninin `3` katına sırasıyla `0`, `1` ve `2` eklenerek elde edilen indislerdeki köşe noktaları sırasıyla `v0`, `v1` ve `v2` vektörlerine atanmıştır. Işının başlangıç noktası `vPickRayOrig`, doğrultusu `vPickRayDir` ve `v0`, `v1` ve `v2` vektörleri kesişim testini yapacak `IntersectTriangle()` fonksiyonuna parametre olarak yollanmıştır.

`IntersectTriangle()` fonksiyonu kesişimin olup/olmamasına göre boolean bir değer döndürmenin yanında kesişen üçgene olan uzaklığı `fDist` olarak, `(u,v)` doku koordinatlarını veya barisentrik koordinatları da `fBary1` ve `fBary2` olarak hesaplayıp döndürmektedir. Bu değişkenleri içeren `INTERSECTION` türünden structure her bir kesişim için `g_IntersectionArray[]` adlı diziye eklenmektedir. Örneğin o doğrultu boyunca giden ışının kesiştiği üçgenlerden en yakın olanına ait bilgiler `g_IntersectionArray[0]` indisi altındaki değişkenlerde tutulmaktadır. `g_bAllHits` boolean değişkeni bütün kesişimlerin diziye alınıp/alınmayacağını belirler. Bu işlemlerin gerçekleştirildiği kod bloğu aşağıda verilmiştir :

```
for( DWORD i = 0; i < faceCount; i++ )
{
	D3DXVECTOR3 v0 = verticesObjModel[3 * i + 0].Pos;
	D3DXVECTOR3 v1 = verticesObjModel[3 * i + 1].Pos;
	D3DXVECTOR3 v2 = verticesObjModel[3 * i + 2].Pos;

	if( IntersectTriangle(vPickRayOrig,vPickRayDir, v0, v1, v2, &fDist,&fBary1,&fBary2))
	{
		if(g_bAllHits|| g_nNumIntersections==0 || fDist<g_IntersectionArray[0].fDist)
		{
			if( !g_bAllHits ) g_nNumIntersections = 0;
			g_IntersectionArray[g_nNumIntersections].dwFace = i;
			g_IntersectionArray[g_nNumIntersections].fBary1 = fBary1;
			g_IntersectionArray[g_nNumIntersections].fBary2 = fBary2;
			g_IntersectionArray[g_nNumIntersections].fDist = fDist;
			g_nNumIntersections++;
			if( g_nNumIntersections == MAX_INTERSECTIONS )    break;
		}
	}
}
```

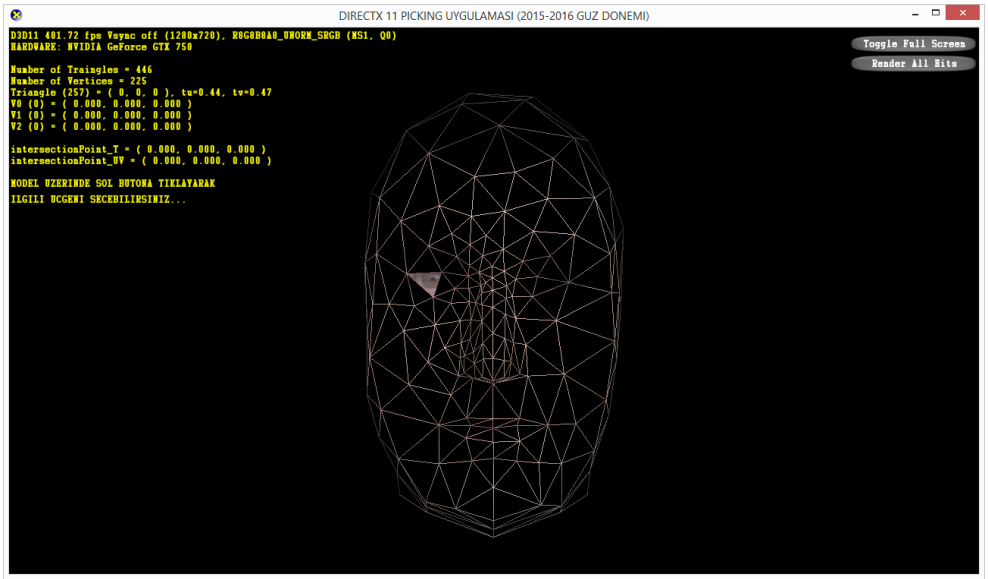
Model çizilirken yalnızca vertex buffer kullanılmıştır. Başka bir deyişle `verticesObjModel[]` dizisindeki köşe noktaları sırasıyla 3'erli gruplar halinde modelin üçgenlerini temsil etmektedir.

3. Kesişen Üçgenlerin Çizilmesi

Kesişen üçgenlerin çizimi `OnD3D11FrameRender()` fonksiyonunda aşağıdaki kod ile yapılır. Bu üçgenlerin köşe noktaları `PickedTriangle` isimli vertex bufferda tutulmaktadır. Buffera köşe noktalarını kopyalama işlemleri için `Map()` ve `Unmap()` emirleri kullanılmıştır (870. ve 894. satırlar). Çizim modu `RSSetState(g_pRasterizerStateSolid)` ile setlendikten sonra `Draw()` emri ile kesişen üçgen(ler)in çizimi gerçekleştirildikten sonra çizim modu `RSSetState(g_pRasterizerStateWireframe)` olarak setlenerek kafa modeline ait diğer üçgenler wireframe (sadece kenarlar) çizilmektedir. Böylece kesişen üçgen(ler) daha belirgin hale getirilmiştir.

Ekranda tıklanan noktadan yollanan ışın, kafa modeli ile önden ve arkadan olmak üzere iki noktada kesişmektedir. Bunlardan sadece en yakın olan öndekinin mi yoksa ikisinin birden mi Render edileceği "`Render All Hits`" butonu ile setlenir. Şekil-1'de programdan bir ekran görüntüsü verilmiştir :

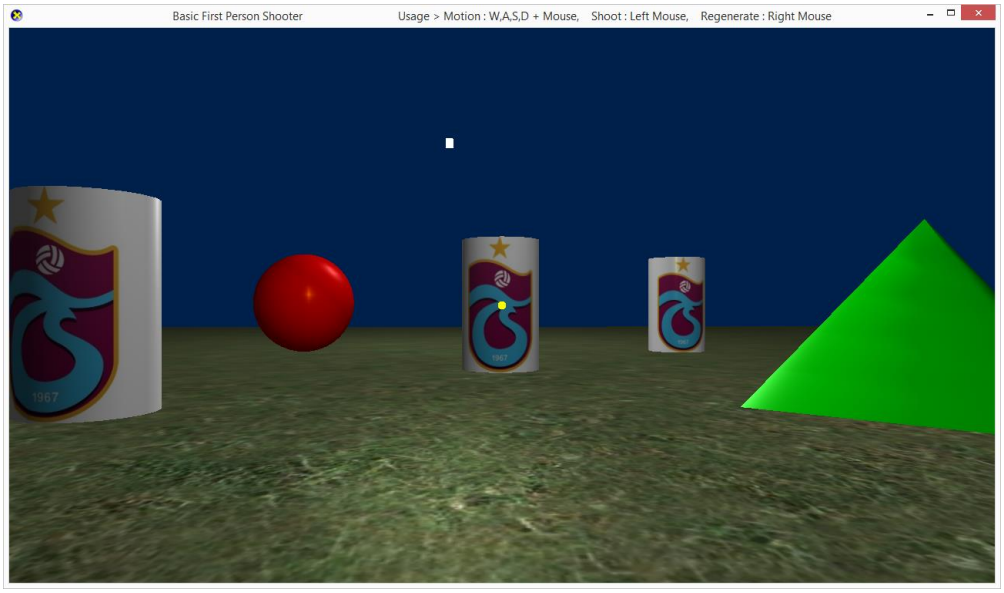
```
if( g_nNumIntersections > 0 )
{
	UINT Strides[1];                          UINT Offsets[1];
	Strides[0] = sizeof(SimpleVertex);        Offsets[0] = 0;

	pd3dImmediateContext->IASetVertexBuffers(0, 1, &PickedTriangle, Strides, Offsets);
	pd3dImmediateContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
	pd3dImmediateContext->PSSetShaderResources(0, 1, &g_pTextureObjModel);
	pd3dImmediateContext->PSSetSamplers( 0, 1, &g_pSamLinear );
	pd3dImmediateContext->Draw( 3*g_nNumIntersections, 0 );

	// Set render mode to Wireframe
	pd3dImmediateContext->RSSetState( g_pRasterizerStateWireframe );
}
```



Şekil-1: DirectX Picking Uygulaması

4. Basit bir FPS Oyunu

Bu bölümde basit bir FPS oyunu olan "`FPS_Game`" adlı DirectX 11 uygulamasından bahsedilecektir. Uygulamada ateş edildiğinde rakibin vurulup/vurulmadığı önceki bölümde anlatıldığı gibi ışın-üçgen kesişim testleriyle yapılmaktadır. Bunun için `testIntersection()` adlı fonksiyon yazılmıştır. Fonksiyonun başındaki `for()` döngüsü içinde test edilecek rakip oyuncuyu temsil eden modelin üçgenlerinin köşe noktaları 3'er 3'er ışın-üçgen kesişim testine tabi tutulur. Eğer kesişim varsa `TRUE` döndürülür. Nişan alınan modele mouse'un sol butonuna tıklanarak ateş edilir ve yollanan ışın modelle keşisirse modelin çizilip/çizilmeyeceğini temsil eden boolean değişken `FALSE` olarak setlenerek çizimi engellenir (öldürülür). Boolean değişken `TRUE` yapılarak tekrar çizilmesi (canlandırılması) için mouse'un sağ butonuna tıklanır. Uygulamanın ekran görüntüsü Şekil-2'de verilmiştir:



Şekil-2: DirectX FPS_Game Uygulaması

`testIntersection()` adlı fonksiyon kesişimin olup/olmamasına bağlı olarak boolean değer döndürmenin yanında modele olan `t` uzaklığını da döndürmektedir. Bu uzaklık belli bir değerin altına düştüğünde hareket durdurularak duvarların içinden geçilmesi engellenmiştir. `testIntersection()` fonksiyonu basılan tuş veya mouse butonu ile ilgili kodları barındıran `DetectInput()` fonksiyonu içinde çağırılmıştır.

5. Deney Hazırlığı

❖ Seçilen üçgene ait köşe noktalarının indisleri (`index_v#`) ve koordinatları (`_v#`) başlangıç değeri olarak `0`'a setlenmiştir. Doğru değerler ekrana yazılacak şekilde programda `861.` satırdan itibaren gerekli güncellemeleri yapınız. Hesaplanan değerlerin ekrana yazılması ile ilgili kodlar `RenderText()` adlı fonksiyonda vardır.

❖ Seçilen üçgende mouse'un hangi koordinatlara işaret ettiği 2 şekilde hesaplanabilir:

`fDist` uzaklık değerini kullanarak → `intersectionPoint_t`
`fBary1`,`fBary2` barisentrik koordinatlar ile → `intersectionPoint_uv`

`intersectionPoint_t` ve `intersectionPoint_uv` değişkenlerini hesaplamak için gerekli kodları yine programa `861.` satırdan itibaren ekleyiniz. Yazdığınız kodları (dosya boyutunu azaltmak üzere `.sdf` uzantılı dosya, `ipch` ve `Debug` klasörlerini sildikten sonra projenin tamamını .rarlayıp en geç deney saatine kadar) hem mustafayazici61@gmail.com adresine grup adına e-mail ile gönderiniz hem de USB bellekte deneye getiriniz. Programın tamamlanmış haline ait ekran görüntüsü `PickScreen.wmv` isimli video olarak kaynak kodların olduğu klasördedir.

6. Deney Tasarımı ve Uygulaması

❖ `IntersectTriangle()` isimli ışın-üçgen kesişim testi fonksiyonu yerine "Alan Testi" yöntemini gerçekleyecek şekilde `IntersectTriangleAlan()` fonksiyonuna gerekli kodları ekleyiniz. Kodları yazarken şunlara dikkat ediniz :

Skaler ve vektörel çarpımlar için sırasıyla `D3DXVec3Dot()` ve `D3DXVec3Cross()` emirlerini; vektör boyunu hesaplamak için de `D3DXVec3Length()` emrini kullanacaksınız.

Fonksiyon, kesişim varsa `true` döndürmenin yanı sıra `intersectionPoint_t` hesabında `fDist` olarak kullanılacak `t` değişkenini gerektiği gibi setlemelidir.

Kod tamamlandıktan sonra `Pick()` fonksiyonu içindeki `IntersectTriangle()` satırını kapatıp `IntersectTriangleAlan()` satırını açınız (`840.` ve `841.` satırlar).

❖ FPS_Game uygulamasında aynı doğrultudaki modellere ateş edildiğinde hepsi ölmektedir. Yalnızca en öndekinin ölmesi için gerekli kod güncellemelerini yapınız.

❖ FPS_Game uygulamasında `Render()` fonksiyonu içindeki `if(renderTSilindir2){}` kod bloğundaki kapalı (comment) satırlar açıldığında silindirlerden biri 5 birim yarıçapla dönmeye başlamaktadır. `testIntersection()` fonksiyonunda gerekli güncellemeleri yaparak hareket halindeki nesnelerin de vurulabilmesini sağlayınız.

7. Deney Raporu

Deney Raporunu, **`Rapor.docx`** adlı şablon belgeye göre grup adına hazırlayıp **Deneye Hazırlık** bölümünde istenilen kodlarla birlikte **en geç deney saatine kadar** mustafayazici61@gmail.com adresine gönderiniz. Mailin başlığına grubunuzu ve deneyin ismini yazınız. Rapor ayrıca printer çıktısı olarak **deneye de getirilecektir**.

Pürüzlü Yüzey Üretimi

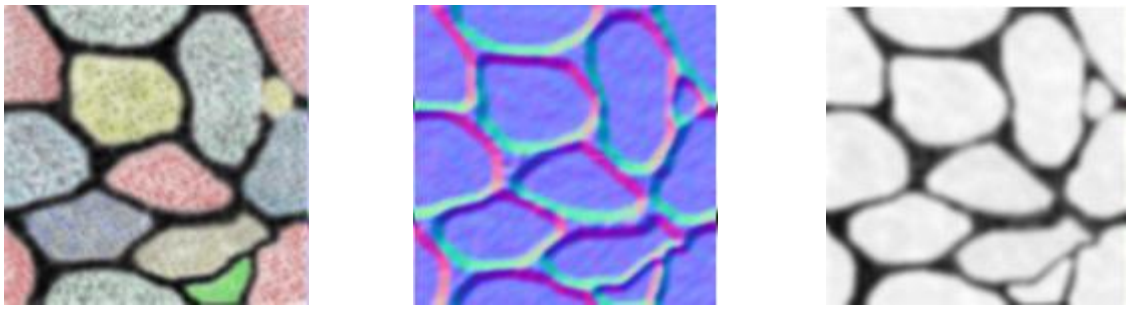
1. Giriş

Yüzeylerin özel bir doku kaplama yapılarak pürüzlü görünmesini sağlayan 2 temel yöntem vardır : 1.Bump Mapping, 2.Parallax Mapping.

Bump Mapping yönteminde Şekil.1(a)'da kaplanacak dokudaki renk değişimlerinden elde edilen (b)'deki "**Normal Map**" dokusu kullanılır. Bu normal mapdeki **(R,G,B)** değerleri yüzeyin normalinin **(X,Y,Z)** değerleri olarak alınır. Yüzeyin diffuse ve specular renk bileşenleri yüzey normaline bağlı olarak hesaplandığından **(0,1,0)** gibi tek bir normal değerine sahip düzlemsel bir yüzeye bump mapping yöntemine göre doku kaplandığında herbir piksel için normalmap dokusundan okunan farklı normallerle hesaplanan renk değerleri sanki yüzlerce farklı poligona sahip pürüzlü bir yüzey varmış hissi verecektir.

Parallax Mapping yöntemi **(R,G,B,A)** renk bileşenlerinden **A-**alpha parlaklık bileşeninden elde edilen Şekil.1(c)'deki "**Height Map**" dokusu ile yüzeyin yüksekliğini değiştirerek görüntüler ve böylece tümsekler/çukurlar oluşur. Bump mappingden farklı olarak yüzeyin koordinatları y-ekseni boyunca piksel mertebesinde gerçekten artar/azalır. Dolayısıyla elde edilen pürüzlü yüzeylerdeki tümsekler/çukurlar, bump mapping yöntemine nazaran daha fazladır. Hatta Parallax mapping yönteminde bu derinlik değerini ayarlamak bile mümkündür.

Deneyde **Jason Zink**'in, "**A Closer Look At Parallax Occlusion Mapping**" başlıklı makalesi ve örnek HLSL programından yararlanılarak geliştirilen DirectX 11 uygulaması ile yukarıda bahsedilen özel doku kaplama yöntemleri incelenecektir.



(a) (b) (c)

Şekil 1: (a)'daki dokudan elde edilen (b) Normal Map ve (c) Height Map dokuları

Bilindiği gibi DirectX uygulamalarında **.cpp** uzantılı program dosyasında çizilecek grafiği oluşturan poligonlara ait koordinat, normal, doku bilgileri ve **World, View, Projection** matrisleri setlenir. Bu bilgiler kullanılarak en son ekrana çizilecek şekle ait renk değerleri **.fx** uzantılı HLSL programındaki Vertex ve Pixel shader fonksiyonları aracılığıyla hesaplanır. Uygulamada üzerine doku kaplanacak yüzey düzlemsel olduğundan **.cpp** programında iki üçgen ile temsil edilmiş ve normal, doku koordinatları gerektiği gibi setlenmiştir.

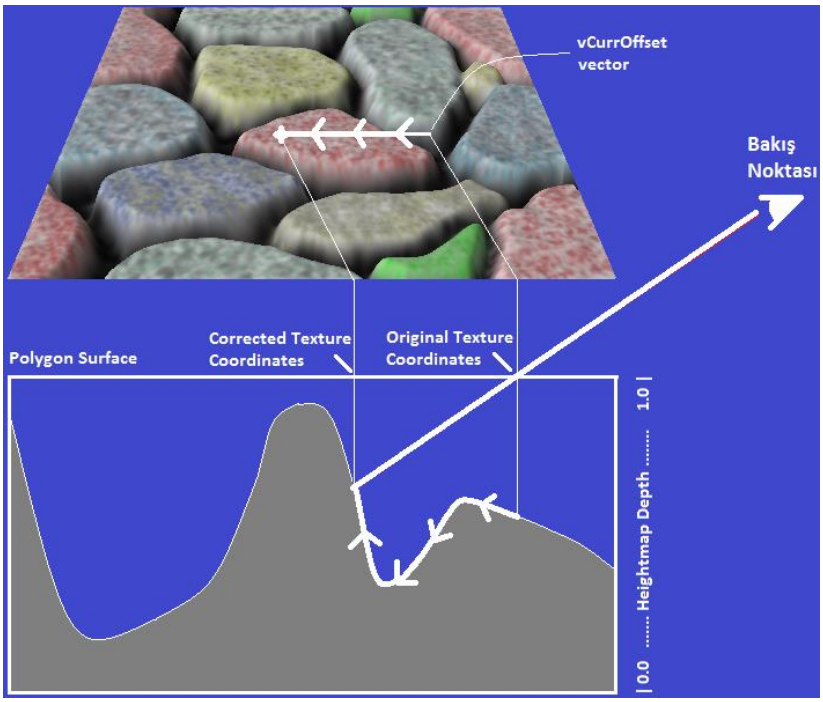
2. Parallax Mapping Yöntemi

Parallax mapping yöntemi ile pürüzlü yüzey üretilirken ilk işlem bakış noktasından **eye** ışını yollamaktır. Bu ışının yüzeye kaplanacak dokuda hangi renk ile kesiştiğinin belirlenebilmesi için **3D** uzaydan **2D (u,v)** doku uzayına bir dönüşüm yapılmalıdır. Bu dönüşüm **Normal (0,1,0)**, **Tangent (1,0,0)** ve bu ikisinin vektörel çarpımları ile hesaplanan **Binormal (0,0,-1)** vektörlerinden elde edilen matris ile ışının doğrultusu çarpılarak HLSL programındaki Vertex shader fonksiyonunda yapılır. **2D** doku uzayına izdüşürülen ışının dokuda kesiştiği koordinatlardaki rengin alpha değerine ve gerekirse ışının izdüşüm doğrultusu **vCurrOffset** boyunca başka alphalara bağlı olarak **Parallax Mapping** yönteminin nasıl gerçeklendiği aşağıda verilen Pixel shader kodu ve Şekil.2 üzerinden anlatılacaktır:

while döngüsünden önce **SampleGrad** fonksiyonu **IN.texcoord + vCurrOffset** parametresi ile çağrılmıştır. Burda **IN.texcoord** doku koordinatına eklenen **vCurrOffset**, Şekil.2'den de görüldüğü gibi doku üzerinde ilerlemede kullanılan doğrultu vektörüdür. Hesaplanması dolaylı yoldan **eye** vektörüne dayanır. **SampleGrad** fonksiyonunun sonundaki **.a** ifadesi ile **NormalHeightMap** isimli dokudan **fCurrSampledHeight a**lpha değeri okunur. Bu değer ile bakış noktasından yollanan **eye** ışınının, başlangıç değeri **1**'e setlenmiş yüksekliği **fCurrRayHeight** karşılaştırılır. **fCurrSampledHeight < fCurrRayHeight** yani dokudan okunan alpha, ışının yüksekliğinden küçük olduğu müddetçe **fCurrRayHeight** değeri **fStepSize** kadar azaltılır. Işının yüksekliği değiştikçe yeni okunan alphaların değerleri Şekil.2'deki Height Map eğrisi ile temsil edilmiştir. **fStepSize * vMaxOffset** ile **vCurrOffset** vektörü güncellenerek yeni **fCurrSampledHeight** değeri okunur. Dokudan okunan alpha değeri ışının yüksekliğinden **>=** olduğunda döngüden çıkılır. Bu nokta aynı zamanda ışının Height Map eğrisiyle kesiştiği noktadır. Dokuda bu noktaya karşılık gelen renk ekrana basılarak parallax mapping yöntemi gerçeklenmiş olur. Şekilde doku kaplanacak yüzey **polygon surface** ile temsil edilmiştir. Pürüzlü yüzeyin oluşması, ışının **polygon surface** ile kesiştiği nokta değil, alpha bileşeninden elde edilen height map eğrisi ile kesiştiği noktaya karşılık gelen doku koordinatlarındaki **vFinalColor** renginin görüntülenmesine dayanmaktadır.

```
fCurrSampledHeight = NormalHeightMap.SampleGrad( samLinear,
                                    IN.texcoord + vCurrOffset, dx, dy ).a;

while ( fCurrSampledHeight < fCurrRayHeight )
{
      fCurrRayHeight     -= fStepSize;
      vCurrOffset        += fStepSize * vMaxOffset;
      fCurrSampledHeight  = NormalHeightMap.SampleGrad( samLinear,
                                    IN.texcoord + vCurrOffset, dx, dy ).a;
```

```
}
float2 vFinalCoords  = IN.texcoord + vCurrOffset;
float4 vFinalColor   = ColorMap.SampleGrad( samLinear, vFinalCoords, dx, dy );

float4 vFinalNormal  = NormalHeightMap.SampleGrad(samLinear, vFinalCoords, dx, dy );
vFinalNormal         = vFinalNormal * 2.0f - 1.0f;

float3 vAmbient      = vFinalColor.rgb * 0.3f;
float3 vDiffuse      = vFinalColor.rgb * max(0.0f, dot( L, vFinalNormal.xyz )) * 0.7f;
vFinalColor.rgb      = vAmbient + vDiffuse;
OUT.color            = vFinalColor;
```



Şekil 2: Alpha eğrisi ve ışının yüksekliğine bağlı olarak piksel renginin bulunması

3. Bump Mapping Yöntemi

Giriş bölümünde de bahsedildiği gibi **Bump Mapping** yöntemi yüzey normalini "normal map" denilen özel bir dokudaki renk değerleri olarak almaya dayanır. Kaynak kodlardaki **`Textures`** klasöründe bulunan dokulara dikkat edilirse HLSL'deki **`ColorMap`** için **`*_colormap.dds`**; **`NormalHeightMap`** için de **`*_normalmap.dds`** gibi iki doku vardır. **`*_colormap.dds`** dokusundan hangi **`vFinalColor`** renginin okunacağına parallax mapping yöntemine göre yukarıda anlatıldığı gibi karar verilir. **`*_normalmap.dds`** dokusunu hem parallax mapping hem de bump mapping yöntemi kullanır. Gerçekte **`*_heightmap.dds`** gibi bir doku yoktur. Çünkü parallax mapping **`*_normalmap.dds`** 'nin **`(R,G,B,A)`** bileşenlerinden **`A-`**alphayı; bump mapping de yukarıdaki kod parçasından da görüldüğü gibi **`(R,G,B)`**'yi kullanır ve yeni yüzey normali **`vFinalNormal`** olarak alır. Böylece bump mapping yöntemi de gerçeklenmiş olur. Her bir piksel için farklı yüzey normali kullanılması her bir pikselin için farklı diffuse ve specular renk değeri hesaplanacağı anlamına gelir. Böylece yüzeyde pürüzler varmış gibi görülür. **`*_normalmap.dds`**'deki **`(R,G,B)`** değerleri **`[0..1]`** arası değiştiğinden **`3D`** uzayda normalize edilmiş **`[-1..1]`** aralığına map etmek için **`vFinalNormal * 2.0 - 1.0`** işlemi yapılmıştır.

Örnek programda klavyenin sağ/sol tuşları **fStepSize** değişkenini; yukarı/aşağı tuşu da yükseklik değerini arttırıp/azaltmaktadır. Ayrıca **"Q"**, **"W"** ve **"E"** tuşları ile değişik dokular arasında geçiş yapmak mümkündür.

Şekil 3: Specular renk bileşeni eklenmiş görüntü

4. Deney Hazırlığı

Örnek program ambient ve diffuse renk bileşenlerini hesaplamaktadır. Gerekli kodları yazarak bunlara Şekil.3'teki gibi specular renk bileşenini ekleyiniz. Bakış noktasına doğru olan vektör olarak **E** vektörünü kullanınız. Işık kaynağından gelen vektör olarak da **L** vektörünü kullanınız. Projenin tamamını **(.sdf** dosyası, **ipch** ve **Debug** klasörlerini sildikten sonra) **en geç deney saatine kadar** hem mehmetcemil@gmail.com adresine grup adına e-mail ile gönderiniz hem de USB bellekte deneye getiriniz.

5. Deney Soruları

1. Yüzey normali olarak **vFinalNormal** yerine yüzeyin kendi normali **N** kullanılacak şekilde kodu güncelleyiniz ve öncekinden farklı yönlerini açıklayınız. Örneğin iki görüntü arasındaki specular renk farklılıkları neden oluşmuştur?
2. Yüzeyin kendi normali **N** kullanılırken yukarı tuşuna sürekli basılıp yükseklik sıfırlandığında oluşan görüntüyü yorumlayınız. Yükseklik sıfır iken **vFinalNormal**'e göre çizilen görüntü ile normal **N** alındığında oluşan arasındaki farkın sebebi nedir?
3. Örnek program hem parallax hem de bump mapping yöntemini gerçeklemektedir. Sadece bump mapping yönteminin etkisini veya sadece parallax mapping yönteminin etkisini görmek için hangi değişiklikleri yapmak gerekir?

6. Deney Tasarımı ve Uygulaması

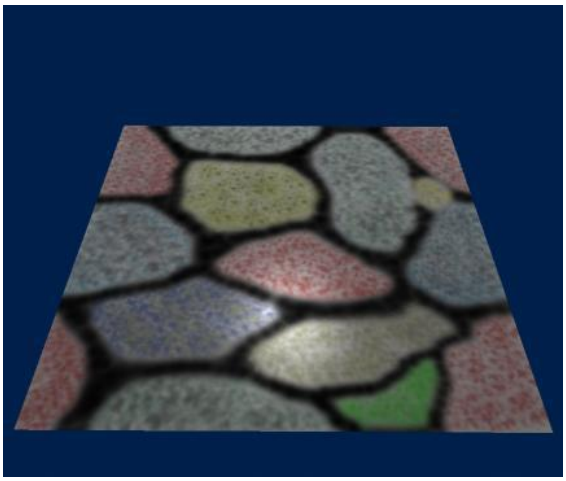
"**0**", "**1**", "**2**" ve "**3**" tuşlarına basıldığında Şekil.4'teki görüntüler elde edilecek şekilde programı güncelleyiniz. Görüntülerde :

"**00**" : ne bump mapping ne de parallax mapping var,
"**01**" : bump mapping yok, parallax mapping var,
"**10**" : bump mapping var, parallax mapping yok,
"**11**" : hem bump mapping hem de parallax mapping var

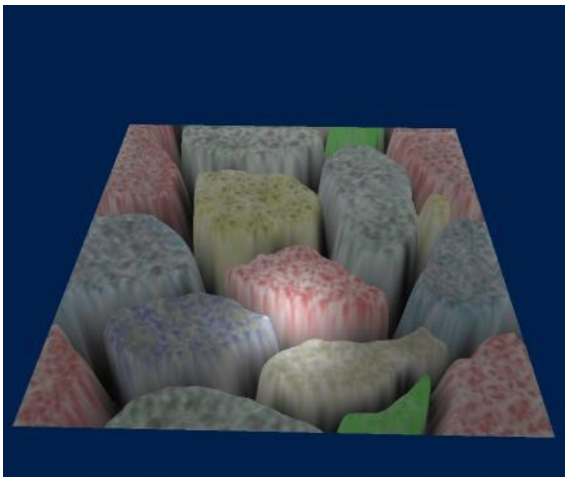
olarak özetleyebileceğimiz etkileri görüyoruz.

Klavyeden okunan değere göre **.fx** programını güncellemeyi `ConstantBuffer` adlı structure üzerinden yapınız. Yani **cb** adlı constant buffera ekleyeceğiniz yeni bir **int** değişkeni basılan tuşa göre setleyip **.fx** programında bu değere göre ilgili değişkenleri güncelleyiniz.

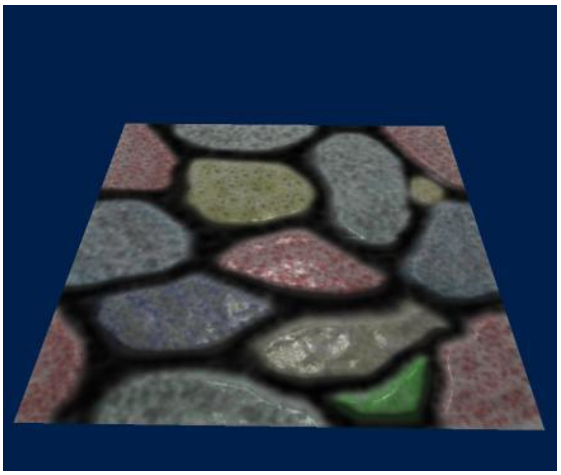
İpucu → Parallax mapping yüksekliğini **fParallaxLimit** değişkenini `0`'a setleyerek sıfırlayınız. Bump mapping etkisi ekleme/kaldırma için de **vFinalNormal** değişkenini uygun şekilde setleyiniz.



00 : no bump, no parallax



01 : no bump, yes parallax



10 : yes bump, no parallax

11 : yes bump, yes parallax

Şekil 4: bump/parallax mapping modları

Şekil.4'teki bump/parallax mapping modlarının görülebileceği **bpModes.wmv** isimli video, kaynak kodların olduğu klasördedir.

7. Deney Raporu

Deney Raporunu, **Rapor.docx** adlı şablon belgeye göre grup adına hazırlayıp **Deneye Hazırlık** bölümünde istenilen kodlarla birlikte **en geç deney saatine kadar** mehmetcemil@gmail.com adresine gönderiniz. Mailin başlığına grubunuzu ve deneyin ismini yazınız. Rapor ayrıca printer çıktısı olarak **deneye de getirilecektir**.