KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

BIL 4027 BİLGİSAYAR GRAFİKLERİ LABORATUARI

Ders Sorumluları
Öğr.Gör. Ömer ÇAKIR
Dr.Öğr.Üyesi Murat AYKUT

2024-2025 Güz Dönemi

İÇİNDEKİLER

			Sayfa No
	Önsöz		03
1	OpenGL Uygulamaları	Büşra ÖZKELLEKCİ	04
2	WebGL Uygulamaları	Ömer ÇAKIR	15
3	Yüzey Doldurma Teknikleri	Orhan SİVAZ	24
4	MAYA ile 3D Modelleme	Sefa KEKLİK	29
5	MAYA ile Animasyon	Seda EFENDİOĞLU	37
6	Ters Perspektif Dönüşüm ile Doku Kaplama	Murat AYKUT	45
7	DirectX ile Tank Oyunu	Ömer ÇAKIR	54
8	Pürüzlü Yüzey Üretimi	M. Cemil AYDOĞDU	60

Önsöz

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

Laboratuarda, dersin Işın İzleme (Ray Tracing), DirectX 12 ve MAYA ana başlıkları altında anlatılan konularına ilişkin "DirectX ile Tank 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

KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

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.

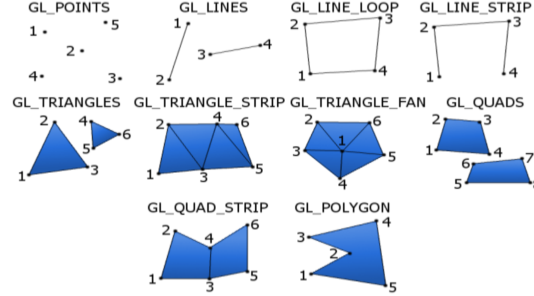
4.3. OpenGL Söz dizimi

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.
```

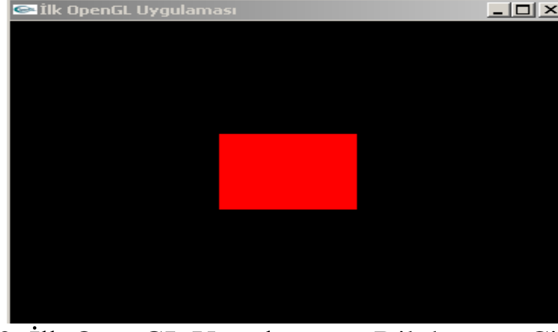


Şekil 1. İlkel (Primitif) Geometrik Nesneler

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 - Dikdörtgen 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.

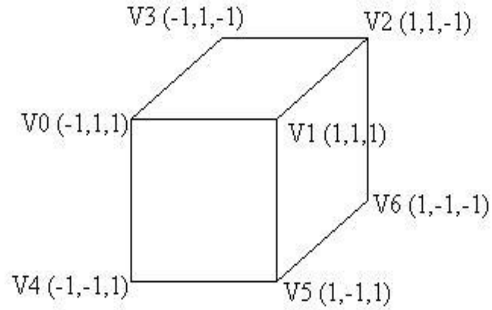
4.6. OpenGL Koordinat Sistemleri ve Dönüşümleri

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.

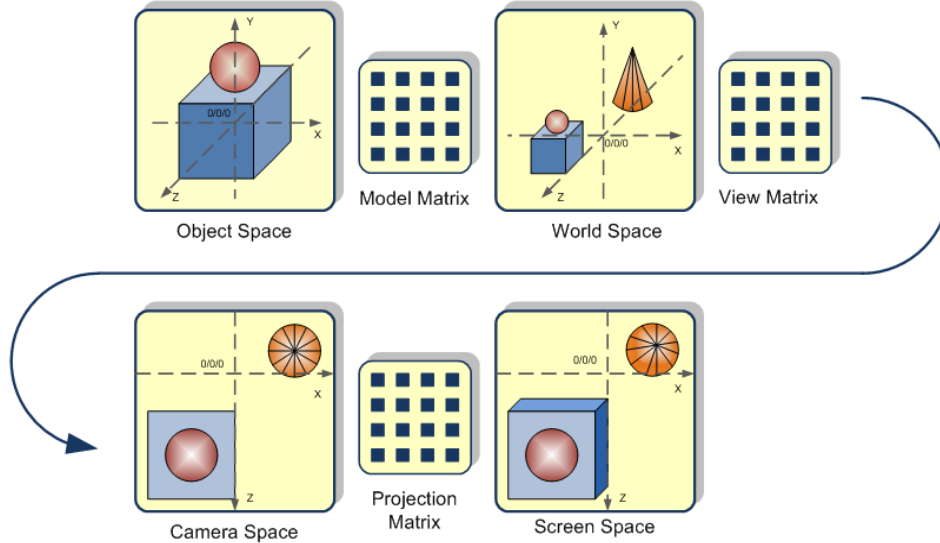
$$
\begin{bmatrix} 1 & 0 & 0 & \mathbf{5} \\ 0 & 1 & 0 & \mathbf{0} \\ 0 & 0 & 1 & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \mathbf{-1} \\ \mathbf{-1} \\ \mathbf{1} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{4} \\ \mathbf{-1} \\ \mathbf{1} \\ 1 \end{bmatrix}
$$

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);
```

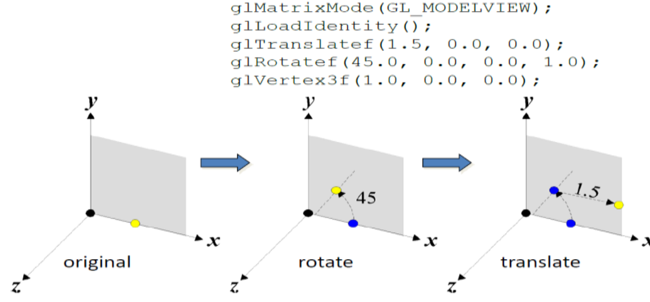
x, y, z parametrelerine geçirilen değerler her bir eksene göre ölçeklendirme değerini belirler. Örneğin, çizilen bir modelin derinlik ve yüksekliğini değiştirmeden, x ekseni üzerindeki genişliğini 2 katına çıkartılmak isteniyosa, aşağıdaki kod bu işlemi gerçekleştirir.

```
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
```

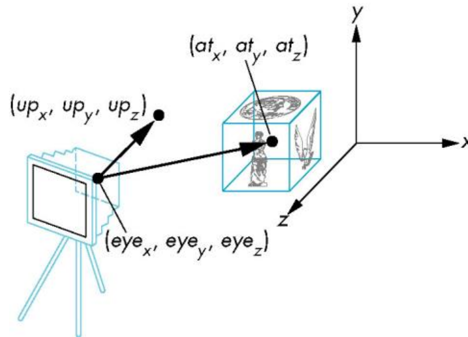
Şekil 6. Dönüşüm İşlem Sırası ve Farklı Çizim Sonuçları

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.

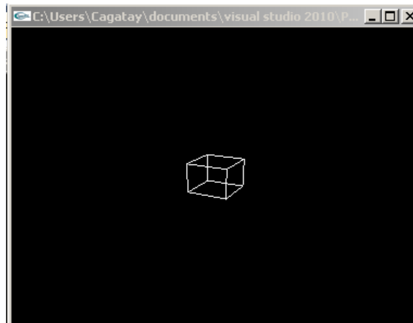


Şekil 7. OpenGL ile Kamera Konumu

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 C++ derleyicisi içeren herhangi bir sürümde Visual Studio IDe'si kurulmalıdır. *http://www.microsoft.com/visualstudio/tur/downloads#d-2010-expres* adresinden veya bölümümüzün DreamSpark Premium sayfasından indirebilirsiniz.
2. Ek-1'de verilen adımlar takip edilerek Microsoft Visual C++ 2010 Express veya diğerleri için (VS 2013 vs.) gerekli GLUT kütüphaneler eklenmelidir. (Farklı olarak VS 2013 için C:\Program Files (x86)\Microsoft Visual Studio 12.0)
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 Tasarımı ve Uygulaması

1. Deneye hazırlıksız gelen ve deney sırasında ilgisiz olan öğrenciler deneyden çıkarılacaktır. Deneye OpenGL uygulamalarının nasıl çalıştırılacağı, Visual Studio IDE'si ile kod düzenleme ve çalıştırmanın nasıl yapılacağını bilmeniz gerekmektedir. Deney ortamında VS 2013 kurulu olacaktır.
2. Deney sayfasında yer alan uygulamalar ve kaynak kodlar çalıştırılır ve incelenir. Deney sırasında uygulamalar üzerinde değişiklik yapar farklı grafik ve animasyon çıktıları elde etmeniz istenecektir. OpenGL hakkında temel bilgiler soru-cevap şeklinde sorgulanır. Uygulama alanları ve güncel bazı OpenGL uygulamalarına örnekler verilir.
3. GLUT kütüphanesi nedir ve ne için kullanılır soruları cevaplanır, kod üzerinde uygulama yapmanız istenir.
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. Parametreler değiştirilerek farklı açılardan kamera görüntüsü incelenir. İlgili fonksiyonların işlevleri soru-cevap şeklinde sorgulanır.

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 deney sayfasındadır. Gerekli açıklamalar ve sorular rapor kapağında verilmiştir. Raporda istenen dışında deneyle ilgili herhangi bir şey yazmayınız.

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

WebGL Uygulamaları

1. Giriş

WebGL serbest kullanım lisanslına sahip çapraz platform destekli OpenGL ES 2.0 temelleri üzerine inşa edilmiş düşük seviye 3D grafik programlama API'dır. WebGL'in en öne çıkan özellikleri:

- OpenGL Shading Language (GLSL) 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 edilir yani eklenti gerektirmez (plugin free). İşletim ve pencere sistemi seviyesinde (operating / window system independence) bağımsızlık sağlar.
- Uygulamalar uzak sunucu üzerinde saklanabilir.
- Farklı web uygulamalarına kolayca bütünleştirilebilir, CSS ve JQuery gibi standart web paketleri ile birlikte kullanılabilir.
- Masaüstü ve taşınabilir cihazlar üzerinde birlikte çalışabilir.
- WebGL her geçen gün daha fazla modern GPU özelliklerinden faydalanmakta ve geliştirilmektedir. Hızlı gelişen bir platformdur.

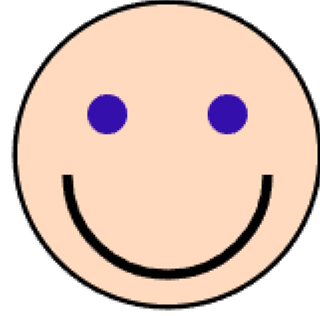
2. Canvas API ile Web Tabanlı 2D Çizim Uygulaması

WebGL uygulamaları 3D grafikler çizmek için *`<canvas>`* elementini, Javascript 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 da 2D çizimler gerçekleştirilebilir. Canvas API ile ilgili dokümantasyona buradan erişebilirsiniz.

WebGL uygulamalarına başlamadan önce Canvas API ile 2D gülen yüz çizen basit bir uygulamadan bahsedilecektir. Örnek programlardaki **`0_canvas.html`** sayfasına ait kodların bir kısmı sonraki sayfada verilmiştir. Gülen yüz uygulamasının temel adımları şu şekildedir:

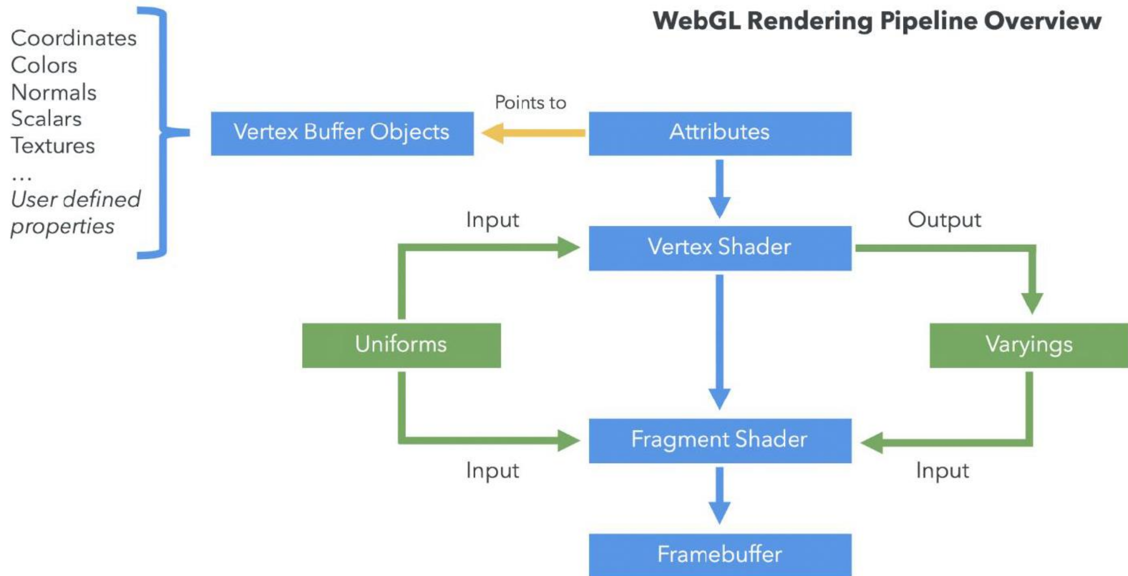
1. 512x512 boyutunda çizim alanı oluşturmak üzere *`<canvas>`* elementi ve Javascript ile *`<canvas>`* elementine erişmek için `id` değişkeni tanımlanmıştır (7. Satır).
2. `getElementById()` Javascript metodu ile *`<canvas>`*'a erişilmiştir (11. Satır).
3. `getContext()` ile 2D çizimler için render değişkeni tanımlanmıştır (12. Satır).
4. `beginPath()`, `arc()`, `stroke()` gibi Canvas API metodları ile 2D çizim işlemleri gerçekleştirilmiştir (15. Satır 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="512" height="512">
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>
38 </body>
39 </html>
```



3. İlk WebGL Uygulaması : Üçgen Çizimi

İlk WebGL uygulaması olarak bir üçgen çizilecektir. Uygulama **`1_triangle.html`** ismi ile kaydedilmiştir. Gerek HTML gerekse de Javascript kodları editlenirken Notepad++ kullanılacaktır. `1_triangle.html` içindeki kodları detaylı olarak anlatmadan önce aşağıdaki şekil üzerinden WebGL rendering pipeline hakkında bilgi vermekte fayda vardır:



WebGL kodları ekran kartında koşacağı için GLSL (GL Shading Language) tabanlı **Vertex Shader** ve **Fragment Shader** fonksiyonlarını içermelidir. Vertex Shader fonksiyonu köşe noktaları üzerinde Perspektif Projeksiyon, Öteleme, Ölçekleme, Döndürme gibi çeşitli transformasyonları yapar. Fragment Shader renk hesabı yapar.

Vertex Buffer Objects, köşe noktalarına ait konum, normal, renk ve doku bilgilerini içeren vertex dizisini tutar. Bu dizideki her bir verinin türüne ait bilgiye **Attribute** denir.

Uniforms, Vertex Shader ve Fragment Shader fonksiyonlarının Javascript metodları ile haberleşmesini sağlayan değişkenlerdir. Vertex Shader'ın çıktısı ve aynı zamanda da Fragment Shader'a input olan verilere **Varyings** denir. Render edilecek grafik browserda görüntülenmeden önce **Framebuffer** adı verilen bellek alanına çizilir.

WebGL rendering pipeline hakkında yukarıda verilen genel bilgilerin ardından üçgen çizimi ile ilgili detaylardan bahsedebiliriz: `1_triangle.html` sayfası Notepad++ ile açıldığında `<script>` taglarıyla ayrılmış 3 tane kod bloğu görülür. Bunlardan **[Blok_1]**:Vertex Shader'ı, **[Blok_2]**:Fragment Shader'ı **[Blok_3]**: de Vertex Shader ve Fragment Shader'ı derleyen, onlara çizilecek üçgenlerin köşe noktası bilgilerini ileten ve nihayet çizim yapan emirleri içerir. Öncelikle [Blok_1]'deki VertexShader ve onunla ilgili [Blok_3]'teki kodlardan bahsedilecektir. Sonrasında [Blok_2]'deki FragmentShader ve onunla ilgili [Blok_3]'teki kodlardan bahsedilecektir:

[Blok_1]: Vertex Shader fonksiyonu :

```
#version 300 es
in vec4 vPosition;
void main()
{
  gl_Position = vPosition;
}
```

[Blok_3]: Vertex Shader ile ilgili kodlar :

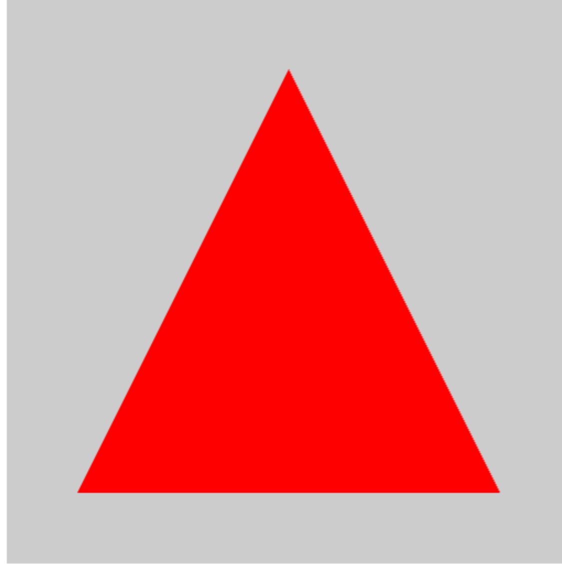
```
var vertexElem = document.getElementById( "vertex-shader" );
var vertexShader = gl.createShader( gl.VERTEX_SHADER );
gl.shaderSource(vertexShader, vertexElem.textContent.replace(/^\s+|\s+$/g,'' ));
gl.compileShader( vertexShader );

var vertices = new Float32Array([0.0,  0.75,  0.75,  -0.75,  -0.75, -0.75]);
var vertex_buffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vertex_buffer );
gl.bufferData( gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW );

var Position = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( Position, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( Position );
```

[Blok_3]'e ait yukarıda verilen kodlardan `gl.compileShader()` ile **Vertex Shader** derlenir. Çizilecek 2D üçgenin (x,y) köşe noktalarına ait konum bilgileri `vertices` dizisinde tutulur. `vertex_buffer` adında bir buffer değişken tanımlanır. `gl.bindBuffer()` ile çizimde `vertex_buffer`'daki vertex bilgilerinin kullanılacağı söylenir ve `gl.bufferData()` ile `vertices` dizisinin içeriği vertex buffera kopyalanır.

`vertices` dizisinde üçgenin köşe noktalarına ait konum bilgilerinin (x,y) çiftleri halinde floating point sayılar olarak tutulduğunu Vertex Shader'a bildirmek üzere `Position` attribute değişkeni tanımlanır. `Position` attribute değişkeninin VertexShader'daki eşdeğerine `vPosition` adı verilmiştir. Başka bir deyişle Vertex Shader tarafında üçgenin köşe noktalarına `vPosition` değişkeni üzerinden erişecektir. `gl.vertexAttribPointer( )`'daki `2` parametresi vertexlerin (x,y) koordinat çiftleri halinde olduğunu, `gl.FLOAT` da x ve y koordinatlarının floating point sayıları olduğunu gösterir.

Üçgenlerin çizimi aşağıdaki kodlarla gerçekleştirilir. `gl.clear()` FrameBuffer'ı `gl.clearColor()` ile setlenen açık gri renge boyar. `gl.drawArrays()` ile üçgen çizilir.

```
gl.clear( gl.COLOR_BUFFER_BIT );
gl.drawArrays( gl.TRIANGLES, 0, 3 );
```



[Blok_2]: Fragment Shader fonksiyonu :

```
out vec4  fColor;
void main()
{
    fColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
```

[Blok_3]: Fragment Shader ile ilgili kodlar :

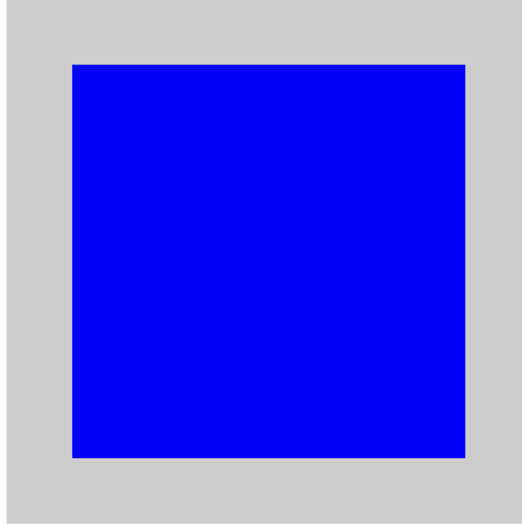
```
var fragmentElem = document.getElementById( "fragment-shader" );
var fragmentShader = gl.createShader( gl.FRAGMENT_SHADER );
gl.shaderSource(fragmentShader,fragmentElem.textContent.replace(\s+$/g,''));
gl.compileShader( fragmentShader );
```

[Blok_3]'e ait yukarıdaki kodlardan `gl.compileShader()` ile **Fragment Shader** derlenir. Fragment Shader'daki `fColor = vec4( 1.0, 0.0, 0.0, 1.0 );` ile üçgen kırmızı renge boyanır. `vec4( 1.0, 0.0, 0.0, 1.0 )` 'deki sayılar sırasıyla (R,G,B,A) değerleridir. Yani kırmızı, yeşil, mavi renk bileşenleri ve alpha değeridir.

[Blok_3]'te hem Vertex Shader hem de Fragment Shader'la ilgili bahsetmediğimiz şu kodlar kaldı:

```
var program = gl.createProgram();
gl.attachShader( program, vertexShader );
gl.attachShader( program, fragmentShader );
gl.linkProgram( program );
gl.useProgram( program );
```

Yukarıdaki kodlarla bir `program` değişkeni tanımlanıp daha önce derlenmiş olan Vertex ve Fragment Shaderlar `attachShader()` ile bu `program` değişkeni ile ilişkilendirilmiştir. Dolayısıyla 1'den fazla Vertex Shader veya 1'den fazla Fragment Shader söz konusu olduğunda onlarla ilgili ayrı program değişkenleri tanımlanabilir. Sonraki ugulamalarda buna örnek verilecektir.



4. WebGL ile Kare Çizimi (Index Buffer)

Bu bölümde 2 uygulamadadan bahsedilecektir: **`2_square_vertexBuffer.html`** dosyası yalnızca vertex buffer ile, **`3_square_indexBuffer.html`** da hem vertex hem de index buffer ile kare çizer. Yalnızca vertex buffer ile çizerken vertex dizisinde iki dik üçgeni temsil etmek üzere 6 tane (x,y) koordinat çifti olur:

```
var vertices = new Float32Array(
              [ 0.75,  0.75,
                0.75, -0.75,
               -0.75, -0.75,
                0.75,  0.75,
               -0.75, -0.75,
               -0.75,  0.75 ]);
```

Dolayısıyla kırmızı ve mavi renkle yazılmış olan köşe noktalarının tekrarı söz konusudur. Kare şekli vertex + index buffer ile çizilirken `vertices` dizisinde 4 köşe noktası tutulur. Bu köşe noktalarının `0..3` arası indisleri 3'er gruplar halinde üçgenleri oluşturur:

```
var vertices = new Float32Array(
              [ 0.75,  0.75,
                0.75, -0.75,
               -0.75, -0.75,
               -0.75,  0.75 ]);
var indices  = new  Uint16Array( [ 0,1,2,   0,2,3 ] );
```

3. bölümde üçgenin köşelerine ait `vertices` dizisinin tutulduğu `vertex_buffer` tanımlanmıştı. Benzer şekilde yukarıdaki `indices` dizisi için `index_buffer` tanımlanır. `gl.bindBuffer()` ile çizimde `index_buffer`'daki indis bilgilerinin kullanılacağı söylenir ve `gl.bufferData()` ile `indices` dizisinin içeriği index buffera kopyalanır. Çizim emirleri açısından fark şudur : Tek başına vertex buffer için `gl.drawArrays()` vertex + index buffer için `gl.drawElements()` kullanılır.

```
var index_buffer = gl.createBuffer ();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, index_buffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
```

5. WebGL ile Döndürme ve Öteleme Dönüşümleri

Bu bölümde, iki farklı uygulama ile önceki bölümde çizdiğimiz kareyi önce döndürüp sonra öteleyeceğiz. Döndürme işlemi ile ilgili kodlar **`4_rotating_square`** klasöründedir. z-ekseninde döndürme için aşağıdaki matris kullanılır:

CCW Rotation around Z-axis

$$\begin{bmatrix} \cos(\beta) & \sin(\beta) & 0 \\ -\sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Yukarıdaki döndürme matrisinin Vertex Shader içinde kodlanmış hali şöyledir:

```
float sin_ = sin(vTheta);
float cos_ = cos(vTheta);
gl_Position.x = vPosition.x * cos_ - vPosition.y * sin_;
gl_Position.y = vPosition.x * sin_ + vPosition.y * cos_;
```

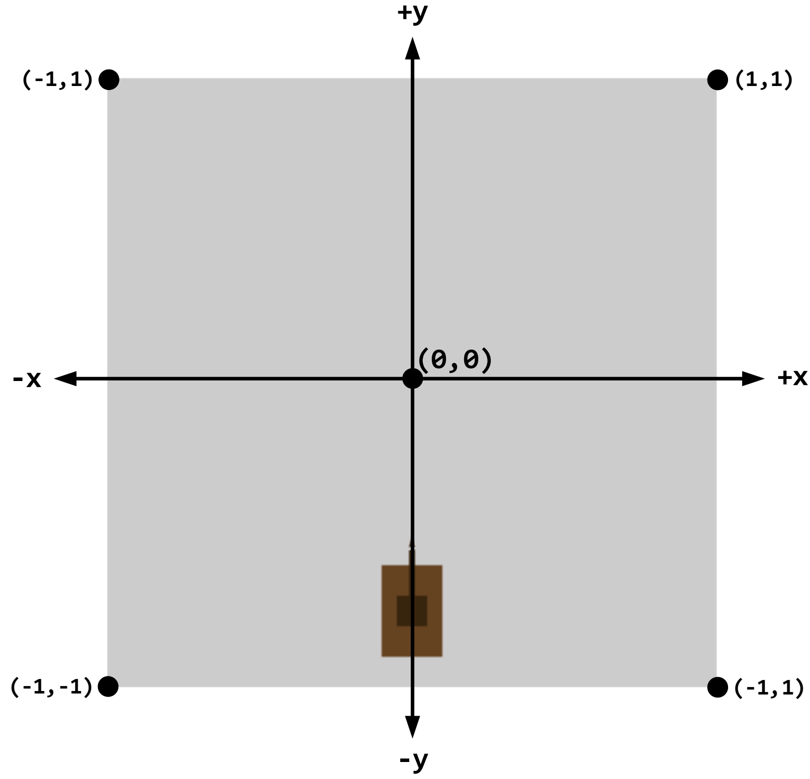
Vertex Shader veya Fragment Shader fonksiyonlarının JavaScript metodlarıyla `uniform` değişkenler üzerinden haberleştiğini daha önce belirtmiştik. Döndürme işlemi için Vertex Shader'da `vTheta` uniform değişkeni tanımlanmıştır. Bu değişkenin değeri JavaScript metodunda `theta` adlı başka bir değişken üzerinden setlenir. `vTheta` ile `theta` değişkenleri `thetaLoc` ara değişkeniyle `gl.uniform1f()` fonksiyonu üzerinden üzerinden ilişkilendirilir.

Öteleme işlemi ile ilgili kodlar **`5_moving_square`** klasöründedir. Öteleme işlemi için Vertex Shader'da `vMove` uniform değişkeni tanımlanmıştır. Bu değişkenin değeri JavaScript metodunda `move` adlı başka bir değişken üzerinden setlenir. Örnek uygulamada kare şekli sağa sola hareket etmektedir. Dolayısıyla köşe noktalarının sadece `x` değerleri değişir. İlgili Vertex Shader kodu:

```
gl_Position.x += vPosition.x + vMove;
gl_Position.y  = vPosition.y;
```

JavaScript tarafındaki `move` değişkeni sağa doğru harekette arttırılmış; sola doğru harekette azaltılmıştır. Sürekli sağa veya sürekli sola gidilmesin diye `moveRight` ve `moveLeft` şeklinde iki boolean değişken tanımlanmıştır.

Bundan önceki uygulamalarda çizim emri yalnızca bir kez koşuyordu. Bu sefer gerek döndürme gerekse de öteleme işlemi bir animasyon gibi sürekli koşmaktadır. Bunun için çizim emirlerini içeren `Render()` fonksiyonu tanımlanmış ve `requestAnimationFrame()` ile sürekli koşması sağlanmıştır.

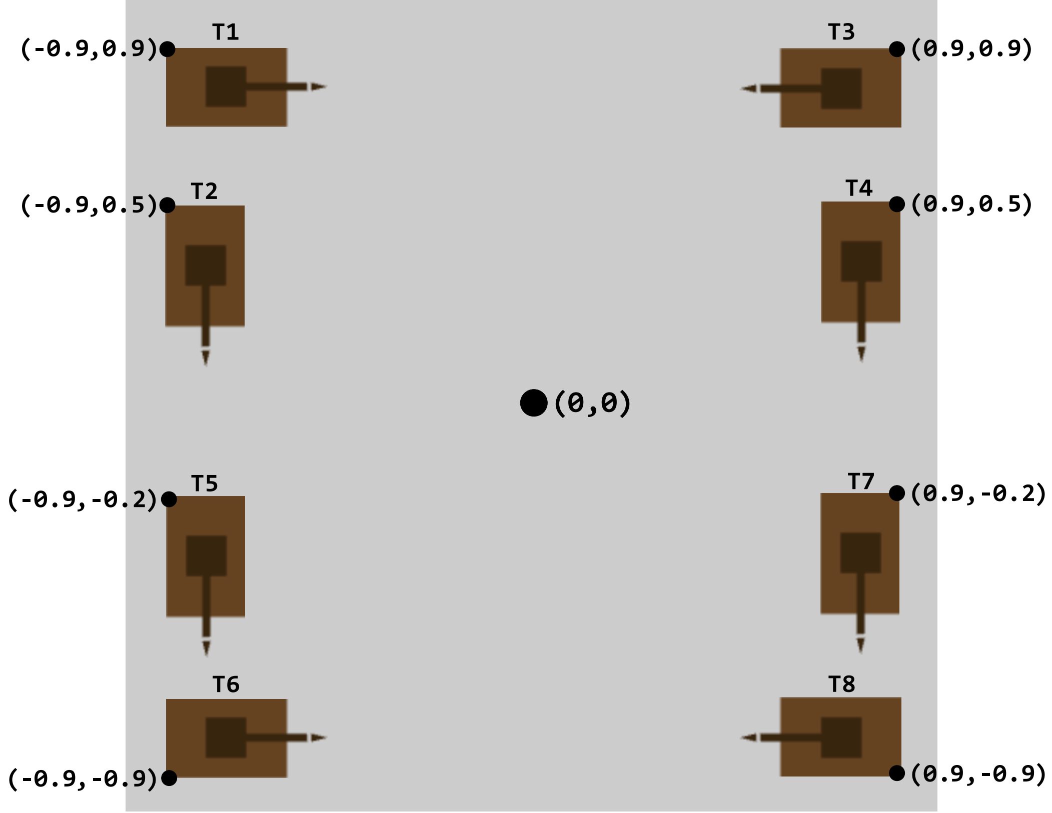


6. Deney Hazırlığı

`6_tank game_beta` klasöründe deney hazırlığı ve deney uygulaması öncesi kodları içeren basit bir Tank Oyununun başlangıç sürümü verilmiştir. Yukarıda çizilen 2D Tank modeli `TankGame.js` JavaScript kod dosyasında `vertices_hull`, `vertices_turret` ve `vertices_missile` olmak üzere üç vertex dizisi ile üretilmiştir :

```
// H1 ve H2 üçgenlerinden oluşmuş Dikdörtgen Ana Gövde: H1(V0,V1,V2) ve H2(V3,V4,V5)
var vertices_hull = new Float32Array([
//V0.x  V0.y    V1.x  V1.y     V2.x  V2.y     V3.x  V3.y     V4.x  V4.y     V5.x  V5.y
  0.10, -0.6,   0.10, -0.9,   -0.10, -0.9,    0.10, -0.6,   -0.10, -0.9,   -0.10, -0.6
]);

// Kare Üst Gövde → G1(V0,V1,V2) ve G2(V3,V4,V5) ; Namlu  → T1(V6,V7,V8) ve T2(V9,V10,V11)
var vertices_turret = new Float32Array([
//V0.x   V0.y    V1.x  V1.y     V2.x  V2.y     V3.x   V3.y     V4.x  V4.y     V5.x  V5.y
  0.05, -0.70,   0.05, -0.8,   -0.05, -0.8,    0.05, -0.70,   -0.05, -0.8,   -0.05, -0.70,

//V6.x   V6.y    V7.x  V7.y     V8.x  V8.y     V9.x  V9.y     V10.x  V10.y   V11.x  V11.y
  0.01, -0.55,   0.01, -0.8,   -0.01, -0.8,    0.01, -0.55,   -0.01, -0.8,   -0.01, -0.5
]);

// Tek üçgenden oluşan Mermi (Missile) → M(V0,V1,V2)
var vertices_missile = new Float32Array([
// V0.x   V0.y    V1.x   V1.y    V2.x  V2.y
  -0.01, -0.54,   0.01, -0.54,   0.0, -0.50
]);
```

Tankın ana gövdesinin eni `0.2`br, yüksekliği `0.3`br'dir. Kare üst gövdenin kenarları `0.1`br'dir. Namlunun eni `0.02`br, yüksekliği `0.25`br'dir.



Deney Hazırlığı olarak yukarıda Takımınızın ismine sahip Tankı çizecek şekilde `vertices_hull`, `vertices_turret` ve `vertices_missile` vertex dizilerini güncelleyiniz.

7. Deney Tasarımı ve Uygulaması

Deneyde yazacağınız ilk uygulama size verilen kodun çizdiği Tank modelinin namlusunun işaret ettiği doğrultu boyunca mermiyi yollamak olacaktır. Aslında bu uygulamaya dair kodun büyük bir kısmı zaten size verilmiştir. Sizden sadece gerekli yerleri güncellemeniz istenmektedir. Önceki sayfada çizilen Tank modelinin namlusunun dolayısıyla merminin doğrultusu `+y` ekseni boyunca yani `(0,1)` idi. O yüzden merminin doğrultusunu temsil eden değişkenler şu şekilde setlenmişti :

```
var Missile_Direction_X = 0.00   var Missile_Direction_Y = 0.02
```

Yani `x` `0`'a, `y` de `0.02` gibi pozitif bir değere setlenmişti. `1` yerine `0.02` yani çok küçük bir değer olmasının sebebi `requestAnimationFrame()` fonksiyonu ile herbir frame'de merminin `y` koordinatına bu değerin eklenmesidir. Çizim alanının `x` ve `y` koordinatları `-1..+1` arası değiştiğinden herbir adımda küçük bir mesafe gitmesi (kısa sürede ekranın dışına çıkmaması) için `0.02`'ye setlenmiştir. Deneyde yukarıdaki şekillerden biri için mermiyi ilgili doğrultu boyunca yollayacak şekilde yukarıdaki 2 değişkeni güncellemeniz istenecektir.

Mermi `'F'` (fire) tuşu ile ateşlenir. Deneyde ikinci uygulama olarak `'R'` (reload) tuşuna basıldığında mermi tekrar namlunun ucuna gelecek şekilde aşağıdaki değişkenleri `'F'` tuşuna basıldığı andaki değerlere setleyiniz:

```
var Move_Missile_X = 0.0;   var Move_Missile_Y = 0.0;
```

Deneyde yazılacak basit uygulamalardan bahsettikten sonra ‘W’, ‘A’, ‘D’ tuşlarıyla Tankın konumunun nasıl güncellendiğini anlatalım. ‘W’ tuşu ilerleme, ‘A’, ‘D’ tuşları da sırasıyla saat yönünün tersi (CCW) ve saat yönünde (CW) döndürme işlemleri içindir. İlerleme doğrultusu merminin doğrultusu ile aynıdır. Başka bir deyişle merminin doğrultusuna göre belirlenir. `render()` fonksiyonunda ilgili kodlar şöyledir:

```
let sin_t = Math.sin(theta);
let cos_t = Math.cos(theta);
let Missile_Direction_x =  Missile_Direction_X*cos_t - Missile_Direction_Y*sin_t ;
let Missile_Direction_y =  Missile_Direction_X*sin_t + Missile_Direction_Y*cos_t ;
if(MoveTank == 1)
{
    centerTank_X += Missile_Direction_x;
    centerTank_Y += Missile_Direction_y;
    moveTank_X += Missile_Direction_x;
    moveTank_Y += Missile_Direction_y;
}
```

‘W’ tuşu ile merminin doğrultusu boyunca `moveTank_X` ve `moveTank_Y` değerleri kadar ilerlenir. ‘A’, ‘D’ tuşları ile döndürme işlemi yapılırken Tank kendi ekseni etrafında dönsün diye Tankın merkez koordinatlarını tutan `centerTank_X` ve `centerTank_Y` değerleri de güncellenir. JavaScript tarafındaki bu güncellemeler `uniform` değişkenler üzerinden Vertex Shader’a şöyle aktarılır:

```
in vec4 vPosition;
uniform float vTheta;
uniform float v_centerTank_X;
uniform float v_centerTank_Y;
uniform float v_moveTank_X;
uniform float v_moveTank_Y;

void main()
{
    float temp_Position_x = vPosition.x + v_moveTank_X; // ‘W’ tuşu ile ilerleme için
    float temp_Position_y = vPosition.y + v_moveTank_Y;

    gl_Position.x = temp_Position_x - v_centerTank_X; // Döndürme öncesi Tankın merkez
    gl_Position.y = temp_Position_y - v_centerTank_Y; // konumunu ÇIKAR

    float sin_ = sin(vTheta);
    float cos_ = cos(vTheta);

    temp_Position_x = gl_Position.x * cos_ - gl_Position.y * sin_; // Döndürme işlemi
    temp_Position_y = gl_Position.x * sin_ + gl_Position.y * cos_;

    gl_Position.x = temp_Position_x + v_centerTank_X ; // Döndürme sonrası Tankın merkez
    gl_Position.y = temp_Position_y + v_centerTank_Y ; // konumunu EKLE

    gl_Position.z = 0.0;
    gl_Position.w = 1.0;
}
```

8. Deney Raporu

Deney Hazırlığı olarak **bireysel** güncellediğiniz **`TankGame.js`** JavaScript kod dosyası ile birlikte **Rapor.docx** adlı şablon belgeyi **takımınız adına** düzenleyip **deney günü akşamına kadar** (takım adına biriniz) dersin Moddle Sayfasına yükleyiniz. Yani Moodle sayfasına, Rapor takım adına, Deney Hazırlığı bireysel yüklenecektir.

KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

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.

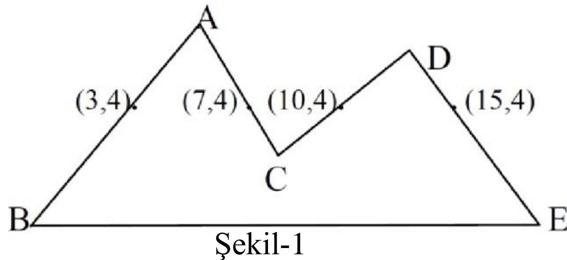
Basit şekilde bu yöntem aşağıdaki adımlardan oluşur:

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. $(y_1=y_2)$

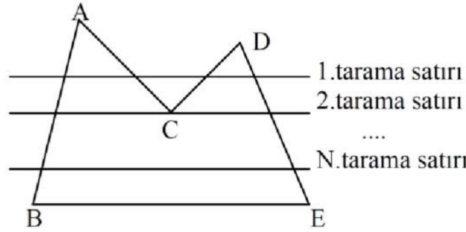


Şekil-1

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

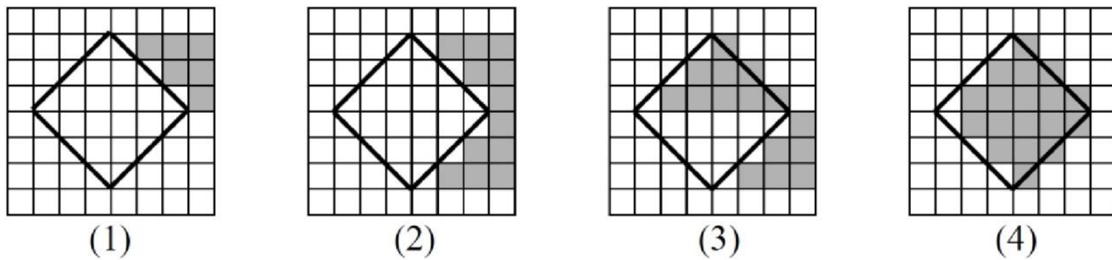
C noktası bir yerel minimum noktasıdır. Yerel minimum noktayı oluşturan kenarlar X-Y düzleminde Y ekseni boyunca artan değerlere sahiptir. Yukarıdaki probleme benzer bir problem D noktası için de vardır ve D noktası bir yerel maksimum noktasıdır. Bu durumda çözüm olarak tarama satırı boyunca yerel maksimum 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

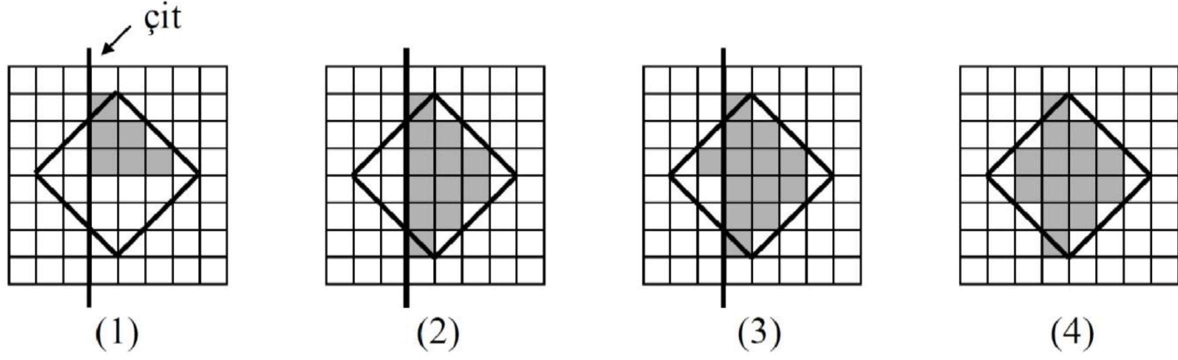
Kenar doldurma tekniğinde kesişim noktalarının tutulduğu listede sıralama ve listeyi düzenleme gibi işlemler yapmadan sadece kenarlar kullanılarak yüzey doldurma işlemi gerçekleştirilir.

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.

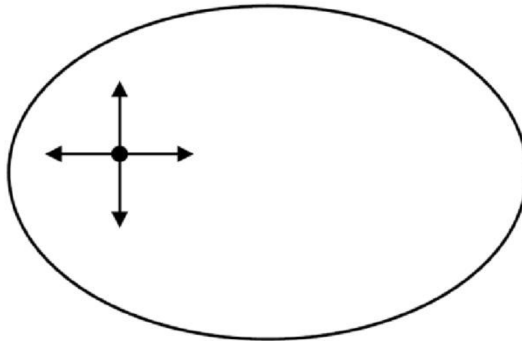


Şekil-4

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, <u>özyinelemeli fonksiyonlar</u> 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

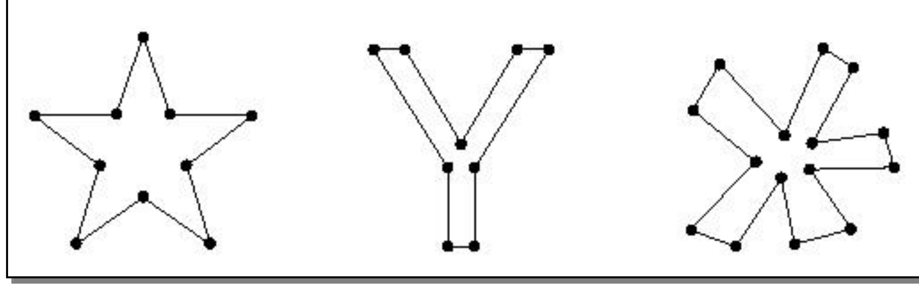
1. Deneyde DDA ve Breshenam doğru çizme yöntemlerinden de bahsedilecektir. Bu yöntemler hakkında bilgi sahibi olunuz.
2. Ekte verilen sıralı kenar liste yöntemini gerçekleyen uygulamayı (ScanLine klasörü içerisinde) çalıştırınız ve kaynak kodları inceleyiniz. (Uygulama OpenGL ile geliştirilmiştir, OpenGL'in 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

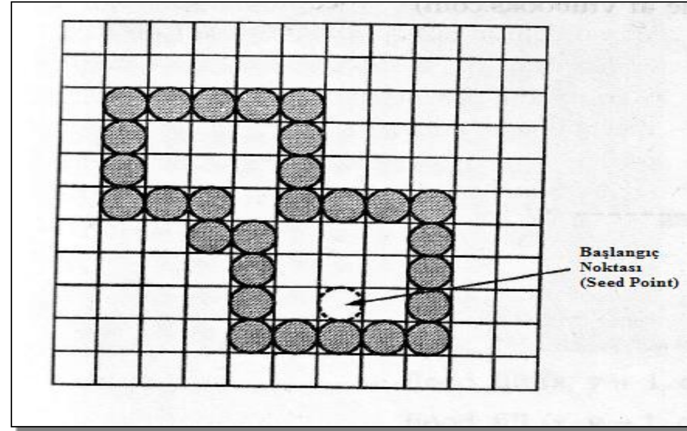
Deneyde, yüzey doldurma tekniklerinin işleyişleri tartışılacak, anlatılan yöntemlerle gerçekleştirilmiş uygulamalar incelenecektir. Bu yöntemlerde karşılaşılan ve karşılaşılabilecek problemler tartışılacaktır. Yöntemlerin hız ve bellek kullanımı yönünden üstünlükleri ve sakıncaları değerlendirilecektir.

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 Raporunu sonraki hafta deneyine kadar (takım adına biriniz) dersin Moodle sayfasına yükleyiniz.

MAYA ile 3D Modelleme

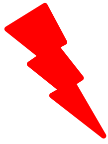
1. Giriş

3D oyunlar ve animasyonlar, günümüzde bilgisayar grafiklerinin 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.

- ✓ Deneye gelmeden önce http://www.autodesk.com/education adresinden üyelik yaptırarak MAYA 2017 öğrenci versiyonunu "Free Software" linki ile indiriniz. Aşağıda bahsi geçen üç ödevi Maya kurulu kişisel bilgisayarlarınızda yapıp deneye ödevleriniz hazır bir şekilde bilgisayarlarınızla geliniz.
- ✓ Ödevlerinizi sorunsuz bir şekilde yapabilmek için deneyin sorumlusundan deneyde anlatılan konularla ilgili videoları temin ediniz ve sorun yaşadığınız yerlerde deney sorumlusundan yardım istemekte tereddüt yaşamayınız.
- ✓ **Ödev 1:** Tabloda yer alan şekillerden kendi deney grubunuzla ilgili olanı çiziniz.

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

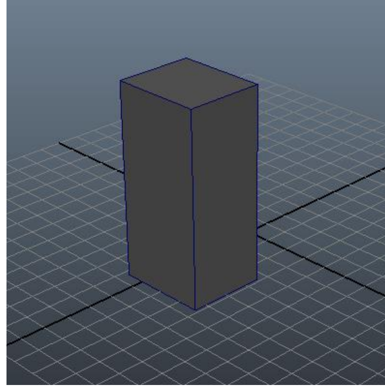
- ✓ **Ödev 2:** Bölüm 3.1'de gösterilen basit bir insan modellemesini kişisel bilgisayarınızda tüm adımları dikkate alarak yapınız.
- ✓ **Ödev 3:** Ömer Hoca'nın Bilgisayar Grafikleri Laboratuarı sayfasında yer alan önden ve yandan resimlerini, kişisel bilgisayarlarınızda front ve side viewlardaki imagePlane'lere yükleyiniz ve yüz modelini çizmeye hazır bir şekilde geliniz.



Yoklama sırasında bilgisayarında MAYA kurulu olmayan, modelleme videoları bulunmayan yada ödevlerini yapmamış olan öğrenciler deneye alınmayacak ve devamsız sayılacaklardır.

2. MAYA Ortamının Tanıtımı

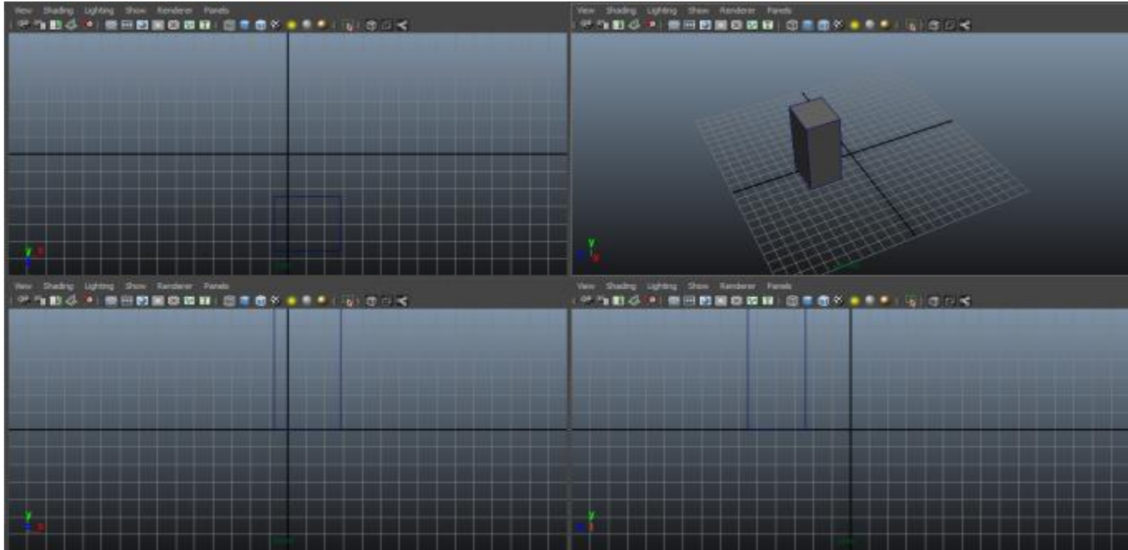
3D modellemeye başlamadan önce MAYA ortamının, sık kullanılan kısayol tuşları ve menüler ile tanıtımında fayda vardır. Şekildeki dikdörtgen prizma Polygons→PolygonCube tıklanıp fare ile çizilmiş olsun.



- ✓ Prizmaya değişik açılardan bakmak için: Klavyenin Alt tuşu sürekli basılı tutulurken farenin sol butonuna basılıp hareket ettirilir.

- ✓ Prizmaya yaklaşıp uzaklaşmak için: Farenin ortasındaki tekerlek kullanılır. Sağa-sola, yukarı-aşağı hareket etmek için Alt+scroll tuşları basılı iken fare hareket ettirilir.

- ✓ Prizmaya önden (front), yandan (side), üstten (top) ve perspektif (persp) olarak 4 farklı pencereden bakmak mümkündür. Pencereler arası geçiş için fare pencerenin üzerine getirilip space tuşuna basılır. Böylece örneğin yandan görünüş aktifken diğer görünüşlerden birine geçiş yapmak için fare o pencerenin üzerine getirilip tekrar space tuşuna basılır. 4 farklı pencere aşağıdaki şekilde gösterilmiştir.



- ✓ Prizmayı hareket ettirmek için: Prizma seçili iken move tooluna tıklanır ve fare ile oklardan çekilerek istenilen eksende hareket ettirilir.

- ✓ Prizmayı döndürmek için: Prizma seçili iken rotate tooluna tıklanır ve fare ile çemberlerden çekildiğinde istenilen eksende döndürülür.

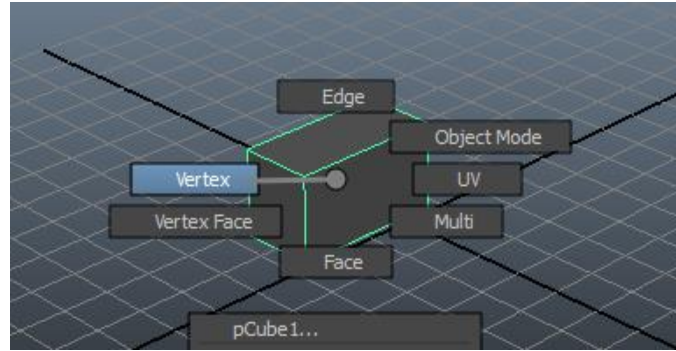
- ✓ Prizmayı ölçeklendirmek (büyütme - küçültme) için: Prizma seçili iken scale tooluna tıklanır ve fare ile küplerden çekildiğinde ölçeklenir.

- ✓ Shift tuşu ile birden fazla cisim seçilebilir.

- ✓ Yapılan işlemler Z tuşu ile geri alınabilir.

- ✓ Bir cismin poligonal (wireframe) görünümü ile boyanmış (shaded) görünümü arasında geçişler yapmak için sırasıyla 4 ve 5 tuşlarına basılır.

- ✓ Cismin üzerinde farenin sağ butonuna tıklanılırsa 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 kenarı; Vertex için o köşesi; Face için o yüzeyi ve Object Mode için cismin tamamı seçilmiş olur.



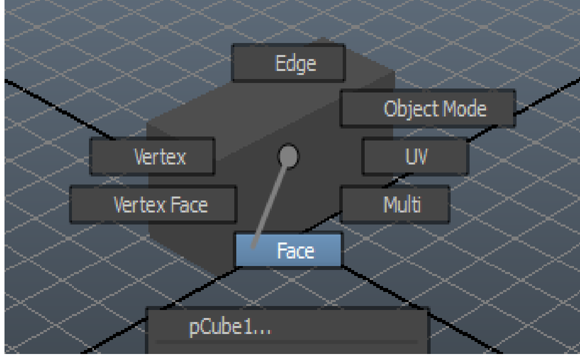
3. 3D Modellemeye Giriş

Herhangi bir cismi poligonal modellerken genellikle plane gibi basit bir şekil çizip bu şekil üzerinde değişiklikler yapılır. Değişiklik daha çok modelin karmaşıklığına bağlı olarak yeni poligonlar üretmek şeklinde gerçekleşir.

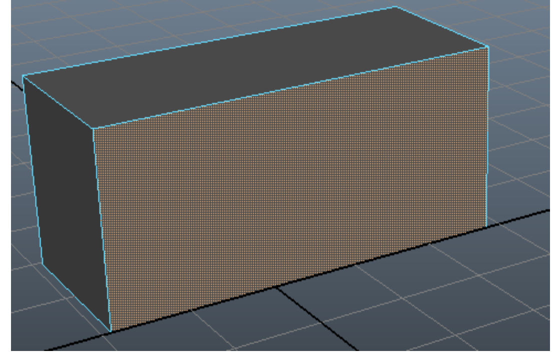
Bu bölümde Extrude, Append to Polygon, Insert Edge Loop, Target Weld, Create Polygon, Combine, Merge ve Multi-Cut toollarının kullanımları yeri geldikçe anlatılmıştır.

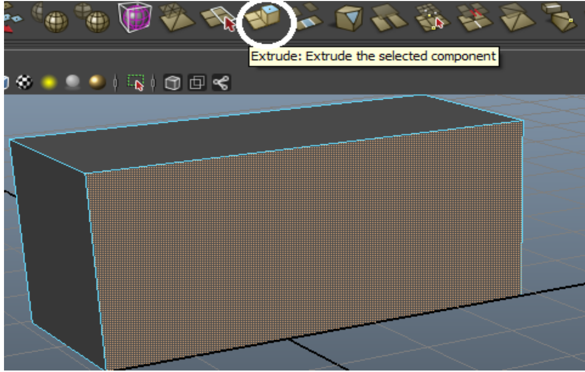
3.1. Basit Bir İnsan Modellemesi

Bu bölümde Extrude tool ile çok basit bir insan modelinin çizimi anlatılacaktır. Extrude (çekme), adından da anlaşılacağı gibi işaretlenen yüzey veya kenarın yenisini oluşturup istenilen doğrultuda çekme (uzatma) işlemidir. Bunun için öncelikle cismin üzerinde farenin sağ butonuna tıklanır ardından Edge, Vertex, Face ve Object Mode'lardan biri farenin sol butonu ile seçilir. Dikdörtgen prizmaya ait bir yüzeyin (face) extrude edilmesi aşama aşama aşağıda gösterilmiştir.

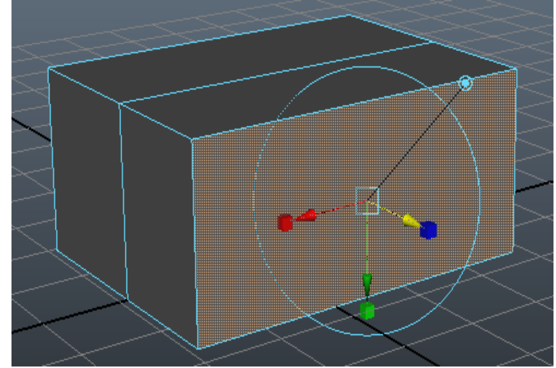


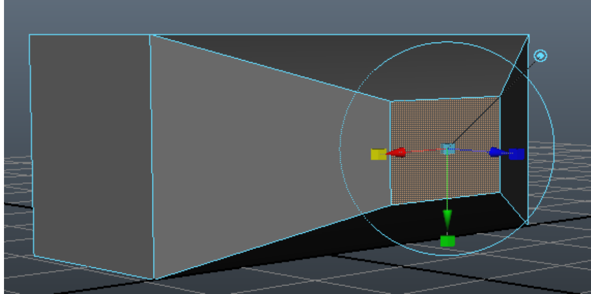
(Farenin sağ butonuna tıklanılır.)


(Sol buton ile extrude edilecek face seçilir)

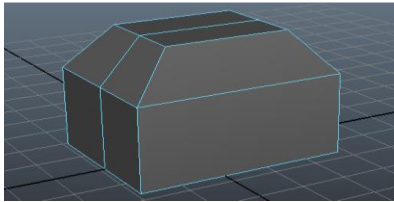


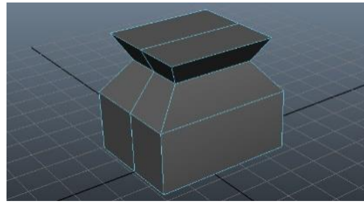
(Polygon→Extrude'a tıklanır)

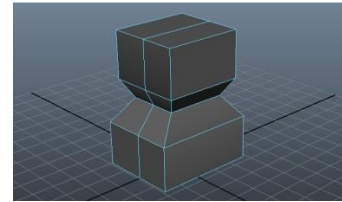

(Okun çekildiği yönde yeni face üretilir)

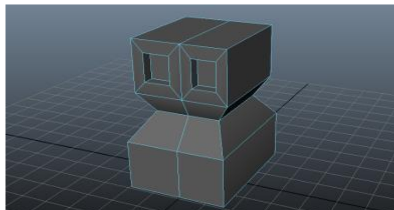

Extrude işleminde istenilen yönde yeni yüzey üretilirken ölçekleme de yapılabilir.

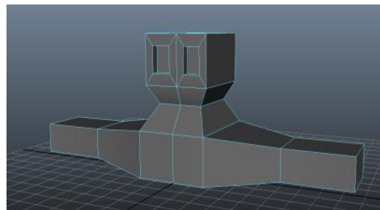
<u>Extrude işlemleri ile basit bir insan modeli çizimi aşağıda aşama aşama gösterilmiştir:</u>

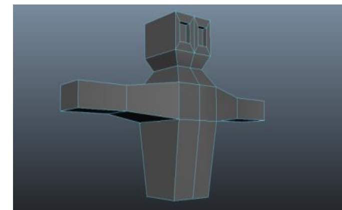

(1)

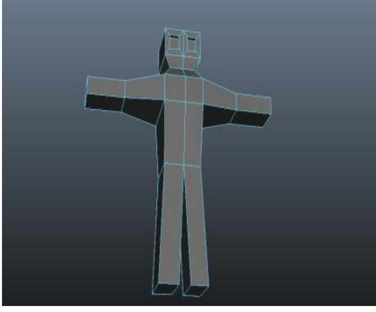

(2)


(3)

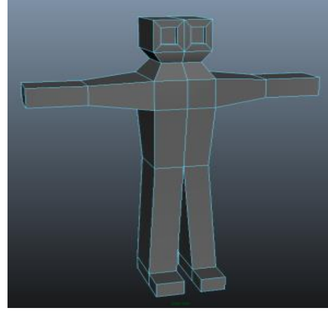

(4)


(5)


(6)



(7)



(8)

3.2. Yüz Modellemesi

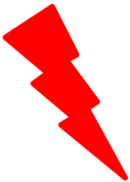
Bu bölümde, insan yüz modellemesi anlatılacaktır. Bunun için öncelikle View→Image Plane→Import Image ile front.jpg olarak kayıtlı resim front view, side.jpg olarak kayıtlı resimde side view pencerelerine yüklenir. Ardından aşağıda anlatılan yedi aşamada bu resimlere ait model elde edilir.

3.2.1 Ağız Modelinin Oluşturulması

İlk olarak front view açılır ve bir Polygon Plane çizilir. Çizilen plane top view'dan görülecek şekilde olduğu için front view'da sadece bir çizgi olarak görülür. Bunu düzeltmek için Channel Box'da pPlane sekmesinde rotate satırı güncellenir. Rotate satırında x ekseninin değeri (ilk sütun) 90.000 olarak güncellenir. Ayrıca bu plane 10*10'luk ızgara görüntüsündedir. Modelleme aşamasında bu durum bize zorluk çıkaracağı için Channel Box'da polyPlane sekmesinde Subdivisions Width ve Subdivisions Height satırları 1 değeriyle güncellenir.

Ardından Extrude ve Append to Polygon toolları yardımıyla model oluşturulur. Burada **Extrude** tool kenarları çoğaltmada, **Append to Polygon** tool da seçilen iki kenar arasına üçüncü bir kenar ekleyerek yeni bir polygon oluşturmada kullanılır.

Not: Bu aşamada ve diğer tüm aşamalarda kullanılan toollardan bazıları doğrudan Shelf Editor'de görülmekte, bazıları ise Mesh Tools sekmesinde yer almaktadır. Kullanmak istediğiniz toolu bu iki kısımda arayınız.



Modelleme aşamalarının tamamında, front view aktifken yaptığınız değişiklikleri hemen ardından side view'da güncellemeyi unutmayınız. Aksi halde tüm noktalar (Vertex) üstüste geleceği için sonradan düzenleme yapamayacak hale gelirsiniz ve üç boyutlu bir model oluşturmada başarısız olursunuz.

3.2.2 Burun ve Ağız Çevresinin Oluşturulması

Kalınan yerden front view açılıp Polygon Plane çizilerek devam edilir. Çizilen Plane'e front view aktifken 3.2.1'de anlatılan ön işlemler yapılır ve elde edilen plane burnun tam ortasına yerleştirilir.

Ardından dört kez Extrude tool ve son olarak Keep Spacing özelliği kullanılarak burun ve ağız çevresinin temel çerçevesi tamamlanır. Burada kullanılan **Keep Spacing** özelliği klavyenin W (Move işleminin kısayolu) tuşuna basılıyken farenin sol tuşuna tıklayınca ortaya çıkan menüde yer almaktadır. Bu özellik sayesinde istediğimiz kenarı, seçtiğimiz kenara paralel hale getirebiliriz.

Son olarak elimizdeki birbiriyle bağı olmayan dudak ve burun-dudak çevresi çerçevelerini birleştirmemiz gerekmektedir. Bu işlem için öncelikle farenin sağ tuşuna basılı tutarak Object Mode'u aktifleştirip bu iki çerçeveyi seçeriz. Ardından Append to Polygon toolu seçip dudak kısmının altında kalan üç kenarla kendilerine paralel olan çene kenarlarının arasına kenarlar ekleyerek kapalı hale getiririz. Fakat dudak kısmının üstünde kalan kenar sayısı az olduğu için aynı işlemi burada doğrudan yapamayız. Bunun için öncelikle Insert Edge Loop toolu kullanırız. **Insert Edge Loop** tool bir seferde birden fazla kenar ekleyerek birbirine paralel kenar sayısını artırmada kullanılır. Bu aşamada üç kez Insert Edge Loop tool kullanılarak dudak bölgesindeki, bir kez kullanılarak da burun çevresindeki kenar sayısı artırılmıştır. Son olarak Append to Polygon tool kullanılarak dudağın üst kısmından burna kadar olan iki paralel kenar, yeni kenar eklenerek kapalı hale getirilmiştir.

3.2.3 Alın ve Çene Arası Çerçevesinin Oluşturulması

Bu adımda ilk olarak beş kez ardarda Extrude tool kullanılarak çeneden başlayıp alında biten (yüzün sol taraf sınırını çizen) bir çerçeve oluşturulmuştur.

Ardından iki kez Insert Edge Loop tool kullanılarak çerçevenin çene kısmındaki kenar sayısı artırılır. Bu adımda oluşturulan çerçeveyi, 3.2.2 adımında elde edilen çerçeveyle birleştirebilmek için önce Object Mode aktifleştirilir. Ardından Target Weld tool kullanılarak bu iki çerçevenin çene kısımları birleştirilir. **Target Weld** tool seçilen iki kenarı veya köşeyi birleştirmede kullanılır. Burada Target Weld tool kullanılarak iki çerçevenin çene kısımlarındaki köşeler birleştirilmiştir. Bu sayede elimizdeki ayrık iki çerçeve arasındaki bağlantı sağlanmıştır.

3.2.4 Göz Modelinin Oluşturulması

Öncelikle en üst satırdaki Create sekmesine tıklanır ardından açılan listede NURBS Primitives → Sphere seçeneğine tıklanarak bir küre elde edilir. Bu kürenin göz boyutuna yakın olması için Channel Box'da Scale X, Scale Y ve Scale Z değerleri 1.2 olarak güncellenir. Ayrıca kürenin kutup kısmının front view'a denk düşmesi için Rotate X değeri 90 olarak güncellenir.

Ardından önce **Make the selected object live**'a, daha sonra en üst satırdaki Mesh Tools sekmesine tıklanır. Açılan listede Create Polygon'a tıklanarak kürenin ön kısmında göz şekli çizilir. Burada kullanılan **Create Polygon** toolu, rastgele konulan noktaların arasına çizgi çizerek, konulan ilk noktayla başlayıp son noktayla biten kapalı bir kenar döngüsü oluşturur. Eğer Create Polygon tool kullanılmadan önce Make the selected object live'a tıklanıp bu özellik aktif yapılmasaydı, kürenin üzerine şekil çizmek mümkün olmazdı.

Bu adımı tamamlamak için önce Make the selected object live'a tıklanarak bu özellik deaktif edilir. Ardından Object Mode aktifken küre seçilir. Küre seçiliyken sağ alt köşede bulunan Attribute Editor'de Display kısmı seçilip en sonda yer alan simgeye (**Create a new layer and assign selected objects**) tıklanır. Oluşan satır Create Polygon'la çizilen göz şeklini temsil etmektedir. Buyüzden ismi eyeTemp_lyr olarak değiştirilip kaydedilir. EyeTemp_lyr adlı satırda **V** kısmına tıklandığında küre kaybolur, sadece Create Polygon'la çizilen göz şekli

kalır. Son olarak bu yüzey Face aktifken seçilir ve Extrude toolla oluştululan kopyası büyütülerek elde edilen yeni yüzey, göz şeklinin tam dışını saracak şekilde ayarlanır, ortada kalan yüzey (ilk çizilen yüzey) silinerek göz oyuğunun oluşturulmasıyla göz modellemesi tamamlanmış olur.

3.2.5 Burun ve Göz Çevresinin Oluşturulması

Bu adımda ilk olarak ardarda dört kez Extrude tool kullanılarak burun üzerindeki kenarlar alna kadar çoğaltılır. Ardından Insert Edge Loop tool kullanılarak gözün tam orta kısmının burna bakan tarafında bulunan yüzey, göz merkezinden geçen yatay çizginin böldüğü iki yüzeye dönüştürülür. Burada polygonların çoğaltılmasının sebebi, bu iki çerçeveyi daha kolay ve doğru birleştirebilmektir.

Daha sonra Object Mode aktifken bu iki çerçeve seçilir ve Combine toola tıklanır. Burada kullanılan **Combine** tool, seçilen polygonları tek bir polygonmuş gibi birlikte işleme sokabilmemizi sağlar. Ardından Append to Polygon tool kullanılarak burnun üst kısmıyla göz arasında kalan üç kenar birleştirilir. Son olarak Insert Edge Loop tool kullanılarak son aşamada elde edilen üç polygonu yukardan aşağıya doğru bölen kenar eklenir.

Bu adımı tamamlamak için önce Insert Edge Loop tool kullanılarak burundan yanağa doğru inen polygon ikiye bölünür. Ardından bu iki parça, Append to Polygon tool kullanılarak bir önceki adımda elde edilen kısma birleştirilir. Oluşan iki büyük polygon ve sol taraflarındaki polygon, Insert Edge Loop tool kullanılarak yatay yönde ikiye bölünür. Bu kısımla göz arasında kalan üçgenin kapatılması için Merge tool kullanılır. **Merge** tool, iki köşeyi bağlı oldukları kenarlarla birlikte birleştirmeye yarar. Bu şekilde göz çerçevesiyle 3.2.3 adımında elde edilen çerçeve birleştirilmiş olur.

3.2.6 Yanak Çerçevesinin Oluşturulması

İlk olarak dudağın üst kısmında bulunan bir kenar silinir ve arkada kalan kenar Extrude tool kullanılarak çoğaltılır. Oluşan yeni kenarın köşelerinden aşağıda olanı Merge tool kullanılarak birleştirilir. Ardından burnun solundaki iki kenar Extrude tool kullanılarak çoğaltılır. Oluşan kenarın üst köşesi gözün altındaki bir köşeyle, alt köşesi yanağın üstündeki bir köşeyle birleştirilir. Burnun üzerindeki bir kenar silinerek kalan alana iki kere ardarda Insert Edge Loop tool uygulanarak daha küçük iki alan elde edilir. Son olarak burnun üzerinde yukarıdan aşağıya kadar olan soldan üçüncü çizgiye ait tüm kenarlar silinir.

Yanak kısmını doldurmak için burnun sağ tarafında kalan dört kenar Extrude Tool kullanılarak çoğaltılır ve oluşan kenarlara ait üstte kalan iki köşe gözün altında kalan iki köşeyle Merge tool kullanılarak birleştirilir. Ardından yanağın altıyla çenenin üstünü birleştirebilmek için çene bölgesindeki kenar sayısı iki kez Insert Edge Loop tool kullanılarak, yanağın altındaki kenarda bir kez Extrude tool kullanılarak çoğaltılır. Üstte kalan iki köşe Merge tool, altta kalan iki kenar da iki kez Append to Polygon tool kullanılarak birleştirilir ve ikinci Append to Polygon tool kullanımında elde edilen üçgen alana ait bir kenar silinir.

Son olarak önce Insert Edge Loop tool kullanılarak çene üzerindeki yatay kenar sayısının artırılması, sonra Multi-Cut tool kullanılarak bir önceki aşamada kenarın silinmesiyle ortaya çıkan büyük yüzeyin yatay olarak ikiye bölünmesiyle çene bölgesinin modellemesi tamamlanmış olur. Burada kullanılan **Multi-Cut** tool, seçilen iki nokta arasına bir kenar çizilmesini sağlar.

3.2.7 Alın ve Yanak Çevresinin Tamamlanması

Bu adımda ilk olarak burnun en üstündeki kenar Extrude tool kullanılarak çoğaltılır ve alnın başlangıcına yakın bir yere Insert Edge Loop tool kullanılarak yeni kenar eklenir. Extrude toolla oluşturulan kenarın sol köşesi alnın başlangıç kenarının alt köşesine, sağ köşesi de Insert Edge Loop toolla eklenen kenarın alt köşesine Target Weld tool kullanılarak birleştirilir.

Ardından tekrar Insert Edge Loop tool kullanılarak kaş çizgisinin ortasındaki kenar çoğaltılır. Oluşan bu kenar Append to Polygon tool kullanılarak göz çevresindeki kenara birleştirilir. Daha sonra birleştirilen bu kenardan yanağın üstüne kadar olan sekiz kenar tek seferde Extrude tool kullanılarak çoğaltılır ve en altta kalan kenarın köşesi yanındaki köşeye Target Weld tool kullanılarak birleştirilir. En üstte kalan kenarın köşesini birleştirebilmek için, Insert Edge Loop tool kullanılarak alna giden polygona yeni kenar eklenir.

Son olarak alnın ortasındaki polygon ardarda üç kez kullanılan Insert Edge Loop toolla dört polygona, kulağın önünde kalan polygonda Insert Edge Loop toolla iki polygona bölünür. Ardından Append to Polygon tool kullanılarak yanak bölgesindeki üç bölge birleştirilir. Benzer işlemler önce alnın sağ kısmı ve gözün sağ üst kısmı arasında sonra alnın üst kısmı ve gözün üst kısmı arasında tekrarlanır. Insert Edge Loop toolu birçok kez ardarda kullanarak alından çeneye kadar olan bölgedeki polygonların küçük polygonlara bölünmesiyle modelleme işlemi tamamlanmış olur.

4. Deney Tasarımı ve Uygulaması

- ✓ Deney föyünde ve videolarında anlatılan aşamaları dikkate alarak Ömer Hoca'nın yüzünü modelleyiniz.

5. Deney Raporu

- ✓ Grubunuzdan bir kişinin yüzünü modelleyiniz ve elde ettiğiniz yüz modelinin ayrı ayrı üç ekrandan alınmış ekran görüntülerini çıktı alarak getiriniz.

Not → Deney Raporunu sonraki hafta deneyine kadar (takım adına biriniz) dersin Moodle sayfasına yükleyiniz.

KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

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.

- ✓ Deneye gelmeden önce http://www.autodesk.com/education adresinden, öğrenci olduğunuzu belgeleyip, üyelik yaptırarak MAYA 2022 versiyonunu indiriniz. Aşağıda bahsi geçen üç animasyonu Maya kurulu kişisel bilgisayarlarınızda yapıp, deneye **mutlaka** maya kurulu bilgisayarlarınızla geliniz. Deneye başlamadan önce herkesin ödevleri kontrol edilecek, deneye hazırlık puanının tamamıyla deney yapılış puanının yarısı bireysel olarak bu ödevlerden verilecektir. Bu sebeple laboratuvara gittiğinizde üç ödevi de kişisel bilgisayarlarınızda açıp hazır bekleyiniz.

- ✓ **Ödev 1:** Bölüm 2.1'de anlatılan topun zıplaması animasyonunu, deformasyon ve dönme efektlerini de katarak yapınız. Bu esnada Graph Editordeki eğrilerde değişiklikler yaparak etkilerini gözlemleyiniz. Bu ödev hazırlık puanının yarısını oluşturur.

- ✓ **Ödev 2:** https://www.ktu.edu.tr/bilgisayar-bilgisayargrafiklerilaboratuari sayfasında eğimli bir zeminde küpün yuvarlanması animasyonunun videosu konmuştur. Bu animasyonu yaparken küpe gravity özelliği vermeyiniz. Bu ödev hazırlık puanının yarısını oluşturur. (Gravity ile yapanlar puan alamayacaktırlar.)

- ✓ **Ödev 3:** Bölüm 3.1'de anlatılan domino taşlarının devrilmesi animasyonunu gerçekleştiriniz. Bu animasyonu yaparken **FX** arayüzünü kullanınız. Bu ödev deney yapılışı puanının yarısını oluşturur.

2. Keyframe ve Graph Editor

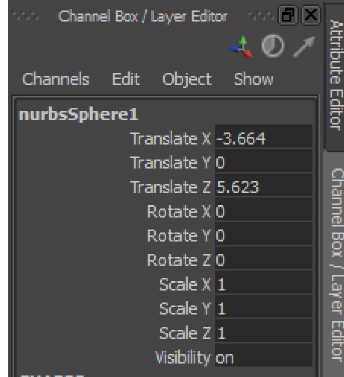
Geliştireceğimiz animasyonu bir video olarak düşünelim. Bilindiği gibi videolar resimlerden oluşmaktadır. Bu resimlere **frame** denir. Animasyonlar geliştirilirken bazı framelerde cisimler üzerinde değişiklikler yapılır ve bu frameler setlenir. Bu framelere **keyframe** denir.

Bir keyframeden diğerine geçerken cismin de bir konumdan diğerine gittiğini varsayalım. Bu hareketin, iki keyframe arasında kalan frameleri nasıl etkilediğini belirlemek için **Graph Editor** kullanılır. Örneğin bir topun havaya atılması animasyonunu yaparken, topun ilk konumu ve havada en yüksek noktaya ulaştığı andaki konumu keyframe olarak setlenmiş olsun. Bu iki nokta arasında topun hızının nasıl değişeceği Grap Editorde kullanılan eğri (Curve) ile belirlenir.

2.1. Topun Zıplaması Animasyonu

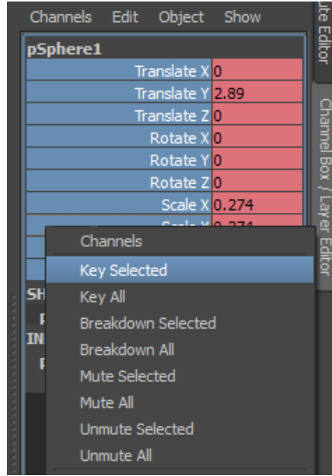
Bu bölümde, animasyon geliştirilirken kullanılan keyframe setleme ve hareketin Graph Editor ile belirlenmesi işlemleri, bir topun zıplaması animasyonu ile anlatılacaktır.

Bunun için öncelikle bir **küre** çizelim. Bu küreyi top olarak düşünürsek topun zıplaması animasyonu yapılırken topun 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.



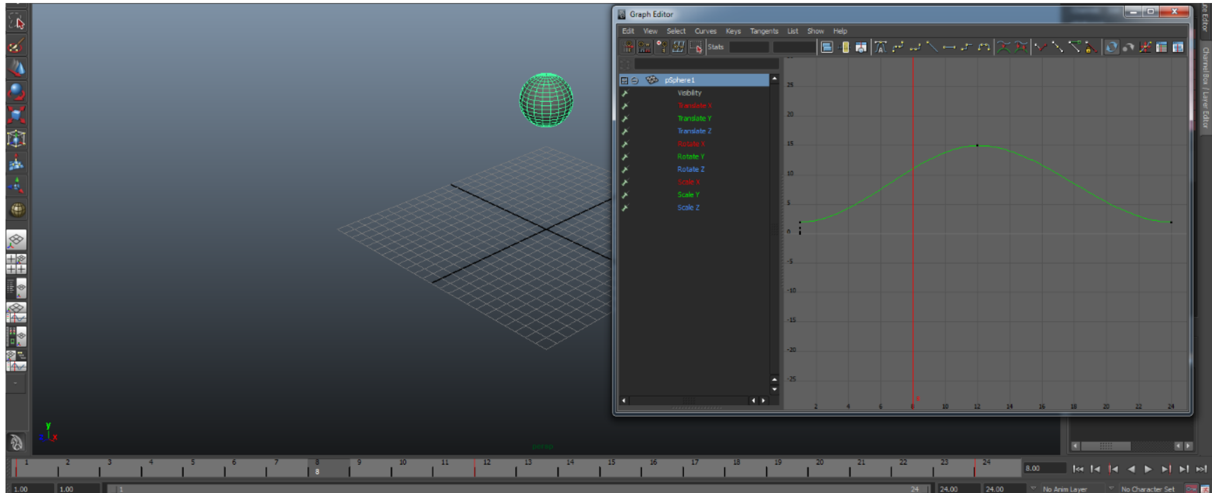
MAYA penceresinin aşağısına yakın **Time Slider** kısmında animasyon yaparken setleyeceğimiz frameler görülmektedir. Time slider'da default olarak 24 frame olduğu görülmektedir. Time slider'ın hemen altında 1...24 şeklinde **Range Slider** barı görülmektedir. Onun sağında da 24.00 ve 48.00 yazıyor. 24 sayısı yukarıda da söylendiği gibi time slider'da 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 fare ile tutulup çekilebilir.

Yukarıda da bahsedildiği gibi küre üzerindeki öteleme, dönme ve ölçekleme işlemleri için Channel Box kullanılacaktır. Channel Box'da yapılan değişikliklerin key frame olarak setlenebilmesi için, farenin sağ butonuyla Translate, Rotate ve Scale özellikleri seçilir ve sol butonuyla **Key Selected** yapılır.

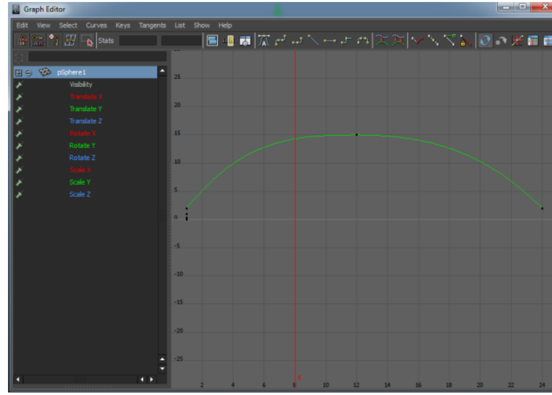


Böylece animasyon için gerekli hazırlıklar tamamlanmış oldu. Geliştirilecek animasyonda hem range slider'ı hem de toplam frame sayısını 24 olarak belirleyelim (24.00 24.00). Time slider'da herhangi bir frame'e, mesela 12. frame'e, tıklayıp Channel Box'tan translation değerini 15 olarak, sonra son frame olan 24. frame'e tıklayıp translation değerini 0 olarak değiştirip bu frame'leri keyframe olarak setlediğimizde zıplama animasyonu tamamlanmış olur. Animasyonu izlemek için Play butonuna tıklanır.

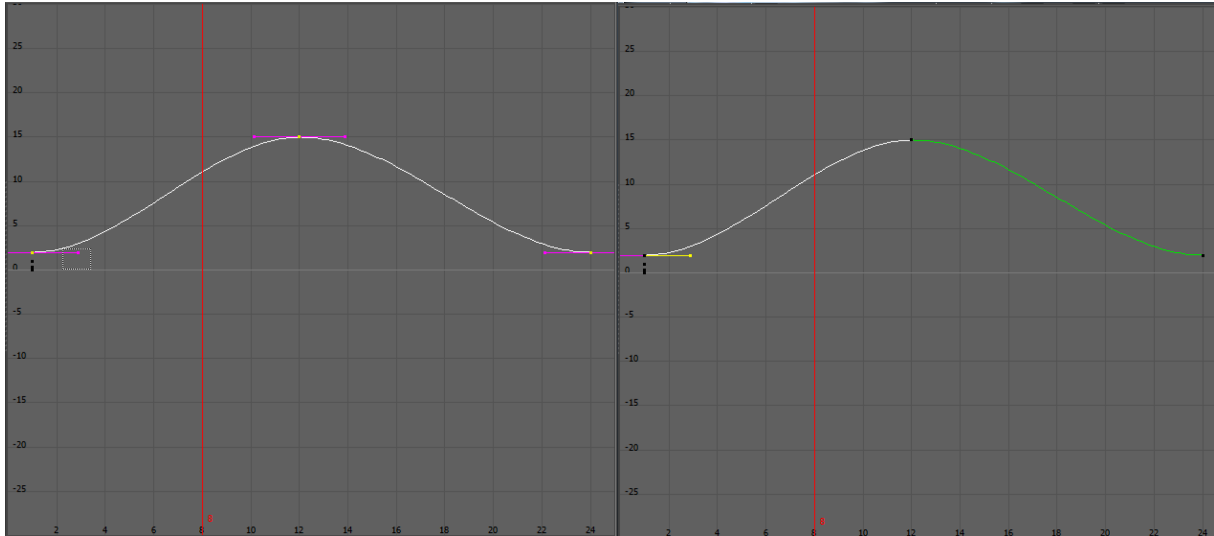
Bahsedildiği gibi animayon için setlenen key frameler arasında cismin nasıl hareket edeceğine Graph Editor'deki eğrilerle karar verilir. Windows→Animation Editors→Graph Editor ile Graph Editor açılabilir. Time slider mouse ile ilerletilirken hem top hareket eder hem de Graph Editor'deki eğri üzerinde düşey bir çizgi ile o anda eğrinin neresinde olunduğu görülür. Aşağıdaki ekran görüntüsü Time slider 8.frame'de iken alındığı için Graph Editor'deki düşey çizgi 8.frame üzerindedir.



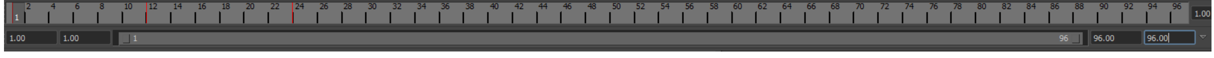
Ayrıca eğrinin tepe noktası da 15’i göstermektedir. Çünkü cismin Y ekseni boyunca yükseldiği tepe noktası değeri 15’tir. Bu noktaya çıkarken hızın değişimini **eğrinin eğimi** belirlemektedir. Dolayısıyla 0..4 arası framelerde eğim düşük olduğundan hız düşük, 4..8 arası hızda doğrusal yakın bir artış var ve 8..12 arası yine eğim giderek azalmakta ve tepe noktasında eğim sıfır olduğundan top durmaktadır. Anlatılan hız değişimi, gerçek bir zıplama hareketini temsil etmemektedir. Normalde aşağıda gösterildiği gibi 0..4 arası top yerden yüksek hızla zıplamalı, bu yüzden eğim yüksek olmalı, ardından eğim sürekli azalmalıdır.



Animasyonumuzdaki eğriyi yukarıdaki şekle getirmek için öncelikle fare ile eğriye tıklanır. Eğri üzerinde **tangent** adı verilen 3 tane yatay çizginin ortaya çıktığı görülür. Eğri üzerindeki değişiklikler bu çizgiler yardımıyla yapılır. Bunun için ilgili tangent fare ile seçilir. Sonra klavyeden W tuşuna bir kez basılır ve farenin orta tuşuna (tekerlek) **basılı** tutularak eğri üzerinde istenilen değişiklik yapılır.

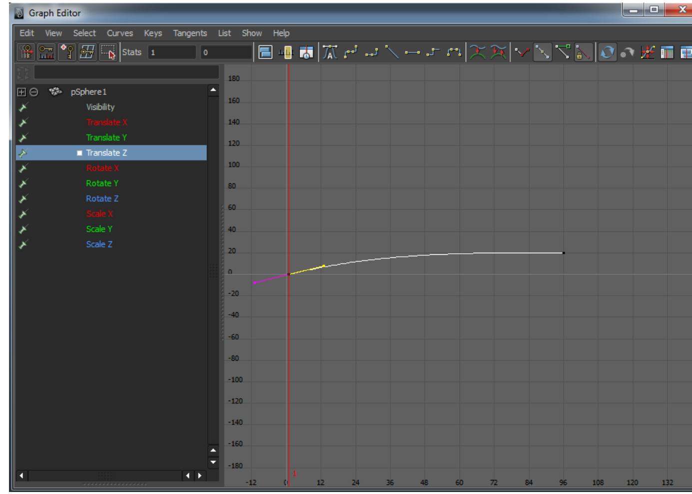


Geliştirilen animasyonda top sadece birkez zıplamaktadır. Birkaç kez mesela 4 kez zıplayıp sonra durması için frame sayısını 24x4=96 yapalım.



Yeni keyframeler setleyerek topun yüksekliklerini 36.frame’de 12, 48.frame’de 2, 60.frame’de 8, 72.frame’de 2, 84.frame’de 3 ve 96.frame’de 2 yapalım. Burada 0 yerine 2 yapılmasının nedeni kürenin yarıçapının 2 olmasıdır. Yani top zıplarken tamamı grid’in üzerinde olsun, alt kısmı grid’in içine girmesin diye böyle setlenmiştir.

Dikkat edilirse topun hareketi yalnız Y eksenindedir. Topun hem zıplaması hem de ileri hareket etmesi için time slider’da 96.frame’e gidip Channel Box’ta bu sefer Translate Z key selected yapılır ve değer olarak 20’ye setlenir. Grap Editor açılıp orada da Translate Z seçilip eğri üzerinde gerekli değişiklikler yapıldığında, Z ekseninde de hareket sağlanmış olur.



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

3. FX Tool

Dinamik, genel anlamda fizik biliminin nesne hareketleriyle ilgilenen parçasıdır. Maya'nın FX tool'u (önceki sürümlerde dynamic isimli tool), animasyonlardaki fiziksel güçleri simule etmek için çeşitli fizik yasalarını dikkate alır. Geliştirici keyframe'ler setlemeden sadece nesnenin hangi fiziksel kuvvetler karşısından etki görmesini istiyorsa bunları ayarlar ve süreç otomatik gerçekleşir.

FX animasyon bileşeni, geliştiricinin keyframe setleyerek çok zor ve uğraştırıcı bir süreç sonrası ulaşabileceği gerçekçi sahneleri, kolayıkla gerçekleştirmesine imkân tanır. Okyanus veya bayrak dalgalanması, patlama efektleri bunlardan bazılarıdır. FX bileşeni kendi içinde çeşitli araçlara sahiptir.

Fields/Solvers: Simülasyonları gerçekliğe ve doğallığa yaklaşılır. Örneğin akışkan, bulut veya saç hareketleri çeşitli field'larla birlikte kullanılarak sağlanabilir. Maya'da kullanıma sunulan field çeşitlerinden bazıları 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.

Rigid Body: Rigid body polygonal ya da nurbs yüzeyleri sert bir yüzeye çevirmek için uygulanır. Surface'lerden farklı olarak rigid body'ler animasyon süresince birbirleri ile hareket eder ve gerekli ayarlar yapılırsa sahnedeki kuvvetlerin etkisi altında kalabilirler.

Maya aktif ve pasif olmak üzere iki tip rigid body'ye sahiptir. Aktif body olarak tanımlanan cisimler, ortamın yerçekiminden, rüzgarından ya da çakışma gibi dinamikliğinden etkilenirler ve key setlemeleri yapılmasına izin vermezler. Pasif body olarak tanımlanan cisimler ise, aktif olarak setlenen cisimlere etki etmek için oluşturulurlar ve key setlemelerine uygundurlar. Öteleme, dönme ve ölçekleme işlemlerine izin verirler ama dinamik etkilerden etkilenmezler.

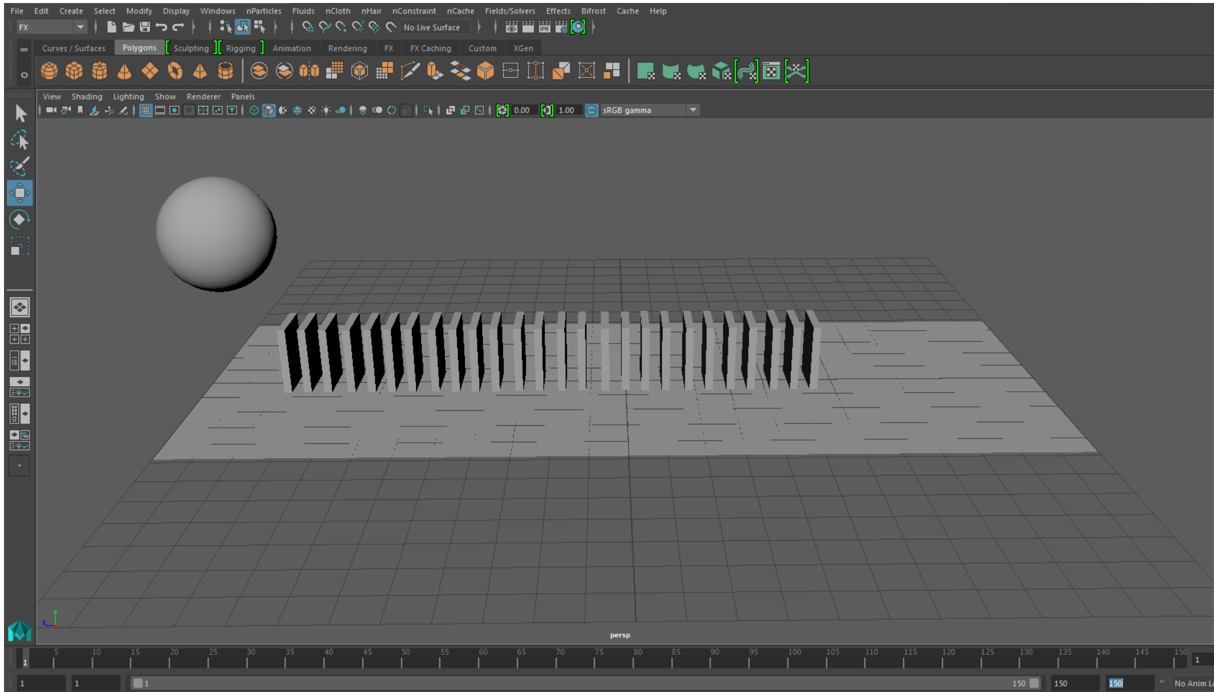
Bu iki body türüne örnek vermek için topun bir zeminden sekmesini kullanabiliriz. Top aktif body olarak tanımlanmalıdır. Çünkü yer çekiminden etkilenerek yere düşmeli ve daha sonra yere çarpmanın etkisi ile yerden yükselmelidir. Ancak zemin pasif olarak setlenmelidir. Çünkü top sektiğinde zemin yerinden oynamamalıdır ve zeminde herhangi bir bozulma olmamalıdır. Rigid body'nin dinamik animasyonunun kontrolü rigid body solver denilen maya componenti ile yapılır.

3.1. Domino Taşlarının Devrilmesi

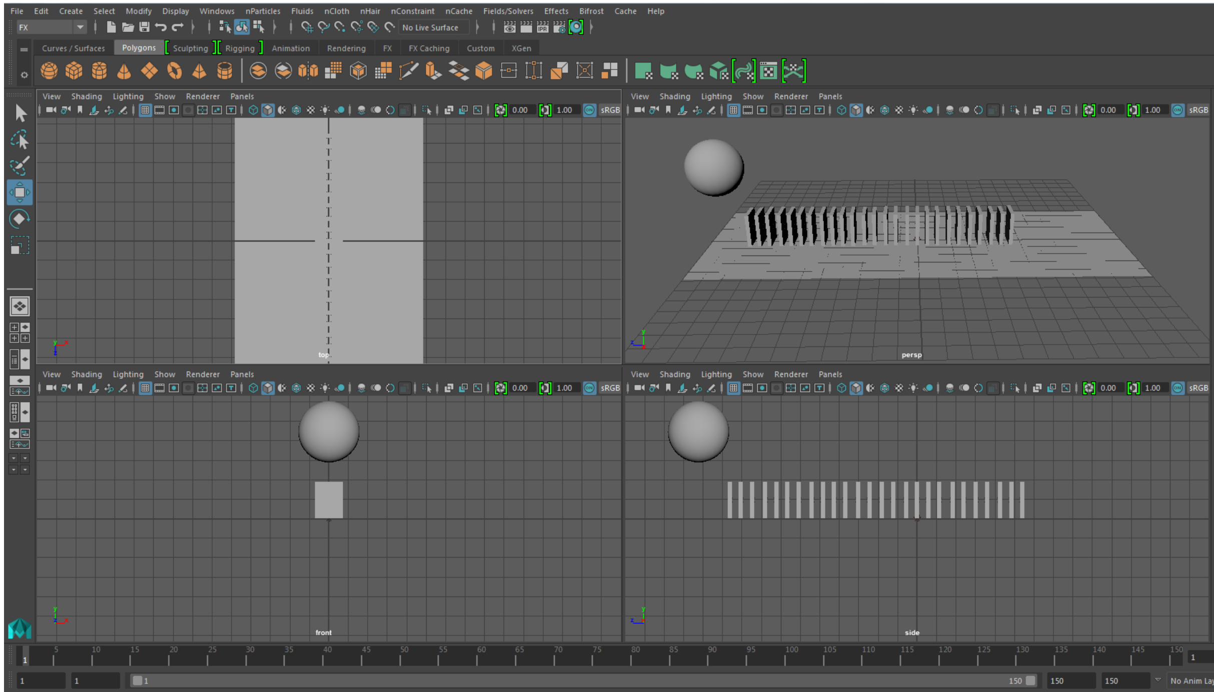
Domino taşlarının Dynamic Tool ile devrilmesi animasyonunda öncelikli olarak poligonlardan yararlanılarak bir zemin, devrilecek olan taşlar ve bu taşları devirecek bir top oluşturulur. Bunun için;

- 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.



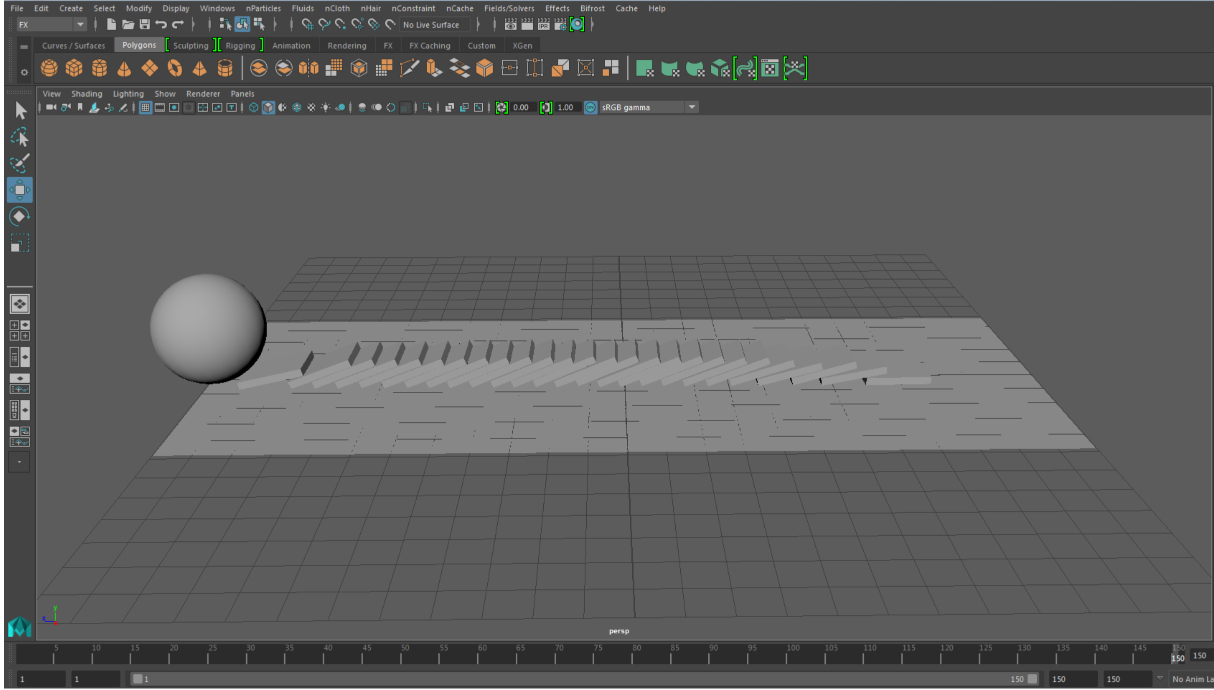
Domino taşlarının oluşturulmasında; bir Polygon Cube oluşturulduktan sonra diğer taşlar, oluşturulan Polygon Cube çoğaltılarak üretilir. Burada oluşturulan topun yukarıdan bırakıldığında ilk domino taşına çarpacak biçimde yerleştirilmiş olmasına dikkat edilmelidir.



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

- Zemin seçili iken ; **FX→Field/Solvers→Create Passive Rigid Body** özelliği,
- Domino taşlarının hepsi seçili iken (Shift tuşuna basılı tutularak fare ile tıklanıp) ; **FX→Field/Solvers→Create Active Rigid Body** özelliği,
- Top seçili iken (Shift tuşuna basılı tutularak fare ile tıklanıp) ; **FX→Field/Solvers→Create Active Rigid Body** özelliği eklenir.

Topun yukarıdan aşağıya, domino taşlarının da birbirlerine çarpıp düşebilmesi için, top ve taşların tamamı seçilir ve **FX→Field/Solvers→Gravity** özelliği kazandırılır.



4. Deney Tasarımı ve Uygulaması

- Domino taşlarının devrilmesi simülasyonunu bölüm 3.1’de anlatıldığı şekilde yapınız. Ardından topa gravity özelliği verme yerine topa zıplama animasyonu vererek aynı simülasyonu gerçekleştiriniz.
- Deney sorumlusunun belirttiği oyunun animasyonunu gerçekleştiriniz.

5. Deney Raporu

Deney sırasında geliştirmeye başladığınız oyun animasyonunu tamamlayınız. Oluşturduğunuz animasyonun önemli gördüğünüz yerlerinde ekran görüntüsü alınız. Ekran görüntülerini düzgün sırada içeren bir dokümanı **Takım_No.pdf** ismiyle kaydedip Deney Raporu olarak sonraki hafta deneyine kadar dersin Moodle sayfasına yükleyiniz.

KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

Ters Perspektif Dönüşüm ile Doku Kaplama

1. Giriş

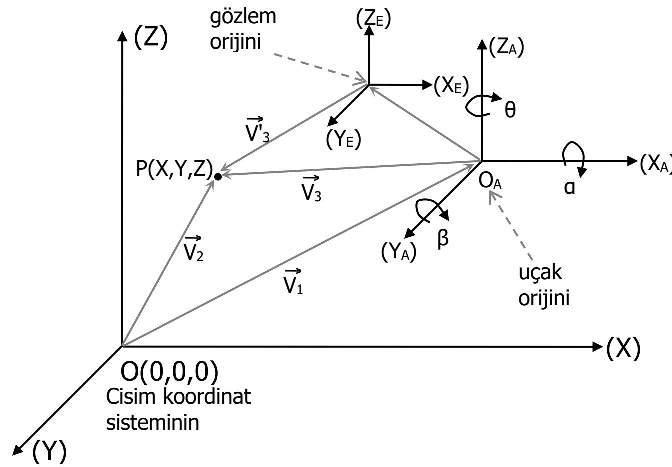
Bu deneyde, genel hatları ile paralel ve perspektif izdüşüm teknikleri, ters perspektif dönüşüm ile doku kaplama ve örtüşme (aliasing) problemi üzerinde durulacaktır. Deney sonunda öğrencilerin, gerekli matematiksel ve geometriksel bilgileri edinmeleri ve bu tür programlar geliştirebilmeleri amaçlanmaktadır.

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:

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

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.

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

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.

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

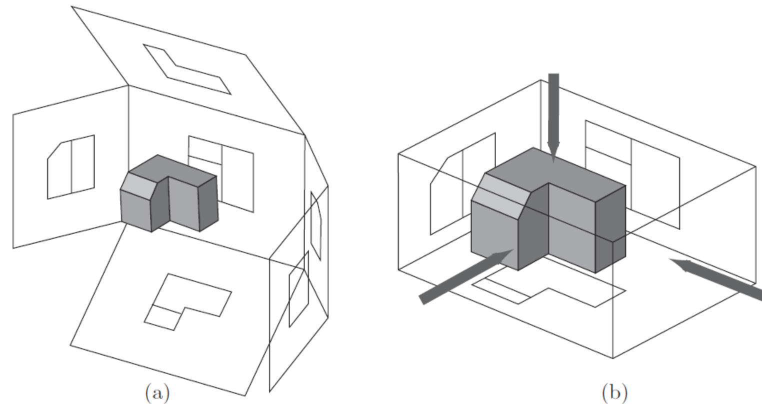
3. Paralel İzdüşüm Teknikleri

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 $\Phi(P)$ 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.

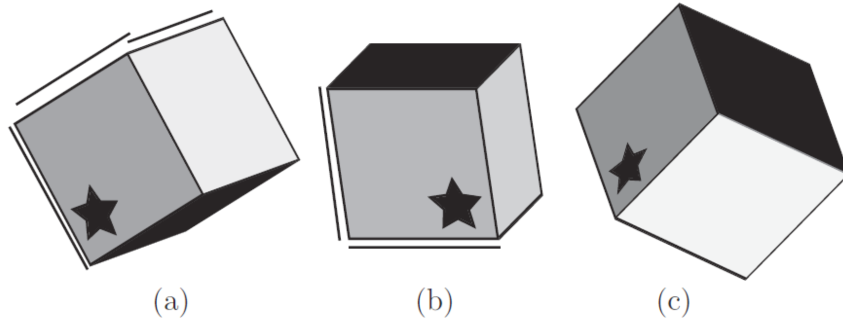
Eğer çevreleyen kutunun bir kenarı xy düzlemi ise izdüşüm işlemi sadece z bileşeninin silinmesiyle gerçekleştirilebilir Benzer şekilde, diğer eksenler silinerek farklı izdüşümler de elde edilebilir.

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.

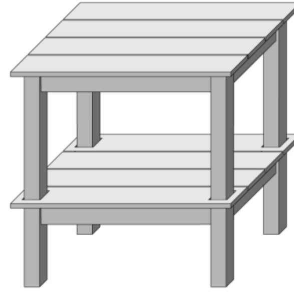


Şekil-4 Aksonometrik izdüşüm örnekleri (a) trimetric (b) dimetric (c) isometric.

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.

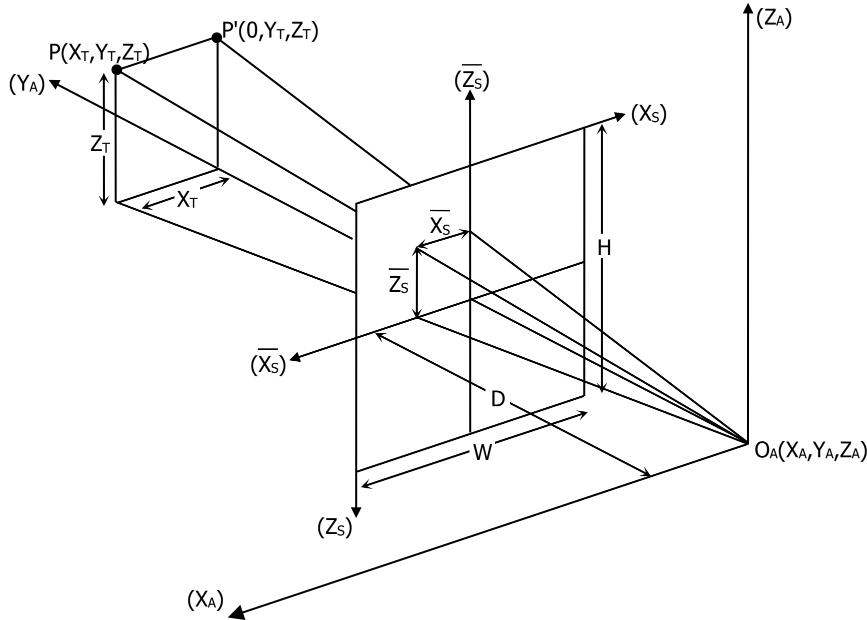


Şekil-5 Eğik izdüşüme bir örnek.

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.

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

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.

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

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.

$$\vec{V}_3^{\,r} = K * P^*, \quad P^* = \begin{bmatrix} \vec{X}_S \\ D \\ \vec{Z}_S \end{bmatrix}$$

$$\vec{V}_2 = \vec{V}_1 + [\beta]^{-1}[\alpha]^{-1}[\theta]^{-1} K * P^*$$

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

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:

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

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

6. Doku Kaplama (Texture Mapping)

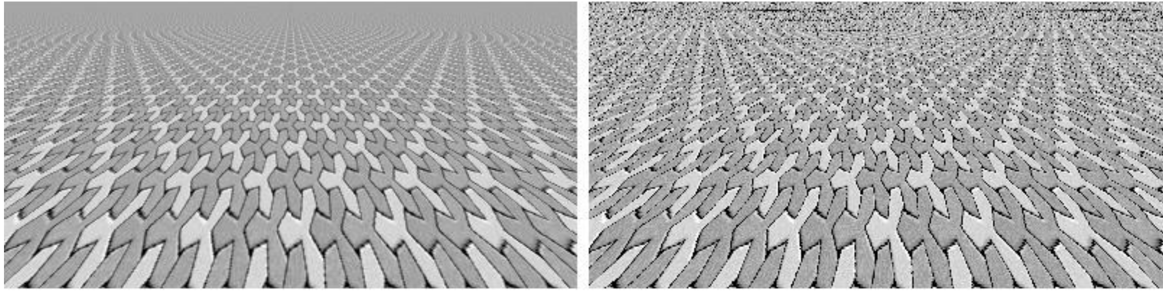
Bilgisayar grafiklerinde bir cismin yüzey ayrıntıları doku olarak adlandırılır. Tuğlalardan örülmüş büyük bir duvara yeterli uzaklıktan baktığımız zaman duvardaki her bir tuğlayı bir doku elemanı olarak düşünebiliriz. Ama bu duvara yakından bakıldığında tuğlalar artık ayrı cisimlerdir ve cisim üretme 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.

DirectX ile Tank Oyunu

1. Giriş

Oyunlar, Bilgisayar Grafikleri'nin en popüler uygulama alanlarından biridir. Bu deneyde DirectX 12 API ile basit bir 3D Tank Oyununun nasıl geliştirilebileceğinden bahsedilecektir.

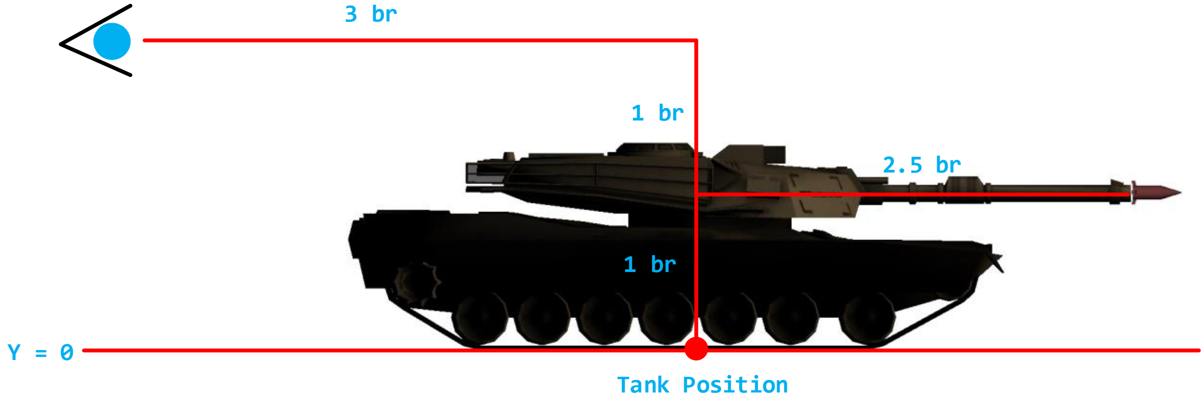
2. Tank Oyunu Fonksiyonları ve Görevleri

Genel olarak oyunlar 3 temel fonksiyondan oluşurlar:

- **`OnInit()`** : Oyuncular ve oyun ortamı bufferlara (vertex, index, texture) yüklenir. Oyunla ilgili bazı matris ve vektörler ilk değerlerine setlenir.
- **`OnUpdate()`** : Klavye/mouse etkileşimlerine göre oyuncuların konumları güncellenir. Oyuncuların birbirleriyle (veya ortamla) olan etkileşimlerine göre örneğin birbirlerine ateş etmişlerse (veya duvara yaklaşmışlarsa) isabet (kesişim) testleri yapılır.
- **`OnRender()`** : Oyuncular güncel konumlarına çizilir veya vurulan oyuncu artık çizilmez.

`OnInit()`, uygulama çalıştırıldığında ilk koşan ve bir kere koşan fonksiyondur. Dolayısıyla oyuncuların, oyun ortamının yüklenmesi gibi uygulamada bir kez yapılacak işlemlere dair kodlar burada yazılmalıdır. Ayrıca vektörlere/matrislere ilk değer atama da burada yapılır. Tank oyununda oyun ortamını oluşturan zeminin, duvarların, oyuncular olan tankların **`OnInit()`** fonksiyonunda kendilerine ait köşe noktası, doku ve normal koordinatları bilgilerini içeren **`.obj`** formatında model dosyaları okunup ilgili vertex/index/texture bufferlarına yüklenir. **`OnInit()`** fonksiyonunda ayrıca **`Eye`**, `At` ve `Up` vektörleri ilgili vektör değerlerine setlenir bu vektörler kullanılarak **`XMMatrixLookAtLH()`** ile View matrisi setlenir. **`OnInit()`** fonksiyonunda son olarak, görüş açısı (field of view angle), pencerenin yatay/düşey oranı (aspect ratio), yakın ve uzak **`z`**-düzlemlerini (near/far z-planes) parametre olarak alan **`XMMatrixPerspectiveFovLH()`** ile perspektif izdüşüm (projeksiyon) matrisi setlenir.

```
Eye = XMVectorSet(0.0f, 4.0, -30.0, 0.0);
At  = XMVectorSet(0.0f, 0.0,   1.0, 0.0);
Up  = XMVectorSet(0.0f, 1.0,   0.0, 0.0);
g_View = XMMatrixLookAtLH(Eye, At, Up);

g_Projection = XMMatrixPerspectiveFovLH(XM_PIDIV4, 1280/720, 0.01, 1000);
```



`OnUpdate()` fonksiyonunda klavye/mouse etkileşimine göre ilgili değişkenler setlenmektedir. Örneğin **`W,A,S,D`** tuşları ve mouse ile **`Eye, At`** vektörleri ve dolayısıyla **`g_View`** matrisi güncellenmektedir (836. satır). Böylece 3D ortamda mouse doğrultusunda tuşlarla ilerlenmektedir. Tankın konumu (**`Tank_Position`**) da 3rd Person Shooter tarzı olarak bakış noktasından (**`Eye`**) hareket doğrultusunca (**`At-Eye`**) 3 birim ilerisinde ve 2 birim aşağısında olacak şekilde şöyle hesaplanmakdır (847. satır):

```
XMVECTOR Tank_Position = Eye + 3 * (At - Eye) + XMVectorSet(0, -2, 0, 0);
```

Tank düşmana (enemy) **`SPACE`** tuşu ile ateş etmektedir. Bu tuşa basıldığında namlunun ucundan top mermisi çıkmakta ve düşmana doğru hareket etmektedir. Merminin başlangıç noktası namlunun ucu olacak şekilde World matrisi aşağıdaki kodla setlenir (859-873 arası):

```
Ro_Tank_Missile = Eye + XMVectorSet(0, -1, 0, 0);
Rd_Tank_Missile = XMVector3Normalize(At - Eye);

if (FireTankMissile)
{
	XMVECTOR initialPosition = Ro_Tank_Missile + (3+2.5)*Rd_Tank_Missile;
	XMFLOAT4 initialPosition_F4;
	XMStoreFloat4(&initialPosition_F4, initialPosition);
	g_World_Missile = g_World_Tank;
	XMFLOAT4X4 g_World_Missile_4x4;
	XMStoreFloat4x4(&g_World_Missile_4x4, g_World_Missile);

	g_World_Missile_4x4._41  = initialPosition_F4.x;
	g_World_Missile_4x4._42  = initialPosition_F4.y;
	g_World_Missile_4x4._43  = initialPosition_F4.z;
	g_World_Missile          = XMLoadFloat4x4(&g_World_Missile_4x4);
}
```

`initialPosition = Ro_Tank_Missile + ( 3 + 2.5 ) * Rd_Tank_Missile` ile merminin namlu ucundaki konumu hesaplanır. **`Eye + XMVectorSet(0, -1, 0, 0)`** ile merminin başlangıç noktası hesaplanır. Namlunun yerden yüksekliği `1`'i elde edebilmek için bakış noktasının yüksekliği **`2`**'den **`1`** çıkarılmıştır. **`(3+2.5)*Rd_Tank_Missile`** eklendiğinde başlangıç noktası namlunun ucu olur. Burada **`2.5`** tankın merkezinden namlu ucuna kadar

olan mesafedir. Tank bakış noktasından **3** birim ileride olduğundan ayrıca **3** eklenmiştir. **Rd_Tank_Missile** merminin doğrultusudur.

Merminin 4x4 World matrisinin (**g_World_Missile**) öteleme (translation) bileşenleri **_41**, **_42** ve **_43** merminin başlangıç noktasının **x**, **y** ve **z** değerlerine setlenerek başlangıç noktası olarak namlunun ucuna çizilmesi sağlanır. **SPACE** tuşuna basıldığında **FireTankMissile** boolean değişkeni **false** **TraceTankMissile** **true** yapılarak merminin hareket edebilmesi sağlanır. Merminin hareketi için World matrisinin **_41**, **_42** ve **_43** bileşenleri yukarıdakine benzer aşağıdaki kodla setlenir (877-884):

```
if (TraceTankMissile)
{
	XMStoreFloat4x4(&g_World_Missile_4x4, g_World_Missile);
	g_World_Missile_4x4._41 += 0.5 * Rd_Tank_Missile_Float4.x;
	g_World_Missile_4x4._42 += 0.5 * Rd_Tank_Missile_Float4.y;
	g_World_Missile_4x4._43 += 0.5 * Rd_Tank_Missile_Float4.z;
	g_World_Missile = XMLoadFloat4x4(&g_World_Missile_4x4);
}
```

Merminin düşmana isabet edip/etmediği "*Ray Tracing*" ile test edilir (899-924). **Ro_Tank_Missile** başlangıç noktası **Rd_Tank_Missile** doğrultusuna sahip ışın ile ortamdaki tank, duvarlar ve zemin arasında kesişim testleri yapılır. Mininum **t** uzaklığına sahip cisim düşman tankı ise ve mermi düşman tankına değmişse düşmanın render edilip/edilmeyeceğini belirleyen **RenderEnemy** boolean değişkeni **false** yapılarak düşman öldürülür yani çizilmez. **renderTankMissile** da **false** yapılarak mermi de artık çizilmez. Düşman tankı canlandırmak (tekrar çizmek) için sol mouse butonuna tıklanır.

Merminin düşman tankına değip değmediğinin belirlenmesi için kesişim testlerinden dönen en yakın cismin **nearest.t** uzaklığı ile düşman üzerindeki kesişim noktasının konumu **RedDot_Position** hesaplanır (906). **RedDot_Position** ile merminin o andaki konumu **Missile_Position**'ın fark vektörünün boyu **Missile_RedDot_Distance** çok küçük bir değerin (**0.5**) altına düşünce merminin düşmana isabet ettiği sonucuna varılır. **Missile_RedDot_Distance** şöyle hesaplanır:

```
XMVectorGetX(XMVector3Length(RedDot_Position - Missile_Position))
```

XMVector3Length() aslında skaler bir değer döndürmelidir ama DirectX'in matematik kütüphanesindeki vektörel işlem fonksiyonlarının tamamı vektör döndürmektedir. Dönen bu vektörden örneğin yukarıdaki gibi fark vektörünün boyunu okuyabilmek için **XMVectorGetX()** kullanılmıştır. Bunun yerine **XMVectorGetY()** ya da **XMVectorGetZ()** de olabilirdi. DirectX'in matematik kütüphanesi fonksiyonlarına buradan erişebilirsiniz. Oyunda kullanılan fonksiyonlar "Geometric Functions" ve "Transformation Functions" linklerindedir.

Oyun ortamında ilerlerken duvarların içinden geçilmemesi için de Ray Tracing yöntemi kullanılabilir. **OnUpdate()** fonksiyonunda **W** ve **S** tuşlarının test edildiği **if(){}** bloklarında ışının başlangıç noktası ve doğrultusu **Ro** ve **Rd** vektörleri örneğin **W** için aşağıdaki gibi setlenip **IntersectTriangle()**'dan dönen **t** uzaklığı belli bir değerin altına inince hareket engellenir. Başlangıç değeri **true** olarak setlenen bool bir değişken **t** belli bir değerin altına inince **false** yapılır. **moveBackForward+=speed** ve **moveBackForward-=speed** güncellemeleri bu değişkene bağlı olarak (**true** ise) yapılır:

```
XMVECTOR Ro = Eye;
XMVECTOR Rd = XMVector3Normalize(At - Eye); // S için : (Eye - At)
```

`OnRender()` fonksiyonunda sırasıyla zemin, duvarlar, oyuncuyu temsil eden tank, o tankın ateş etmesi sonucu yollanacak mermi (tank missile), nişangâh (reddot), düşman tankı (enemy) ve son olarak düşman tankının ateş etmesi sonucu yollanacak mermi (enemy missile) kendileri için `OnUpdate()`'te yapılan `m_constantBufferData.mWorld` constant buffer setlemelerine göre transform edilip çizilir.

3. Deney Hazırlığı

- Sağ mouse butonu tıklanılmış olarak tutulurken zoom yapılacak ve mousedan el çekildiğinde zoom öncesine dönülecek şekilde 800 ve 805. satırlardaki `if(){}` bloklarına gerekli kodları yazınız.
  İpucu → Projeksiyon matrisinin görüş açısı (`FovAngleY`) daha küçük bir değere (örneğin `XM_PI/12`) setlenerek zoom yapılabilir.

- Duvar arkasından ateş edildiğinde mermi duvarı delip geçmeyecek şekilde kodu güncelleyiniz.

- Oyun ortamında ilerlerken duvarların içinden geçilmemesi için önceki sayfanın son paragrafında anlatılan yönteme göre 742 ve 747. satırlardaki `if(){}` bloklarına gerekli kodları yazınız.

- Tank düşmana ateş ettiğinde mermi dönerek ilerleyecek şekilde kodu güncelleyiniz.

- Tank düşmana ateş ettiğinde mermi düşmana ulaşana dek namluyu oynatmamak gerekir. Eğer oynatılırsa merminin yönü değişir ve kesişim testleri merminin (namlunun) o anki doğrultusuna göre yapılır. Programdaki bu hatayı düzeltiniz. Yani ateş ettikten sonra namlu oynatılsa bile mermi ateş edilen doğrultuda ilerlesin.

4. Deney Tasarımı ve Uygulaması

Düşmanı temsil eden tank (enemy) bizim kontrol ettiğimiz tanka (tank) ateş edip vuracak şekilde kodu güncelleyiniz. Basitten zora doğru adım adım ilerlemek için öncelikle namluyu döndürmeksizin sanki dönmüş ve hedefe kilitlenmiş gibi varsayarak ateş ettirebilirsiniz. Bunun için `OnUpdate()` fonksiyonunda 945. satırdan itibaren kod yazacaksınız. Ekleyeceğiniz kodlar 859-924 arasındaki kodlara benzer olacaktır. Yani biz düşmana ateş ettiğimizde gerçekleşen olayların benzerleri düşman bize ateş ettiğinde yaşanacaktır. Öncelikle `if(FireEnemyMissile){}` ve `if(TraceEnemyMissile){}` şeklinde iki kod bloğu ekleyip bu bloklara sırasıyla merminin başlangıç konumunun setlenmesi ve ilerlemesi için gerekli kodlar yazılır:

`if(FireEnemyMissile){}` içinde merminin başlangıç noktası namlunun ucu olarak setlenirken namlunun doğrultusu olan **`(0,0,1)`** vektörü namlunu boyu **`2.5`** ile çarpılıp düşman mermisinin 4x4 World matrisinin (**`g_World_Enemy_Missile`**) öteleme (translation) bileşenleri **`_41`**, **`_42`** ve **`_43`** setlenir. Bizim kontrol ettiğimiz tank için ayrıca **`Ro_Tank_Missile`** değişkenini de kullanmıştık. Çünkü biz 3D ortamda hareket ediyoruz. Düşman tankı ise sabit olduğundan düşmanın konumu **`(0,0,0)`** olarak varsayılabilir ve dolayısıyla merminin konumu hesaplanırken **`Ro`**‘ya gerek yoktur.

`if(TraceEnemyMissile){}` içinde mermi ilerletilmek üzere hareket doğrultusu belirlenmelidir. Bizim kullandığımız tankta merminin yönü **`XMVector3Normalize(At-Eye)`** ile belirleniyordu. Bu sefer mermi yönü bizim kullandığımız tankın konumundan merminin konumu çıkarılarak bulunacaktır. İlgili World matrislerinin **`_41`**, **`_42`** ve **`_43`** bileşenlerinin farkları **`XMVectorSet()`** ile bir vektöre setlenip **`XMVector3Normalize()`** ile normalize edilerek yön belirlenebilir. Herhangi bir tuşa (örneğin ‘**`F`**’) basıldığında **`FireEnemyMissile`** boolean değişkeni **`false`** **`TraceEnemyMissile`** **`true`** yapılarak mermi ilerletilir.

Düşman tankından ateşlenen merminin bizi vurup/vurmadığı da benzer şekilde test edilbilir. Merminin başlangıç noktası `if(FireEnemyMissile){}` içinde, doğrultusu da `if(FireEnemyMissile){}` içinde hesaplanmıştı. **`testIntersections()`** fonksiyonuna bu vektörler **`Ro, Rd`** değerleri olarak verilir. 3. parametre de bizim tankın World matrisidir.

`testIntersections()` fonksiyonu **`IntersectTriangle()`** fonksiyonunu zemin, duvarlar ve tank için koşar ve (varsa) kesişimleri `intersect` türünden structlar olarak `intersections` adlı STL vektöre ekler. `intersect` struct’ı `t` uzaklığına ek olarak `isWall` ve `isEnemy` olmak üzere iki tane boolean değişken tutar. **`testIntersections()`** fonksiyondaki kesişim testlerine göre duvarla kesişim varsa `isWall`, tankla kesişim varsa da `isEnemy` **`true`** yapılır. **`nearestObject()`** fonksiyonu en yakın kesişimi `intersect` olarak döndürür. Eğer **`isEnemy=true`** ise (burada enemy bizim kontrol ettiğimiz tanktır) ve nişan alınan nokta (**`RedDot_Position`**) ile merminin o anki konumu arasındaki uzaklık çok küçük bir değerin altına düşmüşse **`renderTank`** ve **`renderEnemyMissile`** **`false`** yapılır yani artık render edilmezler.

Deney uygulamasının bu noktaya kadarına ait videoyu buradan izleyebilirsiniz.

Şimdi sıra geldi düşmanın dönerek hedefe (bize) kilitlenip ateş etmesine. Düşman tankını döndürmek üzere **g_World_Enemy** matrisi **0.02** gibi küçük bir açı ile y-ekseninde dönme yapacak şekilde **XMMatrixRotationY(0.02)** matrisi ile çarpılarak güncellenir. **(0,0,1)** başlangıç değerine setlenmiş namlu doğrultu vektörü, **g_World_Enemy** matrisine göre **XMVector3TransformNormal()** fonksiyonu ile transform edilerek güncellenir. Yukarıda da anlatıldığı gibi World matrislerinin **_41, _42** ve **_43** bileşenlerinin farkları **XMVectorSet()** ile bir vektöre setlenip **XMVector3Normalize()** ile normalize edilerek düşmanı temsil eden tanktan bizim tanka doğru olan vektör hesaplanır. Bu vektör ile **XMVector3TransformNormal()** ile transform edilen vektör arasındaki açının kosinüsü **XMVector3Dot()** ile hesaplanır. Bu değer **1** olana kadar (veya **>0.9999**) yani transform edilen namlu doğrultu vektörü ile düşmandan bize doğru olan vektör arasındaki açı **0** derece olana (bize kilitlenene) kadar bu paragraftaki işlemler tekrarlanır.

Deney uygulamasının bu noktaya kadarına ait videoyu buradan izleyebilirsiniz.

5. Deney Raporu

Kaynak kodlardaki **Rapor.docx** adlı şablon belge **deney saatine kadar** takım adına doldurulup **Deney Hazırlığı** olarak istenen güncellemeleri içeren **D3D12TankOyunu.cpp** kod dosyası ile birlikte dersin Moodle Sayfasına yükleyiniz. **Deney Hazırlığı** bireysel, **Rapor** takım adına yüklenecektir.

KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR GRAFİKLERİ LABORATUARI

Pürüzlü Yüzey Üretimi

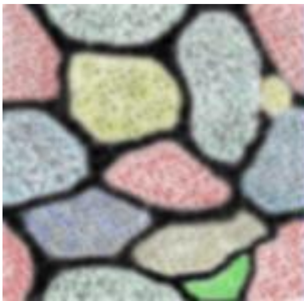
1. Giriş

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

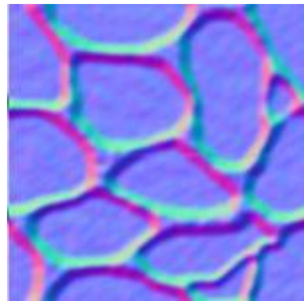
Bump Mapping yönteminde Şekil.1(a)'da kaplanacak dokudaki renk değişimlerinden elde edilen (b)'deki "**Normal Map**" dokusu kullanılır. Bu normal mapdeki `(R,G,B)` değerleri yüzeyin normalinin `(X,Y,Z)` değerleri olarak alınır. Yüzeyin diffuse ve specular renk bileşenleri yüzey normaline bağlı olarak 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. Dolayısyla elde edilen pürüzlü yüzeylerdeki tümsekler/çukurlar, bump mapping yöntemine nazaran daha fazladır. Hatta Parallax mapping yönteminde bu derinlik değerini ayarlamak bile mümkündür.

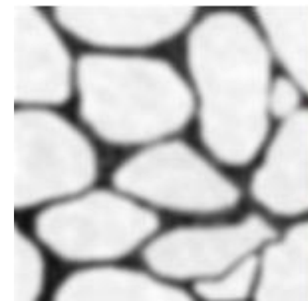
Deneyde **Jason Zink**'in, "**A Closer Look At Parallax Occlusion Mapping**" başlıklı makalesi ve örnek HLSL programından yararlanılarak geliştirilen DirectX 11 uygulaması ile yukarıda bahsedilen özel doku kaplama yöntemleri incelenecektir.



(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.

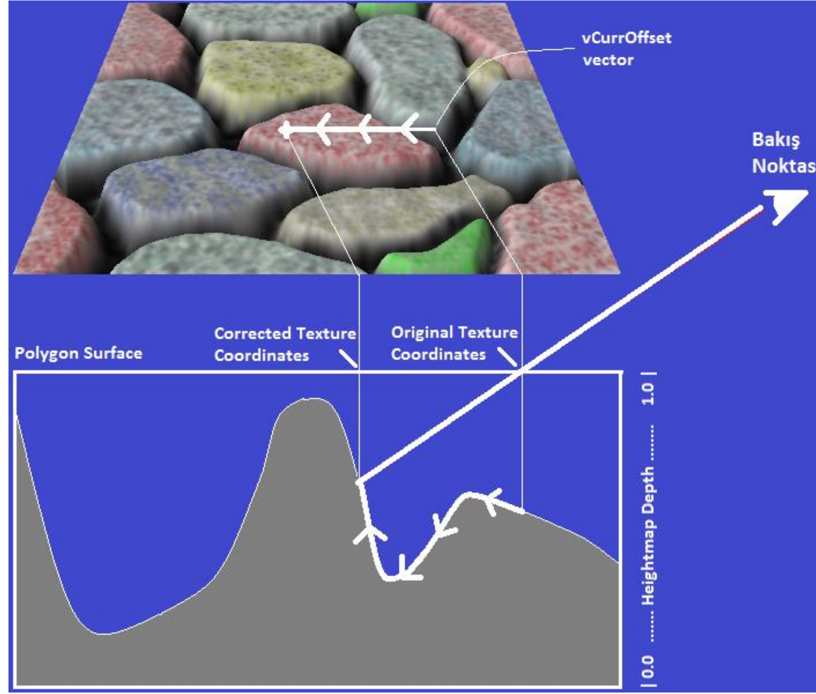
2. Parallax Mapping Yöntemi

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 yalnızca ambient ve diffuse renk bileşenlerini hesaplamaktadır. **`ParallaxMapping11.fx`**’e 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.

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

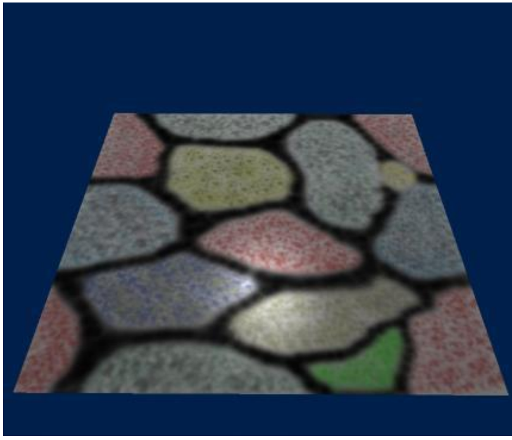
“**`0`**”, “**`1`**”, “**`2`**” ve “**`3`**” tuşlarına basıldığında Şekil.4’teki görüntüler elde edilecek şekilde programı güncelleyiniz. Görüntülerde :

“**`0`**” : bump mapping **yok**, parallax mapping **yok**,
“**`1`**” : bump mapping **yok**, parallax mapping **var**,
“**`2`**” : bump mapping **var**, parallax mapping **yok**,
“**`3`**” : bump mapping **var** parallax mapping **var**

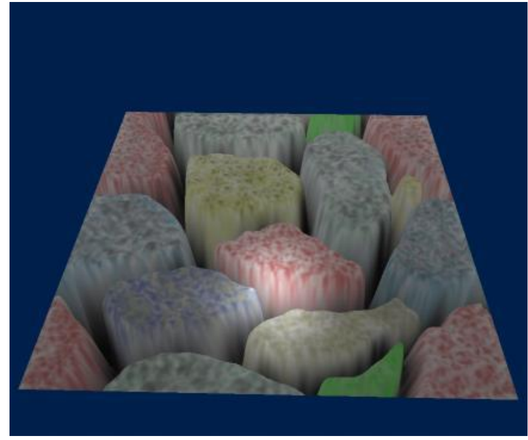
olarak özetleyebileceğimiz etkileri görüyoruz.

İpucu → `.cpp` programında 24. satırdaki `ConstantBuffer` adlı struct'a `int key` değişkeni ekleyiniz. 201. satırdaki `WndProc()` içinde `case 'E':` 'den sonra 4 tane daha `case` ekleyiniz ve basılan tuşa göre `key`'i setleyiniz. `.fx` programının 12. satırında da `.cpp`'dekinin benzeri bir `ConstantBuffer` vardır. Buna da bir `int key` değişkeni ekleyiniz. `.cpp`'deki `key`'in değerinin `.fx`'teki eşdeğerine otomatik aktarıldığını varsayınız. Basılan tuşun değerini tutan `key`'e göre `PS()` adlı pixel shaderda 100. satırdaki `fParallaxLimit` değişkenini `0`'a setleyerek parallax mapping etkisini kaldırabilirsiniz. Bump mapping etkisini kaldırmak için de 200. satırdaki `vFinalNormal` değişkenini yine `key`'e göre setleyiniz. Program default olarak hem bump hem de parallax etkisini gerçeklediğinden `key=3` için `.fx`'te kod yazmanız gerekmiyor. Fakat `.cpp`'de basılan tuşa göre `key=3` setlemesini yapmalısınız.

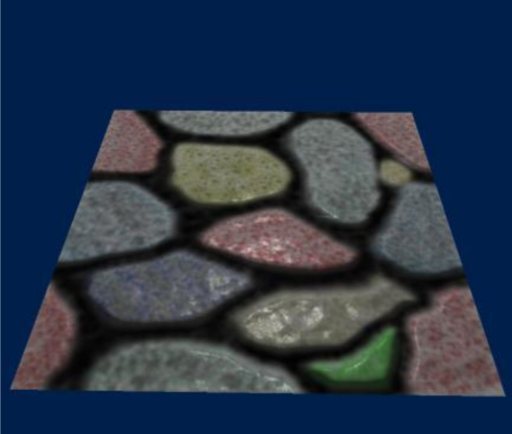
Kaynak kodların olduğu klasörde Şekil.4'teki bump/parallax mapping modlarını içeren `bpModes.mp4` isimli video paylaşılmıştır.



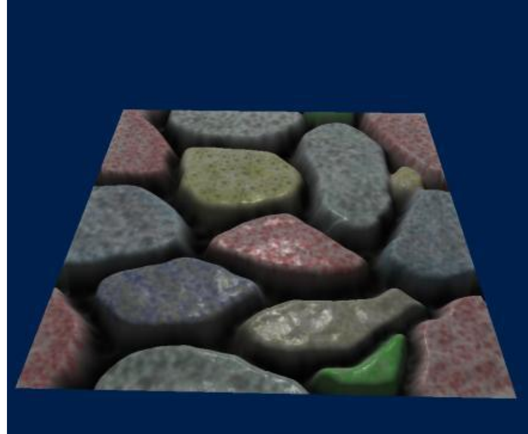
`0 : bump yok, parallax yok`



`1 : bump yok, parallax var`



`2 : bump var, parallax yok`



`3 : bump var, parallax var`

Şekil 4: bump/parallax mapping modları

7. Deney Raporu

Deney Raporunu, `Rapor.docx` adlı şablon belgeye göre takım adına hazırlayıp **Deney Hazırlığı** çalışması `ParallaxMapping11.fx` ile **deney saatine kadar** (takım adına biriniz) dersin Moodle Sayfasına yükleyiniz.