WebGL Uygulamaları

1. WebGL Nedir?

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

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

Şekil 1. WebGL Destekli Bazı Tarayıcılar

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

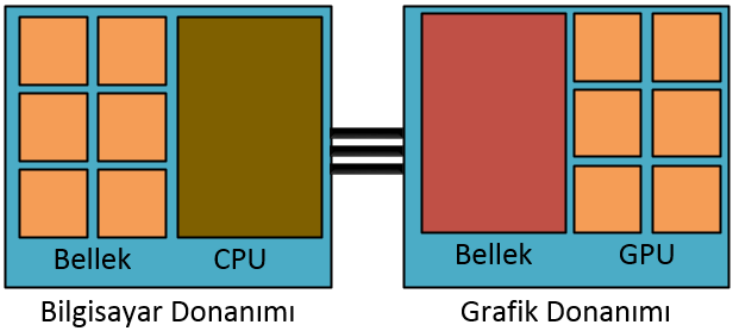
Deney Hazırlığı: WebGL 3D grafik API'si hakkında bilgi sahibi olunuz. Örnek uygulamaları araştırıp, inceleyiniz. WebGL'in eksiklikleri nelerdir, geleceğe yönelik midir, araştırınız.

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

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

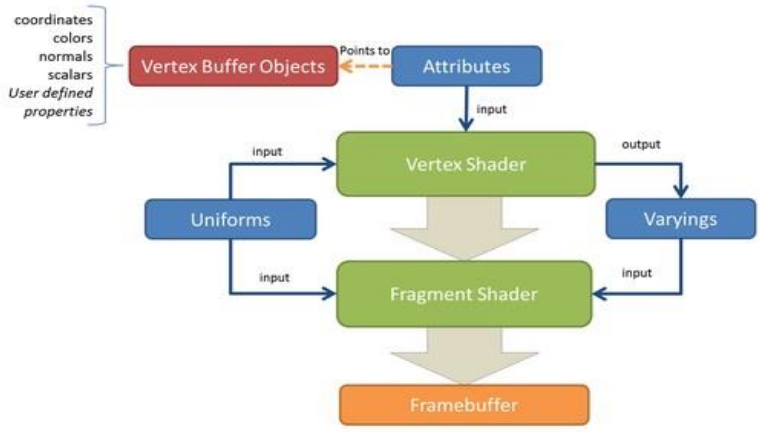
WebGL'in temellendiği fikirde burada ortaya çıkmaktadır. Amaç, iki birim arasında gerçekleşen haberleşme yükünü azaltmaktır. Bu işlem şu şekilde gerçekleştirilir. Komutları ve grafik kaynaklarını (çizim nesnesi köşe nokta bilgileri vd. veriler) sürekli ve parçalı olarak CPU GPU arasında göndermek yerine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

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



Şekil 2. Donanım

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



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

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

2. Deney Uygulamaları

2.1. Canvas API ile Web Üzerinde 2D Çizim Uygulaması

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

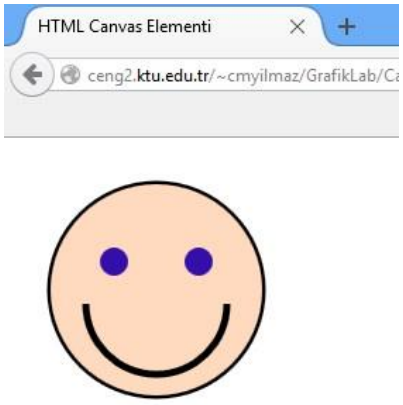
Ekstra: Canvas yerine SVG (Scalable Vector Graphics)'de kullanılabilir [11][12].

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

Verilen uygulamada çizim için gerekli temel adımlara kısaca bakacak olursak;

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

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



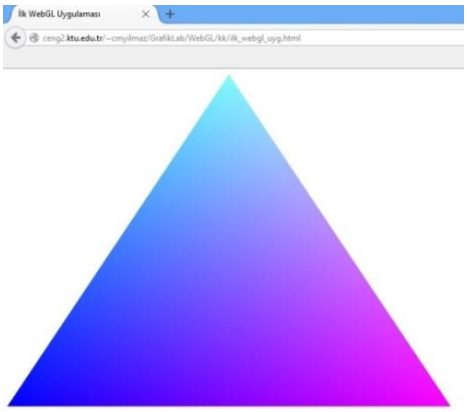
Şekil 4. İlk Canvas API uygulaması

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

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

2.2. İlk WebGL Uygulaması

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



Şekil 5. İlk WebGL Uygulaması

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

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

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

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

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

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

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

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

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

2.2.1.Vertex Shaderlar

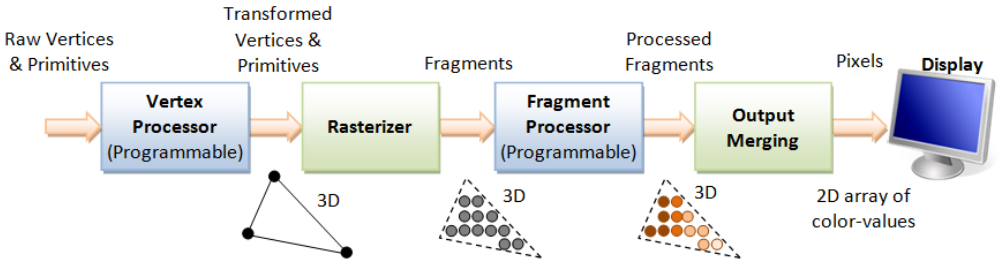
Vertex shaderlar vertex (poligonların köşe noktaları şeklinde düşünülebilir) attributelarının tanımlanması ve işlenmesini sağlarlar (mesh koordinatlarının setlenmesi şeklinde düşünebilirsiniz). Bu işlemler rendering sürecinde çalıştırılacak olan main() fonksiyonları içerisinde gerçekleştirilir. Vertex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

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

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

2.2.2. Fragment Shaderlar

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



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

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

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

Uygulamaya dönecek olursak, vertex shaderlar kısaca şu şekilde çalışmaktadır:

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

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

Uygulamada fragment shader programı kısaca şu şekilde çalışmaktadır:

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

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

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

```
// -- getElementById ve the id attribute shader kaynak kodunu edin
shaderScript = document.getElementById("2d-vertex-shader");
shaderSource = shaderScript.text;
// -- boş bir shader nesnesi oluştur
vertexShader = gl.createShader(gl.VERTEX_SHADER);
// -- shader nesnesine shader kaynak kodunu ekle
gl.shaderSource(vertexShader, shaderSource);
// -- shader kodunu derle
gl.compileShader(vertexShader);

...

// -- program nesnesi oluştur
program = gl.createProgram();
// -- vertex ve fragment shaderları programa yükle
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
// -- shaderları program nesnesine bağla
gl.linkProgram(program);
// -- mevcut rendering aşamasında bu programı kullan
gl.useProgram(program);
```

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

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

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

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

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

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

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

2.3. WebGL ile 3D Nesne Üretimi

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

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

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

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

2.4. Three.js ile WebGL Uygulamaları

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

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

2.4.1. Three.js ile İlk Uygulama

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

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

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

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

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

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

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

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

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

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

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

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

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

Uygulamada aşağıdaki değişiklikleri yaparak farklı bir animasyon elde edebilirsiniz.

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

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

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

3. Deney Hazırlığı

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

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

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

4. Deney Tasarım ve Uygulanışı

Hazırlık soruları ile deneye hazırlık seviyesi ölçülecek.

Öğrencilerin deney sorularını cevaplandırması istenecek.

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

5. Deney Raporu

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

6. Kaynaklar

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

12. http://www.svgopen.org/2010/papers/58-WebGL__SVG/ Embedding WebGL documents in SVG, 2015.
13. http://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html WebGL Fundamentals, 2015.
14. http://www.html5rocks.com/en/tutorials/webgl/shaders/ An Introduction to Shaders, 2015.
15. http://www3.ntu.edu.sg/home/ehchua/programming/opengl/cg_basicstheory.html 3D Graphics with OpenGL, 2015.
16. https://blog.mayflower.de/4584-Playing-around-with-pixel-shaders-in-WebGL.html Playing around with frag. shaders in WebGL, 2015.