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.

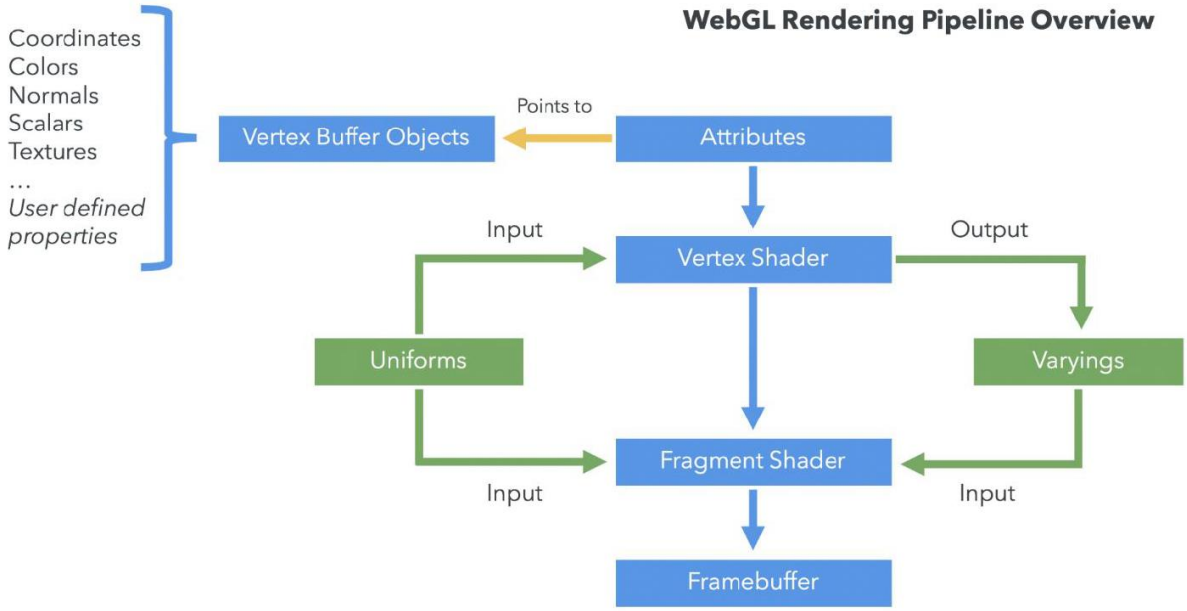
WebGL uygulamalarına başlamadan önce Canvas API ile 2D gülen yüz çizen basit bir uygulamadan bahsedilecektir. Örnek programlardaki **`0_canvas.html`** sayfasına ait kodların bir kısmı sonraki sayfada verilmiştir. Gülen yüz uygulamasının temel adımları şu şekildedir:

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 :

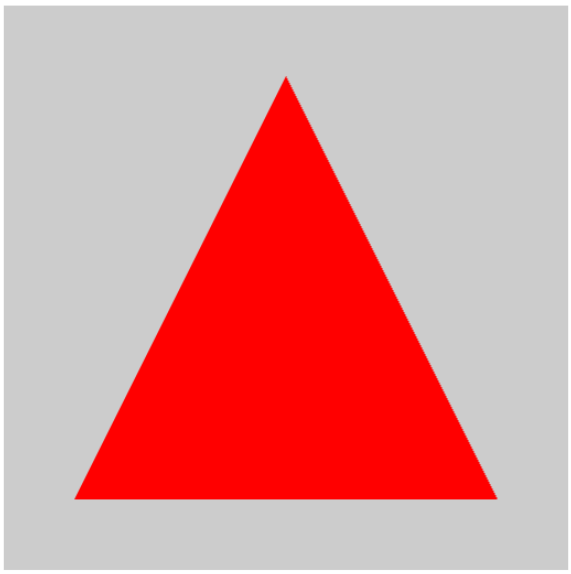
```
var vertexElem = document.getElementById( "vertex-shader" );
var vertexShader = gl.createShader( gl.VERTEX_SHADER );
gl.shaderSource(vertexShader, vertexElem.textContent.replace(/^\s+|\s+$/g,'' ));
gl.compileShader( vertexShader );

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 :

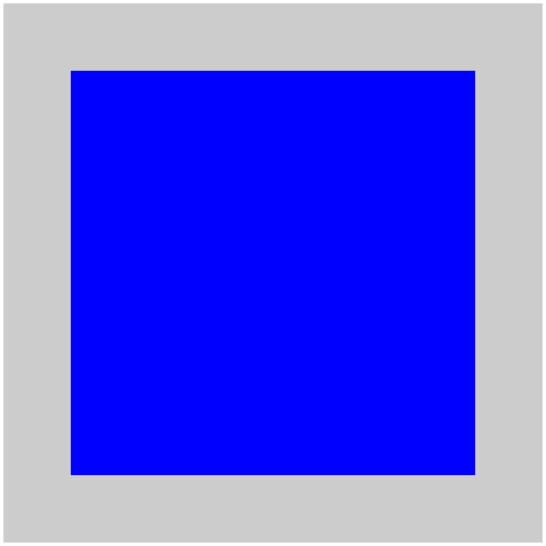
```
var fragmentElem = document.getElementById( "fragment-shader" );
var fragmentShader = gl.createShader( gl.FRAGMENT_SHADER );
gl.shaderSource(fragmentShader,fragmentElem.textContent.replace(\s+$/g,''));
gl.compileShader( fragmentShader );
```

[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:

$$\text{CCW Rotation around Z-axis}$$

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

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_;
```

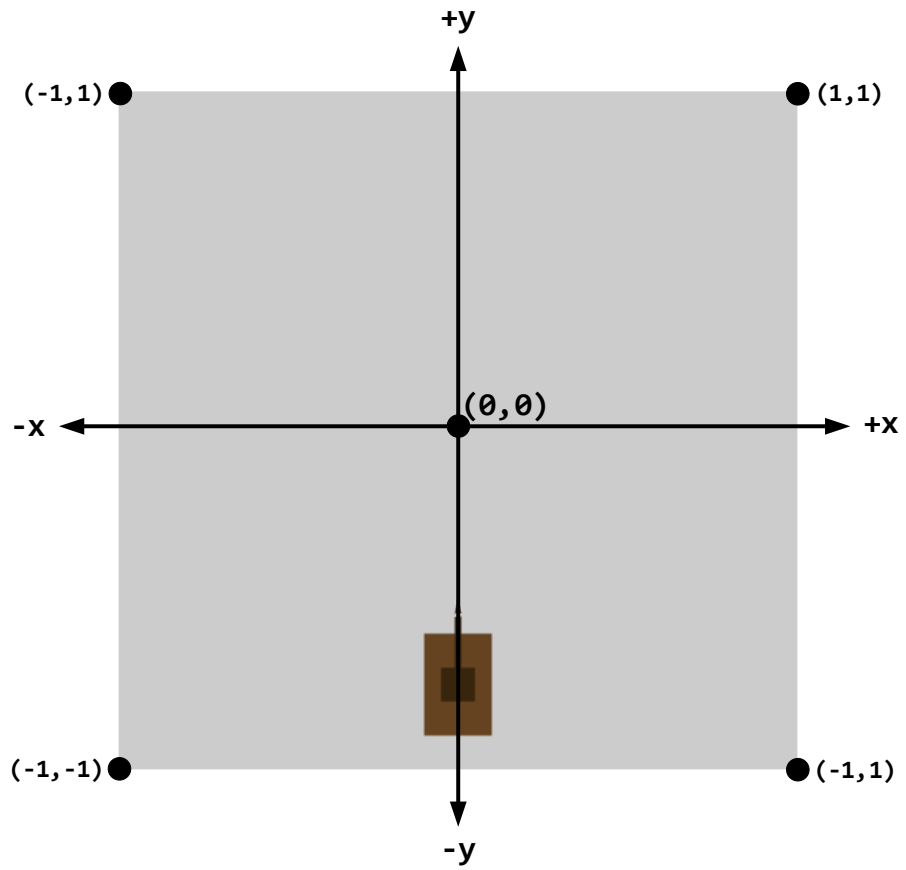
Vertex Shader veya Fragment Shader fonksiyonlarının JavaScript metodlarıyla `uniform` değişkenler üzerinden haberleştiğini daha önce belirtmiştik. Döndürme işlemi için Vertex Shader'da `vTheta` uniform değişkeni tanımlanmıştır. Bu değişkenin değeri JavaScript metodunda `theta` adlı başka bir değişken 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.

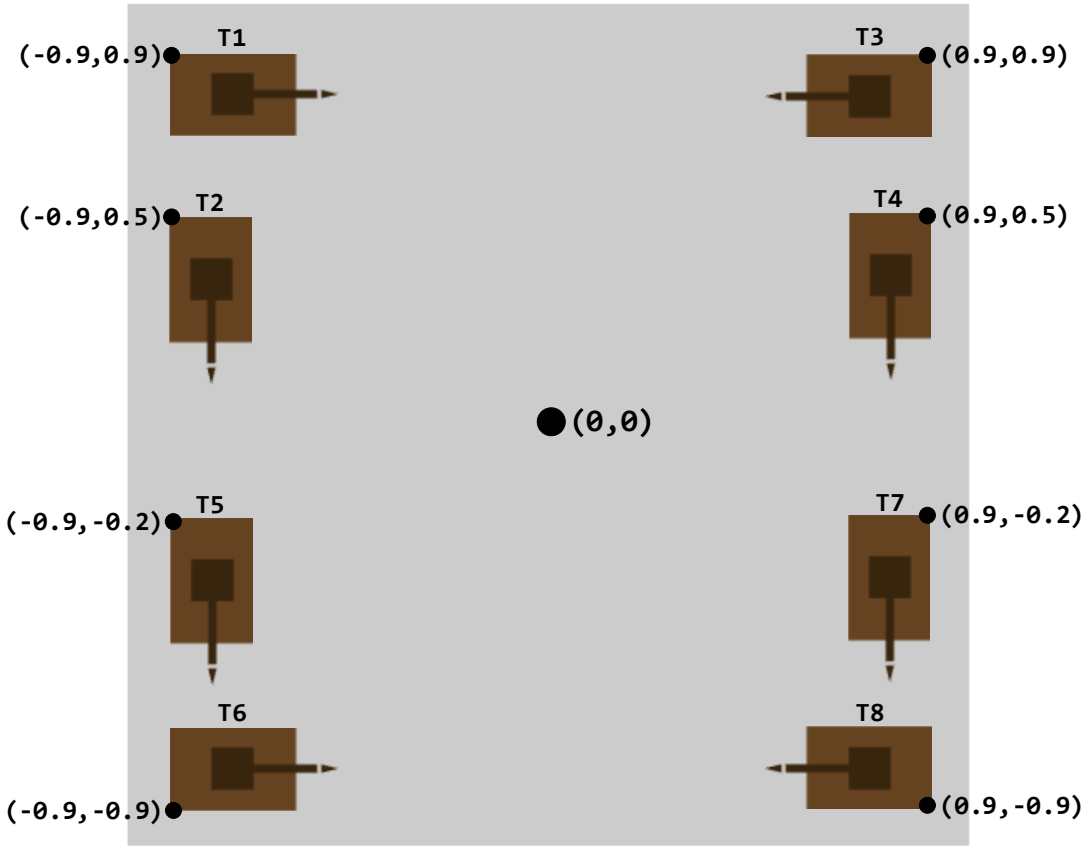


6. Deney Hazırlığı

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 0.2br, yüksekliği 0.3br'dir. Kare üst gövdenin kenarları 0.1br'dir. Namlunun eni 0.02br, yüksekliği 0.25br'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 çıkar

    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.