Tartalmi összefoglaló

- ***A téma megnevezése:***

Interaktív 3D grafika a weben WebGL segítségével

- ***A megadott feladat megfogalmazása:***

A WebGL technológia bemutatása: alapfogalmak (pufferek, shaderek), egyszerű alakzatok modellezése, geometriai transzformációk, megvilágítási jellemzők, átlátszóság, perspektíva beállítása WebGL-ben, hasonlóságok és különbségek az OpenGL-hez képest. OpenGL modellek implementálása WebGL-ben. Egy interaktív, böngészőben, plug-in nélkül futó háromdimenziós játék fejlesztése keretrendszer felhasználásával.

- ***A megoldási mód:***

A háromdimenziós alakzatok modellezése a WebGL grafikus programkönyvtár segítségével történik, melyet a HTML5 és JavaScript nyelvekbe ágyaztam be. A játék fejlesztéséhez az O3D keretrendszert is felhasználtam, mely egy nyílt forráskódú, a Google által kifejlesztett JavaScript API.

- ***Alkalmazott eszközök, módszerek:***

WebGL, JavaScript, HTML5, O3D, Blender, Google Chrome böngésző (15.0.874.121-es verzió)

- ***Elért eredmények:***

A WebGL technológiát példaprogramok segítségével bemutatom, különös tekintettel azon részekre, melyek eltérnek az OpenGL-től. A többféle példaprogram a legegyszerűbb modellektől az összetettebbekig szemlélteti, mire képes a WebGL és ez hogyan valósítható meg. A játék a WebGL-t támogató Google Chrome böngészőben közvetlenül futtatható, az egér és a billentyűzet segítségével irányítható, és a háromdimenziós modellezés nagyban növeli a felhasználói élményt.

- ***Kulcsszavak:***

WebGL, 3D modellezés, interaktív webes alkalmazás, számítógépes grafika, JavaScript, HTML5

Tartalomjegyzék

Feladatkiírás 2

Tartalmi összefoglaló 3

Tartalomjegyzék 5

BEVEZETÉS 7

1. A WEBGL TECHNOLÓGIA 10

1.1. A kezdeti lépések 10

1.2. Pufferek 12
1.2.1. Pufferek létrehozása, feltöltése 12
1.2.2. Pufferek használata 14

1.3. Geometriai transzformációk 15

1.4. Shaderek 17
1.4.1. Egyszerű vertex és fragment shader 17
1.4.2. A GLSL nyelv 19
1.4.3. Shaderek használata a WebGL programban 20

1.5. Textúrák 22

1.6. Megvilágítás 24
1.6.1. Phong modell 25
1.6.2. Egyszerű megvilágítást kezelő shaderek írása 26
1.6.3. A megvilágítás jellemzőinek megadása JavaScriptben 29

1.7. Átlátszóság 30

1.8. Perspektíva beállítása 32

1.9. Renderelés 32

2. MODELLEK IMPLEMENTÁLÁSA WEBGL SEGÍTSÉGÉVEL 34

2.1. Fogaskerék 2D-ben 34
2.1.1. Kitöltött és drótvázas modellek 34
2.1.2. Billentyűzet kezelése 38

2.2. Tűzijáték 2D-ben 40
2.2.1. Animálás 40

2.3. Kopácsoló harkály 43
2.3.1. Modell készítése a Blender modellező programmal 44
2.3.2. Ambiens és direkcionális megvilágítás, animálás 44

2.4. Választható alakzatok modellezése 46
2.4.1. Négyféle alakzat modellezése 47

2.4.2. Egér események kezelése 50
2.4.3. Per-fragment módszerrel számoló shaderek írása 52

3. O3D KERETRENDSZER 57

3.1. Keretrendszer választása: O3D 57

3.2. Első lépések az O3D használatához 58

3.3. Teljesítmény 59

3.4. Csomagok 59

3.5. Transzformok és transzform-gráf 60

3.6. Render-gráf 62

3.7. Alakzatok, anyagok, effektek, textúrák 63

3.8. Eseménykezelés 64

4. HÁROMDIMENZIÓS JÁTÉK ÍRÁSA O3D SEGÍTSÉGÉVEL 66

4.1. A játék szabályai 66

4.2. Az O3D által kínált lehetőségek felhasználása a játékban 70
4.2.1. Az objektumok kezelése: csomagok, transzformok 70
4.2.2. Globális változók, beállítások 72
4.2.3. A játék vezérlése 73
4.2.4. A keretrendszer további lehetőségei 75

Irodalomjegyzék 80

Nyilatkozat 81

Köszönetnyilvánítás 82

Függelék 83

BEVEZETÉS

Szakdolgozatom témája interaktív webes alkalmazások készítése a WebGL technológia segítségével. A WebGL (Web-based Graphics Library) egy grafikus programkönyvtár, mely a HTML5 és a JavaScript programozási nyelvekbe ágyazva lehetővé teszi 3D-s grafikák megjelenítését közvetlenül a böngészőben, plug-inek használata nélkül. Nagyon friss technológiáról van szó, a hivatalos specifikáció 1.0-s verziószámmal 2011. március 3-án jelent meg (WebGL 1.0 Specification [10]), és amikor elkezdtem foglalkozni a szakdolgozattal, még csak egyes böngészők fejlesztői verzióiban – Minefield (Mozilla Firefox), Chromium (Google Chrome), WebKit (Safari) – volt támogatott a WebGL. Időközben már a Google Chrome és a Mozilla Firefox, valamint az Operának a fejlesztői verziója is támogatja.

A témával való foglalkozás során tehát többek között a technológia új mivolta és gyors változása jelentette a nehézséget és a kihívást is. Szakdolgozatom első felében magáról a technológiáról lesz szó, arról, hogy mire képes és hogyan.

Miért jó a WebGL? Legfontosabb előnye a közvetlenül böngészőből, egyéb program illetve plug-in telepítése nélkül elérhető 3D élmény, ezen felül a böngésző- és platformfüggetlenség. További előnye még a gyorsaság, mivel a modellezés során a számítások nagy részéhez közvetlenül a grafikus kártyát használja. Említésre méltó előny az is, hogy a WebGL az OpenGL ES 2.0-n alapul, ahhoz hasonló grafikai API-t kínál, márpedig ez az API igen széles körben ismert és használt. Emellett a WebGL a HTML5 canvas elemében fut, és mivel DOM (Document Object Model) API, így bármilyen DOM-kompatibilis nyelvbe beágyazható (pl. JavaScript, Java).

Milyen jövő vár erre a technológiára? A jelek biztatóak, mert egyre több böngészőben válik támogatottá. Véleményem szerint napjainkban egyre nagyobb jelentősége lesz a hálózatoknak, főként az internetnek, a WebGL pedig elhozhatja azt a korszakot, amikor bármilyen játék vagy alkalmazás elérhető lesz közvetlenül a böngészőből, grafikailag is kiváló minőségben. Erre jó példa a Google Chrome OS projekt, melynek lényege, hogy a böngésző vegye át az operációs rendszer funkcióját. A Google új böngészőjével felszerelt első notebook-ok 2011. június 15-én kerültek kereskedelmi forgalomba, a webböngésző egyelőre fájlkezelőt és médialejátszót foglal magában, így leginkább azokat a felhasználókat célozza meg, akik főleg internetezésre használják a gépet. A jövő azonban nyitva áll az új fejlesztések

előtt, és a felhasználói élményt nagyban növelő WebGL valószínűleg meghatározó eleme lesz a jövő webes alkalmazásainak.

Mint korábban említettem, a WebGL tulajdonképpen az OpenGL ES webes implementációjának tekinthető. Az OpenGL ES (OpenGL for Embedded Systems) az asztali gépen használt OpenGL-en alapuló 2D és 3D grafikai API-t biztosít beágyazott rendszerekben, mint például mobiltelefonok, járművek, navigációs eszközök. Ugyanakkor számos optimalizálást és bővítést tartalmaz, melyek alkalmassá teszik a beágyazott rendszerekben való használatra – a ritkán használt szolgáltatások el lettek távolítva belőle (például nem támogat NPOT textúrákat, vagyis olyan textúrákat, melyeknek dimenziói nem 2 hatványai, továbbá nem támogatja a 3D-s textúrákat sem), valamint mobil-kompatibilis adattípusokat használ. Fontos különbség van a modellezésben is, míg OpenGL-nél az alakzat pontjai a *glBegin* és a *glEnd* függvények között kerülnek megadásra, addig OpenGL ES-ben ez nem támogatott, hanem vertextömbben tároljuk a pontokat, és ezt adjuk át a shadernek a modellezéshez. OpenGL ES-ben csak a pont, vonal és a háromszög raszterizálása támogatott, OpenGL-ben ezen felül a négyszögé és a poligoné is.

A WebGL az OpenGL ES-re épül, például használja az OpenGL shading nyelvet, a GLSL-t, valamint engedélyezi a grafika hardveres gyorsítását. Azonban akadnak különbségek is. A képek betöltésénél a WebGL a böngésző képbetöltését használja, OpenGL-ben ezzel szemben több lehetőség is van rá, bár standard módszer nincsen. OpenGL-ben expliciten kell memóriát foglalni és felszabadítani, WebGL-ben a memória menedzsment automatikus.

Szakdolgozatom további részében, a WebGL technológia bemutatása után annak gyakorlati alkalmazásaival foglalkozom. Egy fejezeten keresztül vizsgálom és példákkal illusztrálom, hogyan implementálhatók WebGL-ben már létező OpenGL modellek, hogyan lehet látványos és interaktív alkalmazásokat készíteni. A kiinduló OpenGL programok a Számítógépes grafika tantárgy gyakorlatának példaprogramjai. Az OpenGL témájában a Francis S. Hill Jr. – Stephen M. Kelley szerzőpáros Computer Graphics Using OpenGL című könyvét [4] ajánlom.

Az utolsó két fejezetben egy 3D-s, böngészőben futó játék megvalósításáról és az ehhez felhasznált WebGL keretrendszerről, az O3D-ről lesz szó. Röviden áttekintem a WebGL keretrendszereket, majd bővebben is bemutatom az O3D-t, annak lehetőségeit és előnyeit. Ezután ismertetem magát a játékot, és kitérek arra is, hogy mely részeknél használtam fel az O3D kínálta lehetőségeket, ezek milyen előnyt jelentettek.

A dolgozat írásakor az elméleti alapokhoz sok segítséget nyújtott Andries van Dam – James D. Foley – John F. Hughes – Steven K. Feiner Computer Graphics című könyve [2]. A

WebGL kapcsán sok hasznos információt találtam a Kronos WebGL Public Wiki oldalán [11], valamint a learningwebgl.com weboldalon [7].

1. A WEBGL TECHNOLÓGIA

Ebben a fejezetben bemutatom a WebGL technológiát az alapoktól indulva. Szó lesz arról, hogyan kell inicializálni a WebGL-t, és hogyan lehet egyszerű alakzatokat modellezni, textúrákat használni, alapszinten kezelni a megvilágítást és az átlátszóságot, beállítani a perspektívát. A fejezethez tartozik a CD mellékleten a HelloCube nevű program (hellocube_pelda.html), mely egy egyszerű, folyamatosan forgó, textúrázott kockát modellez.
Hogyan próbálhatjuk ki a fejlesztést WebGL-ben? Először is le kell töltenünk egy olyan böngészőt, melyben a WebGL támogatott, hogy majd láthassuk a munkánk eredményét működés közben. A Google Chrome esetén nincs is több teendőnk a böngészővel, Firefox esetén viszont még engedélyeznünk kell a WebGL-t, mert alapértelmezetten nincsen engedélyezve. Ehhez írjuk be a címsorba, hogy „about:config”, majd a megjelenő oldal tetején a szűrő sorába azt, hogy „webgl”. Keressük meg a „webgl.disabled” elemet, majd engedélyezzük (dupla kattintással válthatunk a *true* és a *false* értékek között). A szakdolgozathoz tartozó programok fejlesztéséhez a Google Chrome 15.0.874.121-es verzióját használtam, és a technológia frissessége valamint a gyakori változások miatt nem garantált, hogy más böngészőkben is hibátlanul működik.

Ezután már akár neki is állhatunk programozni. Megkönnyíti azonban a dolgunkat, ha megismerünk és használunk néhány segédkönyvtárat, például a glUtils, a Sylvester és a glMatrix nevű JavaScript könyvtárakat, melyeket én is felhasználtam a példákban. Ezek főként vektor- és mátrixműveleket tartalmaznak, a későbbiekben még lesz szó arról, hogy hol vettem hasznukat.

A JavaScript programozási nyelvhez jó bevezetőt nyújtott John Pollock JavaScript, A Beginner’s Guide című könyve [5], a JavaScript újdonságainak megismeréséhez pedig sok hasznos ismeret található a 2011-ben megjelent JavaScript: The Definitive Guide: Activate Your Web Pages című könyvben, melyet David Flanagan írt [3].

1.1. A kezdeti lépések

Mivel a WebGL tartalom a HTML canvas elemében (magyarul ezt vászonnak fordíthatjuk, és HTML5-től létezik ilyen elem) jelenik meg, így az első lépés a WebGL program megírásakor a canvas elem létrehozása. A HTML5 által kínált újdonságokról áttekintést kaphatunk például Mark Pilgrim HTML5: Up and Running című könyvéből [6]. A canvas létrehozását szemlélteti az alábbi kódrészlet:

```
<body onload="webGLStart();">
  <canvas id="myCanvas" style="border: none;" width="600" height="500"></canvas>
</body>
```

A vászonnak mindenképpen adnunk kell id-t, mert ezzel fogunk hivatkozni rá, emellett be kell állítanunk a méretét is, ezzel ugyanis kijelöljük annak a területnek a nagyságát a weboldalon, ahol a WebGL tartalom meg fog jelenni. Már az oldal betöltődésekor mehgívódik a *webGLStart* nevű JavaScript függvény, ezen belül történik a WebGL, a shaderek, a pufferek és a textúrák inicializálása, valamint néhány alapbeállítás (milyen legyen a vászon alapszíne, legyen-e mélység ellenőrzés) és maga a modellezés is. A *setInterval* függvény a második paraméterében megadott időközönként hívja meg az első paraméterben megadott függvényt, mely a rajzolást és az animálást végzi. Amennyiben statikus tartalmat helyezünk el a vásznon, úgy nincs szükség a bizonyos időközönkénti újrarajzolásra, így akkor a *setInterval* helyett egyszerűen csak a saját modellező függvényünket hívjuk. A HelloCube példában azonban folyamatosan animálom a kockát, tehát a modellező függvény egyszeri hívása nem lenne elég. A *webGLStart* függvény tehát a következőképpen néz ki:

```
<script type="text/javascript">
function webGLStart() {
    var canvas = document.getElementById("myCanvas");
    initGL(canvas);
    initShaders();
    initBuffers();
    initTexture();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);
    setInterval(tick, 10);
}
</script>
```

Ezen belül az első lépés a WebGL inicializálása, melyet az *initGL* függvény valósít meg, melynek át kell adnunk az előbb létrehozott vásznat:

```
var gl;
function initGL(canvas) {
        try {
                gl = canvas.getContext("webgl");
                gl.viewportWidth = canvas.width;
                gl.viewportHeight = canvas.height;
        } catch(e) {
        }
        if (!gl) {
```

```
                alert("Could not initialise WebGL");
        }
}
```

Létrehozunk egy változót (a fenti esetben ez a *gl*) a WebGL tartalom számára, majd a vászonból lekérjük bele a tartalmat. Mivel JavaScriptben bármilyen objektumhoz felvehetünk tulajdonságokat (angolul property), így a *gl*-hez is felveszünk kettőt, ezekben tároljuk a vászon szélességét és magasságát. Amennyiben nem sikerül inicializálni a WebGL-t (például mert az adott böngésző nem támogatja), tájékoztató hibaüzenetet írunk ki.

Az *initShaders*, *initBuffers*, *initTexture* függvények és a modellező függvény megírása a következő alfejezetekben kerül tárgyalásra.

1.2. Pufferek

1.2.1. Pufferek létrehozása, feltöltése

A következő fontos lépések egyike a pufferek feltöltése. A pufferekben tárolódik minden adat, amire szükség van az objektum modellezéséhez, és mivel a pufferek a grafikus kártyán tárolódnak, így a modellezés a lehető leggyorsabb, mert a kódban csak azt kell megmondani, melyik puffert szeretnénk használni.

Egy vertexeket, vagyis az objektum (3D esetén térbeli) koordinátáit tároló pufferre mindenképpen szükség van. A puffer létrehozása a *gl.createBuffer()* függvény meghívásával történik, a létrehozása után pedig be kell állítanunk az új puffert aktuálisnak a *gl.bindBuffer* függvény segítségével. Fontos tudni, hogy a puffereken végrehajtódó műveleteknek nem adunk át paraméterként puffert, hanem azok mindig az aktuálisnak beállított pufferen hajtódnak végre. Következő lépésként a puffert egy JavaScript tömbből töltjük fel értékekkel.

A felsorolt lépések láthatók az alábbi kódban (feltételezve, hogy a *vertices* egy korábban már deklarált és értékekkel feltöltött tömb):

```
var vertexPuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
```

Mivel a puffer is JavaScript objektum, így létrehozhatunk neki tulajdonságokat, amit hasznos is megtennünk, mert így eltárolhatjuk, hogy hány vertex található a tömbben, és egy vertex hány számértékkel van megadva. Például egy kocka modellezésekor a kocka egy lapját két háromszögből modellezzük, egy laphoz azonban elég 4 vertexet megadni, mivel a két

érintkező háromszögnek 2 koordinátája közös. Tehát a kocka 6 lapjának definiálására 6x4, vagyis 24 vertex megadására van szükség. Ahhoz pedig, hogy egy pontot térben elhelyezzünk, az x, y és z koordináták megadása szükséges, tehát a kocka egy vertexét három számmal határozzuk meg. Így kapjuk meg, hogy a tömb 24x3-as, és ennek megfelelően beállítjuk a tulajdonságokat:

```
puffer.itemSize = 3;
puffer.numItems = 24;
```

A legegyszerűbb esetben az alakzat modellezéséhez elég a vertexpufferből kiolvasni a megfelelő vertexeket, azonban ha szeretnénk textúrát vagy megvilágítást rendelni a modellhez, akkor szükség van textúra-koordináták, valamint normálvektorok megadására, tárolására. Sőt, létezik egy speciális puffer, az indexpuffer, melyben azt adhatjuk meg, hogyan szeretnénk párosítani egymással az egyes vertexeket – például a kocka esetén több vertex csak egyszer van megadva, és azt, hogy melyik háromszög melyik vertexekből épüljön fel, az indexpufferben adhatjuk meg.

A szín-, textúra- és normálvektor-puffereket a vertexpufferhez hasonlóan hozzuk létre és töltjük fel értékekkel. Az indexpuffer létrehozása annyiban tér el az előzőktől, hogy a típusa *GL.ARRAY_BUFFER* helyett *GL.ELEMENT_ARRAY_BUFFER*, mert csak ilyen típusú puffert használhatunk majd a modellező részben annak megadására, hogy hogyan párosítsa a WebGL az egyes vertexeket. A tömbben egymás után megadott számok az indexpufferben a vertexek sorszámait jelentik, vagyis a vertexek az indexpufferben megadott sorrendben lesznek kirajzolva. Jelenleg azonban csak a puffer létrehozása és feltöltése látható a lenti kódrészletben, a tényleges rajzolás majd a modellező függvényben történik.

```
vertexIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, vertexIndexBuffer);
var vertexIndices = [
      0, 1, 2,      0, 2, 3,    // Front face
      4, 5, 6,      4, 6, 7,    // Back face
      8, 9, 10,     8, 10, 11,  // Top face
      12, 13, 14,   12, 14, 15, // Bottom face
      16, 17, 18,   16, 18, 19, // Right face
      20, 21, 22,   20, 22, 23  // Left face
]
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(vertexIndices),
gl.STATIC_DRAW);
vertexIndexBuffer.itemSize = 1;
vertexIndexBuffer.numItems = 36;
```

Az összes szükséges puffert létrehozhatjuk és feltölthetjük értékekkel például az *initBuffers()* függvényen belül, mely egyszer hívódik meg a program elején, onnantól kezdve pedig csak felhasználjuk a már létező, a grafikus kártyán tárolt értékeket.

1.2.2. Pufferek használata

A puffereket a modellező függvényben használjuk fel. Ennek a függvénynek az elején elvégzünk néhány nézőponttal és perspektívával kapcsolatos beállítást, majd megadjuk a szükséges geometriai transzformációkat – ezekről mind lesz szó a későbbi alfejezetekben. Most a modellezésnek arra a részére térek rá, ahol felhasználjuk a puffereket. Minden puffer használata előtt be kell állítani az adott puffert aktuális puffernek a *gl.bindBuffer()* függvénnyel. Jelen esetben a vertexpuffer lesz az aktuális. A következő lépésben pedig a puffert átadjuk a shader programot tároló objektumnak, amin keresztül a rajzolás történik majd – a shader programról szintén egy későbbi alfejezetben lesz szó.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPuffer);
gl.vertexAttribPointer (shaderProgram.vertexPositionAttribute,
vertexPuffer.itemSize, gl.FLOAT, false, 0, 0);
```

Ezután a *gl.drawArrays* függvény meghívásával adjuk ki a parancsot arra, hogy a pufferek tartalma alapján rajzolódjon ki a modell. A *gl.TRIANGLE_STRIP* azt mondja meg, hogy a pufferben átadott ponthalmazt háromszögekké kell összerakni, mégpedig úgy, hogy miután az első három pontból összeállt a háromszög, mindig egy további pontot vegyen hozzá, és az előző kettő meg az új pont segítségével jöjjön létre a soron következő háromszög. Így például egy négyzet modellezéséhez négy vertexre van szükség, mert két háromszögből áll, és van két olyan vertex, amit mindkét háromszöghöz felhasználunk.

```
gl.drawArrays(gl.TRIANGLE_STRIP, 0, vertexPuffer.numItems);
```

Amennyiben nemcsak a vertexek koordinátáit, hanem a színeiket is át szeretnénk adni, akkor színpuffert is létre kell hoznunk, és át kell adnunk a shadernek. A színpufferben minden egyes vertexhez négy értéknek kell tartoznia: R, G, B és alfa (utóbbi az átlátszóság mértékét adja meg). Ha létrehoztuk a puffert (például az *initBuffers* függvényen belül, a vertexpufferhez hasonló módon), akkor a vertexpuffer átadása után kiválasztjuk aktuálisnak, és a vertexpuffernél látott módon azt is átadjuk a shadernek még a *gl.drawArrays* meghívása előtt:

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
```

```
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute, colorBuffer.itemSize,
gl.FLOAT, false, 0, 0);
```

Hasonló módon történik a textúra-, illetve normálvektor-koordinátákat tároló pufferek átadása is. Az indexpuffer használata kicsit eltér a többitől, azt ugyanis egyrészt nem adjuk át a shadernek, hanem már hamarabb felhasználjuk, másrészt a rajzolásnál nem a *gl.drawArrays* függvényt hívjuk, hanem a *gl.drawElements* nevűt. Ennek a függvénynek megmondjuk, hogy milyen alakzatokat szeretnénk rajzolni az átadott pontokból, az pedig az indexpufferben megadott sorrendben felhasználja a vertexeket a választott alakzat rajzolásához. A rajzoló parancs segítségével rajzolható elemi alakzatok a pont (*POINTS*), a vonalak (*LINES*, *LINE_LOOP*, *LINE_STRIP*) és a háromszögek (*TRIANGLES*, *TRIANGLE_STRIP*, *TRIANGLE_FAN*), ezekből azonban az összes többi összetett alakzat is előállítható, ha megfelelően adjuk meg a vertexeket. Az indexpuffer használata az alábbi kódrészletben látható:

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, vertexIndexBuffer);
gl.drawElements(gl.TRIANGLES, vertexIndexBuffer.numItems, gl.UNSIGNED_SHORT, 0);
```

Az előző alfejezetben látott indexpuffert a rajzoló parancs ismeretében már jobban tudjuk értelmezni: a hármasával megadott indexek az egyes háromszögek kirajzolásához lesznek felhasználva. Az első háromszög a 0., 1. és 2. sorban megadott vertexpufferbeli vertexekből áll össze, a második a 0., 2. és 3. vertexekből. A két háromszög a két közös csúcs mentén egymáshoz illeszkedik, így alkotnak egy négyszöget. A többi háromszögpárból szintén négyszögek állnak majd össze, és ha a vertexek pozíciói megfelelően vannak megadva a háromdimenziós koordináta-rendszerben, akkor kockát kapunk eredményül. (Mind a *TRIANGLES*, mind a *TRIANGLE_STRIP* elemi alakzatok felhasználásával rajzolhatunk kockát, de ha előfordul olyan eset, amikor az egyes háromszögeket külön szeretnénk kezelni – például a kocka különböző oldalai különböző színűek legyenek –, azt csak a *TRIANGLES* használata esetén tehetjük meg, ennek alapján érdemes mérlegelni.)

1.3. Geometriai transzformációk

Modellezésnél fontos, hogy pontosan meg tudjuk adni az alakzatok egymáshoz képesti távolságát, szükség esetén eltoljuk, elforgassuk vagy nagyítsuk/kicsinyítsük az alakzatokat a megadott mértékben. Ezeket a geometriai transzformációkat az OpenGL-hez hasonlóan a WebGL-ben is transzformációs mátrixokkal adjuk meg, amit mi magunk hozunk majd létre és kezelünk, mivel (az OpenGL-lel ellentétben) a WebGL-be ez nincsen beépítve.

Mindig egy egységmátrixból indulunk ki, mely az alapállapotot reprezentálja, vagyis egy olyan transzformációs mátrixot ad meg, ami lényegében semmit nem csinál. Az egységmátrixot szorozzuk meg sorban az egyes transzformációs mátrixokkal, melyek 3D esetén 4x4 dimenziósak, és az összes transzformációs mátrix szorzata adja a model-view mátrixot.

Ám mint említettem, míg OpenGL-ben megtaláljuk az egységmátrix létrehozásához, a transzformációk végrehajtásához szükséges függvényeket, WebGL-ben ezek nem beépített funkciók, ezért nekünk kell őket implementálni, vagy használhatunk segédkönyvtárakat. A megfelelő mátrixokat végül át kell adnunk a shader programnak, ami ez alapján a végső pozíciót ki fogja számolni.

Például egy kocka eltolását és forgatását x és y tengelyek mentén a következő transzformációkkal végezhetjük el (feltételezve, hogy *forgas* egy már korábban létrehozott változó, melynek értéke bizonyos idő elteltével változik, így a kocka forgása folyamatos):

```
loadIdentity();
mvPushMatrix();
    mvTranslate([0.0, 0.0, -1.0])
    mvRotate(forgas, [1, 0, 0]);
    mvRotate(forgas, [0, 1, 0]);
    //kocka modellezése itt
mvPopMatrix();
```

A fenti példában használt, model-view mátrixot manipuláló függvények a glUtils és a Sylvester segédkönyvtárak felhasználásával készültek, mindegyikük egy már korábban létrehozott globális változón, a model-view mátrixot tartalmazó *mvMatrix*-on hajtja végre a megfelelő műveleteket. A következőkben példát adok néhány geometriai transzformáció végrehajtását segítő függvény megvalósításának menetére, de valamennyi függvény implementációja a HelloCube program forráskódjában megtekinthető.

Az *mvPushMatrix* függvényben, amennyiben nem kapott paramétert, akkor a model-view mátrix másolatát a verembe teszem, amennyiben valamely mátrixot kapott paraméterül, akkor annak másolatát teszem a verembe, a mátrixot pedig beállítom az aktuális model-view mátrixnak. A verem egy globális változóként deklarált JavaScript tömb, a másolás és a verembe mentés pedig a Sylvester segédkönyvtár *dup* és *push* függvényeinek segítségével történik.

Az eltolást megvalósító *mvTranslate* függvény egy vektort vár paraméterül, az eltolás értékeit x, y, illetve z tengely mentén (amennyiben háromdimenzióban vagyunk). Ezt a

vektort átadjuk a glUtils segédkönyvtár *Translation* nevű függvényének, mely a megfelelő transzlációs mátrixszal tér vissza. Ezt még meg kell szorozni a model-view mátrixszal, a mátrixok szorzásához a Sylvester által kínált, mátrixokat szorzó függvényt használom fel.

A geometriai transzformációknál fontos figyelni arra, hogy a transzformációk közül először a legbelső hajtódik végre, azután kifelé haladva sorban a többi, tehát esetünkben először az y, majd az x tengely tengely menti forgatás történik meg, csak ezután az eltolás a z tengely mentén. A transzformációk sorrendjét felcserélve pedig más eredményt kapunk, mert a mátrixok szorzása nem asszociatív művelet. Arról is gondoskodni kell, hogy a model-view mátrixot átadjuk a shader programnak:

```
gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, new
Float32Array(mvMatrix.flatten()));
```

1.4. Shaderek

A következő fontos elemei egy WebGL programnak a shaderek. Semmit sem rajzolhatunk addig, amíg nincsen betöltve egy vertex shader és egy fragment shader (másik nevén pixel shader), ezek megléte ugyanis alapvető követelmény a rendereléshez. A shaderek sok mindenben segítségünkre lehetnek, mert nagyon sokféle változtatást tudnak végrehajtani a modellen még a renderelés előtt, és nagy előnyük, hogy a grafikus kártyán futnak, így a változtatásokat gyorsan hajtják végre.

1.4.1. Egyszerű vertex és fragment shader

A vertex shader program meghatározza, milyen műveleteket kell végrehajtani az egyes vertexeken. Bemenete lehet *attribute*, *uniform* vagy *sampler* típusú változó, kimenete pedig *varying* típusú változó. Az *attribute* típusú változó pontonkénti adatok (pl. vertextömbök) tárolását támogatja, a *uniform* típusúba a vertex shader által használt konstans kerül. A *sampler* egy speciális típus, mely textúrákat reprezentál. A vertex shader outputja ezeknek a bemenő változóknak a segítségével áll elő, és ezt a minden fragmensre lineáris interpolációval számolt outputot kapja majd a fragment shader bemenetként.

A vertex shadert vertex-alapú műveletekhez használjuk, amilyen például a geometriai transzformálás mátrix segítségével, a vertexenkénti színszámítás, a textúra-koordináták generálása, transzformálása. Mind a vertex, mind a fragment shader GLSL (OpenGL Shading Language) nyelven íródik, melyet a HTML kódba ágyazunk be. Egy egyszerű vertex shader a következőképpen néz ki:

```
attribute vec4 vPosition;
void main()
{
 gl_Position = vPosition;
}
```

Először létrehozunk egy *attribute* típusú változót az egyes vertexek számára, ez egy 4 komponensű vektor, hiszen 3D koordináta-rendszerben 3 számmal adjuk meg a vertex helyét, a negyedik pedig a szín tárolásához használható. A *gl_Position* nevű, impliciten mindig létrejövő változó a vertex shader outputja lesz, mely esetünkben nagyon egyszerűen számolódik, hiszen csak átadjuk neki az aktuális értékeket. Ezt a *main* függvényen belül tesszük meg, mivel az a shader program belépési pontja.

Geometriai transzformációk és perspektíva beállítása esetén azonban a megfelelő mátrixokkal való szorzásról is gondoskodni kell. A mátrixok *uniform* változóként adódnak át a shadernek (az átadásukról az 1.4.3-as alfejezetben lesz szó bővebben), és minden egyes vertex pozícióját beszorozzuk a megfelelő mátrixok, a lenti példában a model-view és a projekciós mátrix értékével, így kapjuk meg a végleges pozíciót.

Ha színt is szeretnénk kezelni, akkor a szín értékét is minden egyes vertex esetén át kell adni – a példában ez az *aVertexColor* változóba fog kerülni, mely értékül adódik a *vColor-nak*, a vertex shader egyik outputjának (a másik output itt is az alapértelmezetten mindig létrejövő *gl_Position*). Az alábbi példában az előzőnél kissé bonyolultabb, geometriai transzformációkat, perspektívát és színeket kezelő vertex shader látható:

```
attribute vec3 aVertexPosition;
attribute vec4 aVertexColor;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
varying vec4 vColor;
void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vColor = aVertexColor;
 }
```

A fragment shader az egyes fragmensek rajzolását határozza meg, inputja a vertex shader outputja. Egy egyszerű fragment shader a következőképpen néz ki:

```
precision mediump float;
void main()
{
 gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
```

```
}
```

Ahogy az előző esetben a vertex shadernél, úgy a program belépési pontja itt is a main függvény. A *gl_FragColor* egy speciális beépített változó, mely a végleges színt tartalmazza az egyes fragmensekre. Esetünkben minden fragmens színe piros lesz, a shader így nagyon egyszerű, viszont csak piros színű alakzatokat tudunk modellezni.

A második vertex shaderhez tartozó fragment shader alább látható. Ezzel már az általunk megadott színnel jeleníthetjük meg a különféle alakzatokat:

```
precision highp float;
varying vec4 vColor;
void main(void) {
    gl_FragColor = vColor;
}
```

Ami a vertex shadernél output volt, itt az az input, vagyis a *vColor*. A fragment shaderben pedig nem teszünk mást, csak értékül adjuk a már kiszámolt színeket minden egyes fragmensnek.

Látható tehát, hogy a modellezéshez fontos információk végül mind a shaderhez kerülnek, ami felhasználja őket a végleges modell kirajzolásához. Az itt bemutatott egyszerű shadereket ennek megfelelően tovább bővíthetjük attól függően, hogy milyen információkat szeretnénk feldolgoztatni velük (pl. megvilágítás, átlátszóság jellemzői).

1.4.2. A GLSL nyelv

A GLSL nyelv szintaxisa sokban hasonlít a C nyelvéhez, ám számos jelentős különbség van a két nyelv között. A GLSL nyelvben a változók három típusba sorolhatók: skalár, vektor vagy mátrix. Skalár változók a *float*, *int* és *bool* típusok, minden egyéb változó vagy vektor, vagy mátrix, ezekből azonban sokféle áll a programozó rendelkezésére (lebegőpontos, integer, illetve boolean egy-, két-, három- és négykomponensű vektorok, 2x2-es, 3x3-as, illetve 4x4-es mátrixok), valamint minden beépített változónak van konstruktora.

A terjedelemre való tekintettel részletesebben nem térnék ki erre a nyelvre, amennyit szükséges tudni róla a WebGL kapcsán, azt a shaderek írásánál megemlítem. További információkért ajánlom Aaftab Munshi – Dan Ginsburg – Dave Shreiner OpenGL ES 2.0 Programming Guide című könyvét [1], melyből az információkat merítettem, és ahol egy teljes fejezetben tárgyalják.

1.4.3. Shaderek használata a WebGL programban

A shadereket tehát megírtuk GLSL nyelven, ezután beszúrhatjuk szkriptként a HTML kódba:

```
<script id="shader-fs" type="x-shader/x-fragment">
<!--ide kerül a fragment shader kódja -->
</script>
```

A fenti példa a fragment shader beszúrását mutatja, de a vertex shadernél is hasonlóan kell eljárni, csak ott a szkript típusa *x-shader/x-vertex* lesz. Az *id* mindkettőnél tetszőleges, ez alapján fogunk hivatkozni rájuk.

Ha visszalapozunk a fejezet első alpontjában említett *webGLStart* függvényhez, láthatjuk, hogy a puffereket inicializáló függvény mellett egy shadereket inicializáló függvényt is meghívunk, ez az *initShaders*. Ez a függvény a következőket tartalmazza: először id alapján lekérjük a két shadert egy-egy változóba, ehhez felhasználjuk a *getShader* függvényt, ami a HTML kódban a megadott id alapján megtalálja azokat a szkripteket, amik a shaderek kódját tartalmazzák.

```
var fragmentShader = getShader(gl, "shader-fs");
var vertexShader = getShader(gl, "shader-vs");
```

A *getShader* függvényben az alapján, hogy a szkript típusa *x-shader/x-fragment* vagy *x-shader/x-vertex*, a *gl.createShader* függvény meghívásával létrehozzuk a megfelelő shadert:

```
var shader = gl.createShader(gl.FRAGMENT_SHADER);
```

Vertex shader esetén pedig:

```
var shader = gl.createShader(gl.VERTEX_SHADER);
```

A *gl.shaderSource* függvénnyel átadjuk a forráskódot is a shadernek, majd a *gl.compileShader-rel* lefordítjuk, így innentől kezdve már a grafikus kártyán fog futni a kód. (A *getShader* függvény teljes kódja a függelékben található.)

A shaderek létrehozása után a *createProgram* függvény segítségével létrehozunk egy olyan programobjektumot, ami a WebGL része, és a grafikus kártyán képes futni, majd társítjuk hozzá a shadereket. Fontos ismét hangsúlyozni, hogy egy programhoz egyetlen vertex shader és egyetlen fragment shader társítható. Ezután a *linkProgram* függvény segítségével belinkeljük a shadert – ez a művelet akkor lehet sikertelen, ha a fordítás nem sikerült, vagy ha nem pont egy vertex és egy fragment shader van, valamint ha a GLSL-ben

megengedettnél több változót használunk. Végül a *useProgram* függvény segítségével megadjuk, hogy a létrehozott programobjektumot kívánjuk használni a programunkhoz. (A programobjektumot érdemes globális változóként deklarálni már a program elején.)

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

Ha eddig eljutottunk, akkor már csak az a dolgunk, hogy a megfelelő pufferek (pl. vertex-, textúra-, normálvektor pufferek) és mátrixok (pl. model-view mátrix, projekciós mátrix) értékeit átadjuk a shader programnak. A vertexpuffer átadása például a következőképpen néz ki:

```
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
"aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
```

Mivel a *shaderProgram* JavaScript objektum, így bármikor hozzáadhatunk tetszőleges tulajdonságokat, ahogy ezt meg is tesszük a fenti kódban, amikor létrehozunk egy tulajdonságot a vertexpozíciók számára. Az *aVertexPosition* a vertex shader programban a változó neve lesz, az *enableVertexAttribArray* függvénnyel pedig ténylegesen engedélyezzük a shaderben a puffer tartalmának felhasználását, jelezve azt is, hogy tömbből fogjuk olvasni az adatokat. Hogy az átadott puffer tényleg a vertexeket tartalmazza-e, az természetesen a mi felelősségünk, a hozzárendelés a modellezés részben történik a *gl.vertexAttribPointer* függvény segítségével (ezt a korábbi, pufferekről szóló alfejezetben már láthattuk).

Bármilyen puffer tartalmát hasonlóan kezelhetjük, mint a vertex pufferét az előbb bemutatott példában, mert a pufferek pontonkénti adatot tartalmaznak, így belőlük a shader programban *attribute* típusú változók lesznek.

A model-view és a projekciós mátrix, mivel konstans adatokat tartalmaznak, *uniform* változók lesznek a vertex shaderben. A következő példa a model-view mátrix értékének átadását mutatja be:

```
shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");
```

A modellezés részben jelöljük ki, hogy melyik mátrixot jelöli a fent létrehozott *mvMatrixUniform* tulajdonság, mely *uMVMatrix* néven kerül majd a vertex shaderhez (az *mvMatrix* esetünkben a mátrixot tartalmazó változó, mely már korábban létre lett hozva):

```
gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, new
Float32Array(mvMatrix.flatten()));
```

A többi mátrix átadása is a bemutatott példához hasonlóan történik. A megfelelő értékek átadása után pedig már nincsen több teendőnk a shaderrel.

1.5. Textúrák

A textúrázás WebGL-ben lényegében egy speciális módja az egyes pontok színbeállításának. A textúraként használni kívánt képet betöltjük a programunkba, majd átadjuk a fragment shadernek, ami a végleges színt fogja kiszámolni. Hogy a kép melyik képpontját szeretnénk az egyes fragmensekhez használni, azt az információt a textúrapuffer segítségével adjuk át a fragment shadernek. Textúrát a következőképpen tölthetünk be:

```
var textura = gl.createTexture();
textura.image = new Image();
textura.image.onload = function() {
      handleLoadedTexture(textura)
}
textura.image.src = "minta_csikos.gif";
```

A *gl.createTexture* függvény csak egy textúrára mutató referenciát hoz létre, ezután JavaScript függvénnyel hozzuk létre a kép objektumot, melyet az image tulajdonságba teszünk bele. A betöltődés eseményéhez beállítunk egy callback függvényt (amit majd szintén megírunk), majd megadjuk a kép forrását, esetünkben relatív útvonallal.

Amint a betöltődés befejeződik, meghívódik a textúra kezelésére általunk írt függvény (mely az előbb létrehozott textúra változót várja paraméterként). Ennek tartalmára egy példa az alábbi kódrészlet, melyet igényeinknek megfelelően lehet alakítani:

```
gl.bindTexture(gl.TEXTURE_2D, texture);
```

Első lépésként a *gl.bindTexture* függvény segítségével kiválasztjuk az átadott textúrát aktuálisnak – a puffereknél már láttunk ehhez hasonló parancsot. Ezután az összes további textúrára vonatkozó művelet az aktuálisnak kiválasztott textúrán fog végrehajtódni.

```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

A következő paranccsal azt adjuk meg, hogy a betöltött képet vertikális irányba forgassa át, ez lényegében a kényelmünket szolgálja, mert így az általunk megszokott koordináta-rendszer szerint adhatjuk majd meg a pozíciókat, vagyis az Y tengelyen felfelé nőnek az

értékek, lefelé csökkennek (ellentétben például a GIF formátum által használt lefelé növekvő vertikális tengellyel).

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, texture.image);
```

Következő lépésként átadjuk a betöltött képet a grafikus kártyának, a paraméterekben pedig megadjuk a tulajdonságait, valamint azt, hogy a grafikus kártyán milyen formában szeretnénk tárolni, az utolsó paraméter pedig maga a kép.

```
 gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

Ezt követően a textúra nagyítására, illetve kicsinyítésére vonatkozó módszereket adjuk meg. Többféle módszer létezik, egyesek nagyításnál adnak szebb eredményt, mások kicsinyítésnél, de ettől még tény marad, hogy minőségromlás nélkül nem lehet egy kép méretét megváltoztatni, tökéletes eredményt egyik sem ad.

A *gl.NEAREST* filter kicsinyítésnél elfogadható eredményt ad, nagyításnál viszont nagyon blokkos lesz a kép, viszont ez a leggyorsabb, mivel mindig az eredeti kép legközelebb eső pontját veszi, semmit nem számol. A *gl.LINEAR* az eredeti képpontok közötti lineáris interpolációval számolja ki a megváltozott méretű kép pontjait, nagyításnál a *gl.NEAREST*-nél láthatóan szebb képet ad. A legszebb eredményt azonban a *gl.LINEAR_MIPMAP_NEAREST* filter adja, cserébe ez a legösszetettebb módszer. Lényegében az történik, hogy a textúra különböző méretekben (az eredeti fele, negyede, nyolcada) eltárolódik, ezek összességét nevezzük mipmapnek. Ilyen jó minőségű, pixelről pixelre történő kicsinyítés futási időben nagyon lassú lenne, ezért végződik el már előre. Nagyítás és kicsinyítés esetén ez a módszer is lineáris interpolációval számol, de nem az eredeti képből indul ki, hanem a mipmapből kiválasztja a méretben legközelebbi képet. Ha mipmap filterezést választunk, akkor a filter kiválasztásán felül egy másik parancsot is ki kell adni a mipmap generálására:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_NEAREST);
gl.generateMipmap(gl.TEXTURE_2D);
```

Végül az aktuális textúrát null-ra állítjuk – ez a lépés csak afféle takarítás, nem feltétlenül szükséges. Ha minden textúrákat manipuláló utasítás használata előtt kiválasztjuk az aktuális textúrát, akkor az elhagyása semmilyen változást nem okoz az eredményben.

```
gl.bindTexture(gl.TEXTURE_2D, null);
```

A puffereket feltöltő részben a textúrapufferben minden vertexhez két számot kell megadni (x és y koordinátát). A textúra kifeszítését az alakzaton az OpenGL-hez hasonlóan adhatjuk meg: (0,0) a bal alsó sarok, (1,1) a jobb felső. A kép valódi felbontását a megadottak alapján a WebGL számolja ki.

A létrehozott textúrapuffert átadjuk a shadernek, ahogy arról a puffereknél már volt szó, majd még az alakzatot kirajzoló parancs (*gl.drawArrays* vagy *gl.drawElements*) előtt aktívvá tesszük a 0. (vagy a soron következő) textúra használatát, kiválasztjuk, melyik textúra legyen az aktuális, és azt is átadjuk a shadernek:

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, textura);
gl.uniform1i(shaderProgram.samplerUniform, 0);
```

A WebGL maximum 32 textúrát tud kezelni egyszerre, ezekre TEXTURE0-tól TEXTURE31-ig a megfelelő azonosítóval hivatkozhatunk. A megfelelő azonosító sorszámát a *gl.uniform1i* függvénynek adjuk át második paraméterként. Ennek a sampler értékek átadására szolgáló függvénynek a segítségével adódik át a textúra a shadernek *uniform* változóként – ahogy ezt a transzformációs mátrixok esetében már láthattuk, a textúrák azonban nem általános *uniform* típusú változók lesznek, hanem azon belül a speciális *sampler* változók közé fognak tartozni, ami kimondottan textúrák kezelésére való.

Miután a shadernek átadtuk a textúrára vonatkozó összes információt, belekódoltuk a shaderbe, hogy hogyan dolgozza fel ezeket, további teendőnk nincsen, a végső fragmensek jellemzőinél a textúra jellemzői is figyelembe lesznek véve.

1.6. Megvilágítás

Az OpenGL-lel ellentétben a WebGL-ben semmi beépített támogatás nincsen a megvilágítással kapcsolatban, de a shaderek segítségével mi magunk megvalósíthatjuk, ahogy az a geometriai transzformációknál is történt.

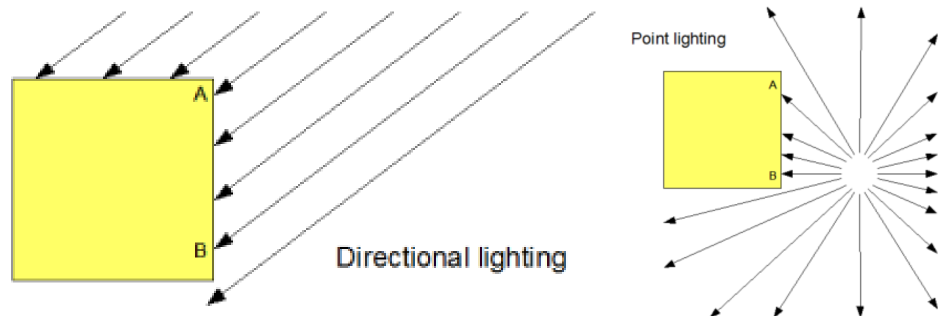
Ha megvilágítással szeretnénk bővíteni a modellünket, akkor az annyit jelent, hogy definiálunk fényforrásokat, melyek maguk nem látszanak, viszont a hatásuk igen: a testek megfelelő felületei világosak, illetve sötétek. Tehát amikor a shader az egyes pontok végleges színét kiszámítja, akkor hozzá kell venni a számításhoz, hogyan változtat a fény a színen. Alapvetően kétféleképp kezelhetjük ezt. Az egyik esetben vertexenként számítjuk a színt (per-vertex módszer), a vertexek között pedig lineáris interpolációval, a másik módszernél pedig

minden pixelre külön számoljuk ki a színt (per-fragment vagy per-pixel módszer). Előbbi gyorsabb, viszont hajlított felületekre csak a per-fragment módszer ad realisztikus eredményt.

1.6.1. Phong modell

Mielőtt a megvilágítás tényleges WebGL megvalósításra rátérnék, röviden bemutatom annak elméleti hátterét és a Phong modellt, ugyanis ezen alapszik az egész.

A számítógépes grafika világában kétféle fényt különböztetünk meg. Az egyik esetében a meghatározott irányból jövő fényforrás messze van, ilyenkor a fény egyformán verődik vissza a megvilágított tárgy felületéről. Ezt nevezzük direkcionális fénynek, példa rá a nap fénye. A másik esetben a pontszerű fényforrás közel van a tárgyhoz, a fény iránya és a visszaverődés a megvilágított objektum és a fény pozíciójától függ. Ezt pozicionális fénynek nevezzük, példa rá a lámpa fénye. A direkcionális és pozicionális fény közti különbséget az 1.1-es ábra szemlélteti.

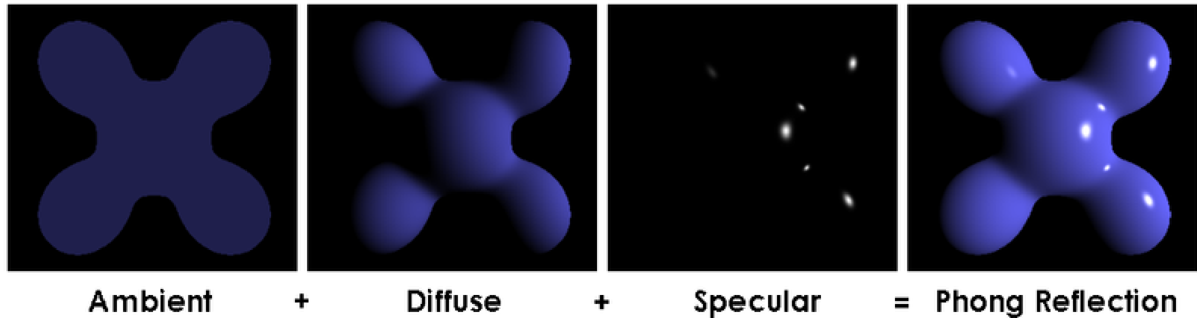


1.1. ábra: direkcionális és pozicionális fény – forrás: www.learningwebgl.blog.com

Egy másik csoportosítás szerint különbséget tehetünk ambiens, diffúz és spekuláris fény között. Az ambiens fénynek nincsen konkrét iránya, mindenhol ott van és mindent megvilágít, ahogyan például borús időben nem látszik a nap, árnyékok sincsenek, a tárgyakról visszaverődő fény miatt azonban mégis olyan, mintha minden egyforma mértékben meg lenne világítva. Diffúz fény esetén adott irányból jön a fény, és a fény iránya, valamint a test normálvektora alapján számítjuk ki a visszaverődést. Spekuláris fény esetében ezen felül az is fontos, hogy hol van a nézőpontunk, a fény irányától, a test nomálvektorától, a nézőpont pozíciójától és a test anyagának jellemzőitől függ a visszatükröződés.

A Phong-féle modell lényege, hogy minden anyagnak négy tulajdonsága van: RGB értékek az ambiens, diffúz és spekuláris fényhez, amit visszavernek, valamint a tárgy fényessége, ami befolyásolja a spekuláris visszaverődést. A fényeknek két tulajdonságuk van, az általuk előállított diffúz, illetve spekuláris RGB értékek. Adott pont színe a modellben tehát a fény színének, az anyag színének és a fényhatásnak a kombinációjaként áll elő. Tehát

ha kiszámoljuk a megfelelő színértékeket egy adott pontra külön-külön ambiens, diffúz, illetve spekuláris fény esetében, végül pedig ezeket összegezzük, akkor megkapjuk a pont végleges színét. Ezt szemlélteti az 1.2. ábra.

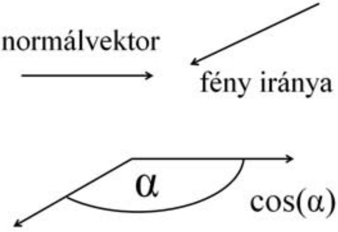


1.2. ábra: Phong-féle megvilágítási modell - forrás: Wikipedia

1.6.2. Egyszerű megvilágítást kezelő shaderek írása

Amennyiben a fényforrásunk nem mozog és az adott sugarak iránya állandó, akkor erről az információt *uniform* vektorváltozókban tárolhatjuk, és ezt átadhatjuk a shadernek. A megvilágítás számításához azonban nemcsak a fénysugarak irányát szükséges tudni, hanem azt is, hogy azok milyen szögben érik az egyes testek felületét, tehát tárolnunk kell minden felületről annak orientációját. Erre a legjobb módszer háromdimenziós térben normálvektor tárolása minden egyes vertexhez.

Ha adva van mindkét vektor, a fény irányának a vektora és a test adott vertexének normálvektora, akkor ebből kiszámolhatjuk a visszaverődés mértékét – az 1.3. ábrán látható módon vesszük a két vektor cosinusát:



1.3. ábra: visszaverődés számítása a fény irányvektora és a test normálvektora alapján

Amennyiben a vektorok által bezárt szög nagyobb 90-nél, akkor a cosinus negatív lenne, aminek természetesen nincs értelme, mivel negatív mértékű visszaverődés nincsen, tehát ilyen esetben a visszaverődést nullának tekintjük. A két vektor közötti szög cosinusának kiszámolására létezik függvény a shader nyelvben (*dot* függvény), ezt felhasználva egy ambiens és direkcionális megvilágítást kezelő vertex shader a következőképpen nézhet ki: először deklaráljuk a használni kívánt változókat, melyeknek majd a JavaScriptben deklarált

shader programobjektum segítségével adunk értéket a már korábban látott módon. A vertex- és textúrakoordináták tárolása mellett ezúttal szükség lesz a normálvektor koordináták tárolására is:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
attribute vec2 aTextureCoord;
```

A perspektívát és a geometriai transzformációkat tároló mátrixok mellé szintén fel kell venni még egy mátrixot a normálvektornak, hiszen a geometriai transzformációk során a normálvektor is változik:

```
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
```

Az egyes *uniform* változók az ambiens és direkcionális fény RGB értékeinek tárolására szolgálnak, illetve direkcionális fénynél az irányt is tároljuk. A logikai típusú változó segítségével a programunkban engedélyezhetjük, illetve letilthatjuk majd a megvilágítást.

```
uniform vec3 uAmbientColor;
uniform vec3 uLightingDirection;
uniform vec3 uDirectionalColor;
uniform bool uUseLighting;
```

Az implicit pozíciót tároló változó mellé további két output változót definiálunk, melyekbe a textúra és a megvilágítási információk kerülnek majd a fragment shader számára.

```
varying vec2 vTextureCoord;
varying vec3 vLightWeighting;
```

A vertex shader lényegi részében kiszámoljuk a vertexek és textúrák pontos pozícióit, és ha a megvilágítás nincsen engedélyezve, akkor lényegében semmit nem változtatunk a színeken. Ha engedélyezve van, akkor ennél jóval több dolog történik: először a geometriai transzformációktól függően kiszámoljuk a normálvektorok aktuális helyét. Ezután a már említett *dot* függvénnyel kiszámoljuk a normálvektor és a fénysugár vektora közötti szög cosinusát. Annak érdekében, hogy a visszaverődésre sose kapjunk negatív értéket, ezt az eredményt a max függvénynek adjuk paraméterül a 0 érték mellé. A végeredménnyel súlyozzuk a direkcionális fény színértékeit, majd a Phong modellnek megfelelően összegezzük az egyes fények hatásait, esetünkben az ambiensét és a direkcionálisét.

```
void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vTextureCoord = aTextureCoord;
    if (!uUseLighting) {
      vLightWeighting = vec3(1.0, 1.0, 1.0);
    } else {
      vec4 transformedNormal = uNMatrix * vec4(aVertexNormal, 1.0);
      float directionalLightWeighting = max(dot(transformedNormal.xyz,
uLightingDirection), 0.0);
      vLightWeighting = uAmbientColor + uDirectionalColor *
directionalLightWeighting;
    }
  }
```

Ha pozicionális fényt is szeretnénk kezelni, akkor további változókkal bővül a shader program, ugyanis a pozicionális fénynek is van színe és pozíciója:

```
uniform vec3 uPointLightingLocation;
uniform vec3 uPointLightingColor;
```

A fény iránya a fényforrás és a megvilágított test pozíciójából számolható, ezután pedig a test normálvektorát és a fény irányvektorát felhasználva ugyanúgy számolunk, mint az előbb a direkcionális fény esetén:

```
vec3 lightDirection = normalize(uPointLightingLocation - mvPosition.xyz);
float specularLightWeighting = max(dot(transformedNormal, lightDirection), 0.0);
```

Az eredményül kapott súllyal megszorozzuk a pozicionális fény színértékeit, és ezt is hozzáadjuk a végső értékhez.

A fragment shader minden esetben ugyanúgy néz ki, ha megvilágítást kezelünk, mert hozzá már csak a végső érték kerül (*vLightWeighting)*, amit figyelembe veszünk a fragmensek végleges színének kiszámolásakor. A *vLightWeighting* egy háromelemű vektor, mely a vertex shader által a fényekre kiszámított vörös, zöld és kék értékeket tartalmazza, ezt beleszámoljuk a *gl_FragColor* változó értékébe minden egyes fragmens esetén.

```
varying vec2 vTextureCoord;
varying vec3 vLightWeighting;
uniform sampler2D uSampler;
void main(void) {
       vec4 textureColor = texture2D(uSampler, vec2(vTextureCoord.s,
vTextureCoord.t));
     gl_FragColor = vec4(textureColor.rgb * vLightWeighting, textureColor.a);
}
```

Az itt bemutatott shaderek a per-vertex módszer alapján számolnak, ami viszonylag gyors, ám gyakran nem ad realisztikus eredményt. A per-pixel módszer jóval számításigényesebb, de végül sokkal szebb és valósághűbb modellt kapunk eredményül. Ezt a módszert a következő fejezet egyik példájában mutatom majd be részletesen.

1.6.3. A megvilágítás jellemzőinek megadása JavaScriptben

A megvilágítás kezelését tehát lényegében a shaderek végzik el, azonban a JavaScript kódot is bővítenünk kell egy kicsit. Minden testhez szükség van a normálvektorokat tároló pufferre, valamint ennek tartalmát az eddig megismert pufferek tartalmához hasonlóan átadjuk a shadernek, a hozzá tartozó mátrix tartalmával együtt, ami a transzformációkat kezeli.

Ezen felül a megfelelő fényekhez tartozó adatokat is át kell adni a shadereknek a shader programobjektumon keresztül. Ehhez először a shader programobjektumhoz felvesszük a megfelelő tulajdonságokat, és mindegyikhez megadjuk, hogy annak tartalma a shader melyik változójába kerüljön majd:

```
shaderProgram.useLightingUniform = gl.getUniformLocation(shaderProgram,
"uUseLighting");
    shaderProgram.ambientColorUniform = gl.getUniformLocation(shaderProgram,
"uAmbientColor");
    shaderProgram.lightingDirectionUniform = gl.getUniformLocation(shaderProgram,
"uLightingDirection");
    shaderProgram.directionalColorUniform = gl.getUniformLocation(shaderProgram,
"uDirectionalColor");
```

A modellező függvényen belül pedig tartalommal töltjük fel az egyes tulajdonságokat:

```
gl.uniform1i(shaderProgram.useLightingUniform, lighting);
```

A *lighting* egy logikai változó, amit a felhasználótól is bekérhetünk, ezzel interaktívvá téve a programot. A shaderben csak akkor lesz szükségünk a többi megvilágítással kapcsolatos változóra, ha ez a logikai változó igaz, ezért csak ebben az esetben kell a többi változónak is értéket adnunk:

```
if (lighting) {
gl.uniform3f(shaderProgram.ambientColorUniform, red, green, blue);

var lightingDirection = Vector.create([directionX, directionY, directionZ]);
var adjustedLD = lightingDirection.toUnitVector().x(-1);
var flatLD = adjustedLD.flatten();
gl.uniform3fv(shaderProgram.lightingDirectionUniform, flatLD);
```

```
gl.uniform3f(shaderProgram.directionalColorUniform, directionalR, directionalG,
directionalB);
}
```

A *red, green, blue, directionX, directionY, directionZ, directionalR, directionalG, directionalB* változókat szintén előre definiáltnak tekintem, melyeknek értékeit akár a felhasználó is állíthatja, vagy mi magunk beállíthatjuk korábban. A megfelelő színértékeket át kell adni ambiens és direkcionális fény esetén is, direkcionális fénynél emellett még a fény irányára vonatkozó tulajdonságot is fel kell tölteni értékekkel. Ezen néhány változtatást végrehajtunk a glMatrix segédkönyvtár felhasználásával, mielőtt átadnánk a shadereknek: normalizáljuk a vektor hosszát, hogy egységnyi legyen (a vertex shaderben egységnyi hosszú vektorokkal dolgozunk), majd megszorozzuk -1-gyel, hogy megfordítsuk az irányát. Erre azért van szükség, mert mi az alapján definiáljuk a fényt, hogy merre tartanak a sugarak, a számításoknál azonban azt vesszük alapul, hogy honnan jönnek.

Ha pozicionális fényt is használunk, akkor ahhoz további két tulajdonságot veszünk fel, egyiket a színértékekkel, másikat a fényforrás pozíciójának koordinátáival töltjük fel.

1.7. Átlátszóság

Akárcsak az OpenGL, a WebGL is a mélységpuffer segítségével dönti el, hogy az egymás takarásában levő testek közül melyik rész látszódjon, és melyik legyen eltakarva. Már volt szó arról, hogy a fragment shader minden egyes fragmensre visszaadja a végleges színt, ami a képpufferbe kerül (ennek tartalma jelenik majd meg a képernyőn), emellett a képpufferbe minden egyes fragmenshez egy mélységérték is tartozik, ami lényegében az általunk megadott z koordinátától függ. Már a legelső WebGL programunk is tartalmazta a *gl.enable(gl.DEPTH_TEST)* parancsot, ezzel engedélyezzük a mélységpuffer használatát, vagyis a WebGL eldönti, hogy a soron következő új pont az eddigiek elé vagy mögé kerüljön.

Amikor egy testet átlátszóvá szeretnénk tenni valamilyen mértékben, akkor a test mögött levő, általa eltakart testnek is látszania kell olyan mértékben, amilyen mértékben átlátszó az őt takaró test. Ahhoz, hogy ezt meg tudjuk valósítani, először is ki kell kapcsolni a mélységpuffert, és blendinget kell használni helyette, amit tekinthetünk egy másik módszernek az egymás takarásában levő fragmensek kezelésére. Ekkor minden fragmenshez megadunk egy alfa értéket is, ami az átlátszóságát jellemzi (0-tól 1-ig terjed az értéke, 0 jelenti a teljesen átlátszó, 1 a teljesen átlátszatlan testet).

Első lépésként beállítjuk, hogy a blending milyen módszert használjon, ez történik a következő kódrészletben:

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE);
```

A *gl.blendFunc* függvény két paramétert vár, a forrás- és a célfaktort. A forrásfaktor a forrásfragmenshez tartozik, ez az, amit épp aktuálisan rajzolunk, míg a célfaktor a célfragmenshez, ami már a képpufferben van. Ebben a példában a forrásfaktor a forrásfragmens alfa értéke, a célfaktor pedig konstans egy. (A *gl.SRC_ALPHA* és a *gl.ONE_MINUS_SRC_ALPHA* használata forrás-, illetve célfaktorként némileg javíthat majd a kapott eredményen.) Ezek a faktorok lényegében azt adják meg, hogy milyen súllyal kell beszorozni az adott fragmens adott színkomponensét.

Tegyük fel, hogy a forrásfragmens RGBA értékei rendre R_s, G_s, B_s, A_s, a célfragmensé R_d, G_d, B_d, A_d, a forrásfaktoré S_r, S_g, S_b, S_a, a célfaktoré D_r, D_g, D_b, D_a. Ekkor a végső színt a vörös komponens esetén a következő módon számolja a WebGL:

$$R = R_s * S_r + R_d * D_r \text{ (1.1. képlet)}$$

A többi komponensnél is hasonlóan történik a számolás a megfelelő értékek felhasználásával. A következő lépés a blending engedélyezése, és a mélységpuffer letiltása, majd a shader programobjektum segítségével a már ismert módon átadjuk az alfa értékét a fragment shadernek (a példában alfa értéke 0.5, vagyis 50%-ban lesz átlátszó a test):

```
gl.enable(gl.BLEND);
gl.disable(gl.DEPTH_TEST);
gl.uniform1f(shaderProgram.alphaUniform, 0.5);
```

Tehát az alfa értéke a fragment shaderhez kerül *uniform* valós változóként (a példában *uAlpha*), és ott a végleges szín kiszámításánál vesszük figyelembe:

```
gl_FragColor = vec4(textureColor.rgb * vLightWeighting, textureColor.a * uAlpha);
```

A blending nem teljesen ugyanaz, mint az átlátszóság, de nagyon jó módszer az átlátszóság-hatás elérésére. Azonban ez a módszer csak akkor ad szép eredményt, ha megvilágítást használunk, és az átlátszóság valósághűsége függhet a beállított fényektől. Még egy fontos szabályt érdemes betartani, ha átlátszó és átlátszatlan testek is vannak a modellben: először a teljesen átlátszatlan testeket modellezzük, csak utána a valamilyen mértékben átlátszókat.

1.8. Perspektíva beállítása

A modellezés végén születő látványt a nézőpont és a perspektíva beállítása határozza meg, ezeknek a beállításait még a rajzolás megkezdése előtt elvégezzük. A *gl.viewport* függvénnyel a vászon méretét adjuk meg a *gl* objektumhoz már az inicializálásnál felvett tulajdonságok segítségével, a következő lépés pedig a vászon „letakarítása” a rajzolás előtt, ami lényegében a szín- és mélységpufferek kiürítését jelenti.

```
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Fontos lépés annak beállítása, hogy milyen perspektívából szeretnénk látni a modellt. A WebGL alapértelmezetten ortografikus projekciót használ, ami azt jelenti, hogy a közelebb elhelyezkedő elemek ugyanolyan méretűek lesznek, mint a távolabbiak. Ahhoz, hogy a megjelenítést realisztikusabbá tegyük, beállíthatjuk, hogy milyen szögből lássunk rá a modellre, valamint hogy bizonyos távolságnál (a példakódban 0.1 egység) közelebbi és bizonyos távolságnál távolabbi tárgyakat (a példakódban 100 egység) ne lássunk. Ezen kívül megadjuk a vászon szélesség-magasság arányát is. Ahogy a transzformációs mátrixok kezelése sincsen beépítve a WebGL-be, úgy a projekciós mátrix kezelése sem, viszont használhatunk előre megírt segédkönyvtárat. A példában a *glUtils* segédkönyvtárából a *makePerspective* függvényt felhasználva írtam a *perspective* nevű függvényt, mely lényegében csak az előbb említett segédfüggvény által visszaadott mátrixot menti el egy globális változóba, melyet a perspektíva tárolására hoztam létre. Ennek a függvénynek a segítségével a perspektíva a következőképpen állítható be:

```
perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
```

1.9. Renderelés

Ebben az alfejezetben röviden felvázolom, hogyan történik a renderelés WebGL-ben. Minden egyes alkalommal, amikor kiadunk egy rajzolási parancsot, az adatok *attribute* vagy *uniform* változók formájában továbbítódnak a vertex shader felé, de míg az *attribute* változók értéke mindig az aktuális vertex adatait tartalmazza, addig a *uniform* változók értéke nem változik hívásról hívásra. A vertex shader elvégzi a megfelelő számításokat az adatokon, és ún. *varying variable* változókat ad eredményül – egy változót mindenképpen eredményül kell adnia, ez az impliciten minden vertex shaderben deklarált *gl_Position*, mely tartalmazza a vertexek végleges koordinátáit.

A vertex shader után a következő lépés a primitív assembly. A primitív egy olyan objektum, mely a megfelelő rajzolási parancsokkal 2D-ben lerajzolható, mint például egy vonal vagy egy háromszög. Ebben a szakaszban a vertexek ilyen elemi egységként lerajzolható alakzatokból állnak össze. Itt dől el az is, hogy mi fog látszani a látótérben és mi nem. Ehhez a szakaszhoz szorosan kapcsolódik a következő, a raszterizáció, melynek során a megfelelő primitívek ténylegesen kirajzolásra kerülnek, vagyis a primitív kétdimenziós fragmensek halmaza lesz.

A kétdimenziós fragmensek pixeleket reprezentálnak, melyek ezután a fragment shaderhez kerülnek. A fragment shader minden egyes pixelre meghívódik, azokra is, melyekhez nem tartozik vertex, az értékek pedig lineáris interpolációval számolódnak a vertexek alapján.

A következő a per-fragmens operációk szakasza, ahol nagyrészt ellenőrzések futnak le (például mélység-ellenőrzés, átlátszóság beállítása), a végén pedig a fragmens vagy el lesz utasítva, vagy a megfelelő értékei bekerülnek a képpufferbe a hozzá tartozó helyre. Ezzel el is érkeztünk a folyamat végére, ugyanis a képernyőn a képpuffer tartalma jelenik majd meg. A renderelés folyamatát a függelékben található 1.4. ábra szemlélteti.

2. MODELLEK IMPLEMENTÁLÁSA WEBGL SEGÍTSÉGÉVEL

Az előző fejezetben bemutatásra kerültek a WebGL technológia alapjai, valamint az, hogy hogyan készíthetők egyszerű modellek WebGL segítségével. Ez a fejezet arról szól, mivel bővíthetjük az alapokat, hogyan tehetők a WebGL programok interaktívvá, hogyan érhetünk el minél realisztikusabb fényhatásokat, mozgásokat, hogyan implementálhatók OpenGL modellek WebGL-ben. (A kiinduló OpenGL modellek a Számítógépes grafika tantárgy gyakorlatának példaprogramjai közül kerültek ki.) Minden program teljes kódja megtalálható a CD mellékleten.
A programokat a Google Chrome böngésző 15.0.874.121-es verziójában teszteltem, és a dolgozat elején már említett okok miatt hibátlan működésük csak ebben a verzióban garantált.

2.1. Fogaskerék 2D-ben

Az első példa egy kétdimenziós OpenGL modell implementálása WebGL-ben. Az eredeti OpenGL modell egy fogaskereket ábrázol, és a felhasználónak lehetősége van kitöltött vagy drótvázas nézetek között váltani. A WebGL programot ezen felül további interaktivitással bővítettem: megadható a fogaskerék fogainak száma, valamint a PageUp és PageDown, illetve a Numpad + és – gombjainak segítségével növelhető/csökkenthető a fogaskerék belső, illetve külső sugara. Belső sugár alatt a beírható, külső sugár alatt a köréírható kör sugarát értem. (A program teljes kódja a CD mellékleten a fogaskerek_interaktiv.html nevű fájlban található.)

2.1.1. Kitöltött és drótvázas modellek

Az OpenGL modellhez képest az volt a nehézség, hogy a WebGL-ben nem lehet kitöltött, illetve drótvázas módot beállítani (mint OpenGL-ben a *glPolygonMode* függvény segítségével), így valójában kétszer kellett elkészíteni a modellt. A kitöltött nézet esetén a fogaskereket háromszögekből raktam össze, drótvázas nézet esetén vonalcsíkokból. Legelső lépésként elterveztem, milyen elemi alakzatokból állítható elő a fogaskerék, majd kiszámoltam a megfelelő pontok helyét, és ezekkel feltöltöttem a puffereket.

Kitöltött mód esetén egy körből és a fogak számával megegyező darab háromszögből modellezhető a fogaskerék. A megfelelő sugarú kör megfelelő magasságú háromszögekből hozható létre, elegendő háromszög esetén az emberi szem számára már nem érzékelhetők a törések a körvonalban. Hogy a számolást (és ezáltal a modellezést is) gyorsabbá tegyem, a

háromszögekből 45 fokos körcikkeket hoztam létre, így a pufferekben csak egy körcikkhez tartozó pontok pozíció- és színinformációt kell tárolni. A pufferek tartalmát nyolcszor használtam fel a nyolc körcikk létrehozásához, melyeket egymáshoz képest 45 fokkal elforgattam a z tengely mentén, így összeilleszkedve pontosan a teljes kört alkotják. A vertexpufferbe kerülő értékek számolásához annyit fontos tudni, hogy háromszög-legyezőkből rakom össze a körcikket, ebben az esetben pedig – akárcsak OpenGL-ben – először két pont megadására van szükség, majd mindig további egyet adunk meg a soron következő háromszöghöz. Minden további pont az elsőnek megadott ponttal lesz összekötve az óramutató járásával megegyező irányban, és az előző háromszöghöz fog csatlakozni.

Egy vertex megadásához két értékre, az x és y koordinátákra van szükség, mivel most 2D-ben modellezünk. A kezdeti két vertex megadása után mindig egy újabbat fűzünk a tömbhöz. Hogy hány vertexet kell megadnunk, az attól függ, hogy hány háromszöget szeretnénk létrehozni belőle. Minél kisebb – és ebből kifolyólag több – háromszögből áll össze a körcikk, annál inkább valóban körvonalra fog hasonlítani, és természetesen annál több vertexet kell megadnunk. Tökéletes kör valójában sohasem lesz, de nem nehéz elérni azt a küszöböt, ahol az emberi szem már nem tudja érzékelni a sok kis háromszögből összerakott körvonalban a töréseket.

A kódban látható *sugar* változó a felhasználó által a belső sugárra beállított értéket tartalmazza, a vertexpuffer létrehozására pedig függvényt írtam, mely ezt a változót paraméterül kapja, a vertexek pozícióinak számolásánál pedig megszorozzuk vele mind az x, mind az y koordináta értékét. A függvény végül a feltöltött vertexpufferrel tér vissza.

```
function newKorPuffer(sugar){
  puffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, puffer);
  //tömb létrehozása, értékek számolása
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  puffer.itemSize = 2;
  puffer.numItems = 10;
  return puffer;}
```

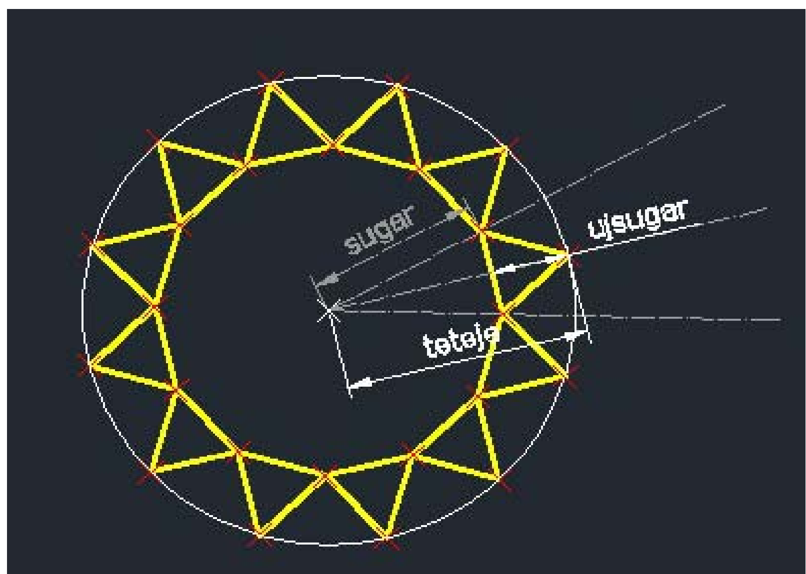
A vertexek pozícióinak megadásán felül a vertexek színét is meg kell határoznom. A színpuffert igen egyszerű feltölteni: annyiszor négyelemű (r, g, b, alfa) tömböt adok át, ahány vertexből áll a kör, és mivel minden pont ugyanolyan színű, így minden négyes ugyanazokat az értékeket fogja tartalmazni. A vertex- és a színpuffer segítségével a kör modellezhető, a vertexpuffer megadásánál a puffer helyett a puffert létrehozó függvényt hívjuk a megfelelő paraméterrel, ami a megfelelő puffert adja majd vissza:

```
gl.bindBuffer(gl.ARRAY_BUFFER, newKorPuffer(sugar));
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
newKorPuffer(sugar).itemSize, gl.FLOAT, false, 0, 0);
gl.drawArrays(gl.TRIANGLE_FAN, 0, newKorPuffer(sugar).numItems);
```

A fogaskerékhez azonban a körön kívül szükség van még fogakra, melyek háromszög lesznek. Annyi háromszöget kell létrehozni, ahány foga van a fogaskeréknek, mivel azonban mind a fogak száma, mind a belső és a külső kör sugara (vagyis a háromszögek szempontjából a magasság és a kezdőpozíció) a felhasználó által változtatható, ezért a háromszögek modellezésére is függvényt írtam. A függvény lényegében egy ciklusból áll, mely minden iterációban egy, az előbbi körre illeszkedő következő háromszöget hoz létre, az iterációk száma pedig annyi, ahány foga lesz a keréknek. A puffert létrehozó függvény tehát adott iterációhoz négy paramétert vár: belső sugár (*sugar*), külső sugár (*teteje*), hanyadik iteráció (*iteracio*), hány foga legyen a keréknek (*fogak*). Ez alapján a függvény törzsében kiszámoltam a három pontot, amiből a körre illeszkedő következő háromszög fog összeállni:

```
var ujsugar = sugar + (teteje-sugar);
var ujiteracio = iteracio+1;
var oszto = fogak/2;
vertices2 = [
sugar*Math.sin(iteracio*Math.PI/oszto), sugar*Math.cos(iteracio*Math.PI/oszto),
(ujsugar*Math.sin(iteracio*Math.PI/oszto)+ujsugar*Math.sin(ujiteracio*Math.PI/oszto
))/2,
(ujsugar*Math.cos(iteracio*Math.PI/oszto)+ujsugar*Math.cos(ujiteracio*Math.PI/oszto
))/2,
sugar*Math.sin(ujiteracio*Math.PI/oszto), sugar*Math.cos(ujiteracio*Math.PI/oszto)
]
```

A pontok kör mentén való elhelyezéséhez a sinus és cosinus függvényeket használtam, megszorozva a *sugar* és *ujsugar* értékekkel, melyek a koordináta vászon közepétől való távolságát befolyásolják, ezért a kért alakzat külső és belső sugarától függnek. A háromszög magasságának kiszámolásához kivontam a külső sugárból a belsőt, a két alappont a belső sugárnak megfelelő koordinátára került, a csúcspontot pedig korrigáltam a magassággal. Ezt szemlélteti a 2.1-es ábra.



2.1. ábra: a fogaskerék külső és belső sugara, valamint a háromszögek magasságának számítása

Azt, hogy a körnek melyik pontján tartunk, vagyis melyik irányban kell venni adott távolságot a vászon közepétől, a sinus függvény belsejében levő *iteracio* és *ujiteracio* értéke alapján számolom. Az *ujiteracio* segítségével a következő háromszög első alappontját adtam meg, ami egyben az aktuális háromszög második alappontja is. A csúcspont esetén a két alappont átlagából számoltam az x és y koordinátákat. Az *oszto* értéke pedig egyszerűen fogalmazva annyit ad meg, hogy mekkora lépésekkel haladok végig a kör mentén. Minél több fogat kell rajzolni, annál több lépés van, és annál kisebbek a lépések.

Az így kiszámolt pontokat beletesszük a vertexpufferbe, a függvény ezzel tér vissza, hasonlóan ahhoz, ahogy a kör esetében is történt. A kitöltött modell elkészítéséhez szükséges kör és háromszögek tehát megvannak. Azonban ha drótvázas megjelenítést is szeretnénk, akkor el kell készítenünk a modellt vonalcsíkokból összeállítva is.

Drótvázas esetben azonban elég a háromszögeket modellezni, hiszen azok alapjaikkal egymáshoz illeszkedve (a felhasználó által definiált fogak számától függően) úgyis megközelítőleg kört fognak bezárni, ezúttal azonban nem kell a kört kiszínezni, tehát valójában nincs is szükség a megrajzolására.

A drótvázas háromszög rajzolása nagyon hasonló az előbbi kitöltött háromszögéhez, két fontos különbség van: az egyik, hogy vonalcsíkokat rajzolunk a megadott pontokból, nem háromszögeket. A másik különbség ebből következik, hogy négy vertexet kell megadnunk minden háromszög esetén, ha be akarjuk zárni a háromszöget, és a negyedik vertexnek meg kell egyeznie az elsővel.

2.1.2. Billentyűzet kezelése

Ezek alapján mind a kitöltött, mind a drótvázas modell elkészíthető, amennyiben ismerjük a felhasználó által megadott paramétereket. Ebben az alfejezetben arról lesz szó, hogyan figyelhetjük a felhasználó által leütött billentyűket. A billentyűzetet JavaScriptben kezeljük, WebGL-ben csak a megváltozott paraméterek éreztetik a hatásukat.

A billentyűzet alapvetően kétféleképpen működhet egy programban. Az egyik esetben egyszer kell leütni a billentyűt, és ez vált ki valami változást (ami esetleg egy újabb billentyűleütéssel módosítható), a másik esetben egy esemény addig folyamatosan fennáll, amíg nyomjuk a billentyűt. Előbbire példa lehet egy játékban a lövés, amennyiben minden gombnyomásra egy rakétát lövünk ki, és ahányszor lenyomjuk a lövés gombját, annyi rakétát indítunk útnak – viszont attól, hogy hosszan nyomva tartjuk a lövés gombot, ugyanúgy egy rakétát fogunk kilőni. A második esetre példa, amikor egy figurát nyilakkal mozgatunk, ebben az esetben a mozgás az adott irányba addig áll fenn, ameddig a megfelelő billentyűt nyomva tartjuk. Ezt a két esetet kétféleképpen kezeljük JavaScriptben.

Először az egyszeri billentyűlenyomás estét mutatom be, ezt a példaprogramban a kitöltött és a drótvázas nézet közötti váltásnál használom. A billentyűzet kezelését már a legelején, a betöltődéskor meghívódó függvényben (a példákban ez a *WebGLStart*) érdemes beállítani:

```
document.onkeydown = handleKeyDown;
document.onkeyup = handleKeyUp;
```

A fenti két sorban azt mondjuk meg a JavaScriptnek, hogy billentyűlenyomás, illetve billentyűfelengedés események hatására mely függvényeket hívja meg. Ezen függvények megírásán kívül további teendőnk nincsen, mivel a JavaScript minden esetben figyeli a billentyűzet és az egér eseményeit, ha a fókusz az adott oldalon van.

A példaprogram esetében elég a *handleKeyDown* függvényt használni az egyszeri billentyűlenyomás vizsgálatához, hiszen a felengedés pillanata teljesen lényegtelen. A függvény a JavaScripttől automatikusan megkapja az *event* objektumot, melynek *keyCode* tulajdonságában elérhető az eseményhez tartozó billentyűkód, majd ezt sztringgé konvertálhatjuk az egyszerűbb kezelhetőség érdekében. A programban a „D” és a „T” billentyűkhöz rendeltem a drótvázas, illetve kitöltött (teli) nézetet, ennek megfelelően állítom a logikai változót mindkét esetben:

```
function handleKeyDown(event) {
```

```
  if (String.fromCharCode(event.keyCode) == "D") {
        drotvazas = 1;
  }
 if (String.fromCharCode(event.keyCode) == "T") {
        drotvazas = 0;
 }
}
```

A második eset a folyamatos billentyűlenyomás, ennél az információk tárolására létrehozunk egy objektumot:

```
var currentlyPressedKeys = Object();
```

Ebben az objektumban az asszociatív tömbhöz hasonlóan tároljuk minden egyes billentyűhöz, hogy a leütés esemény éppen igaz-e vagy hamis az adott billentyű esetén. Az eseményhez tartozó billentyűt ezúttal is az *event* objektum *keyCode* paramétereként érhetjük el. A *handleKeyDown* függvényt a következő sorral egészítjük ki:

```
currentlyPressedKeys[event.keyCode] = true;
```

A *handleKeyUp* függvényben is hasonló történik, csak fordítottjára változtatjuk a logikai értéket:

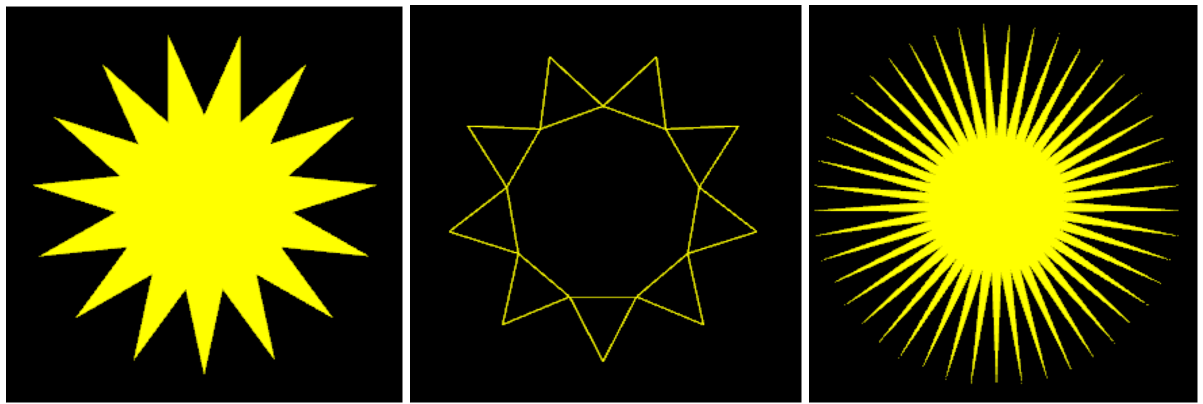
```
function handleKeyUp(event) {
    currentlyPressedKeys[event.keyCode] = false;
}
```

Így kaptunk egy asszociatív tömböt, melyből minden billentyű aktuális állapota megtudható. A következő lépés, hogy felhasználjuk ezt a tömböt. A példaprogramban számomra csak négy billentyű állapota volt érdekes: a PageUp és PageDown, valamint a Numpad + és – billentyűi, ezért csak ezeket vizsgáltam. A billentyűket a kódjukkal értem el az asszociatív tömbben, és amennyiben állapotuk lenyomott volt, a megfelelő (már hamarabb deklarált és alapértelmezett értékkel rendelkező) változók értékét módosítottam:

```
if (currentlyPressedKeys[33]) {// Page Up
      belsoSugar += 0.1;}
if (currentlyPressedKeys[34]) {// Page Down
      belsoSugar -= 0.1;}
if (currentlyPressedKeys[107]) {// +
      kulsoSugar += 0.1;}
if (currentlyPressedKeys[109]) { // -
      kulsoSugar -= 0.1;}
```

Ahhoz, hogy ezek a vizsgálatok folyamatosan végrehajtódjanak, beletettem őket egy *handleKeys* nevű függvénybe, melyet időzítő segítségével ugyanúgy adott időközönként meghívtam, mint a *drawScene* modellező függvényt. A modellező függvényben pedig a beállított változók segítségével adott időközönként kirajzolom az aktuális modellt.

A shaderek megírása nem jelent nehézséget, az előző fejezetben említett egyszerű, vertexek pozícióit és színeit, valamint a transzformációkat és perspektívát tároló mátrixokat kezelő shader megfelelő. A végeredmény a 2.2. ábrán látható.



2.2. ábra: kitöltött és drótvázas fogaskerék modellezése, a fogak száma és a sugár változtatható

2.2. Tűzijáték 2D-ben

A következő példaprogram szintén egy OpenGL modell megvalósítása WebGL segítségével. A feladat adott helyen, adott számú, eltérő méretű, vonalvastagságú, vonalstílusú és színű robbanás modellezése animálva. Az általam megvalósított program egy 400x400-as vászonra egyszerre négy robbanást modellez véletlenszerű helyen. A robbanások mérete eltérő, mely a maximális sugár méretének és a megjelenő sugarak számának eltérését jelenti – mindkettő véletlenszerűen változik adott intervallumon belül. A szín szintén minden robbanás esetén véletlenszerű. A robbanások animáltak, folyamatosan nő a sugár, ha pedig elérte a maximális méretet, akkor a robbanás eltűnik, és a vászon valamely részén új jelenik meg helyette. A vonalvastagság és vonalstílus kezelésére az OpenGL-lel ellentétben a WebGL nem biztosít lehetőséget, ezek megvalósítása nehézkes, illetve sokat lassít a programon, ezért a WebGL modellből ezeket kihagytam. (A program teljes kódja a CD mellékleten a tuzijatek.html nevű fájlban található.)

2.2.1. Animálás

Az előbb megismert programhoz képest a lényegi újdonságot az animálás jelenti, vagyis az, hogy a modell felhasználói beavatkozástól függetlenül folyamatosan változzon. Az animálás

alapelve WebGL-ben nagyon egyszerű: adott időpillanat elteltével újra és újra kirajzoljuk a modellt, vagyis időről-időre meghívjuk a modellezést végző függvényünket.

Ennek az elvnek a megvalósítása a kódban úgy történik, hogy a WebGL, a shaderek, pufferek és minden egyéb inicializálását végző függvényben (a példákban *WebGLStart*) nemcsak egyszer hívjuk meg a modellező függvényt, hanem megadott időközönként mindig újrahívjuk, a forgó kocka példájában már látott *setInterval* függvény segítségével:

```
setInterval(tick, 15);
```

A fenti példában *tick*-nek nevezett függvény törzsében helyezünk el minden olyan részt, amit változtatni szándékozunk: a példaprogram esetén ez a modellező függvény és az animáló függvény – utóbbiban változtatom és ellenőrzöm a mozgatáshoz szükséges változók értékeit.

```
function tick(){
        drawScene();
        animate();
}
```

A mozgatás vezérlésére globális változókat deklarálok. Jelen esetben szükség van változókra az aktuális x és y koordináták, az aktuális színek és nagyság (tűzijáték kezdő-, valamint maximális sugara, valamint a sugarak száma) tárolására, valamint logikai változók, melyek segítségével eldönthető, szükség van-e újabb robbanásra. Mivel egyszerre négy tűzijáték robban, és mindegyikhez szükség van ezekre az adatokra, négyelemű tömbökben tárolom őket, a kezdő sugárhosszoknak és a logikai változóknak pedig kezdőértéket is adok: kezdetben minden sugár 0, és minden logikai változó igaz (ez azt jelzi, hogy szükség van újabb robbanásokra, hiszen kezdetben üres a vászon).

A többi változó a modellező függvényen belül kap értéket, amennyiben szükség van újabb robbanásra. Mindegyik értéknek véletlenszerűnek kell lennie, ehhez a JavaScript *Math.random* függvényét használom, mely 0 és 1 közötti véletlen valós számot generál. Az R, G és B színkomponensek megadásához ez a generált érték meg is felel, hiszen annak éppen 0 és 1 közötti valós számnak kell lennie. A többi esetben kicsit módosítok az eredményen. A maximális sugárhossz értékét például a 0.2 és 1.2 közötti intervallumban szeretném generálni, ezért a kapott véletlen számhoz mindig hozzáadok 0.2-t:

```
Math.random()+0.2.
```

A robbanás sugarainak számát 8 és 20 között szeretném változtatni, és ennek a számnak nyilvánvalóan egésznek kell lennie. A generálás ez esetben így fog kinézni:

```
Math.floor(Math.random()*12)+8;
```

A *Math.floor* lefelé kerekít, tehát először 0 és 12 közötti egész számot kapok, majd ehhez hozzáadok 8-at, mivel ennyivel szeretném eltolni az intervallumot.

Mind a négy logikai változóra megvizsgálom, hogy szükség van-e az adott robbanás helyett újat létrehozni, amennyiben igen, akkor generálom a változók értékeit, majd ezeket paraméterként átadom a robbanást modellező függvénynek. Az első robbanásra ez a következőképpen fog kinézni (*randomX* és *randomY* általam írt függvények, melyek -2 és 2 közötti értéket generálnak):

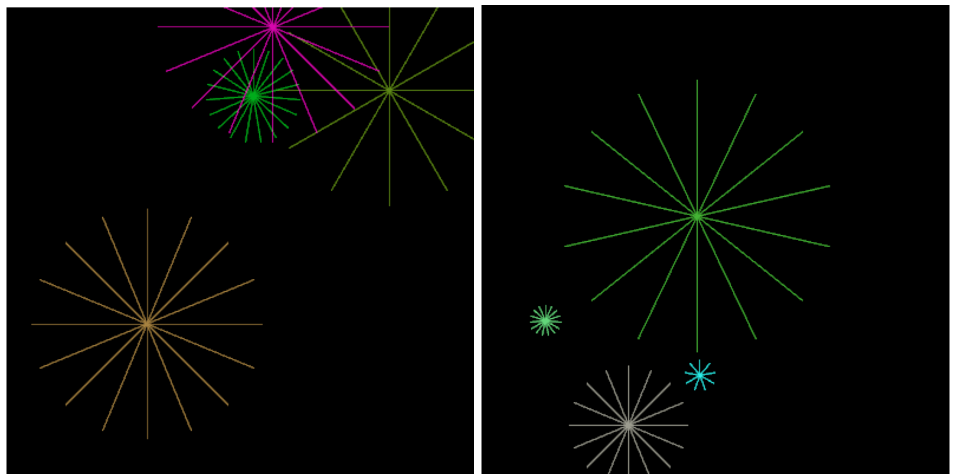
```
if(ujabb[0]){
csikok[0] = Math.floor(Math.random()*12)+8;
        r[0] = Math.random();
        g[0] = Math.random();
        b[0] = Math.random();
        xKoord[0] = randomX();
        yKoord[0] = randomY();
        maxsugarhossz[0] = Math.random()+0.2;
}
newRobbanas(csikok[0], sugarhossz[0], xKoord[0], yKoord[0], r[0], g[0], b[0]);
```

Látható, hogy a modellezést az *ujabb* nevű logikai változókkal vezéreljük, melyeknek értékét az animáló függvényben határozzuk meg minden újabb modellezés után újból. Az animáló függvényben azt vizsgálom, hogy az aktuális sugárhossz elérte-e már a maximálist az adott robbanásra. Amennyiben nem, úgy növelem az értékét, és az *ujabb* logikai változó hamis lesz, amennyiben igen, akkor az aktuális sugárhosszt nullára csökkentem (vagyis eltüntetem a robbanást), és az *ujabb* értéke igaz lesz, hiszen az eltűnt robbanás helyett modellezhetek egy újat. Az első robbanásra a vizsgálat így néz ki:

```
if (sugarhossz[0]<=maxsugarhossz[0]){
        sugarhossz[0] += 0.035;
        ujabb[0] = false;
}
if (sugarhossz[0]>maxsugarhossz[0]){
        sugarhossz[0] = 0.0;
        ujabb[0] = true;
}
```

A többi robbanásra is hasonló a vizsgálat, csak a tömbök megfelelő elemeit vizsgálom és változtatom. Nem volt még szó a robbanást modellező függvényről, mely a következő paramétereket kapja: sugarak száma (*mennyi*), aktuális sugár (*sugar*), x és y koordináta (*x, y*),

R, G és B komponensek (*red, green, blue*). A robbanást vonalakból (*gl.LINES*) rakom össze, és annyi vonalat használok, ahány sugárból áll a robbanás. Mindegyik vonal két vertexből áll, az egyik a (0,0), a másik a (0, *sugar*) koordinátára kerül. A vonalakat a megadott x és y koordinátákkal eltolom, és annak megfelelően, hogy hanyadik sugár a robbanásban, a kiszámolt szöggel elforgatom a z tengely mentén. A színpuffernek pedig az átadott R, G és B értékekkel feltöltött tömböt adom át. A shaderek az előző programban használt shaderekkel megegyeznek, a végeredmény a 2.3. ábrán látható.



2.3. ábra: véletlenszerűen robbanó tűzijátékok modellezése

2.3. Kopácsoló harkály

Ebben az alfejezetben szintén egy OpenGL-es példa néhány funkcióval kibővített változatát implementálom WebGL-ben. Az OpenGL modell egy rudat és egy körülötte kopácsoló harkályt ábrázol – a rúd egy elnyújtott, zöld téglatest, a harkály teste szintén téglatest, csőre pedig egy piros kúp. A harkály folyamatosan kopácsol, ami azt jelenti, hogy teste középpontja körül forog egy irányban egy adott intervallumon belül. Kopácsolás közben folyamatosan halad lefelé, és amint a rúd aljára ér, ismét felülre kerül. Lefelé mozgás közben folyamatosan kerüli körbe a rudat. A színtér a jobbra és balra nyilakkal körbeforgatható.

A WebGL megvalósításban ezen felül billentyűkkel vezérelhető (akár meg is szüntethető) a lefele mozgás, illetve a körbeforgás sebessége, valamint a színtér a PageUp és PageDown billentyűk lenyomásával folyamatosan nagyítható/kicsinyíthető. A színtér ambiens és direkcionális fénnyel van megvilágítva, a megvilágítás paraméterei (ambiens fénynél a színkomponensek, direkcionális fénynél az irány és a színkomponensek) űrlap segítségével változtathatók. (A program teljes kódja a CD mellékleten a 3D_kopacsolo_harkaly.html nevű fájlban található.)

2.3.1. Modell készítése a Blender modellező programmal

Az előző példákban már volt szó az animálásról és a billentyűzet kezeléséről, ez a harkály és a színtér mozgatásánál is hasonló. A modellezésnél annyi a különbség, hogy a vertexpufferen felül textúra- és normálvektor-koordinátákat tároló puffere is szükség van minden alakzat esetében a textúrák használata és a megvilágítás miatt. A pufferekbe kerülő értékek egyszerű alakzatok (téglalap, kocka, háromszög) esetében viszonylag könnyen számolhatók, de bonyolultabb alakzatok esetében nehezebb ezeket meghatározni, különösen akkor, ha nem is szabályos, szimmetrikus alakzatokról van szó. Ilyen esetekben általában egyszerűbb egy háromdimenziós modellező programmal elkészíteni a modellt, és onnan például OBJ formátumban (mely egy geometriai definíciós, nyílt fájlformátum) kiexportálni a vertexeket.

Én erre a célra a Blender nevű programot használtam. A modell exportálásánál a vertexek mellett a normálvektorok és anyagok exportálását is kértem, ezért a vertexkoordináták mellett a textúra- és a normálvektor-koordináták is belekerültek az OBJ fájlba. Még egy fontos dolgot be kellett állítanom, ez pedig a triangularizálás, vagyis hogy először háromszögekre bontsa le a modellezett alakzatokat, azután a megfelelő értékeket ezekhez a háromszögekhez határozza meg. Ez azért lényeges, mert a WebGL-ben elemi alakzatként csak háromszögeket tudunk rajzolni, négyszögeket és egyéb sokszögeket nem.

Az információk kinyeréséhez az OBJ formátumról annyit kell tudni, hogy a fájl sorai elején levő betűk jelzik, hogy az adott sorban milyen információk találhatók. A vertexeket tartalmazó sorok „v" betűvel, a textúra-koordinátákat tartalmazók „vt"-vel, a normálvektorokat tartalmazók pedig „vn"-nel kezdődnek. Ezen felül még az „f" betűvel kezdődő sorok lesznek lényegesek, melyek az egyes felületeket jelölik, vagyis azt, hogy az adott felület melyik vertexekből áll össze (esetünkben mindig három vertexnek kell meghatároznia egy felületet). A vertex megadását egy „/" karakter követi, mely után a textúra-koordináta következik, újabb „/" után pedig a normálvektor. Tehát egy felülethez három „v/vt/vn" hármas tartozik. Ezen jelölések ismeretében egy fájlbeolvasó program írásával kiválogathatjuk a számunkra lényeges információkat az OBJ fájlból. A példaprogramban is az OBJ fájlból kinyert információkkal töltöttem fel a vertex-, textúra- és normálvektor-puffereket.

2.3.2. Ambiens és direkcionális megvilágítás, animálás

A másik újdonság ebben a példaprogramban az előzőekhez képest az ambiens és direkcionális fénnyel való megvilágítás, de ennek jó részéről az első fejezetben már volt szó. A

megvilágítás jellemzőit HTML űrlapon keresztül kérjük be a felhasználótól. Az űrlap egy input mezője például:

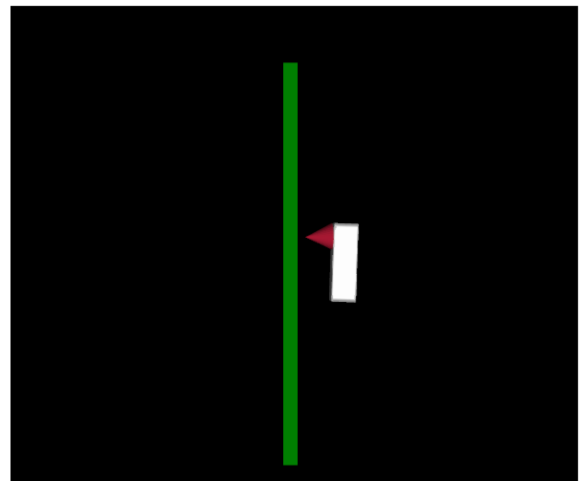
```
<input type="text" id="lightDirectionX" value="-1.0" />
```

A többi input is ehhez hasonló, mindegyikhez tartozik egy alapértelmezett érték, valamint egy id, aminek segítségével az értékét JavaScriptben lekérjük majd. A lekért értékeket először valós számmá kell konvertálnunk (hiszen az input mezők értéke szöveg), majd az első fejezetben már látott módon átadjuk őket a shadernek. Az ambiens fény paramétereinek lekérése és átadása látható alább:

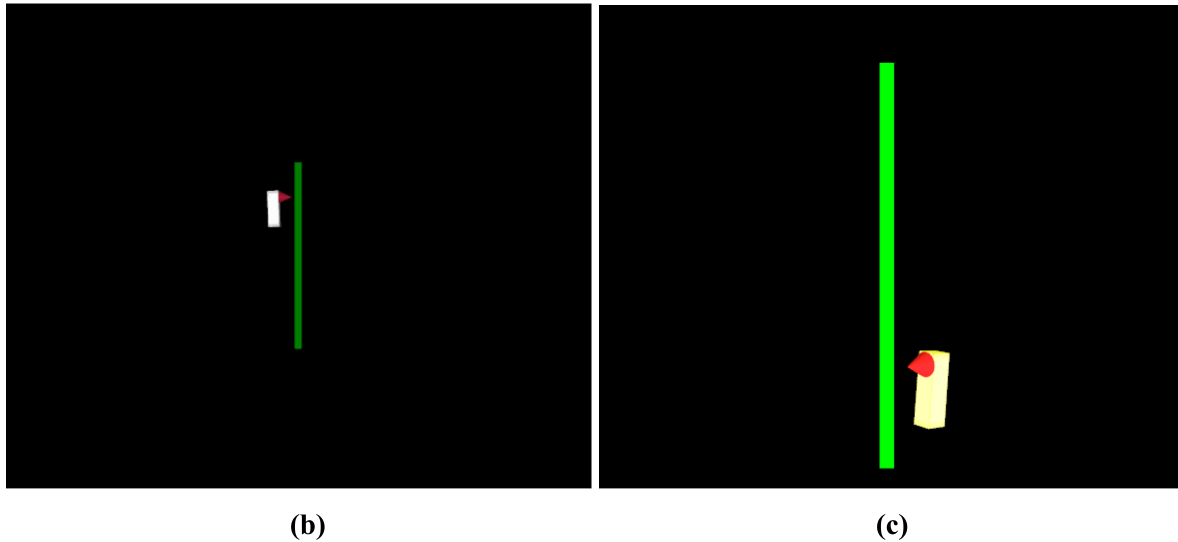
```
gl.uniform3f(shaderProgram.ambientColorUniform,
parseFloat(document.getElementById("ambientR").value),
parseFloat(document.getElementById("ambientG").value),
parseFloat(document.getElementById("ambientB").value)
);
```

A megvilágítással valójában tehát csak a shaderekben foglalkozunk, amiket az előző két példaprogramhoz képest több mindennel is ki kell egészítenünk. Az ambiens és direkcionális megvilágítást kezelő shadert már az első fejezetben bemutattam, így itt nem részletezem újból.

Ezen kívül ebben a programban is van animálás, a tűzijátékoshoz képest annyi a különbség, hogy a mozgási és forgási sebességeket meghatározó változók értéke nem konstans, hanem a lenyomott billentyűknek megfelelően változik. A végeredmény a 2.4. ábrákon látható.



(a)



(b) (c)

2.4. ábra: az (a) ábrán a harkály modellje látható a beállításokat tartalmazó űrlappal együtt, a (b) ábrán a modell kicsinyítése látható, a (c) ábrán pedig a modell megváltoztatott fénybeállításokkal

2.4. Választható alakzatok modellezése

Az utolsó példa nem OpenGL program alapján készült, ez egy összefoglaló jellegű példa, melyben bemutatom a megvilágítás és átlátszóság használatát, az egér, a billentyűzet és a textúrák kezelését. A program négy alakzatot tud modellezni (gömb, kocka, henger, ház), a felhasználó a legördülő menüből választhatja ki, hogy melyiket szeretné látni ezek közül. Számos más beállításra is lehetőség van, három fajta megvilágítás (ambiens, direkcionális, spekuláris) ki- és bekapcsolható, paramétereik állíthatók, spekuláris fénynél pedig a spekuláris csillogás értéke is megadható. Az átlátszóság szintén bármikor ki- és bekapcsolható, az átlátszóság mértéke állítható.

Kétféle textúra választható, egyszínű zöld és egy másik, kis figurát ábrázoló (ház esetén másfajta, realisztikusabb), de ki is lehet kapcsolni a textúrázást, ha a felhasználó azt az opciót választja. A kocka modellezése esetén a figurát ábrázoló textúra mozaikos, a mozaikok nagysága és mennyisége billentyűk segítségével változtatható. (Természetesen ha egyszínű textúra van, vagy a textúrázás ki van kapcsolva, akkor ezen billentyűk lenyomására semmilyen változást nem észlelünk. Ugyanez a helyzet akkor is, ha nem a kocka modellezése van kiválasztva.)

A billentyűk segítségével a modell nagyítható/kicsinyíthető, valamint folyamatosan forgatható, illetve a forgás megállítható. Míg azonban a jobbra és balra nyilak segítségével csak az y tengely mentén forgatható az éppen modellezett alakzat, addig az egérrel tetszőleges irányban és mértékben forgathatjuk, és a kétféle vezérlés egyszerre is alkalmazható.
(A program teljes kódja a CD mellékleten az osszefoglalo_webgl.html nevű fájlban található.)

2.4.1. Négyféle alakzat modellezése

A felhasználó tehát négyféle alakzat közül választhat, de mindig csak azt az alakzatot modellezem, amit éppen kiválasztott. Ehhez HTML-ben létrehoztam egy legördülő listát:

```
<select id="mi">
<option selected value="gomb">Gömb</option>
<option value="kocka">Kocka</option>
<option value="henger">Henger</option>
<option value="haz">Ház</option>
</select>
```

A lista értékét az id alapján a JavaScript kódban, a fő modellező függvényen belül lekérem.

```
var melyik = document.getElementById("mi").value;
```

Mind a négy alakzathoz külön modellező függvényt készítettem, ami lényegében csak a megfelelő puffereket tölti fel a megfelelő értékekkel, valamint végrehajtja a geometriai transzformációkat. A HTML-ből lekért érték alapján mindig csak a megfelelő függvény fog meghívódni:

```
if (melyik == "gomb"){
        GombKeszit();
} else if (melyik == "kocka"){
        KockaKeszit();
} else if(melyik == "henger"){
        HengerKeszit();
} else if (melyik == "haz"){
        HazKeszit(texture);
}
```

A megvilágításra, átlátszóságra, csillogásra vonatkozó beállítások a modellezett alakzattól függetlenek, tehát ezeket a fő modellező részben, még az alakzatot modellező függvény meghívása előtt beállítom. Szintén a HTML űrlap egyes elemeinek értékei alapján határozom meg, hogy milyen jellemzők vannak bekapcsolva, illetve mik az egyes jellemzők felhasználó által beállított paraméterei.

Először a fényeket állítom be, amennyiben szükséges. A *feny* logikai változó jelentősége, hogy értékétől függően a shaderben számolunk megvilágítással kapcsolatos jellemzőket, vagy ezek hatását teljesen kihagyjuk a számításból. Amennyiben bármilyen

megvilágítás is bekapcsolt állapotban van, ez a változó igaz lesz. Az ezt megvalósító kódrészlet alább látható:

```
var feny = false;
var light_a = document.getElementById("ambiens").checked;
var light_b = document.getElementById("diffuz").checked;
var specularHighlights = document.getElementById("specular").checked;
if((light_a==true)||(light_b==true)||(specularHighlights==true)){
feny = true;
}
gl.uniform1i(shaderProgram.showSpecularHighlightsUniform, specularHighlights);
gl.uniform1i(shaderProgram.useLightingUniform, feny);
```

Mivel spekuláris fény esetén még több dolgot figyelembe kell venni az anyagok végső színének meghatározásánál, így logikai változó formájában azt is átadjuk a shadernek, hogy ez a fajta fény aktív-e (a shader felépítésénél majd látni lehet, hogy ettől függően lép vagy nem lép be a program egy feltételes ágba). Emellett az egyes fények jellemzőit csak akkor adjuk tovább a shadernek, ha azok éppen bekapcsolt állapotban vannak, hiszen csak akkor kell őket figyelembe venni a számításoknál.

A következő az átlátszóság vizsgálata. Ez a hatás akkor igazán szép, amikor a fények is be vannak kapcsolva, de a felhasználó a fényektől függetlenül állíthatja. Amennyiben be van kapcsolva, akkor engedélyezem a blendinget, letiltom a mélységellenőrzést, valamint átadom az alfa értékét a shadernek a shader programobjektumon keresztül. Kikapcsolt állapot esetén pont ennek fordítottja történik: a blendinget letiltom, engedélyezem a mélységellenőrzést, alfa értékét pedig 1-re állítom – utóbbi inkább csak a biztonságot szolgálja, hogy tényleg minden esetben teljesen átlátszatlan legyen a modell az átlátszóság kikapcsolása után. A megvalósítás az alábbi kódrészletben látható:

```
var blending = document.getElementById("blending").checked;
if (blending) {
        gl.blendFunc(gl.SRC_ALPHA, gl.ONE);
        gl.enable(gl.BLEND);
        gl.disable(gl.DEPTH_TEST);
        gl.uniform1f(shaderProgram.alphaUniform,
        parseFloat(document.getElementById("alpha").value));
} else {
        gl.disable(gl.BLEND);
        gl.enable(gl.DEPTH_TEST);
        gl.uniform1f(shaderProgram.alphaUniform, 1.0);
}
```

Ezután következik a textúrák beállítása, ami már nem teljesen független a modellezett alakzattól, mert ház esetén másmilyen a nem egyszínű textúra, mint a többi alakzatnál. Szintén HTML-ből, legördülő menü segítségével kérjük le a felhasználó beállítását, először azt ellenőrizzük, hogy be van-e kapcsolva a textúrázás, és amennyiben nem, akkor egy hamis logikai értéket adunk át a shadernek, így az erre vonatkozó számítások nem kerülnek majd végrehajtásra.

```
var texture = document.getElementById("texture").value;
gl.uniform1i(shaderProgram.useTexturesUniform, texture != "none");
```

Ezt követően megvizsgáljuk, hogy a *melyik* változóban az alakzatok közül melyiknek az azonosító neve szerepel, és amennyiben nem a házé, akkor beállítjuk a választott textúrát, ami annyit jelent, hogy a választott textúrát aktiváljuk a *gl.bindTexture* segítségével:

```
if (melyik != "haz"){
        gl.activeTexture(gl.TEXTURE0);
if (texture == "szin") {
                gl.bindTexture(gl.TEXTURE_2D, texturaTomb[0]);
        } else if (texture == "smiley") {
                gl.bindTexture(gl.TEXTURE_2D, texturaTomb[1]);
        }
        gl.uniform1i(shaderProgram.samplerUniform, 0);
}
```

A ház textúráját külön állítom be a házat modellező függvényben, mivel külön textúrát kapnak a falak és a tető.

Az eddigi beállítások után pedig a választott alakzatnak megfelelő modellező függvényt hívom meg. Ezek közül csak egyet mutatok be részletesebben, mivel a többi is hasonló. Vegyük a kockát modellező függvényt, ami azért érdekesebb, mert a textúrapuffer értékei minden alkalommal újraszámolódnak annak a globális változónak megfelelően, amit a felhasználó billentyűk segítségével változtathat. Ez a *mozaik* nevű globális változó határozza meg, hogy mennyi (és ennek megfelelően mekkora) mozaikból áll össze a kocka textúrája. A kockát modellező függvény törzse tehát a következőképpen néz ki:

```
mat4.identity(mvMatrix);
mat4.translate(mvMatrix, [0, 0, -3+z]);
mat4.rotate(mvMatrix, degToRad(yRot), [0, 1, 0]);
mat4.multiply(mvMatrix, forgasMatrix);
```

Először a geometriai transzformációkat hajtjuk végre, melyek a felhasználó által állítható globális változókat (*z*, *yRot*, *forgasMatrix*) kapják értékül.

```
gl.bindBuffer(gl.ARRAY_BUFFER, kockaVertexPuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
kockaVertexPuffer.itemSize, gl.FLOAT, false, 0, 0);
```

A már előzőleg feltöltött, a vertexpuffer értékeit tartalmazó változót a már ismert módon átadjuk a shadernek, ezután következik a textúrapuffer átadása. Ehhez egy függvényt hívunk meg, mely létrehozza az aktuális felhasználói beállításoknak megfelelő textúrapuffert:

```
gl.bindBuffer(gl.ARRAY_BUFFER, newKockaTexturaPuffer(mozaik));
gl.vertexAttribPointer(shaderProgram.textureCoordAttribute,
newKockaTexturaPuffer(mozaik).itemSize, gl.FLOAT, false, 0, 0);
```

Minél nagyobb a *mozaik* változó értéke, annál több mozaikdarabból fog összeállni a textúra. Az egyes mozaikok mérete pedig a számuk nagyságával arányosan csökken, hiszen a felület, amire a textúrát felfeszítjük, változatlan nagyságú. Ezután a már ismert módon a normálvektorok pufferét és az indexpuffert, valamint a megfelelő mátrixokat is átadjuk a shadernek, végül pedig a *gl.drawElements* függvény segítségével a háromszögeket megfelelő módon felrajzolva összeáll a kocka.

2.4.2. Egér események kezelése

Újdonság ebben a programban az egér eseményeinek kezelése is. A billentyűzet kezelésénél már láttuk, hogy a JavaScript képes detektálni az egyes eseményeket, nekünk csak azt kell beállítani, hogy milyen függvény hívódjon meg az egyes esetekben. Ezért az oldal betöltődéskor meghívódó függvényben (a példában ez a *webGLStart*) a megfelelő egéreseményekhez hozzárendeljük a megfelelő függvényeket:

```
canvas.onmousedown = handleMouseDown;
document.onmouseup = handleMouseUp;
document.onmousemove = handleMouseMove;
```

Mivel csak akkor szeretném az egér mozgásával arányosan forgatni a modellezett alakzatot, ha a felhasználó a vásznon belül kattint, ezért az *onmousedown* eseményt csak a vásznon belül vizsgálom. Viszont ha a lenyomott egeret a felhasználó úgy mozgatja, hogy az egér a vásznon kívülre kerül, esetleg a felengedés is a vásznon kívül történik, azt szeretném érzékelni – ellenkező esetben hiába engedné fel a felhasználó a vásznon kívül az egeret, újra a

vászon fölé navigálva úgy tűnne, hogy még mindig le van nyomva, ami igen zavaró. Ezért az *onmouseup* és *onmousemove* eseményeket az egész oldalon vizsgálom.

Az aktuális elforgatás értékét egy mátrixban tárolom (*forgasMatrix*), emellett létrehozok globális változókat annak tárolására, hogy az egér éppen le van-e nyomva (*mouseDown*), valamint hogy melyek voltak az egér előző pozíciójának x és y koordinátái (*lastMouseX*, *lastMouseY*):

```
var mouseDown = false;
var lastMouseX = null;
var lastMouseY = null;
var forgasMatrix = mat4.create();
mat4.identity(forgasMatrix);
```

A *handleMouseDown* függvény nagyon egyszerű, mindössze az egér lenyomását vizsgáló logikai változót állítom igazra benne, valamint az eseményhez tartozó x és y pozíciókat adom értékül az előbb létrehozott, előző egérpozíciót tároló változóknak. Az *event* JavaScript objektumban megtalálhatók az ehhez szükséges információk.

```
function handleMouseDown(event) {
        mouseDown = true;
        lastMouseX = event.clientX;
        lastMouseY = event.clientY;
}
```

A *handleMouseUp* függvény törzse még ennél is rövidebb, ott csak az egér lenyomását vizsgáló logikai változó értékét állítom hamisra. A *handleMouseMove* függvény a legbonyolultabb a három közül, hiszen ebben számolom ki, hogy pontosan mennyit mozgott az egér, és ennek megfelelően hogyan kell forgatni az alakzatot. Ha a *mouseDown* változó értéke hamis, akkor a függvény azonnal visszatér, ellenkező esetben azonban a következő számításokat végzi:

```
var newX = event.clientX;
var newY = event.clientY;
```

Egy-egy változóban eltároljuk a *handleMouseMove* esemény meghívásakor aktuális egérpozíció x és y koordinátáját.

```
var deltaX = newX – lastMouseX
var newRotationMatrix = mat4.create();
mat4.identity(newRotationMatrix);
mat4.rotate(newRotationMatrix, degToRad(deltaX), [0, 1, 0]);
```

Ezután az x koordináta különbségét számoljuk ki az előző pozícióhoz képest, majd létrehozunk egy új mátrixot, melyben az adott *handleMouseMove* meghívásakor számolt x tengely menti forgást tároljuk (ez természetesen minden újabb eseményhíváskor új értéket vesz fel).

```
var deltaY = newY - lastMouseY;
mat4.rotate(newRotationMatrix, degToRad(deltaY), [1, 0, 0]);
```

Az y koordináta esetén szintén kiszámoljuk az elmozdulást, és az előbb létrehozott transzformációs mátrixot az y tengely menti forgással bővítjük.

```
mat4.multiply(newRotationMatrix, forgasMatrix);
lastMouseX = newX
lastMouseY = newY;
```

Végül az aktuális forgatást tároló mátrixot megszorozzuk a forgást tároló globális változóval, tehát lényegében a *handleMouseMove* minden egyes meghívódásakor kiszámolt kis forgások hatását mindig hozzáadjuk a globális mátrixhoz, hogy megkapjuk a tényleges elforgatást. Az utolsó x és y koordinátákat tároló változókba pedig az egér aktuális pozíció kerülnek, hiszen a következő mozgás onnan indul, ahol az előző befejeződött.

Az előző alfejezetben, az alakzatok modellezésnél láthattuk, hogy a geometriai transzformációkat tároló mátrix meg van szorozva ezzel a bizonyos globális mátrixszal, ami a forgást tárolja:

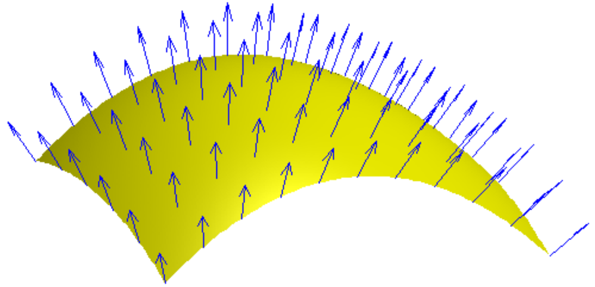
```
mat4.multiply(mvMatrix, forgasMatrix);
```

Ennek köszönhetően az alakzat a kiszámolt értékekkel elfordul, és mivel a forgást tároló mátrix független az alakzatoktól, így ha váltunk az alakzatok között, az új alakzat is pont annyival lesz elforgatva, amennyivel az előtte modellezett volt.

2.4.3. Per-fragment módszerrel számoló shaderek írása

A megvilágítás számításánál említettem, hogy számolhatunk vertex alapon, vagyis minden vertexre kiszámoljuk a megfelelő értékeket, köztük pedig lineáris interpolációval határozzuk meg a többi értéket, illetve számolhatjuk minden egyes pixelre külön a fények hatásait. Utóbbi módszer neve per-pixel vagy per-fragmens módszer, mely jóval idő- és erőforrás-igényesebb ugyan a per-vertex módszernél, ám hajlított felületek esetén csak így kaphatunk realisztikus látványt végeredményül. Mivel ebben a példaprogramban gömböt és hengert is modellezek, így ezt a módszert választottam a számításokhoz.

A per-fragment módszernél a vertex shader a pontok pozícióin felül a normálvektort is átadja a fragment shadernek minden vertex esetén, így a lineáris interpoláció nemcsak a vertexekre, hanem a normálvektorokra is végrehajtódik, ezt szemlélteti a 2.5. ábra. Ezáltal nem lesznek olyan élesek az átmenetek árnyék és megvilágított felületrész között, mert nem az fog történni, hogy az alakzatot felépítő egyik háromszög még meg van világítva, a másik pedig már nem, hanem a változó normálvektor értékek miatt fokozatosan vált át a megvilágított felület árnyékosba. Ez a különbség főként a hajlított felületekből álló alakzatoknál tűnik szembe, de igazából bármely test esetén többé-kevésbé észrevehető.



2.5. ábra: Phong-féle fényesség, a normálvektorok interpolációja – forrás: Wikipedia

Az ebben a programban használt vertex shader abban különbözik a korábban bemutatott, megvilágítás kezelésére alkalmas vertex shadertől, hogy a fények és átlátszóság jellemzőit nem kapja meg, mert ezek közvetlenül a fragment shaderhez fognak kerülni, ellenben három változót is továbbad a fragment shadernek: a transzformált vertexek és normálvektorok értékeit, valamint a textúrakoordináták értékeit. Nem is történik több a vertex shaderben, mint a megfelelő vertexek és normálvektorok beszorzása a megfelelő geometriai transzformációkat és perspektívát tároló mátrixokkal.

A fragment shader ennél jóval több műveletet hajt végre. A három bemenet mellé, amiket a vertex shadertől kap, közvetlenül neki adjuk át a csillogás, átlátszóság és megvilágítás jellemzőit, valamint a textúrát tároló *sampler* változót is. A *main* függvényben először a megvilágítással foglalkozunk, amennyiben be van kapcsolva. A direkcionális fény esetében a már ismert módon számoljuk a súlyt, amivel beszorozzuk majd az adott pont direkcionális színértékeit. Pontfény esetén először kiszámoljuk a fény irányát, majd a normálvektor alapján a visszaverődés irányát. A visszaverődés és a nézőpont tengelyei által bezárt szög határozza meg a spekuláris súlyozást, ahogy ezt már korábban láthattuk, azonban ezt még a fényesség értéke által megadott hatványra emeljük. Az összes fény hatását tehát a következőképpen összegezzük:

```
lightWeighting = uAmbientColor + uDirectionalColor * directionalLightWeighting
                + uPointLightingSpecularColor * specularLightWeighting
```

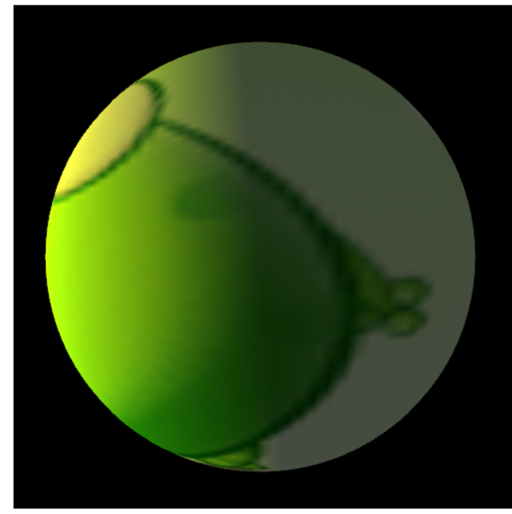
```
+ uPointLightingDiffuseColor * diffuseLightWeighting;
```

Az ambiens fénynél nem kellett számolnunk semmit, azt az értéket használjuk, amit a shader bemenetként kapott. A többi fény hozzáadásakor az egyes színeket minden vertex esetén megszorozzuk az általunk kiszámolt súlyok értékével. Így adjuk hozzá a direkcionális színt, majd a csillogás színét, végül pedig a pontfény színét. A lényeges különbség a másik módszerhez képest a számításokban az, hogy minden fragmens esetén az aktuális normálvektor alapján számolunk

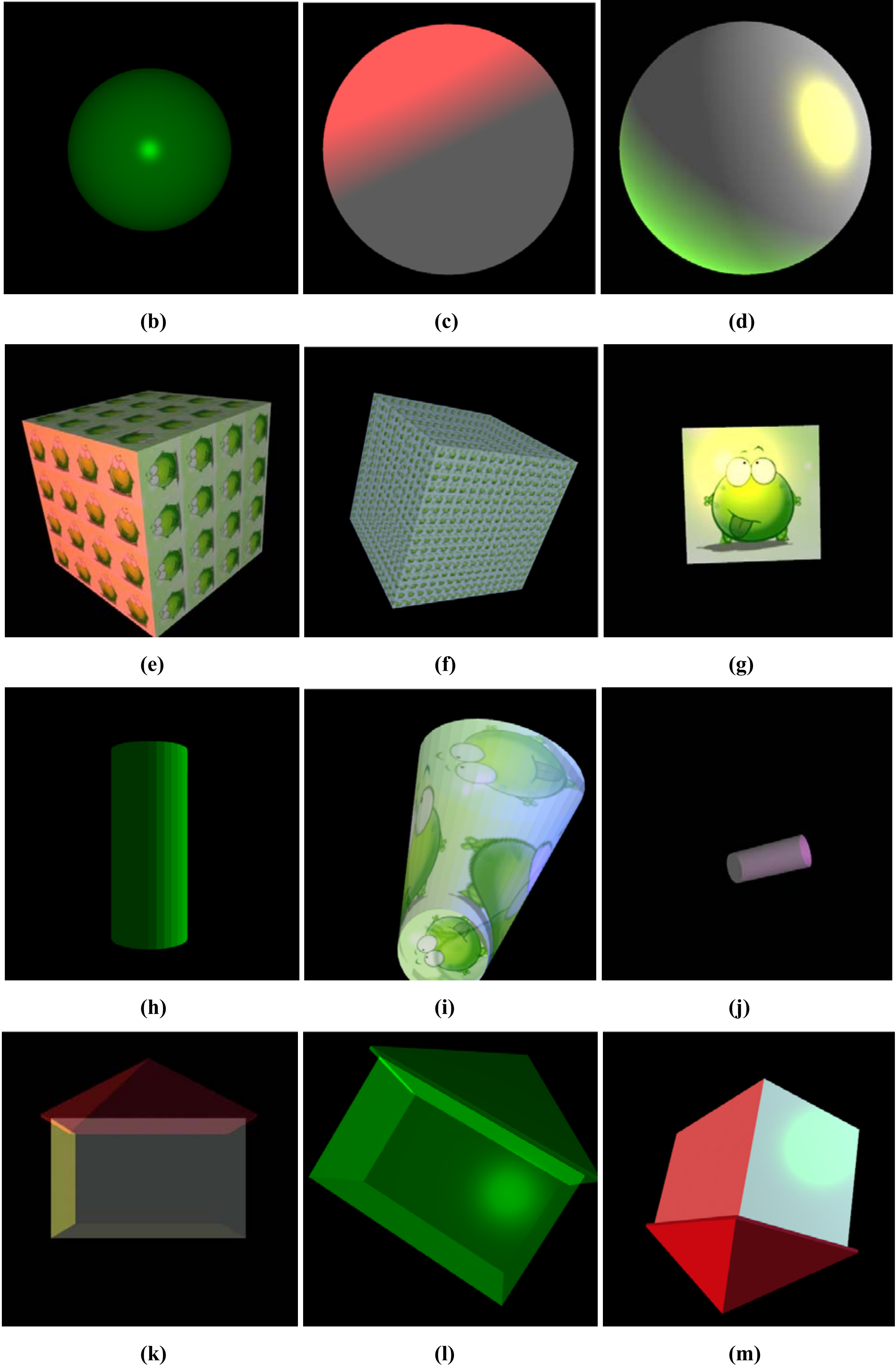
Amennyiben a textúrázás be van kapcsolva, akkor ezután a textúra beszámítása következik minden egyes vertexre – ez az már megismert módon történik. A végső szín kiszámolásához, ami a *gl_FragColor* változóba fog kerülni, minden vertex színénél figyelembe vesszük azt, hogy mennyit módosít rajta a fény, illetve mekkora az átlátszóság értéke:

```
gl_FragColor = vec4(fragmentColor.rgb * lightWeighting, fragmentColor.a* uAlpha);
```

A shaderek teljes kódja a függelékben megtekinthető. A program által kínált lehetőségek közül néhányat a 2.6. ábra szemléltet.



(a)



(b) (c) (d)
(e) (f) (g)
(h) (i) (j)
(k) (l) (m)

2.6. ábra: az (a) ábrán az alapértelmezetten megjelenített, megvilágított gömb látható a beállításokat tartalmazó űrlappal együtt; a (b) ábrán a textúra nélküli gömg látható spekuláris csillogással, a (c) ábrán a gömb vöröses fénnyel van megvilágítva, a (d) ábrán pedig zöld direkcionális és sárga spekuláris fénnyel; az (e) ábrán a mozaikosan textúrázott kocka látható ambiens és vöröses direkcionális fénnyel megvilágítva, az (f) ábrán már jóval kisebbek a mozaikok a kockán és csak ambiens megvilágítás van, a (g) ábrán sárga spekuláris fény világítja meg a kockát; a (h) ábrán a direkcionális fénnyel megvilágított, egyszínű henger látható, az (i) ábrán a henger már átlátszó és a textúrája is ki van cserélve, a (j) ábrán pedig egyáltalán nincs a hengeren textúra, le van kicsinyítve és lilás fény világítja meg; a (k) ábrán az átlátszó, oldalról megvilágított ház látható, az (l) ábrán ugyanez az egyszínű textúrával, elforgatva, spekuláris fénnyel is megvilágítva, az (m) ábrán pedig az átlátszatlan ház a feje tetejéte fordítva

3. O3D KERETRENDSZER

Az előző fejezetekben bemutattam a WebGL technológiát és néhány egyszerűbb példaprogramot. Ebben a fejezetben a WebGL keretrendszerekről lesz szó, melyek hasznosak lehetnek összetettebb alkalmazások fejlesztésénél, ezen belül bővebben az O3D keretrendszert ismertetem, melyet az általam írt 3D-s játék fejlesztésénél is felhasználtam. Az ebben a fejezetben felhasznált példakódok a játék forráskódjából származnak, de nem kizárólag ehhez a játékhoz kapcsolódnak, hanem általános megvalósításokat mutatnak be. A játék konkrét megvalósításáról az ezt követő fejezetben lesz szó.

3.1. Keretrendszer választása: O3D

Ahogyan számos más programozási nyelv esetén, úgy WebGL esetében is igaz, hogy egy keretrendszer használata gördülékenyebbé teszi a programozást, segíti az optimalizálást és kibővíti a lehetőségeket. Ezért döntöttem a játék írásakor amellett, hogy keretrendszert fogok használni, a következő lépés pedig az volt, hogy felderítsem, milyen keretrendszerek léteznek WebGL-hez, és kiválasszam közülük a számomra megfelelőt.

A kutatást a WebGL Public Wiki weboldalon kezdtem, ahol találtam egy fejezetet, melyen össze vannak gyűjtve a linkek a neten fellelhető WebGL-lel kapcsolatos oldalakhoz, így a keretrendszerekhez is. Annak ellenére, hogy a technológia még nagyon új, máris számos keretrendszer fellelhető hozzá. Azonban ezeknek egy része (pl. GTW) még fejlesztés alatt van, más részük kimondottan egy cél támogatására lett tervezve (pl. a TDL JavaScript segédkönyvtár, mely a sebességen hivatott javítani). A legtöbb esetben az a legnagyobb probléma, hogy kevés anyag található a keretrendszerről az interneten, ami van, az is főként a keretrendszer honlapján lelhető fel, és néhány egyszerűbb leckét, példaprogramot meg egy többé-kevésbé részletes dokumentációt foglal magában (pl. PhiloGL, Three.js, SpiderGL, SceneJS, Oak3D). Néhány keretrendszer kimondottan játékok, interaktív alkalmazások fejlesztéséhez ajánlott (pl. CopperLicht, O3D, PhiloGL). Mivel azonban maga a WebGL technológia nagyon friss, így folyamatosan jelennek meg újabb keretrendszerek, valamint a már meglevő keretrendszereknek újabb változatai, illetve újabb segédanyagok, demók, példaprogramok. Így akár fél év elteltével is bővülhet, változhat az általam ismertetett kínálat.

A játék írásának megkezdésekor a választásom végül az O3D keretrendszerre esett, mely egy nyílt forráskódú, a Google által fejlesztett JavaScript API. Fő célja, hogy látványos, interaktív, 3D-s böngészős alkalmazások fejlesztését támogassa. Az eredeti, ma már nem

támogatott keretrendszer egy plug-int is tartalmazott, mely a meglevő grafikus képességek mellett újabbakkal ruházta fel a böngészőket – ez az új képesség a 3D-s grafika támogatása volt. Az új megvalósítás már nem tartalmazza és támogatja többé a plug-in használatát, emellett a GLSL shader nyelvet használja, ellentétben a régi verzióval, mely az Nvidia és a Microsoft által fejlesztett Cg (C for Graphics) shader nyelvet használta. Az új keretrendszer továbbra is tartalmazza a rengeteg lehetőséget kínáló JavaScript API-t, valamint egy konvertert COLLADA (COLLAborative Design Activity) formátumú modellek importálásához. Ez a formátum számos 3D-s modellező által támogatott (SketchUp, Maya, MeshLab), azzal a céllal hozták létre, hogy a különböző 3D-s alkalmazások egységes fájlformátuma legyen.

További előnye az O3D keretrendszernek, hogy abban az időpontban, mikor a játék fejlesztését elkezdtem, ehhez tartozott a legtöbb segédanyag: egy 16 részes leckesorozat, egy részletes dokumentáció, valamint körülbelül 60 példaprogram egyszerűbbtől az összetettebbig, forráskóddal és néhány magyarázó kommenttel együtt. Ezek mind megtalálhatók az O3D honlapján, a http://code.google.com/p/o3d/ címen [8].

3.2. Első lépések az O3D használatához

Mi szükséges a keretrendszerrel való fejlesztéshez? Egy szövegszerkesztő és a JavaScript könyvtár, mely az O3D honlapjáról letölthető. A program írásának megkezdésekor a HTML kódban importálnunk kell a base.js JavaScript fájlokat, melyek az alapjai az összes többi segédfájlnak, segítségükkel a szükségeseket importálhatjuk.

```
<script type="text/javascript" src="../o3d-webgl/base.js"></script>
<script type="text/javascript" src="../o3djs/base.js"></script>
<script type="text/javascript" id="o3dscript">
        /*A JavaScriptben megírt program.*/
</script>
```

Ezután a JavaScript kódon belül importáljuk a további szükséges könyvtárakat az alábbi példában látható módon:

```
o3djs.require('o3djs.primitives');
```

Az oldal betöltésének eseményéhez hozzárendeljük azt a függvényt, amit legelőször szeretnénk meghívni, az *unload* eseményhez pedig egy olyan függvényt, melyben a szükséges takarítást végezzük majd el az oldal elhagyása előtt.

```
window.onload = initClient;
window.onunload = unload;
```

A példa esetében tehát az *initClient* függvény hívódik majd meg az oldal betöltése után, melybe egyetlen fontos utasításnak mindenképpen bele kell kerülnie:

```
o3djs.webgl.makeClients(initStep2);
```

A *makeClients* függvény létrehozza az O3D objektumokat, paraméterként pedig egy callback függvényt kap, mely az O3D objektumok létrejötte után meghívódik. Ezután a callback függvényben további globális változókat deklarálunk, melyek közül a legfontosabbak az alábbi kódrészletben láthatók:

```
g_o3dElement = clientElements[0];
window.g_client = g_client = g_o3dElement.client;
```

A *g_o3dElement* az előbb létrehozott O3D objektumok egyike, a *g_client* pedig az O3D alkalmazás belépési pontja.

3.3. Teljesítmény

Amennyiben a GPU (Graphics Processing Unit), a grafikus kártya központi egysége támogatja az O3D szolgáltatásait, akkor a renderelés mindig hardveres gyorsítást használt. (Ellenkező esetben szoftveresen történik a renderelés.) Mivel a JavaScript teljesítménye az egyes böngészőkben különböző lehet, így az O3D alkalmazások teljesítménye is változó lehet a böngésző függvényében. Itt rögtön említést is kell tennem egy fontos előnyről, ez a teljesítmény-különbség ugyanis kiküszöbölhető, ha beállíthatjuk, hogy az alkalmazásunk a V8 JavaScript motort használja (ugyanazt, amit a Chrome böngésző használ), így némileg nőhet és egységesebb lehet a teljesítmény az egyes böngészőkben. A beállítás egyszerűen megtehetően, a következő sort kell az alkalmazást elejére, még az O3D objektumok létrehozása (a példában a *makeClients* függvény) elé beilleszteni:

```
o3djs.util.setMainEngine(o3djs.util.Engine.V8);
```

3.4. Csomagok

Az előbb bemutatott kezdeti lépések után el lehet kezdeni a fejlesztést O3D-ben. A felhasználható függvények csoportosítva külön fájlokba vannak rendezve az API-ban, amiket

ezekből felhasználunk, azokat az előbbi alfejezetben bemutatott módon importálni kell, ezután pedig a teljes elérési útvonallal hivatkozhatunk rájuk.

Ha az O3D keretrendszert említjük, akkor mindenképpen beszélnünk kell a csomagokról, melyek a memória menedzsment szempontjából fontosak. A csomag tartalmaz minden O3D objektumot és felügyeli az életciklusukat. Amikor létrehozunk egy objektumot O3D-ben, akkor ez egy ún. csomaghoz (pack) lesz hozzáadva. Ez a megoldás azt hivatott megelőzni, hogy egy objektum, ami még használatban van, törlődjön. Számoljuk, hogy az objektumra hány helyről van hivatkozás, és az objektum csak akkor fog törlődni, ha már semmi nem hivatkozik rá (adott törlés eggyel csökkenti a hivatkozások számát). A csomag maga is tartalmaz hivatkozást minden objektumára, tehát ha már semmi más nem hivatkozik egy adott objektumra, akkor a csomagból is el kell távolítanunk ahhoz, hogy a rendszer törölni tudja.

Csomagot a *createPack* függvénnyel hozhatunk létre, ha pedig objektumot szeretnénk beletenni, illetve eltávolítani, akkor a csomag objektum *createObject* és *removeObject* metódusait kell meghívnunk.

Az alábbi kódrészletek ezekre adnak példát. Először a csomag létrehozását láthatjuk:

```
var g_pack = g_client.createPack();
```

Ezután létrehozunk egy új transzformot (ehhez rendeljük majd magát az alakzatot, hamarosan bővebben is lesz róla szó), amit már a létrehozáskor egy csomaghoz kell rendelnünk:

```
var init_transform = g_pack.createObject('Transform');
```

Amikor már nincsen szükségünk az objektumra, akkor a transzformját töröljük a csomagból:

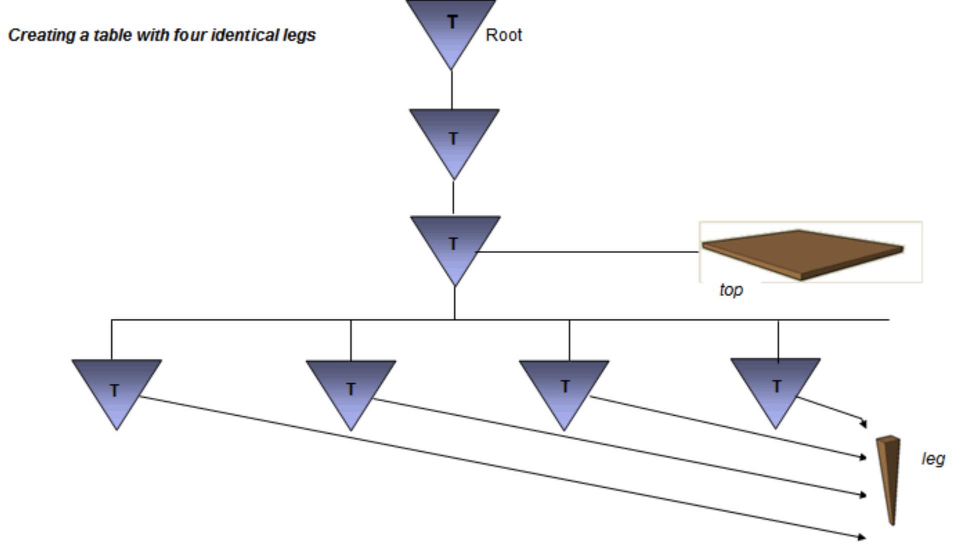
```
g_pack.removeObject(init_transform);
```

3.5. Transzformok és transzform-gráf

Az O3D keretrendszerhez tartozik egy színtér megjelenítését támogató, ún. színtér-gráf (scene graph) API is, melyben kétféle gráf létezik. Ezek egyike a transzform-gráf, mely a pozícióról, méretről, alakzatról, anyagról és shaderekről tárol információkat. A transzform-gráf lényegében szülő-gyerek hierarchiába rendezett transzformok kollekciója. Az alkalmazás transzform-gráfjának mindig van egy gyökér-transzformja, melynek bármennyi gyermeke

lehet. Egy transzformhoz egy vagy több alakzatot társíthatunk. Az alakzat O3D-ben egy egységként kezelt geometriai alakzatot jelent, melyet létrehozása után társítunk egy transzformhoz, ennek következtében az alakzat a transzform lokális koordináta-rendszerében lesz elhelyezve.

Vegyük például egy asztal létrehozását: először modellezünk egy asztallábat, majd négyszer hivatkozunk rá (ugyanis négy különböző transzformhoz rendeljük hozzá a négy különböző lábat). Ezután modellezzük az asztallapot, szintén hivatkozunk rá egy transzformból. Mind az öt transzformot a megfelelő módon pozicionáljuk, hogy asztalt alkossanak. Végül létrehozunk egy szülő transzformot, melynek az említett öt transzform lesz a gyereke, így az asztal egy csoportként kezelhető – például a kívánt helyre mozgatható. A transzform-gráfok tehát ebben az esetben a 3.1. ábrán látható módon fognak kinézni:



3.1. ábra: egy asztal transzform-gráfja - forrás: code.google.com/intl/hu-HU/apis/o3d/docs/

Ez a megoldás azért előnyös, mert a transzformok külön-külön is kezelhetők, viszont a szülőtranszformok segítségével az összetartozó transzformokon egységesen is végezhetők módosítások. Tehát az asztal lábai egymáshoz képest külön-külön eltolhatók, nagyíthatók vagy kicsinyíthetők, illetve elforgathatók, ugyanakkor ezen transzformációk bármelyike a teljes asztalon, tehát mind a négy lábon és az asztallapon is végrehajtható.

A másik dolog, melyről mindenképpen említést kell tenni: a transzformok referenciát tartalmaznak a hozzájuk rendelt alakzatra, tehát elég egyszer létrehozni és tárolni egy alakzatot, viszont annyi transzformhoz rendelhető hozzá, amennyi példányban meg szeretnénk jeleníteni. Az asztal példájánál maradva ez annyit jelent, hogy nem kell négy külön alakzatot létrehozni az asztallábaknak, hanem elég egyetlen alakzat, és mindig csak az újabb hivatkozást hozzuk létre az új transzformból. A geometriai transzformációk is különbözők

lehetnek a négy hivatkozott példány esetén, hiszen a transzformációk nem magához az alakzathoz, hanem egy adott transzformhoz kapcsolódnak.

Új transzform létrehozásakor mindenképpen meg kell adnunk a transzform szülőjét is, ennek alapértelmezetten a gyökértranszformot állíthatjuk be:

```
var init_transform = g_pack.createObject('Transform');
init_transform.parent = g_client.root;
```

Törlésnél fontos, hogy a szülő transzformra való referenciát is megszüntessük, csak ezután fog ténylegesen törlődni:

```
g_pack.removeObject(init_transform);
init_transform.parent = null;
```

3.6. Render-gráf

A másik színtér-gráf a render-gráf, mely arról tárol információt, hogyan konvertáljuk a 3D-s objektumokat pixelekké, melyek a felhasználó képernyőjén megjelennek. A render-gráf tárolja például, hogy melyik 3D-s objektumok nem látszódnak, valamint felel a speciális renderelésért például átlátszó objektumok esetén, vagy amikor egyszerre több nézetet jelenítünk meg ugyanarról a modellről. A render-gráf a transzform-gráftól kapott információk alapján végzi el feladatát.

Az O3D egy *DrawContext* nevű objektumban tárolja a nézőpont és a projekció beállításait, mindkettőt egy-egy mátrixban. A nézőpontot tároló mátrix reprezentálja a transzformációt, mellyel a megfelelő koordinátákra transzformáljuk a vertexeket. A projekciós mátrix segítségével pedig levágjuk mindazt a tartalmat, ami kívül esik a látótéren. Először létrehozunk egy rendergráfot az *o3djs.rendergraph.createBasicView* metódus segítségével (a példában *g_viewInfo* néven), majd ehhez rendeljük a két mátrixot:

```
g_viewInfo.drawContext.projection = g_math.matrix4.perspective(
        g_math.degToRad(45),
        g_client.width / g_client.height,
        0.1,
        10000);
```

A projekció beállítása nagyon hasonló ahhoz, ahogy az keretrendszer használata nélkül is történt: megadjuk, hogy milyen szögben nézünk a színtérre, a képarányt, valamint a legközelebbi és a legtávolabbi, még éppen látható pontokat. Természetesen ehhez a keretrendszer megfelelő függvényeit használjuk.

A nézőpont beállítása a *lookAt* függvény segítségével történik, melynek megadtam a szem pozícióját, a célpozíciót, amit nézni szeretnék, valamint egy felfelé mutató vektort (ez a felfelé mozgatás mértékét szabja majd meg):

```
g_viewInfo.drawContext.view = g_math.matrix4.lookAt(
      [0, 0, 1],   // eye
      [0, 0, 0],    // target
      [0, 1, 0]);  // up
```

3.7. Alakzatok, anyagok, effektek, textúrák

Minden alakzat primitívekből áll, és minden primitív tartalmazhat az anyagára vonatkozó paramétert (több primitív is használhatja ugyanazt az anyagot). A shaderek kódját O3D-ben a HTML textarea elemében helyezzük el szövegként, majd az effekt típusú objektumokba olvassuk be őket. Az effekt egy alakzat anyagának paramétere lehet, ezáltal ugyanúgy a shaderek segítségével számoljuk a pixelek színét O3D-ben, ahogy azt eddig is láthattuk.

Effekt létrehozása az alábbi módon történhet (*vshader* és *pshader* a vertex és fragment shaderek kódjait tartalmazó textarea elemek azonosítói):

```
effect = g_pack.createObject('Effect');
var vertexShaderString = document.getElementById('vshader').value;
var pixelShaderString = document.getElementById('pshader').value;
effect.loadVertexShaderFromString(vertexShaderString);
effect.loadPixelShaderFromString(pixelShaderString);
```

A következő lépés az anyag létrehozása:

```
var meteor_material = g_pack.createObject('Material');
meteor_material.drawList = g_viewInfo.performanceDrawList;
meteor_material.effect = effect;
effect.createUniformParameters(meteor_material);
```

Az anyagot a kétféle rajzolási lista közül a *performanceDrawList*-hez rendeljük, mivel átlátszatlan alakzatról van szó (létezik egy másik lista is, oda az átlátszó alakzatok kerülnének), majd beállítjuk hozzá az előbb létrehozott effektet. A *createUniformParameters* függvény létrehozza a shader számára az anyaghoz szükséges változókat.

Amennyiben textúrát is szeretnénk ráfeszíteni az alakzatra, akkor egy sampler objektumot is létre kell hoznunk, melyet az előbb létrehozott anyag paramétereként beállítunk. A sampler objektum textúra paramétereként pedig egy megfelelő formátumú (pl. JPEG, PNG), már korábban beolvasott képet adunk meg.

Miután az effektet, az anyagot és a textúrát létrehoztuk, jöhet magának az alakzatnak a létrehozása:

```
var meteor_shape = o3djs.primitives.createSphere(g_pack, meteor_material, 50, 20,
15);
```

A fenti példában egy gömb létrehozása látható. A *primitives* nevű csomag jó néhány egyszerűbb alakzat elkészítéséhez kínál függvényt – ilyen például a kocka, gömb, henger, kúp, csonkakúp – így ezeket könnyedén modellezhetjük. Mindegyik esetben meg kell adni paraméterként a csomagot, ahová az alakzatot beletesszük, az alakzat anyagát, valamint az alakzat típusától függően további paramétereket – gömbnél ez a sugár, illetve a szélesség és magasság tengelyei mentén történő felosztás mértéke. Kocka esetén további paraméterként elég az oldalhosszt megadni, hengernél a sugár, a magasság, illetve a vízszintes és függőleges tengely menti felosztás mértéke szükséges. A szükséges paraméterek bármely alakzat esetén az API-ból kikereshetők.

Ezután az alakzatot egy már korábban létrehozott transzformhoz rendeltem:

```
meteor_transform.addShape(meteor_shape);
```

Az alakzat természetesen tetszőleges számú transzformhoz hozzárendelhető, melyeken különféle geometriai transzformációk hajthatók végre. A transzform törlésekor pedig csak az alakzat adott transzformhoz való hozzárendelését szüntetjük meg.

3.8. Eseménykezelés

Eseménykezeléshez O3D-ben ez *event* csomagot használhatjuk, mellyel többféle eseményt is figyelhetünk, és az adott eseménynek megfelelően meghívhatunk egy függvényt, mely egy Event objektumot fog kapni paraméterül. Például az egér lenyomásának és mozgatásának figyelését a következőképpen állíthatjuk be (*picking* és *beforePicking* a megfelelő függvények, melyeket az események bekövetkezésekor meg kívánunk hívni):

```
o3djs.event.addEventListener(g_o3dElement, 'mousedown', picking);
o3djs.event.addEventListener(g_o3dElement, 'mousemove', beforePicking);
```

A billentyűzet eseményeinek kezelésére az *event* csomag egy másik metódusát, a *getEventKeyChar* metódust használhatjuk, mellyel a lenyomott, illetve felengedett karakter Unicode kódját kapjuk vissza:

```
document.onkeydown = function(e) {
```

```
var keyChar = o3djs.event.getEventKeyChar(e);
keyIsDown[keyChar] = true;
}
```

A vászon tartalmának megváltozása esetén fontos az újrarenderelés, és a legtöbbször valóban animált vagy mozgatható modellek vannak a vásznon, tehát a tartalom folyamatosan, vagy adott esemény bekövetkezésekor változik. Adott esemény (pl. billenytűzet- vagy egéresemény) bekövetkezésekor adott az időpillanat, amikor újra kell renderelni, folyamatos változás esetén azonban a renderelést nem kötjük a vásznon bekövetkezett eseményekhez, hanem az eltelt idő a befolyásoló tényező. Az utóbbi esetben lehet különösen hasznos a *setRenderCallback* függvény, melynek használatával a színteret automatikusan újrarenderelhetjük minden alkalommal, amikor a hardver frissíti a képernyőt:

```
g_client.setRenderCallback(onrender);
```

A *setRenderCallback* egy általunk megadott metódust (a példában az *onrender* metódust) hív meg minden képernyőfrissítéskor, így ide kell elhelyeznünk minden olyan részét a programnak, melyben az idő múltával változás következik be. Ez az általunk megírt függvény a *setRenderCallback* általi minden egyes híváskor egy *RenderEvent* objektumot fog kapni paraméterül. Ennek segítségével bizonyos információk érhetők el a rendereléssel kapcsolatban, például az, hogy mennyi idő telt el az előző renderelés óta (*elapsedTime*), vagy mennyi időt vett igénybe maga a renderelés (*renderTime*). Az elérhető információk természetesen csak olvashatók.

4. HÁROMDIMENZIÓS JÁTÉK ÍRÁSA O3D SEGÍTSÉGÉVEL

Ebben a fejezetben egy tényleges gyakorlati alkalmazásról lesz szó: egy böngészőben futó, háromdimenziós játékról. A háromdimenziós, közvetlenül böngészőben futtatható játékok világa valóban a WebGL használatának egyik kiemelkedő területe lehet, számos egyszerűbb és bonyolultabb WebGL-lel készült játék található már most is az interneten. A témában Szirmay-Kalos László Háromdimenziós grafika, animáció és játékfejlesztés című könyvét [9] ajánlom.

Az általam írt játékot is a Google Chrome böngésző 15.0.874.121-es verziójában teszteltem, és a korábban említett okok miatt ez esetben sem garantált, hogy más böngészőkben is hibátlanul működik. A játék teljes kódja a CD mellékleten a meteoros_jatek.html nevű fájlban megtekinthető.

4.1. A játék szabályai

Az általam írt játék helyszíne a világűr, „főhőse” pedig egy űrhajó, amit keresztül kell juttatni a meteorzáporon arra törekedve, hogy elkerülje az ütközést a szembe jövő meteorokkal. A játék célja: minél tovább életben maradni, közben pedig minél több pontot gyűjteni.

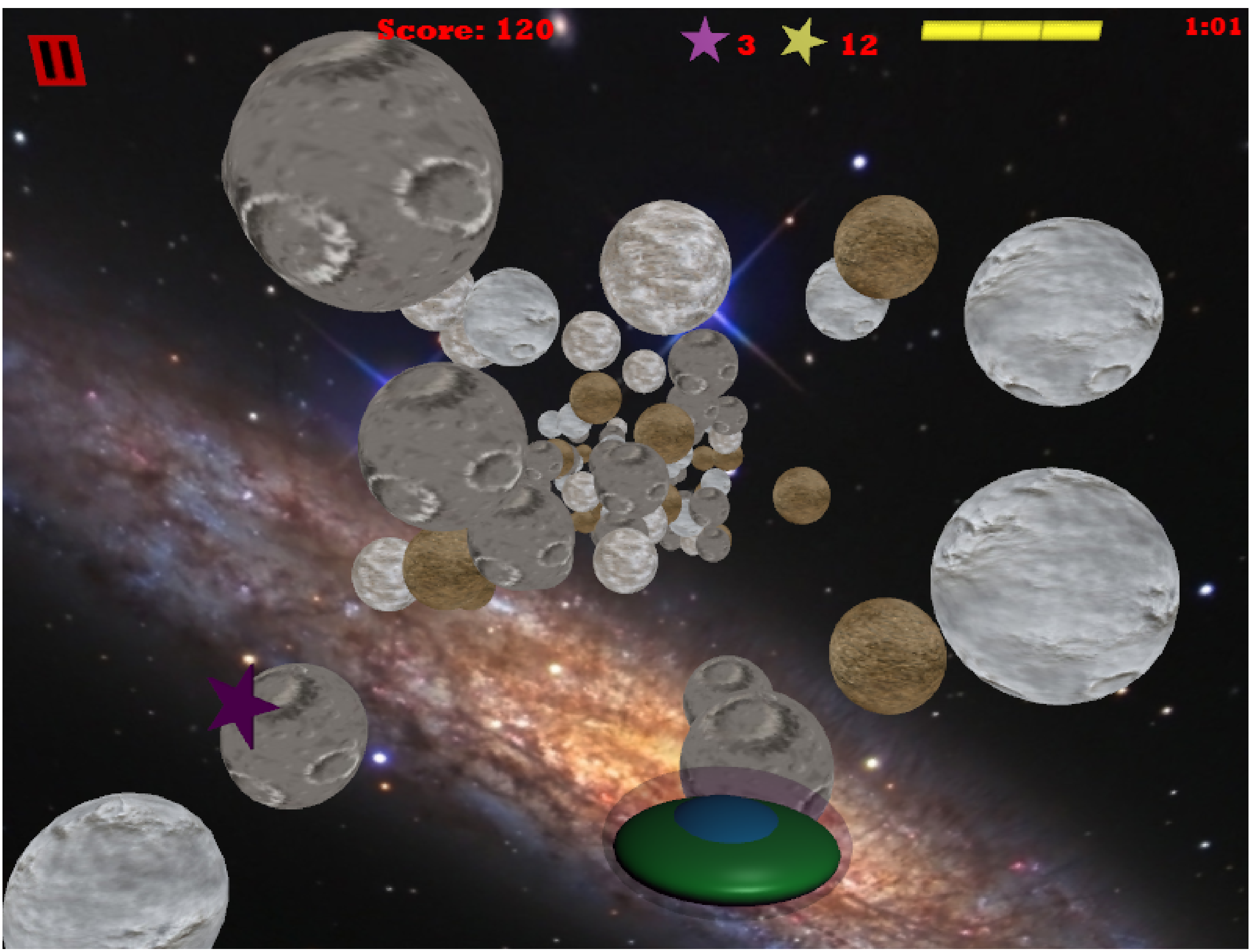
Az űrhajónak van védőpajzsa, ez egy 50%-ban átlátszó, 5 egységnyi erős burok, ami körbeveszi őt. Minden egyes ütközés alkalmával 1 egységgel gyengül a pajzs energiája – ezzel arányosan egyre halványabbnak is látszik –, és amikor eléri a 0 egységet, akkor teljesen eltűnik. Ez a pajzs azonban képes önmagától regenerálódni: amennyiben az űrhajó hosszabb ideig nem ütközik meteorral, úgy egyre erősödik egészen addig, míg visszanyeri eredeti energiáját. Újabb ütközés esetén természetesen újból gyengül. A 4.1. ábrán a játék látható működés közben, az űrhajó pajzsa maximális erősségű.

A pajzs elvesztése után a meteorral való ütközéskor már maga az űrhajó sérül, ami azt jelenti, hogy kezdeti 30 egységnyi energiából kiindulva mindig 10-zel csökken az energiája. A pajzzsal ellentétben az űrhajó energiája nem tud nőni, amennyiben bizonyos ideig nem ütközik, így pajzs nélkül csak kétszer történhet ütközés, a harmadik alkalommal az űrhajó elpusztul, ami a játék végét jelenti. Természetesen ha időközben regenerálódik a pajzs, akkor az a 10 vagy 20 egységnyi energiával rendelkező hajót is ugyanúgy védi, mint a maximális energiájút.

A pontgyűjtésnek két módja van. Egyrészt minél több, véletlenszerű helyen megjelenő sárga csillagot kell összegyűjteni, ezek mindegyike +10 pontot jelent. Másrészt minden eltelt

10 másodperc a nehézségi szinttől függően +1, +2 vagy +3 pontot ér. Van még egy másik fajta csillag is, ami lila színű, ez nem pontszerzésre való, hanem felvételével új, maximális energiájú pajzsot kap az űrhajó. A játékot nehezíti, hogy minden perc elteltével egységnyivel nő a szembejövő meteorok sebessége. A 4.2. ábra a már majdnem megsemmisült pajzsú űrhajót mutatja játék közben, a szerzett pontok, az eltelt idő, a begyűjtött sárga és lila csillagok száma, valamint a hajó energiája a vászon felső részén látható.

4.1. ábra: az űrhajó játék közben, maximális erősségű védőpajzzsal



4.2. ábra: az űrhajó játék közben, a pajzs már majdnem teljesen megsemmisült

A játékban három nehézségi szint közül lehet választani az induláskor: könnyű, közepes és nehéz. Minél nehezebb szintet választunk, annál gyorsabban jönnek szembe a meteorok már a kezdéskor, és ez a sebesség fog percenként nőni. Emellett a nehezebb szinteken ritkábban jelennek meg csillagok, kevesebb ideig léteznek, és ritkábban fordul elő közöttük lila csillag, így még inkább kerülni kell az ütközéseket, hiszen kisebb az esély egy új pajzs beszerzésére. A nehezebb szinten életben töltött idő természetesen többet ér, mint a könnyebben, ezt a már említett módon figyelembe is veszem a pontozásnál. A 4.3. ábrán a játék nyitóképernyője látható, ahol kiválasztható a nehézségi szint, elolvashatók a játék szabályai, valamint a „Start" gombbal elindítható a játék.

4.3. ábra: a játék nyitóképernyője, a nehézségi szint kiválasztása

Ha az űrhajó elpusztul, akkor a „Game Over" feliratú képernyő jelenik meg, de a „New Game" gombra kattintva máris kérhető új játék. A „Game Over" képernyő a 4.4. ábrán látható. További kényelmi funkció, hogy a játékot a pause gombbal menet közben bármikor megállíthatjuk. Ilyenkor a pause gomb eltűnik, helyén a play gomb jelenik meg, aminek megnyomásakor pedig folytatható a játék onnan, ahol abbahagytuk.

4.4. ábra: "Game Over" képernyő; a "New Game" gombbal új játék kezdhető

4.2. Az O3D által kínált lehetőségek felhasználása a játékban

Az előző fejezetben már volt szó az O3D keretrendszer által kínált lehetőségekről. Most azt emelném ki, hogy az általam írt játékban hol és hogyan használtam a keretrendszert, és ez milyen előnyt jelent.

4.2.1. Az objektumok kezelése: csomagok, transzformok

A játék írása során egyetlen csomagot hoztam létre, ebbe tettem bele minden szükséges objektumot, akár a játék előtt, közben vagy után jelenítem meg a vásznon. Így biztos lehettem benne, hogy amíg egy objektumot a csomagból el nem távolítok, addig az biztosan nem fog törlődni.

Ahogy említettem, O3D-ben minden alakzatot a létrehozáskor csomagba teszünk, amikor pedig meg akarjuk jeleníteni, akkor egy transzformhoz is hozzárendeljük, mely tárolja az összes információt az alakzatról és annak geometriai transzformációiról. A transzform és az alakzat azonban nem szétválaszthatatlan, a modellezés során egy transzform több alakzatra is hivatkozhat, létrehozhatunk újabb hivatkozásokat és törölhetünk meglevőket. Az azonban minden esetben teljesül, hogy a transzformon végrehajtott geometriai transzformációk minden alakzaton végrehajtódnak, melyre a transzform hivatkozást tartalmaz. Ezt előnyként különösen a szülő-gyerek transzformok esetében használhatjuk ki, ugyanis a gyerektranszform örököl mindent a szülőtől, tehát a szülőtranszformon végzett geometriai transzformációk a gyerekeken (és a gyerektranszformok által hivatkozott alakzatokon) is végrehajtódnak. Ezt a játék írásánál kihasználtam például az űrhajó modellezése esetében, az űrhajó ugyanis három alakzatból áll: a törzse, az ablaka és a pajzsa is alapvetően egy-egy gömb. A három alakzatot egymáshoz képest is kellett pozícionálni, illetve nyújtani, a kész űrhajónak azonban együtt kell mozognia. Tehát három külön transzformot hoztam létre a három alakzatnak, azonban csak az űrhajó törzsének a szülője a gyökértranszform (a *spaceship_root* a program elején, a globális változók között van deklarálva):

```
spaceship_root = g_pack.createObject('Transform');
spaceship_root.parent = g_client.root;
spaceship = o3djs.primitives.createSphere(g_pack, material, SPACESHIP_SIZE, 30,
20);
spaceship_root.addShape(spaceship);
```

A törzs transzformja a *spaceship* néven létrehozott, konstans (*SPACESHIP_SIZE*) sugarú körre hivatkozik. A másik két alakzat transzformjai gyerektranszformjai az előbb létrehozottnak:

```
pilotafulke_root = g_pack.createObject('Transform');
pilotafulke_root.parent = spaceship_root;
pilotafulke_root.addShape(spaceship);
pajzs_root = g_pack.createObject('Transform');
pajzs_root.parent = spaceship_root;
pajzs_root.addShape(spaceship);
```

Mindkettőhöz ugyanazt az alakzatot rendeljük, amit a törzs transzformjához, hiszen a pilótafülke ablakának és az űrhajót körülvevő pajzsnak az alakja nagyon hasonlít a törzshöz, némi nyújtással és lapítással a törzsből kialakíthatók. A megfelelő nyújtásokat, illetve lapításokat és pozicionálásokat a pilótafülke és a pajzs transzformján végezzük el, így ez csak ezeket az egységeket érinti majd. A pilótafülke esetén például x és z tengely mentén kicsinyítünk, valamint a fülkét a törzshöz képest középre helyezzük (*pilotafulkeTranslate* és *pilotafulkeScale* a megfelelő értékeket tartalmazó globális vektorok):

```
pilotafulke_root.translate(pilotafulkeTranslate);
pilotafulke_root.scale(pilotafulkeScale);
```

A törzs transzformján végrehajtott transzformációk azonban kihatással vannak mindhárom alakzatra, tehát a transzform eltolásával mozgathatjuk például az űrhajót (a három paraméter a transzform három, általam létrehozott tulajdonsága, melyek a folyamatosan számított koordinátákat tartalmazzák, ezeknek megfelelően kerül az űrhajó mindig az aktuális pozícióba):

```
spaceship_root.translate(spaceship_root.x, spaceship_root.y, spaceship_root.z);
```

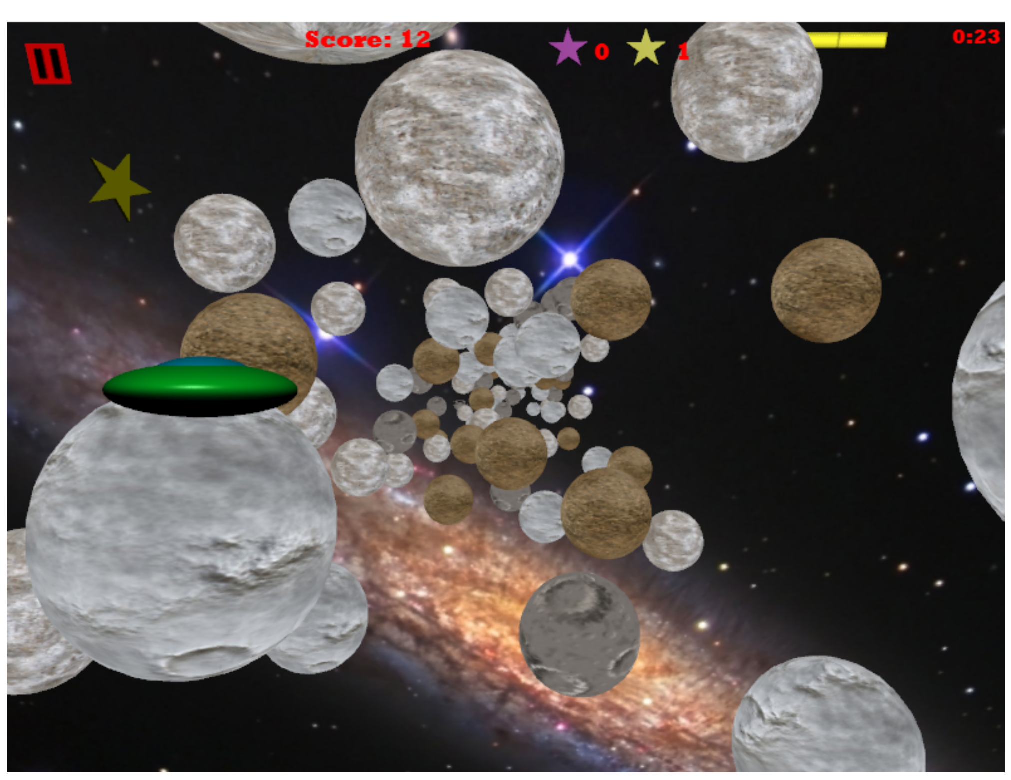
Egy másik példa a szülő-gyerek transzformokra a nyitóképernyőn látható, füstölő csóvát maga után húzó meteor: itt szintén különféle módosításokat kell végrehajtani a gömbön, illetve a füstön, azonban fontos, hogy a füst pontosan a gömbbel együtt mozogjon.

Olyan eset is előfordul a játék modellezése közben, amikor ugyanolyan vagy hasonló alakzatokat kell több példányban modellezni. A legkézenfekvőbb példa erre a folyamatosan szembejövő meteorok modellezése. Itt azt használom ki, hogy a transzform csak referenciát tartalmaz a megfelelő alakzatra, tehát például tíz meteor megjelenítése nem tíz gömb létrehozását jelenti, csak tíz hivatkozást egy adott gömbre. Amennyiben minden meteor textúrája ugyanaz a kép lenne, akkor elég lenne az összes meteorhoz egyetlen alakzatot

létrehozni. A játékban azonban négy különböző textúrát használok, ezért négy gömb kerül modellezésre, a soron következő transzform pedig véletlenszerűen hivatkozik valamelyikre a négy közül. Belátható, mennyi memóriát sikerült megspórolni ezzel a megoldással, hiszen a képernyőn esetenként akár száz meteor is látható, valójában azonban csak négy alakzatot kell létrehozni.

Más esetekben talán nem ennyire kézenfekvő a transzformok segítségével történő példányosítás, de sok más helyen is kihasználom a játékban az általuk kínált előnyt. Például a pajzs megjelenítésénél és eltüntetésénél nem magát a pajzsot (vagyis a megfelelően nyújtott gömböt) törlöm ki és hozom létre újból, hanem csak a transzformból rá mutató hivatkozást szüntetem meg, illetve hozom létre újra. Ez a helyzet az összes időnként eltűnő, máskor újból megjelenő alakzattal, esetükben is valójában csak a hivatkozások változnak.

A négy különböző textúrával rendelkező meteorok példányai és a pajzs nélküli űrhajó a 4.5. ábrán láthatók.



4.5. ábra: az űrhajó játék közben, pajzs nélkül

4.2.2. Globális változók, beállítások

A játék modellezése egymást hívó függvények segítségével történik, tehát a játékban felhasznált változókat (pl. transzformok, alakzatok tárolására használt változók, geometriai transzformációk megadására szolgáló vektorok, a játék vezérlésére használt logikai változók,

konstansok) globális változóként már a legelején létrehozom, hogy mindenhonnan elérhetők legyenek.

Az oldal betöltődésekor meghívódó *initClient* metódusból kiindulva legelőször a globális jellemzőket beállító függvények hívódnak meg. Létrehozom a rendergráfot, a megfelelő nézőpontot és projekciót tároló mátrixokat feltöltöm tartalommal, beállítom a szükséges események figyelését, a *loader* és a *util* csomagok függvényeinek segítségével az URL alapján beolvasom a később használt összes textúrát, és egy *textures* nevű tömb soron következő indexű elemében tárolom. Továbbá létrehozom a csomagot, létrehozok egy közös effektet minden alakzat számára, melybe betöltöm a vertex és fragment shaderek jellemzőit, beállítom a *setRenderCallback* függvényt, és meghívom az üdvözlő képernyőt felrajzoló függvényt.

Már az elején létrehozok minden szükséges alakzatot, és azok folyamatosan léteznek és elérhetők, függetlenül attól, hogy éppen láthatók-e a vásznon, vagy attól, hogy hanyadik új menetet játszuk, hiszen csak a hivatkozásokat változtatom.

4.2.3. A játék vezérlése

Mivel a játékban szinte minden időben változik, így nagyrészt minden fontos dolog a *setRenderCallback* által meghívott *onrender* függvényben történik, tehát a kód áttekintésekor érdemes ezzel a függvénnyel kezdeni, mert a metódusok jó része (közvetve vagy közvetlenül) innen hívódik meg. Itt ellenőrzöm a játék vezérlését segítő logikai változók értékeit, például ennek megfelelően jelenik meg a nyitóképernyő, kezdődik az új játék, illetve szüneteltethető vagy újra elindítható a játék.

A játékban kétféle egéreseményt figyelek, a kattintást és a mozgatást, mivel egy adott gomb fölé mozgatva az egeret, változik a gomb szövegének színe, míg rákattintva a gombra, a megfelelő akció hajtódik végre (pl. nehézségi szint beállítása, szabályok megjelenítése, játék indítása). Az űrhajó vezérlése a fel, le, jobbra, balra nyilak segítségével történik. Hogy ezek közül valamely gomb éppen le van-e nyomva, azt szintén folyamatosan ellenőrzöm az *onrender* függvényen belül.

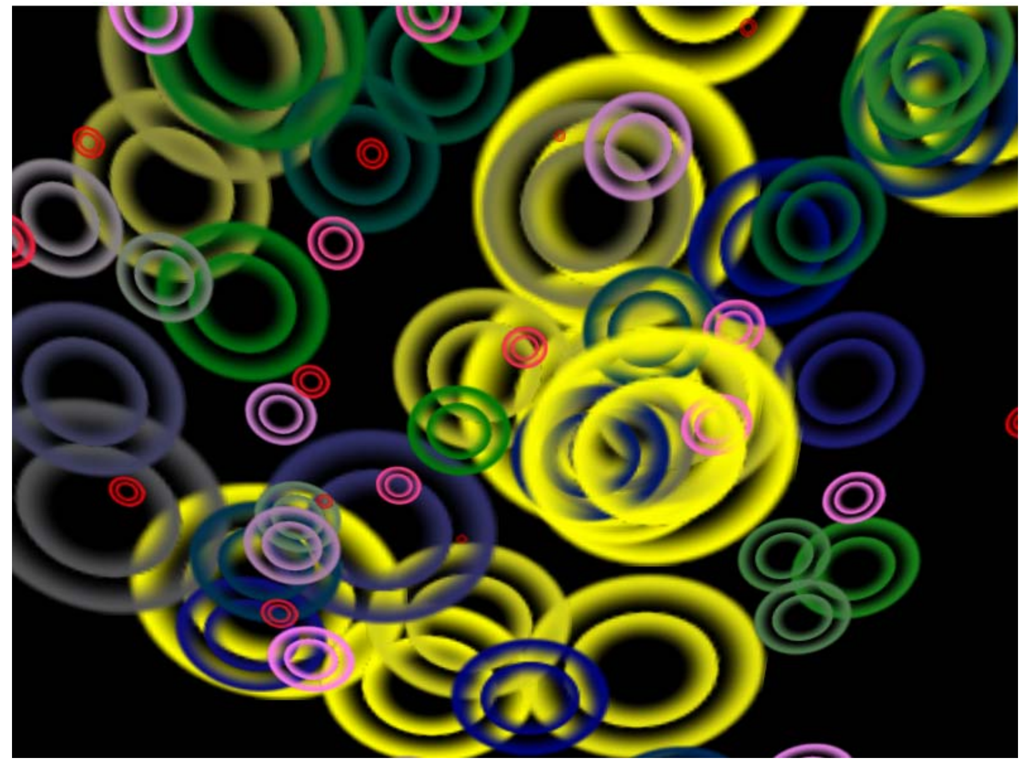
A következőkben az *onrender* függvényen belül található fontosabb utasításokat, függvényhívásokat veszem sorra, ugyanis ebből kiindulva válik érthetővé a kód, fedezhetők fel a kapcsolatok az egyes függvények között.

Az első fontos esemény a villogó színes köröket megjelenítő üdvözlő képernyő eltávolítása, és a játék nyitóoldalának megjelenítése, mely a 150. újrarenderelésnél történik

meg, mivel a *welcomeCount* változó értékét minden újrarenderelésnél 1-gyel növelem (az időt másodpercre is átszámolhatnám, később ennek módját ismertetni is fogom, de ebben az esetben ennek nincs túl nagy jelentősége, a játék idejének mérésénél viszont lesz):

```
welcomeCount++;
if(welcomeCount==150){
removeWelcomeScreen();
        mainMenu();
}
```

Az üdvözlő képernyő a 4.6. ábrán tekinthető meg. Mivel a nyitóoldalon és a játék szabályait ismertető oldalon egy animált meteor látható, ezért a következőkben azt is vizsgáljuk a globális logikai változók lekérdezésével, hogy ezen oldalak valamelyikén vagyunk-e éppen. Ha igen, akkor folyamatosan változtatjuk a füstös csíkot húzó meteor pozícióját, ezáltal az folyamatosan mozog. Ha már nem látszik a képernyőn, akkor újra a kezdőpozícióba helyezzük.



4.6. ábra: üdvözlő képernyő

A nyitóoldalon a játék nehézségének választása is megtörténik, alapértelmezetten a legkönnyebb szint kerül beállításra. A szabály oldal esetén további alakzatokat modellezek és animálok: a mozgó űrhajót és a szikrázó csillagokat. A leglényegesebb rész azonban a következő:

```
if((gameStarted)&&(!gameOver)){
        if(!initialized){
                initializeGame();
                createSpaceshipEnergy();
        } else {
                if(!pause){
                        var elapsedTime = renderEvent.elapsedTime;
                        jatekido += elapsedTime;
                        playGame();
                }
        }
}
```

Szintén logikai változók segítségével eldöntjük, hogy elkezdődött-e már az új menet, amennyiben igen (és még nem is jutott a játékos a menet végére), akkor először azon szükséges dolgokat állítom be és jelenítem meg, melyeket egyszer kell létrehozni a játék elején. Ilyen például a háttér, a képernyő felső részén található jelek, melyek a begyűjtött csillagok számát és a hátralevő energiát mutatják, az űrhajó modellje, a pause/play gomb, és a később felhasználásra kerülő alakzatok is (pl. meteor, robbanás). Egy újabb logikai változóval biztosítom, hogy az inicializáló függvények csak egyszer hívódjanak meg a játék kezdetén.

Amennyiben a pause gomb nincsen lenyomott állapotban, akkor a *playGame* függvényt hívom, mely ténylegesen vezérli az adott játék menetét. Ezen belül mozgatom az űrhajót, jelenítem meg és mozgatom a meteorokat és a csillagokat, vizsgálom az ütközést, változtatom a pajzs és az űrhajó állapotát, animálom a robbanásokat, valamint frissítem a képernyő tetején található információkat.

Egy másik fontos dolog a játékidő mérése, hiszen több dolog is függ az eltelt időtől (pontszám, meteorok sebessége). Erre a *renderEvent* objektum *elapsedTime* tulajdonságát használom, mely az előző renderelés óta eltelt időt mondja meg, azonban egy globális változóhoz minden alkalommal hozzáadva ezt az értéket, kiszámolhatom, hogy mennyi idő telt el összesen a játék kezdete óta. Természetesen a játékidőt tároló globális változó értékét minden egyes új menet kezdetekor nullára állítom. (Hasonlóan a többi, számlálásra vagy vezérlésre használt változó is alapértelmezett értéket kap minden új menet kezdete előtt.)

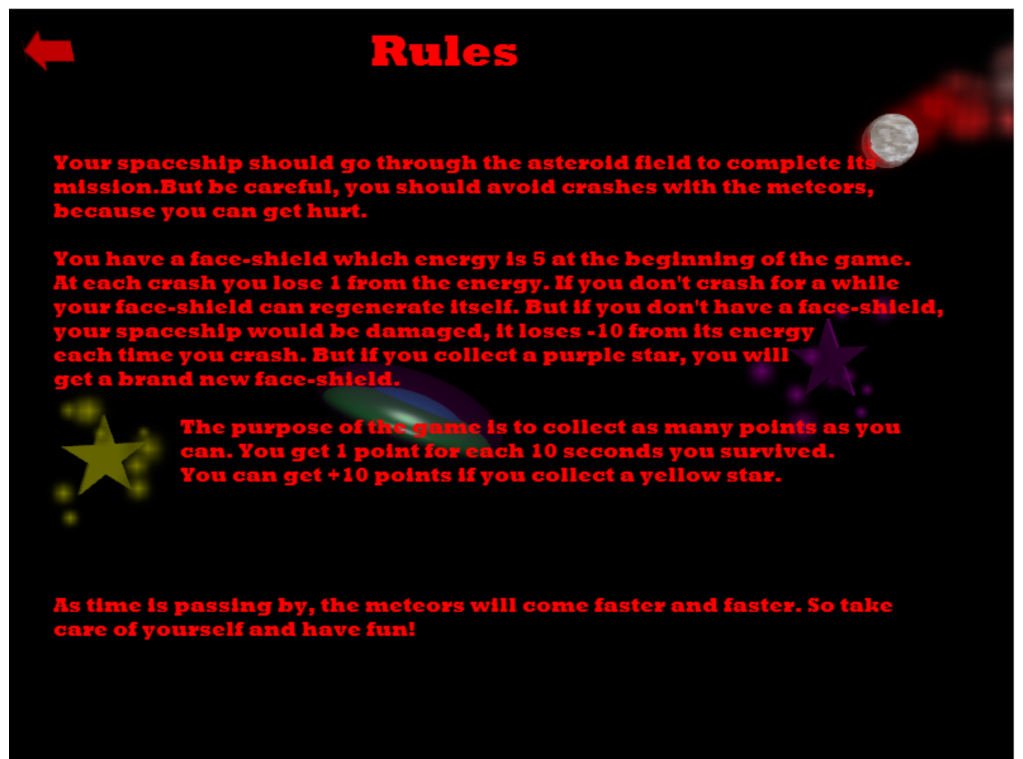
4.2.4. A keretrendszer további lehetőségei

Jó néhány előnyét és lehetőségét említettem már eddig az O3D keretrendszernek, azonban számos más, speciális lehetőségről még nem esett szó. Ezek közül emelnék ki néhányat, melyet a játék fejlesztése során is használtam.

Az egyik ilyen a *particles* nevű csomag, melynek segítségével különféle részecskéket modellezhetünk (maga a particle szó magyarul részecskét jelent), például lángot, füstöt, hullámfodrozódást, esőt. A játékban ennek több helyen is hasznát vettem, a segítségével történt az üdvözlőképernyő színes köreinek, a meteor füstjének, a robbanásoknak, valamint a csillagok szikráinak modellezése. A 4.7. ábrán a füstölgő meteor és a szikrázó csillagok modellezését láthatjuk a játékszabály képernyőjén.

Először egy rendszert kell létrehozni, mely a részecskéket kezelni fogja, ez az úgynevezett részecskerendszer, mely a többi alakzathoz hasonlóan egy csomagba kerül, ezen felül meg kell adnunk egy rendergráfot is, mely a nézet beállításait tartalmazza. Az alábbi példakódban a szikrázó csillag szikráinak létrehozását mutatom be, kezdve a részecskerendszerrel. Továbbra is ugyanabba a csomagba teszem, ahová eddig is került minden, és a globális nézetbeállításokat alkalmazom rá:

```
sparkling_particleSystem = o3djs.particles.createParticleSystem(
                g_pack,
                g_viewInfo);
```



4.7. ábra: a játékszabályok megjelenítése; a képernyőn megfigyelhető részecskék: a meteor füstje és a csillagok szikrái

A részecskerendszer létrehozása után részecske-kibocsátó objektumokat hozhatok létre az adott rendszerben. A következő példa a sárga csillag körül megjelenő szikrák létrehozását mutatja:

```
var yellowEmitter = sparkling_particleSystem.createParticleEmitter();
yellowEmitter.setState(o3djs.particles.ParticleStateIds.BLEND);
yellowEmitter.setColorRamp([0.5, 0.5, 0.0, 1.0]);
```

A részecskének beállíthatunk állapotjellemzőt a blendingre vonatkozóan, melyre különféle konstansok állnak rendelkezésre, valamint megadhatunk egy színskálát, mely alapján folyamatosan változik a részecske színe. A sárga szikrák esetén természetesen csak egy színt adok meg, a sárgát, a meteorfüst esetében azonban már kettőt (piros és szürke), míg az üdvözlőképernyő köreinél ötféle szín váltja egymást.

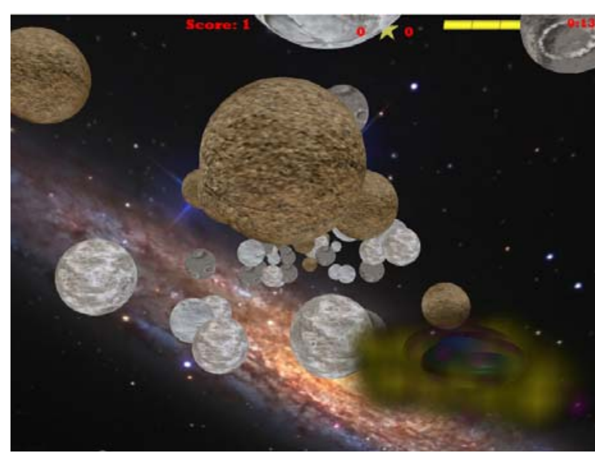
A részecske-kibocsátó objektumnak számtalan paraméterét beállíthatjuk, ennek segítségével lesz a végeredmény robbanás, szikra, láng vagy éppen füst. A szikráknál például beállítjuk a részecskeszámot, egy részecske élettartamát, azt, hogy milyen időközönként jöjjenek létre új részecskék, mekkora legyen a méretük kezdetben és a végén, mekkora területet fedjenek le:

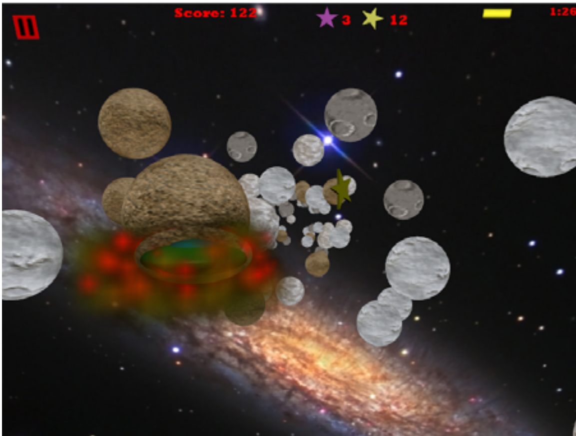
```
yellowEmitter.setParameters({
        numParticles: 10,
        lifeTime: 0.3,
        timeRange: 0.3,
        startSize: 5,
        endSize: 15,
        positionRange: [10, 10, 10]});
```

A paraméterek változtatásával más és más eredményt kaphatunk, valamint ezen felül más paraméterek is használhatók (a teljes lista az API-ban megtalálható). Ha készen van a szikra, akkor már csak hozzá kell rendelni egy transzformhoz, ami hasonló történik, mint más alakzatok esetében, annyi különbséggel, hogy maga az alakzat a részecske-kibocsátó objektum *shape* tulajdonságaként érhető el. A példa esetében szülőtranszformnak a sárga csillag transzformját adjuk meg, így a szikrán mindig ugyanazok a geometriai transzformációk fognak végrehajtódni, mint a csillagon, melyhez tartozik.

```
yellowSparkling = g_pack.createObject('Transform');
yellowSparkling.parent = rules_yellowStar;
yellowSparkling.addShape(yellowEmitter.shape);
```

A 4.8. ábrán a kétféle színű robbanás látható: ütközéskor lilás, illetve pirosas a robbanás attól függően, hogy van-e még pajzs, vagy már nincs.





4.8. ábra: különböző színű robbanások a meteorral való ütközéskor

A másik érdekes lehetőség a *picking* csomag használata. A *PickManager* objektumok a transzform gráf primitívjeinek kiválasztását kezelik, ennek megfelelően adott gyökértranszformhoz hozzuk létre őket, és a gyökértranszformból induló transzform-gráfot figyelik. A létrehozás az alábbi kódrészletben látható:

```
g_pickManager = o3djs.picking.createPickManager(g_client.root);
```

Ahhoz, hogy a *PickManager* adatai valóban helyesek legyenek arra nézve, hogy melyik primitív van kiválasztva, folyamatosan frissíteni kell az *update* nevű metódusának a segítségével:

```
g_pickManager.update();
```

A kiválasztás természetesen adott eseményhez kötődik, a játék esetében ez az egérgomb lenyomása. Tehát ennek az eseménynek a bekövetkezésekor hívódik meg a függvény, melyben a létrehozott *PickManager* objektum *pick* metódusa segítségével megvizsgáljuk, hogy adott primitívet érint-e a kattintás.

Ehhez először definiálnunk kell egy sugarat, melyről megvizsgáljuk, hogy a kattintás pozíciójából kiindulva és a belátható téren áthaladva átdöfi-e valamelyik primitívet. Az egér pozícióját az *Event* (*e*) objektumban megkapjuk, ezen kívül még a vászon nagyságára és a nézőpont és perspektíva jellemzőire van szükség a sugár létrehozásához:

```
var worldRay = o3djs.picking.clientPositionToWorldRay(e.x, e.y,
g_viewInfo.drawContext,       g_client.width, g_client.height);
```

A *pick* metódus csak az előbb létrehozott sugarat várja paraméterül, visszatéréskor pedig a kiválasztásról tartalmaz információkat. Ha semmi nincsen kiválasztva, akkor nullal tér vissza, ellenkező esetben viszont kinyerhetjük belőle azt az információt, hogy melyik alakzat

lett kiválasztva. Ehhez a *shapeInfo* tulajdonságát használjuk, mely maga is egy objektum, adott alakzatot, annak szülőjét és a hozzájuk tartozó *pickManager* objektumot tárolja. Ha az alakzathoz a létrehozáskor felveszünk olyan tulajdonságot, mely alapján könnyen azonosíthatjuk (a példa esetében ez a *name* tulajdonság, mely egy tetszőleges, de minden alakzat esetén különböző karaktersorozat), akkor ezt megvizsgálva könnyen felismerhető, hogy mely alakzatot tartalmazza a *shapeInfo*, vagyis mely alakzat lett kiválasztva. A kiválasztott alakzat alapján pedig végrehajtódhat a megfelelő akció:

```
var pickInfo = g_pickManager.pick(worldRay);
if (pickInfo) {
        picked = pickInfo.shapeInfo.shape.name;
  }
if(picked == 'startbutton'){
removeMainMenu();
gameStarted = true;
}
```

A *picking* csomag tehát lehetőséget nyújt az alakzatok kiválasztásának vizsgálatára úgy, hogy sem az alakzat kiterjedését, sem az alakzatnak az adott koordinátákhoz való viszonyát nem kell a programozónak számolni, mert ezt a csomag által kínált metódusok megteszik, ha átadjuk nekik az információkat, melyekből ki kell indulni.

Irodalomjegyzék

[1] Aaftab Munshi – Dan Ginsburg – Dave Shreiner: OpenGL ES 2.0 Programming Guide. Addison-Wesley, 2009.
[2] Andries van Dam – James D. Foley – John F. Hughes – Steven K. Feiner: Computer Graphics: Principles and Practice. Addison-Wesley, 1999
[3] David Flanagan: JavaScript: The Definitive Guide: Activate Your Web Pages. O'Relly Media, 2011
[4] Francis S. Hill Jr. – Stephen M. Kelley: Computer Graphics Using OpenGL. Prentice Hall, 2006
[5] John Pollock: JavaScript, A Beginner's Guide, Third Edition. McGraw-Hill Osborne Media, 2009
[6] Mark Pilgrim: HTML5: Up and Running. O'Relly Media, 2010
[7] Learning WebGL - learningwebgl.com/blog
[8] O3D homepage - http://code.google.com/p/o3d/
[9] Szirmay-Kalos László: Háromdimenziós grafika, animáció és játékfejlesztés. Computerbooks, 2003
[10] WebGL 1.0 Specification - www.khronos.org/registry/webgl/specs/1.0/
[11] WebGL Public Wiki - http://www.khronos.org/webgl/wiki/Main_Page

Nyilatkozat

Alulírott ……………….………… szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport ……………………….. Tanszékén készítettem, ………………….………..…… diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

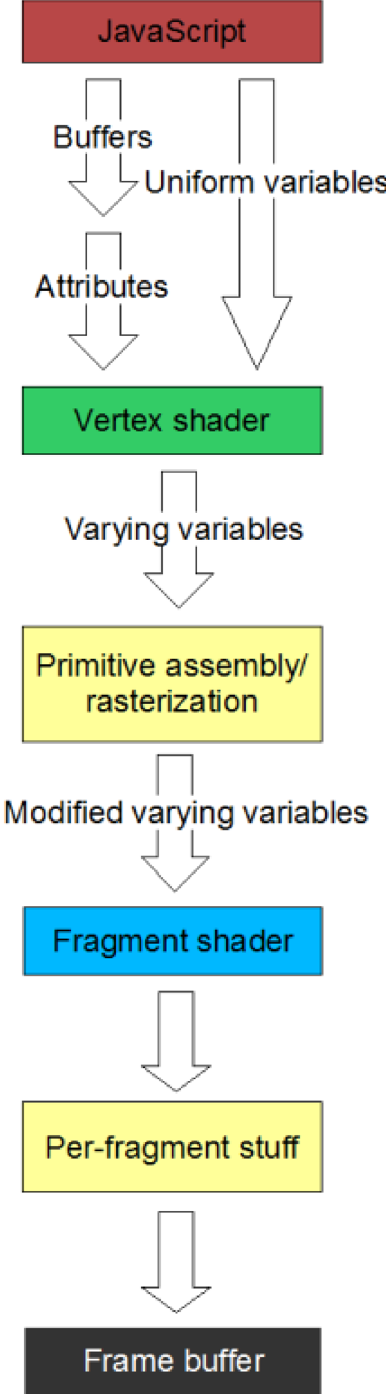
Dátum

Aláírás

Köszönetnyilvánítás

Ezúton szeretnék köszönetet nyilvánítani témavezetőmnek, Dr. Tanács Attilának a szakdolgozat írása során nyújtott segítségért.

Függelék



1.4. ábra: renderelés folyamata - forrás: www.learningwebgl.blog.com

A getShader függvény teljes kódja:

```
function getShader(gl, id) {
    var shaderScript = document.getElementById(id);
    if (!shaderScript) {
      return null;
```

```
    }

    var str = "";
    var k = shaderScript.firstChild;
    while (k) {
      if (k.nodeType == 3) {
        str += k.textContent;
      }
      k = k.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
      shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
      return null;
    }

    gl.shaderSource(shader, str);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert(gl.getShaderInfoLog(shader));
      return null;
    }

    return shader;
  }
```

Per-fragment/per-pixel módszert használó vertex shader teljes kódja:

```
<script id="per-fragment-lighting-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec3 aVertexNormal;
    attribute vec2 aTextureCoord;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    uniform mat3 uNMatrix;

    varying vec2 vTextureCoord;
    varying vec3 vTransformedNormal;
    varying vec4 vPosition;
```

```
    void main(void) {
        vPosition = uMVMatrix * vec4(aVertexPosition, 1.0);
        gl_Position = uPMatrix * vPosition;
        vTextureCoord = aTextureCoord;
        vTransformedNormal = uNMatrix * aVertexNormal;
    }
</script>
```

Per-fragment/per-pixel módszert használó fragment shader teljes kódja:

```
<script id="per-fragment-lighting-fs" type="x-shader/x-fragment">
    #ifdef GL_ES
    precision highp float;
    #endif

    varying vec2 vTextureCoord;
    varying vec3 vTransformedNormal;
    varying vec4 vPosition;

    uniform float uMaterialShininess;
    uniform float uAlpha;

    uniform bool uShowSpecularHighlights;
    uniform bool uUseLighting;
    uniform bool uUseTextures;

    uniform vec3 uAmbientColor;

    uniform vec3 uLightingDirection;
    uniform vec3 uDirectionalColor;

    uniform vec3 uPointLightingLocation;
    uniform vec3 uPointLightingSpecularColor;
    uniform vec3 uPointLightingDiffuseColor;

    uniform sampler2D uSampler;

    void main(void) {
        vec3 lightWeighting;
        if (!uUseLighting) {
            lightWeighting = vec3(1.0, 1.0, 1.0);
        } else {
            vec3 lightDirection = normalize(uPointLightingLocation -
vPosition.xyz);
            vec3 normal = normalize(vTransformedNormal);
```

```
            float specularLightWeighting = 0.0;
                          float directionalLightWeighting = 0.0;

                          directionalLightWeighting =
max(dot(vTransformedNormal.xyz, uLightingDirection), 0.0);

            if (uShowSpecularHighlights) {
                vec3 eyeDirection = normalize(-vPosition.xyz);
                vec3 reflectionDirection = reflect(-lightDirection, normal);

                specularLightWeighting = pow(max(dot(reflectionDirection,
eyeDirection), 0.0), uMaterialShininess);
            }


            float diffuseLightWeighting = max(dot(normal, lightDirection), 0.0);
            lightWeighting = uAmbientColor + uDirectionalColor *
directionalLightWeighting
                + uPointLightingSpecularColor * specularLightWeighting
                + uPointLightingDiffuseColor * diffuseLightWeighting;
        }

        vec4 fragmentColor;
        if (uUseTextures) {
            fragmentColor = texture2D(uSampler, vec2(vTextureCoord.s,
vTextureCoord.t));
        } else {
            fragmentColor = vec4(1.0, 1.0, 1.0, 1.0);
        }
        gl_FragColor = vec4(fragmentColor.rgb * lightWeighting, fragmentColor.a*
uAlpha);
    }
</script>
```