OpenGL alapok

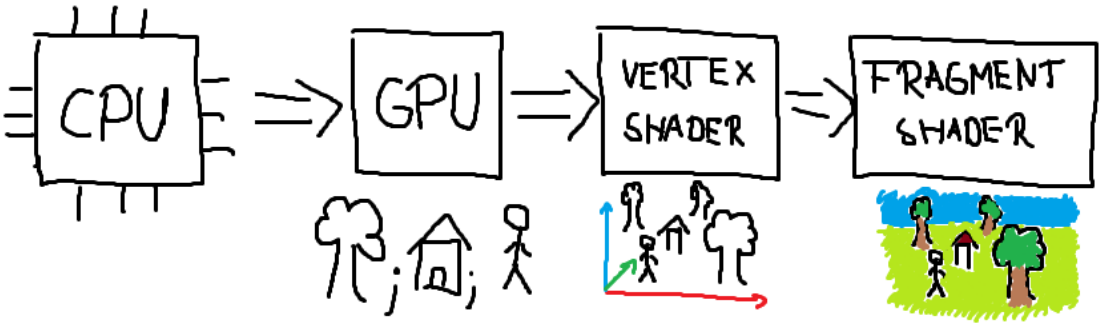
Csala Bálint

Tartalom

Rövid áttekintés 4
Az OpenGL használata 5
Kulcsok és adatok feltöltése 5
Vertexek 6
Háromszögek 7
Koordináta-rendszer 7
VBO-k és VAO-k 8
VBO 8
Létrehozás 9
Aktiválás 9
Feltöltés 9
VAO 10
Létrehozás 10
Aktiválás 10
Feltöltés 10
Végső kód 11
Shaderek 13
Shaderek működése 13
GLSL 14
Típusok 15
Vektorok 15
Konstans változók (const) 16
Bemeneti és kimeneti változók (in és out) 16
Uniform változók 17
Layout meghatározás 17
Beépített változók (gl_* család) 18
Shaderek kezelése a processzoron 18
Létrehozás és fordítás 18
Shaderek felhasználása 21
Uniformok kezelése 21
Végső kód 22
Textúrák 24
A processzoron 24

Létrehozás és feltöltés 24
Paraméterek 25
Mipmapok 27
Textúrákhoz szükséges vertex adatok 28
Felhasználásuk shaderekben....28
Hibakezelés 30
Régi mód....30
Új (egyszerű) mód 30
Egyéb fontos dolgok 32
Lépések a 3D-s rajzolás előtt 32
Feltakarítás 32
Shaderek létrehozása stringekben 32
Példakód 33

Rövid áttekintés



Az OpenGL egy API, aminek segítségével a grafikus kártyával kommunikálhatunk, így különböző 2 vagy 3 dimenziós jeleneteket alkothatunk, amik kihasználják a videókártyák erejét. Cserébe az elsajátítása nehézkes eleinte, de gyorsan megszokható.

A legelső lépés az adatok felvitele a videókártyára (innentől GPU). Ilyenkor minden olyan dolog, amire szükségünk van a rajzolás folyamán, átkerül (amennyiben egy játék vagy vizualizációs program futása során újabb forrásokra van szükségünk, például amikor egy játék új szintet tölt be, akkor ezt megtehetjük menet közben is persze). Ez nem csak 3d-s modelleket tartalmaz, hanem tiszta adatokat, textúrákat és még kisebb programokat is, amit a videókártyának kell később futtatnia (lásd.: shaderek).

Ezután szabadon felhasználhatjuk ezeket bármikor. Kirajzolásnál a GPU végigfut az általunk megírt vertex shaderrel az adatokon, ekkor a különböző objektumok a helyükre kerülnek és a shader felkészíti őket a 2 dimenziós térbe való projekcióra.

Miután a GPU ezt elvégezte, végigmegy az adatok által definiált háromszögeken, illetve vonalakon, azoknak minden pixelén meghívja az általunk megírt fragment shadert és végül eldönti, hogy a kapott pixeleknek egyáltalán látszódnia kell-e (ezt a lépést csak azért a kiszínezés után tudja végrehajtani, mert egyes effektusokhoz szükség van a fragment shadernek módosítania a pixel mélységén). Ennek a lépésnek figyelembe kell vennie a textúrákat, világítást és egyéb effektusokat. A végeredmény egy kiszínezett, elrendezett jelenet.

Megjegyzés

Ez lényegesen le van egyszerűsítve, a való életben még létezik ezeken kívül több shader típus is és a shadereket a CPU látja el extra információkkal még a feltöltés után is stb. Ezekre bővebben kitérek a saját szekciójukban.

Az OpenGL használata

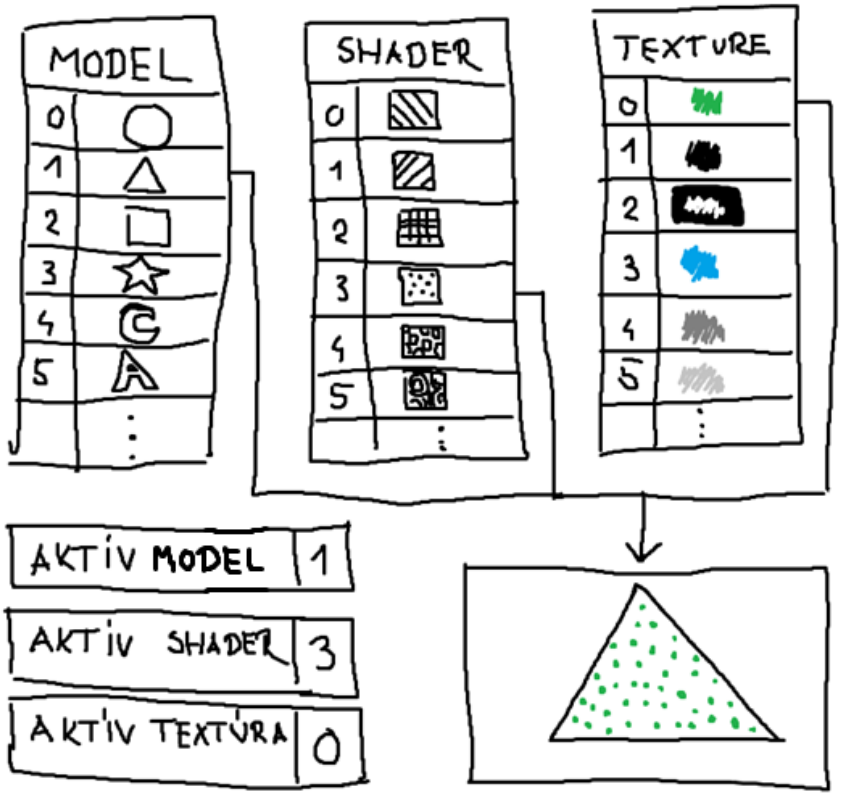
Az OpenGL mögött nincs egy közös, nyílt vagy akár zárt forráskódú implementáció, csak egy szabvány, ami meghatároz pár használható függvényt és a videókártya gyártókra bízza ezeknek a biztosítását. Attól függően, hogy milyen halmaza érhető el egy adott videókártya driveren ezeknek, határozza meg a használható OpenGL verziót.

A legtöbb modern kártya felmegy 4.6-ig (vagy felé, ha ezt a jövőben olvasod) manapság, de a laptopok nagyrészén is elérhető legalább a grafikához elvárt OpenGL 3 is.

Kulcsok és adatok feltöltése

Ahelyett, hogy az adatokat közvetlen kezelnénk a GPU-n (ami egy hagyományos helyzetben nem is lehetséges), az OpenGL kulcsokat ad, amin keresztül kezelni tudjuk azokat. Feltöltésnél ezekkel a kulcsokkal tudjuk megmondani, hogy mit hova szeretnénk rakni és kirajzolásnál pedig, hogy mit szeretnénk felhasználni.

Az OpenGL a háttérben egy hatalmas állapotgép. Mielőtt bármit használni tudnánk, először aktiválni kell azt. Erre gondolhatunk úgy, mint ha listákból raknánk össze a kellő végeredményt.



Ez nagyon hasznos, mivel, ha írunk egy shadert, ami képes egyfajta effektussal kirajzolni egy objektumot (pl.: valósághűen, rajzfilmszerűen) és mi ezt a stílust szeretnénk egy jelenet minden eleméhez felhasználni, akkor nem kell mindegyikhez egy külön shadert létrehozni. Ez persze elvárja, hogy kompatibilisek legyenek a shadereink az objektumainkkal, de ezt könnyen el lehet érni. Továbbá hatékonyabbá is teszi a rendszert, ha képesek vagyunk összevonni az olyan elemeket, amik osztoznak pár erőforásson (ugyanazt a textúrát és shadert használják például), mivel nem kell ki és bekapcsolni ezeket minden rajzolás előtt.

Vertexek

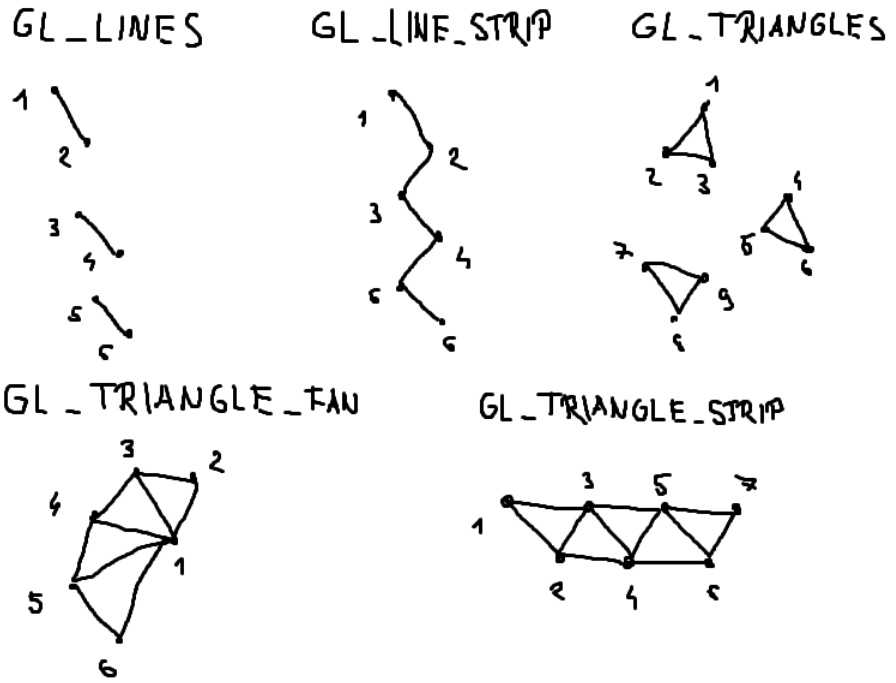
Az OpenGL-ben (és a legtöbb modern 3d API-ban) vertexeket használunk a modelljeink leírására. Ezek összekapcsolt adatok (általában pontok a térben, de nem szükséges adni nekik koordinátát minden esetben). Ilyen adat lehet a pont koordinátája, egy pontban a test normál vektora, a pont színe vagy bármilyen egyéb számokban mérhető tulajdonság.

Ezekből alkotunk különböző módszerekkel vonalakat és háromszögeket.

Megjegyzés

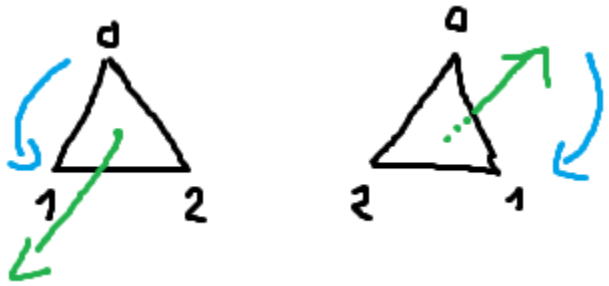
Bár elméletben lehetséges a négyszögek és egyéb sokszögek használata (legalábbis az OpenGL core (= ~asztali) verziójában, ami gépeken fut, de mobilon vagy a weben nem), a gyakorlatban ezek csak háromszögekké bomlanak a videókártyán és használatuk nem ajánlott.

A következő lehetőségeink vannak generálási módokra (OpenGL 3.2 után létezik még 4 ezeken kívűl, de ezekkel nem fogok foglalkozni):



Háromszögek

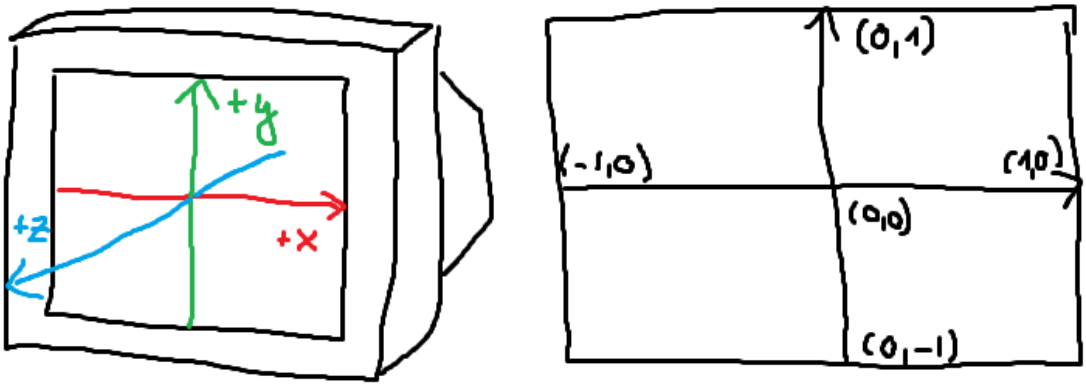
Fontos továbbá, hogy háromszögek esetében nem mindegy, hogy milyen sorrendben határozzuk meg a pontjaikat. Ez fogja eldönteni, hogy melyik a háromszög eleje és háta:



Egy gyakori optimalizálás, hogy kikapcsoljuk a rajzolását az olyan háromszögeknek, amiknek csak a hátát látjuk (mivel egy zárt modellben nem szabadna semmilyen esetben látnunk azokat). Ha viszont rosszul határozzuk meg a háromszögeinket, eltűnhetnek ennek hatására.

Koordináta-rendszer

Az OpenGL koordináta-rendszere közelebb áll a matematikai koordinátákhoz, mint a pixelekhez. Az y tengely felfelé tart és a Z tengely a felhasználó fele néz ki a képernyőből. Ezentúl az értékek (a program ablak felbontásától és képaránytól függetlenül) -1 és 1 közé esnek az x és y tengelyen.



Mivel manapság már elterjedtek a 4:3-tól egészen a 32:9 képarányú képernyők, ezért sokszor érdemes kirajzolás előtt átméretezni a jelenetet, hogy ne legyen kinyújtva az x tengely mentén.

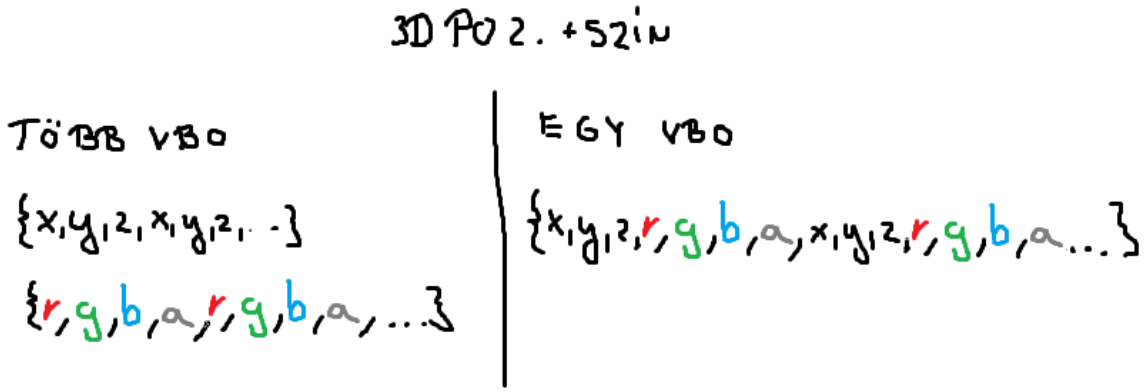
VBO-k és VAO-k

VBO

Modern OpenGL-ben az adatokat vertex buffer objektumokban (innentől VBO) tároljuk.

> ### *Megjegyzés*
>
> A régi megoldásban egyenként kellett bevinni minden pontját az alakzatoknak és automatikusan hozzá lehetett adni bonyolultabb effektusokat is a jelenethez (pl.: köd, bump mapping, cel shading) egyetlen függvényhívással. Ez a módszer mára teljesen elavult, nem is minden grafikus kártya képes kezelni és a mai alternatívák sokkal flexibilisebbek és gyorsabbak is, így érdemes azokat használni.

A VBO-kban a legegyszerűbb esetben egyfajta adatot tárolunk a vertexekről (pl.: a vertex 3d pozíciója, színe, normál vektora), de lehetséges egyszerre többet is, a különböző típusokat egymás után berakva:



A dokumentum keretei közt csak az első megoldással fogok foglalkozni, mivel a különbségek a mi felhasználási módunknál nem jelentősek, de lényegesen egyszerűsödik az első megoldással egy későbbi rész. Amennyiben viszont az írott kódnak minél gyorsabban kell futnia (pl.: modern játékok), akkor érdemes az utóbbit megfontolni. Ennek oka a memória lokalitás, a grafikus kártyának nem kell sokat mozognia a memóriában, hogy beolvassa az adatai egy vertexnek.

A VBO-k elképzelhetőek a GPU-n elhelyezkedő tömbökként. Miután létrehoztuk őket, aktiválhatjuk majd feltölthetjük azokat a kívánt adattal. A következő szekciókban összefoglalom ezeket a lépéseket részletesebben.

Létrehozás

Egy vbo létrehozása:

```
unsigned int vbo;
glGenBuffers(1, &vbo);
```

Több vbo létrehozása:

```
unsigned int vbos[n];
glGenBuffers(n, vbos);
```

A megadott tömb vagy érték fogja tárolni a létrehozott VBO-k kulcsait.

Aktiválás

Egy VBO-nak több fajtája is lehet, ezt az dönti el, hogy miként határozzuk meg aktiválásnál. A két legfontosabb az „array buffer” és az „element array buffer”:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
// vagy
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo);
```

Míg az egyszerű array bufferek adatot tárolnak egy vertexről, addig az element array buffer azt tárolja el, hogy milyen sorrendben kell a megadott pontokat felhasználni, hogy kihozzunk belőle egy modellt, így esetlegesen helyet mentve a duplikált vertexeken. Az esetek nagyrészében az előbbit használjuk.

Feltöltés

Miután aktiváltuk a VBO-nkat, rakhatunk bele adatot. Ehhez a glBufferData függvényt fogjuk használni. Ez a következő értékeket várja:

```
glBufferData(target, size, data, mode);
```

target

Ez lehet `GL_ARRAY_BUFFER` vagy `GL_ELEMENT_ARRAY_BUFFER` az alapján, hogy hogy aktiváltuk a VBO-nk.

size

Ez a feltöltendő adat mérete bájtokban. Érdemes a sizeof függvényt használni ehhez, a méret `sizeof(tipus) * hossz` lesz, ahol a típus a tomb elemeinek típusa, a hossz pedig a tomb elemszáma.

data

A tömb, amiben a feltöltendő adatok vannak tárolva.

mode

Az adat felhasználási módja. Számunkra két fontos értéket vehet fel: `GL_STATIC_DRAW` és `GL_DYNAMIC_DRAW`. Az előbbi érték megmondja a videókártyának, hogy az adaton nem fogunk változtatni, így gyorsabb, de nehezen elérhető memóriában is tárolhatja, míg a második esetben figyelmeztetjük, hogy rendszeres frissítést fog kapni a feltöltött információ, így egy könnyen elérhető helyre kell rakni.

VAO

Tiszta VBO-kkal már lehet dolgozni, viszont ez sokszor kényelmetlenné válhat, mivel kirajzolás előtt mindegyiket egyenként aktiválni kell és betölteni a megfelelő helyre, ezentúl modern OpenGL-ben ez már nincs is mindig megengedve. Ezt oldják meg a vertex array object-ek (innentől VAO-k). Ezek kötik össze a VBO-kat egy közös modellé. Nem közvetlenül a VBO-kat tárolják, hanem azoknak a felhasználási módját a kirajzoláshoz (erről bővebben a shaderek „Layout meghatározás” szekciójában).

Létrehozás

Létrehozásuk a VBO-khoz hasonló:

Egy VAO esetén

```
unsigned int vao;
glGenVertexArrays(1, &vao);
```

Több VAO esetén

```
unsigned int vaos[n];
glGenVertexArrays(n, vaos);
```

Aktiválás

Ezúttal csak egyetlen lehetőségünk van az aktiválásra

```
glBindVertexArray(vao);
```

Feltöltés

Mint ahogy írtam korábban, a VAO-k nem VBO-kat tárolnak, hanem beállításokat. Ahhoz, hogy ezeket hozzáadhassuk, aktívnak kell lennie a VAO-nak, amibe írni szeretnénk és a VBO-nak, amihez a beállításokat akarjuk kötni. Amennyiben több VBO-t is kezelni szeretnénk, egyenként kell rajtuk minden műveletet elvégezni. Miután minden rendben van, a következő két függvényt kell használnunk:

```
glEnableVertexAttribArray(attrib);
glVertexAttribPointer(attrib, size, type, normalized, stride, ptr);
```

attrib

Ez megegyezik a két hívásban, ez határozza meg, hogy melyik attribútumhoz kötjük a VBO-t és később a shadereknél lesz fontos. Érdemes 0-tól kezdve minden VBO-nál 1-gyel növelve megadni ezeket. Ezentúl érdemes több különböző VAO-ban ugyanazokat az adattípusokat ugyanahhoz az attribútumhoz kötni (pl.: A vertex színek legyenek mindig az 1-es attribútumhoz kötve, a normálok a 2-eshez, stb), hogy egy shader később több VAO-val is működhessen.

size

Ez meghatározza, hogy hány elemből áll az adat vertexenként. Például, ha 3d koordinátákat akarunk megadni, akkor ennek az értéknek 3-nak kell lennie. Színeknél (RGBA) 4-nek stb.

type

Az adat típusa. Általában `GL_FLOAT`, de lehet `GL_DOUBLE`, `GL_INT`, `GL_UNSIGNED_INT_2_10_10_10_REV`, stb.

normalized

Lehet `GL_TRUE` vagy `GL_FALSE`, ha `GL_TRUE`-t adunk át, normalizálja a kapott adatokat felöltéskor (általában nem szükséges).

stride

A távolság két vertex információi között. Amennyiben nem tárolunk semmit köztük, ez 0. Akkor jön elő, ha a VBO-ban többféle információt is tartunk egyszerre.

ptr

Az adatunk első eleme előtt lévő adatmennyiség bájtokban. A mi esetünkben majdnem mindig `nullptr`. Akkor jön elő, ha a VBO-ban többféle információt is tartunk egyszerre.

Végső kód

Ebben a példában létrehozok egy háromszöget és minden pontjához hozzárendelek egy színt is:

```
unsigned int vao;
unsigned int vbos[2];
// Vertex array legenerálása és aktiválása
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// A feltöltendő adatok, minden sor egy vertex-nyi
float vertices[] = {
        0, 0.5f,
        -0.5f, -0.5f,
        0.5f, -0.5f
};
float colors[] = {
        1, 0, 0, 1, // Piros
        0, 1, 0, 1, // Zöld
```

```
        0, 0, 1, 0  // Kék
};

// VBO-k legenerálása
glGenBuffers(2, vbos);
// Koordináták feltöltése
glBindBuffer(GL_ARRAY_BUFFER, vbos[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 6, vertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, nullptr);
// Színek feltöltése
glBindBuffer(GL_ARRAY_BUFFER, vbos[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 12, colors, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
```

Shaderek

A shaderek az OpenGL programozható pipeline-jának részei. Több fajtájuk létezik, de most csak kettővel fogunk foglalkozni, a vertex és a fragment shaderekkel. Ezeket egy C-szerű nyelvben, a GLSL-ben (Open**GL S**hading **L**anguage) kell megírni, majd az OpenGL segítségével lefordítani és felhasználni.

A shadereket a kódban stringként, vagy külön fájlokban tároljuk és nem fordítjuk le a többi kóddal együtt eleinte, csak később, amikor szükség van rájuk. Ennek oka, hogy míg a processzorok nagy része kétféle utasításkészlettre bontható (x86 vagy ARM), addig a grafikus kártyáknál nem csak a gyártók közt, hanem az egyes GPU architektúrák közt is jelentősek az eltérések. Ennél fogva egyszerűbb hagyni, hogy minden gyártó implementálja a saját fordítóját és csak futtatáskor lefordítani ezeket.

Shaderek működése

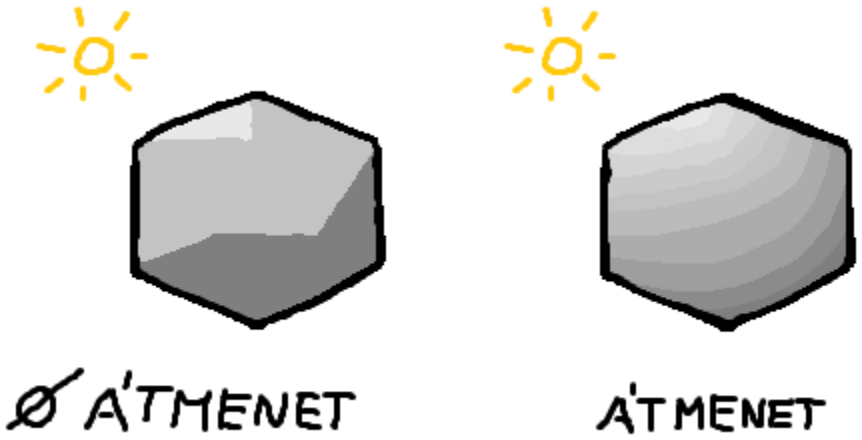
Három alap shader típus létezik, ebből kettőre lesz egyelőre szükségünk: a vertex és a fragment shaderekre.

Miután feltöltöttük az adatokat a grafikus kártyára és elkezdjük a kirajzolást, a GPU először betölti egyenként a vertexekhez az adatokat a vertex shaderbe. Ennek feladata ezekből és egyéb külső változókból (lásd: uniform változók) összerakni a vertexek koordináta-rendszerbeli helyét és továbbítani a maradék információt a fragment shader felé. Itt történik az összes olyan transzformáció is, ami átviszi a modelleinket a saját koordináta rendszerükből az OpenGL koordináta-rendszerébe.

Miután ez megtörtént, a grafikus kártya begyűjti a vertex shader által kiadott adatokat, majd a generált pozíciót leosztja a vektor „w” (negyedik) komponensével (ennek a perspektivikus projekció során van szerepe) és a kapott vertexekből vonalakat vagy háromszögeket alkot a kirajzolási módtól függően. Ezeket raszterizálja (= végigmegy az összes pixelen, amit a síkidom lefed), ellenőrzi, hogy minden feltételt teljesít-e az adott pixel (például 3d-s jelenetekben nincs-e lefedve valahogy), majd a vertex shadertől kapott értékeket súlyozottan átlagolja és átadja a fragment shadernek. Az átlagolásnál figyelembe veszi a távolságot a vertexektől. Erre azért van szükség, hogy sima átmenetek legyenek a pontok között. Vegyük példaként a normál vektorok esetét. Ezek határozzák meg egy felület irányát egy adott pontban. Ha például a következő háromszöget nézzük oldalnézetből a normálvektorokkal együtt:

Itt a normál vektorok a következő módon lesznek eloszlatva:

Ennek hatása, hogy bár a háromszögünk továbbra is lapos, a kirajzolásnál a fény másmilyen normál vektor szerint lesz számolva minden pixelen, így egy sima felület hatását fogja kelteni.



Ezt a folyamatot *interpolációnak* nevezzük.

> ## *Megjegyzés*
>
> Ez csupán az alapműködése a shadereknek. Különböző kulcsszavakkal megoldható például, hogy ne legyenek interpolálva a normál vektoraink, ha rajzfilmes kinézetet szeretnénk mondjuk elérni.

GLSL

Ahogy említettem, a GLSL egy C-szerű nyelv, így a legtöbb elem ismerős lesz mindenkinek, aki valaha is dolgozott már azzal, de pár extra függvényt és típust is találunk, amik megkönnyíthetik az életünket.

Az első sora minden shadernek a következő:

```
#version <verzió>
```

Ahol a „<verzió>” részbe a jelenlegi verzió kerül. Ez a grafika háziban „`330 core`”, ami azt jelenti, hogy a glsl 3.3.0 verziójának „core” halmazát használjuk (ez az „asztali” verzió megkülönböztetője, telefonokon vagy a weben például nem használható). A következő sor általában a float alapú értékeket állítja be:

```
precision highp float;
```

Eszerint a float precizitása magas legyen (highp = **high p**recision). Hasonlóan létezik alacsony (lowp) és közepes (mediump) precizitás is. Ez gépes kártyákon nem jelent sokat, így érdemes a highp-n tartani, de ha mobilon fejlesztünk vizuális appokat, megéri a sebesség kedvéért egy alacsonyagg fokozatra váltani, ha lehetséges.

Egy további közös rész a main függvény:

```
void main() {
    // ...
}
```

Típusok

A GLSL a következő extra, számalapú típusokat határozza meg:

- vec2, vec3, vec4 – float típusú vektorok (2, 3, illetve 4 komponensűek)
- ivec2, ivec3, ivec4 – integer típusú vektorok
- dvec2, dvec3, dvec3 – double típusú vektorok
- bvec2, bvec3, bvec4 – boolean típusú vektorok
- uvec2, uvec3, uvec4 – pozitív szám típusú vektorok
- mat2, mat3, mat4 – négyzetes, 2x2, 3x3, illetve 4x4-es vektorok (hasonlóan léteznek egyéb típusok a vektorok mintájára)
- mat2x3, mat2x4, mat3x2, mat3x4, mat4x2, mat4x3 – Nem négyzetes mátrixok (az első érték a szélesség, a második a magasság, ugyanúgy léteznek egyéb típusok)

Vektorok

A GLSL biztosít pár vektor típust a grafikai számítások egyszerű elvégzéséhez. Ezekkel lehet különböző műveleteket végezni (elemenkénti összeadás, kivonás, osztás, szorzás, amiket a szokásos operátorok segítségével hajtunk végre; +, -, /, *), továbbá biztosít több beépített függvényt is, amik a segítségünkre lehetnek. Ezek közül a fontosabbak a skaláris szorzat (`dot(a, b)`), vektoriális szorzat (`cross(a, b)`), a vektor hossza (`length(v)`) és a vektor egység hosszúvá alakítása/normalizálása (`normalize(v)`).

Továbbá lehetséges vektorokat mátrixokkal megszorozni az * operatorral, ilyenkor az oldaltól függően sor vagy oszlop vektorként is funkcionálhatnak.

A vektoroknak további hasznos funkciói is vannak, ezeket egyenként szeretném részletesen kifejteni.

„Swizzling”

(Igen, ez a neve)

A vektorok négy komponensét le lehet kérni az x, y, z és w nevekkel a következő módon:

```
myVec3.x, myVec3.y, myVec3.z, myVec3.w
```

Ezentúl lehetséges a vektornak egyszerre több komponensét is lekérni, ezzel egy vektort kialakítva. Például:

```
vec3 myVec3 = vec3(1, 2, 3);
vec2 myVec2 = myVec3.xy; // vec2(1, 2)
```

Az egyes komponensek sorrendje is felcserélhető és akár meg is ismételhetőek egynél többször:

```
vec2 myVec2 = vec2(1, 2);
// vec4 myVec4 = vec4(myVec2, myVec2.y, myVec2.x); helyett
vec4 myVec4 = myVec2.xyyx; // vec4(1, 2, 2, 1)
```

Extra elnevezések

A vektorok komponenseinek „x, y, z, w” elnevezései helyett kettő másik is használható, az „r, g, b a”, ami színeknél hasznos, és az „s, t, p, q”, ami UV koordinátáknál (erről bővebben a textúrák „Textúrákhoz szükséges vertex adatok” szekciójában). Ezekkel is alkalmazható az előbb bemuta

Automatikus szétbomlás

Amennyiben egy vektor konstruktorral akarsz átalakítani egy vektort egy magasabb komponensszámúvá, akkor nincs szükség arra, hogy egyenként leírd a vektor komponenseit így:

```
vec4(myVec2.x, myVec2.y, 0, 1);
```

Elég, ha megadod a vektort az első értéknek, utána automatikusan szétosztja a komponenseit:

```
vec4(myVec2, 0, 1);
```

Konstans változók (const)

Olyan változók, melyek értéke konstans marad a program futása alatt, így optimalizálhatóak.

```
const float a = 10;
const vec3 b = vec3(1, 2, 3);
```

Bemeneti és kimeneti változók (in és out)

Ezek jelentése függ attól, hogy éppen vertex vagy fragment shaderekről beszélünk.

Vertex shaderekben a bemeneti változó (in) közvetlen megkapja a jelenlegi vertexhez tartozó adatokat a bekötött VBO-kból az attribútum lista alapján (részletesebben lásd: „Layout meghatározás”). Ezek alapján kell legenerálnia a koordinátákat és átadni a fragment shadernek a további adatokat. Ez utóbbi lépés a kimeneti változókkal történik (out). Példa:

```
in vec4 inputColor;

out vec4 colorToFragmentShader;

void main() {
    colorToFragmentShader = inputColor;
}
```

Fragment shaderekben a bemeneti változók a vertex shader által legenerált értékeket kapják meg, miután átestek az interpoláción. Ahhoz, hogy ez a kapcsolat működjön, a változó neveknek a két shader közt meg kell egyezniük. A kimeneti változók fragment shadereknél a kimeneti színeket tartalmazzák, erről részletesebben a „Létrehozás és fordítás” szekció beszél. az előző vertex shaderhez illő példa:

```
in vec4 colorToFragmentShader;

out vec4 fragmentColor;

void main() {
    fragmentColor = colorToFragmentShader;
}
```

Uniform változók

Az uniform változók olyan értékeket tárolnak, amelyeket nem akarunk megváltoztatni vertexenként. Ilyen például egy test transzformációja, vagy akár a felhasználandó textúra. Ezeket a `uniform` kulcsszóval látjuk el és utólag, a rajzolás előtt töltjük fel. Például:

```
uniform mat4 transformation;
```

Layout meghatározás

Ahhoz, hogy a shaderjeink és a VBO-ink képesek legyenek kommunikálni egymással, kell egy kompatibilis interfészt létrehozni a kettő közt. Ezt részben már meg is tettük, amikor feltöltöttük a VAO-kat az adatokkal. Két megoldás van: Vagy hagyjuk, hogy a shader magától kitalálja a layout-ot és mi ez alapján hozzuk létre a VAO-inkat (enyhény nehezebb, de jobban optimalizált kódot eredményezhet), vagy a VAO-knál határozunk meg egy konstans felépítést és a shadert kérjük meg, hogy ezt kövesse. Mi ez utóbbit fogjuk most használni.

A shaderjeinktől a layout kulcsszóval kérhetünk egy adott elrendezést. Ennek a bemeneti változókon a következő a szintaxisa

```
layout(opcio1 = ertek1, opcio2 = ertek2 /*,,,*/) in vec2 pos;
```

Minket ezúttal a legtöbb opció nem érdekel, csak a location nevű. Ez határozza meg, hogy a változó az attribútum lista melyik elemének felel meg. Korábban ezt kellett beállítani a VAO-knál a glVertexAttribPointer függvényben az index paraméterrel.

Tehát, ha a VAO-nkba a következő adatot vittük be valamelyik VBO-nkról

```
glVertexAttribPointer(5, 2, GL_FLOAT, GL_FALSE, 0, nullptr);
```

Akkor az ehhez kapcsolódó bemeneti változónak a következőképpen kell kinéznie:

```
layout(location = 5) in vec2 valtozoNeve;
```

Beépített változók (gl_* család)

A GLSL nyelv számos beépített változót definiál, ezekből egyet kötelező is használni, a többi opcionális. Két fontosabb:

gl_Position

Ezzel lehet definiálni vertex shaderben a koordinátái a kapott vertexnek:

```
layout(location = 0) in vec2 pos;

void main() {
    gl_Position = vec4(pos, 0, 1);
}
```

gl_FragCoord

Megadja a jelenlegi fragment (~pixel) koordinátáját a fragment shaderben. Ennek segítségével akár egyetlen téglalap segítségével és ray tracing-gel teljes jeleneteket lehet alkotni (vigyázat, lassú!).

Shaderek kezelése a processzoron

Létrehozás és fordítás

> ***Megjegyzés***
>
> A házifeladatokhoz erre a részre nincs szükség, mivel a framework biztosít erre egy beépített osztályt (GPUProgram). Ennek ellenére ajánlom legalább az egyszeri átfutását, amennyiben szükség lesz rá a jövőben vagy akár egy ZH folyamán.

A shaderek felépítése viszonylag egyszerű és hasonlít a VBO-kra. Először létre kell hozni a két shadernek egy-egy shader objektumot a GPU-n, majd feltölteni ebbe a forráskódot, végül lefuttatni. Mivel a shadereket ritkán használjuk egymástól függetlenül, ezért ezután ezeket be kell rakni egy programba, amit linkelni kell.

A shader kezelő függvények stílusa jelentősen eltér az eddig megszokottaktól. Például „`glGen`” helyett a generálló függvények „`glCreate`”-tel kezdődnek, továbbá nem aktiváljuk a shadereket használat előtt, hanem mindig átadjuk a kulcsát a függvényeknek.

Először hozzunk létre egy shader-t:

Vertex shader:

```
unsigned int shader = glCreateShader(GL_VERTEX_SHADER);
```

Fragment shader:

```
unsigned int shader = glCreateShader(GL_FRAGMENT_SHADER);
```

(Innentől a két típus kezelése megegyezik, így érdemes ehhez egy függvényt létrehozni.)

A „`shader`” értéket az előző kódból a shader kulcsának fogom innentől nevezni. Miután létrehoztuk a shadert, adjuk meg a forráskódját:

```
glShaderSource(shader, count, strings, lengths);
```

shader
: A shader kulcsa.

count
: A strings és lengths tömbök hossza.

strings
: A shader kódjait tároló stringekből álló tömb. A tömb elemeit a fordító összerakja és egyben fordítja le, így részletekben is betölthetők a shaderek. Általában elég egy stringet eltárolni és ennek egy pointerét megadni.

lengths
: A sringek hosszai. Amennyiben a stringek null terminálták (\0 az utolsó elem, C-ben általában igaz), akkor ez helyettesíthető `nullptr`-rel.

Amennyiben a shader kulcsa „shader” és a kódot egy `const char *const` típusú, shaderSource nevű változóban tároljuk, akkor a következő kódot használnánk:

```
glShaderSource(shader, 1, &shaderSource, nullptr);
```

Ezután le kell fordítanunk őket. Ehhez a következő függvényt kell használni:

```
glCompileShader(vshader);
```

Ahogy más nyelveknél is, itt is véthetünk programozás közben hibákat, amik megakadályozzák a fordítást. Annak érdekében, hogy ezeket ki tudjuk javítani, az OpenGL biztosít egy log-ot a felmerülő hibákról, azonban ezt le kell külön kérnünk. Ehhez a következő lépéseket kell megtenni:

Először kérjük le, hogy történt-e valami a fordítás közben. Ennek az eredménye 0, amennyiben valami baj van:

```
int statusCode;
glGetShaderiv(shader, GL_COMPILE_STATUS, &statusCode);
if (!statusCode) {
    // ...
}
```

A `glGetShaderiv` függvény, ahogy a neve is sejteti, egy shader paramétert kér le, aminek a típusa integer (innen jön az „i” a végén), majd berakja azt a kapott pointerbe. Minket a

`GL_COMPILE_STATUS` konstanssal elérhető érték érdekel. Innentől az if-en belül fogom a kódot folytatni.

Mivel C++-t használunk, így a stringek hosszát ismernünk kell létrehozás előtt. Ehhez szerencsére szintén lekérhetjük ezt hasonló módon, ezúttal a `GL_INFO_LOG_LENGTH` konstans használatával:

```
int length;
glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length);
char info[length + 1]; // +1 a null terminátor miatt
```

Ezután lekérhetjük és kiírhatjuk a log-ot:

```
glGetShaderInfoLog(shader, length, nullptr, info);
printf(info);
```

Miután ezt elvégeztük, elkészültünk a shaderekkel és létrehozhatunk egy programot, amiben összekötjük őket. Ehhez először hozzuk létre:

```
unsigned int program = glCreateProgram();
```

Ezután adjuk hozzá a két létrehozott shadert a kulcsuk alapján:

```
glAttachShader(program, vshader); // vshader a vertex shaderem kulcsa
glAttachShader(program, fshader); // fshader a fragment shaderem kulcsa
```

Itt érdemes (bár általában nem kötelező) megmondani az OpenGL-nek a kimeneti változó nevét, amiben a végső színt tároljuk el a fragment shaderben. Ehhez a következő függvényt kell meghívni:

```
glBindFragDataLocation(program, colorNumber, name);
```

program

A program kulcsa, amit feljebb hoztunk létre.

colorNumber

A kimeneti szín száma. Ez általában 0. Akkor van rá szükség, ha több színt is szeretnénk egyszerre kiírni, amire más rajzolási módoknak van szüksége.

name

A kimeneti változó neve.

Miután ezt elvégeztük, linkelhetjük a programot. Ezt el lehet képzelni a véglegesítéseként:

```
glLinkProgram(program);
```

A shaderek különlegesek olyan értelemben, hogy az eredeti shader objektumok (nem a program!) törölhető, miután meghívtuk a `glLinkProgram`-ot. Ezt érdemes is azonnal végrehajtani:

```
glDeleteShader(vshader); // vshader a vertex shaderem kulcsa
glDeleteShader(fshader); // fshader a fragment shaderem kulcsa
```

Shaderek felhasználása

Mielőtt egy shadert igénybe akarnánk venni, aktiválni kell azt a VAO-khoz hasonlóan. Ehhez a `glUseProgram` függvényt kell használni:

```
glUseProgram(program);
```

Uniformok kezelése

Amennyiben szeretnénk értékeket rendelni egy uniform változóhoz, először le kell kérnünk hozzá egy kulcsot. Ez egy költséges lépés, így érdemes a program elején ezt elvégezni, majd eltárolni a kapott értéket, amit a következő módon kaphatunk meg:

```
int location = glGetUniformLocation(program, name);
```

Ahol `name` az uniform változó neve. Fontos, hogy ez az egyik egyetlen eset OpenGL-ben, hogy a kulcs típusa nem pozitív (unsigned) változó.

Miután lekértük a kulcsát, fel tudunk tölteni adatokat. Ehhez a `glUniform` függvénycsaládot vehetjük igénybe. Ezeknek két típusa van:

Vektorok:

```
glUniform<szám><típus>
// Például
glUniform1i
glUniform3f
```

Itt található az összes skalárt (egyszerű szám) és vektort feltöltő függvény. Az első szám határozza meg az értékek mennyiségét (1 – skalár, 2 – 2 komponensű vektor, 3 – 3 komponensű vektor, 4 – 4 komponensű vektor), ezután pedig a típus mondja meg az értékek típusát (i – integer, f – float, d – double, stb.). A függvények csak az uniform kulcsát és a megadott mennyiségű értéket kérik. A típus meghatározás után még a „v” is opció a függvény nevében (pl.: `glUniform4fv`), ezekben a paramétereket egy tömbben kell megadni.

Mátrixok:

```
glUniformMatrix<méret><típus>v
// Például
glUniformMatrix4fv
glUniformMatrix2x3dv
```

Itt hasonló módon a méret a mátrix méretét adja meg (egy darab szám egy négyzet mátrixot csinál, két szám egy x-szel elválasztva pedig minden mást képes), a típus pedig az értékek típusát. Ahogy a „v” is mutatja a függvény nevének végén, az értékeket mindig tömbben kell megadni. A függvények a következő értékeket kérik (egy 4x4-es mátrix függvényén bemutatva):

```
glUniformMatrix4fv(location, count, transpose, value);
```

location

Az feltölteni kívánt uniform változó kulcsa.

count

A feltölteni kívánt mátrixok száma. Amennyiben nem tömb típusú az uniform változó, ez legyen 1.

transpose

Szükséges-e transzponálni a mátrixot feltöltés előtt?

value

A tömb, ami eltárolja a mátrix értékeit.

Végső kód

```
unsigned int createShader(unsigned int type, const char *const source) {
    unsigned int shader = glCreateShader(type);
    glShaderSource(shader, 1, &source, nullptr);
    glCompileShader(shader);

    int statusCode;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &statusCode);
    if (!statusCode) {
        int length;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length);
        char info[length + 1]; // +1 a null terminátor miatt
        glGetShaderInfoLog(shader, length, nullptr, info);
        printf(info);
    }

    return shader;
}

unsigned int createProgram(const char *const vertexShaderSource, const char
*const fragmentShaderSource,
                           const char *outColorName) {
    unsigned int vertexShader = createShader(GL_VERTEX_SHADER,
vertexShaderSource);
    unsigned int fragmentShader = createShader(GL_FRAGMENT_SHADER,
fragmentShaderSource);
    unsigned int program = glCreateProgram();
    glAttachShader(program, vertexShader);
    glAttachShader(program, fragmentShader);
```

```
    glBindFragDataLocation(program, 0, outColorName);

    glLinkProgram(program);

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    return program;
}
```

Textúrák

A processzoron

A textúra adatának beszerzésével most nem foglalkozunk, mivel rengeteg módja van (fájból betöltés, procedurális generálás stb.) és nem tartozik egyik sem közvetlen az OpenGL-hez. Ezentúl feltételezem, hogy van egy tömböd, benne a szükséges adatokkal úgy, hogy a színek komponensei (r, g, b, a) egymás után következnek benne.

Létrehozás és feltöltés

Megjegyzés

A házifeladatokhoz erre a részre nincs szükség, mivel a framework biztosít erre egy beépített osztályt (Texture). Ennek ellenére ajánlom legalább az egyszeri átfutását, amennyiben szükség lesz rá a jövőben vagy akár egy ZH folyamán.

A textúrák függvényei a VBO-k és VAO-k függvényeire hasonlítanak, ugyanúgy is kell őket kezelni. Létrehozásnál kell a kulcs tárolásához egy pozitív szám:

```
unsigned int texture;
glGenTextures(1, &texture);
```

Ezután a VBO-khoz hasonlóan meg kell határoznunk a típusát. Ez lehet egy 2 dimenziós textúra (`GL_TEXTURE_2D`), de akár több textúra egymáson (`GL_TEXTURE_2D_ARRAY`) vagy akár még egy 3 dimenziós textúra is (`GL_TEXTURE_3D`) (Ez utóbbinál minden színt egy irányvektorhoz rendelünk. Fő felhasználási területük a skyboxok, a játékost körülvevő, látszólag végtelen távol lévő égboltok). Én a továbbiakban egy átlagos, 2d textúrát feltételezek:

```
glBindTexture(GL_TEXTURE_2D, texture);
```

Végül feltölthetjük a kívánt adattal:

```
glTexImage2D(target, level, internalFormat, width, height, border, format,
type, pixels);
```

target

A módosítani kívánt textúra típusa, például `GL_TEXTURE_2D`.

level

A módosítani kívánt mipmap szint, erről bővebben a „Paraméterek” szekcióban.

internalFormat

A textúra belső formátuma, például `GL_RGBA` vagy `GL_RGB`. Megmondja, hogy a grafikus kártyán milyen formában legyen eltárolva a textúra.

width

A textúra szélessége pixelekben.

height

A textúra magassága pixelekben.

border

Modern OpenGL-ben kötelezően 0.

format

A megadott adat formátuma. Például `GL_RGBA` vagy `GL_RGB`.

type

A megadott adat típusa. Sok esetben `GL_UNSIGNED_BYTE`, mivel a legtöbb kép 8 bites színeket tartalmaz, de különlegesebb esetekben sokféle értéket támogat

pixels

Az adatot tároló tömb.

Paraméterek

A textúráknak vannak beállítható paraméterei. Ezekből néhány kötelező, de a legtöbb opcionális. A beállításokat a következő két függvénnyel elvégezni a beállított érték alapján, de mi csak az integer típusút fogjuk használni:

```
glTexParameterf(target, pname, param)
```

```
glTexParameteri(target, pname, param)
```

A target az előző függvényekben is már használt típus (pl.: `GL_TEXTURE_2D`), a `pname` a megváltoztatni kívánt paraméter kulcsa, a `param` pedig az érték, amivé változtatni szeretnénk. A következő két paramétert kötelező beállítani:

GL_TEXTURE_MIN_FILTER

Beállítja a textúra kirajzolásának módját, ha az kisebbként kerül ki a képernyőre, mint az eredeti mérete. Hat lehetséges értéke lehet a paraméternek, ebből 2 a `GL_NEAREST` és `GL_LINEAR`, a többiről viszont csak a „Mipmapok” szekcióban fogok írni. A `GL_NEAREST` a legközelebbi pixelt választja ki a lefedett pixelekből, míg a `GL_LINEAR` átlagolja őket.

GL_TEXTURE_MAG_FILTER

Beállítja a textúra kirajzolásának módját, ha az nagyobban van kirajzolva, mint az eredeti mérete. A két lehetséges paraméter ebben az esetben a `GL_NEAREST` és `GL_LINEAR`, ezek a `MIN_FILTER`-hez hasonlóan rendre a közelebbi pixelt, vagy a közeli pixelek súlyozott átlagát használják. A `GL_NEAREST` gyakori neve (főleg rajzprogramokban) „nearest neighbor”.

Kiemelném továbbra a következő nem kötelező, de hasznos paramétereket is:

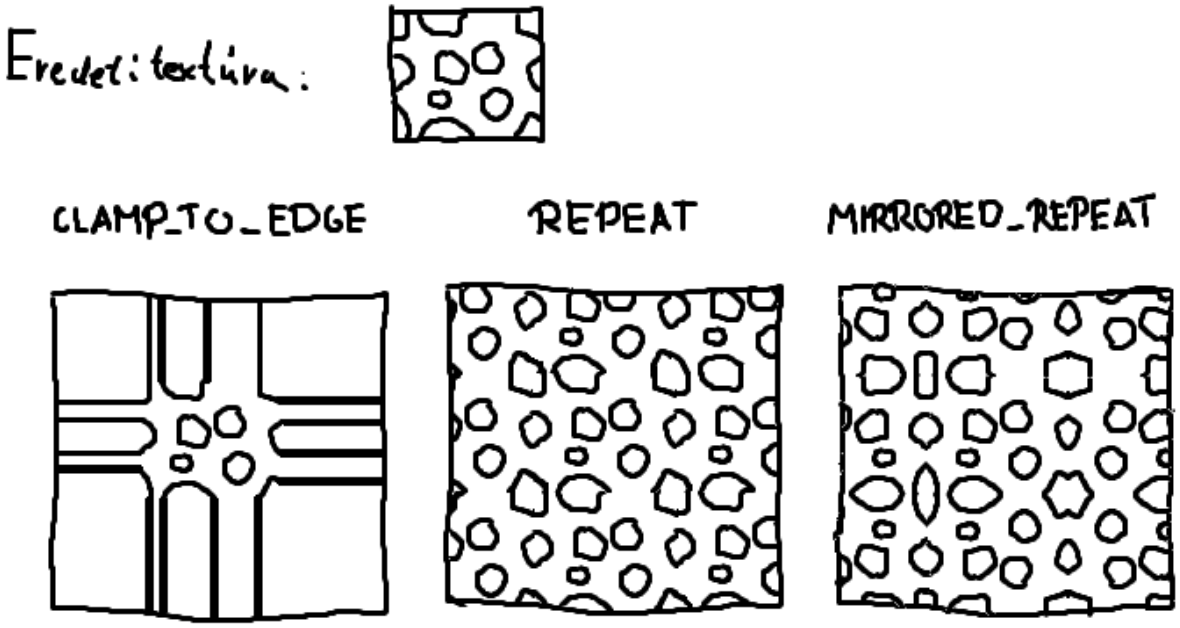
GL_TEXTURE_MAX_LEVEL

Beállítja a maximum mipmap szintjét a textúráknak. Erről bővebben a „Mipmapok” szekcióban.

GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T

Beállítja, hogy mit kell a textúrának tennie, ha túlnyúlunk a szélein. Az „S” végű az x tengelyt állítja be, a „T” végű az y-t. A következő értékeket vehetik fel:

- `GL_REPEAT` – Ismétlődjön a textúra
- `GL_CLAMP_TO_EDGE` – Csak az utolsó pixel ismétlődjön/ne ismétlődjön a textúra
- `GL_MIRRORED_REPEAT` – A textúra legyen tükrözve a szélén túl



Mipmapok

Ha a textúráink a távolban jelennek meg, vagy akár egy talaj részeként elnyúlnak a távolba, nagyon zajosnak fognak tűnni (a pontos megfogalmazás erre, hogy egy Moiré minta jön létre). Ennek oka, hogy a kirajzolt pixelek elmozdulnak a textúra pixelein és megoldása úgy, hogy ne használjunk rá nagy mennyiségű erőforrást, nehézkes. Erre hozták létre a mipmapokat.

Mipmapokkal lényegében minden textúrának több verzióját tároljuk el, az eredeti méretét és egy előre meghatározott mennyiségű csökkentett méretűt. Ezeket egy lassabb algoritmussal is kiszámolhatjuk, mivel erre csak egyszer van szükség, majd ezeket felhasználva sokkal kevésbé zajos képet tudunk alkotni.

A mipmapok engedélyezésére két textúra paramétert kell beállítani, a textúra `MIN_FILTER`-ét és a `MAX_LEVEL`-t.

A `GL_TEXTURE_MIN_FILTER`-nek a következő értékei lehetnek, amivel engedélyezzük a mipmapok használatát:

- `GL_LINEAR_MIPMAP_LINEAR` – A két legközelebbi mipmap szint súlyozott átlagát veszi és a kapott kombinált szinten belül is súlyozott átlaggal dolgozik (mint a `GL_LINEAR`). Ez felel meg a „trilinear filtering” opciónak sok játékban.
- `GL_NEAREST_MIPMAP_LINEAR` – A két legközelebbi mipmap szint súlyozott átlagát veszi, de a kapott kombinált szinten belül a legközelebbi pixelt veszi
- `GL_LINEAR_MIPMAP_NEAREST` – A kirajzolt mérethez legközelebbi textúrát használja, azon belül súlyozott átlaggal határozza meg a tényleges színt
- `GL_NEAREST_MIPMAP_NEAREST` – A szintekből és azon belül a pixelekből is a legközelebbit választja.

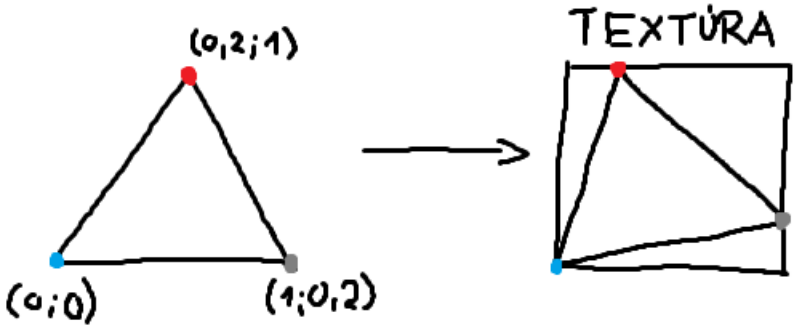
Amennyiben nem egy stílusbeli megkötés dönti ezt el, az elsőt érdemes használni.

Miután engedélyezve lettek, létre kell hoznunk őket. Ez megtehető manuálisan is, ha felhasználjuk a `glTexImage2D` „level” paraméterét. Ha viszont nem szeretnénk ezeket megadni, felhasználható a `glGenerateMipMaps` függvény is. Fontos, hogy ezt csak azután szabad meghívni, hogy elláttuk a 0. szintet a textúra adataival.

Textúrákhoz szükséges vertex adatok

Ahhoz, hogy textúrákat felhasználhassunk egy modellen, kell biztosítanunk egy UV koordinátát minden vertexhez. Ez meghatározza, hogy egy adott vertex hova illeszkedik a textúránkon.

Az UV koordináták (2 dimenziós textúra esetében) origója a textúra bal alsó sarkában található, a jobb felső sarok az (1, 1) pont.



Meghatározásuk a többi VBO-val megegyezik, így ebbe itt nem megyek bele, ezzel kapcsolatban a „VBO” szekció segít.

Felhasználásuk shaderekben

A shaderek nem képesek közvetlen egy textúra értékeit tömbként kezelni, hanem egy `sampler` és a `texture` beépített függvény segítségével lehet kinyerni belőlük egy 4 komponensű vektort.

A sampler egy uniform változónak a típusa. Feltöltésnél egy integert kell megadni, ami megfelel a felhasználandó textúrát tartalmazó textúra slot-nak.

Mivel egy modellhez sokszor (sőt, a való világban általában) több textúrára is szükség van, így nem vagyunk limitálva a textúra típusára. Létezik legalább 80 (a pontos érték implementáció specifikus) textúra slot, amikbe belerakhatjuk őket. Ezeket a `GL_TEXTURE0`, `GL_TEXTURE1`, stb. konstansokkal és a `glActiveTexture` fügvénnyel tudjuk aktiválni. Azt ezt követő `glBindTexture` hívás feltölti a slot-ot:

```
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, texture); // A textúra a 2-es slotba kerül
```

A felhasznált slot indexét kell a `glUniform1i` függvény segítségével feltölteni a shaderben egy `sampler2D` típusú uniform változóba:

Shader

```
uniform sampler2D sampler;
```

Processzor

```
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, texture);
// A textúra a 2-es slotba kerül
int samplerLocation =
glGetUniformLocation(program, "sampler");
// Kirajzolás előtt
// A 2 megfelel az aktív textúra slotnak
glUniform1i(samplerLocation, 2);
```

Ezután a shaderben meghívhatjuk vele és a kapott UV koordinátával a `texture` függvényt:

```
in vec2 vertexUV; // vertex shaderben át kell adni

uniform sampler2D sampler;

out vec4 fragmentColor;

void main() {
    fragmentColor = texture(sampler, vertexUV);
}
```

Hibakezelés

Régi mód

Amennyiben egy függvény hibára fut, beállítja a hibakódot ennek megfelelően. Ezt a `glGetError` fügvénnyel kérhetjük le. Amennyiben több hiba is volt, többszöri hívások során mindegyik megkapható. Ha nincs hiba, `GL_NO_ERROR` értéket ad vissza.

A kapott kódot érdemes átalakítani szöveggé a `gluErrorString` fügvénnyel. Összességében valami hasonlót kaphatunk, amit a rajzoló függvény végén érdemes elhelyezni:

```
unsigned int error;
while ((error = glGetError()) != GL_NO_ERROR) {
    printf("%s\n", (char *)gluErrorString(error));
}
```

Sajnos ez csak egy általános hibaüzenetet ad vissza (általában „invalid value”), azt nem adja meg, hogy hol történt a hiba, így ezt nekünk kell levadászni.

Új (egyszerű) mód

Amennyiben a kártyánk biztosítja az OpenGL 4.3-as verzióját, használhatjuk a beépített debug kimenet opciót. Ehhez engedélyeznünk kell ezt a programunk elején a következő fügvénnyel:

```
glEnable(GL_DEBUG_OUTPUT);
```

Ezután létre kell hoznunk egy függvényt, ami lekezeli a hibákat. Ennek előre meg van határozva a paraméter listája, a következő módon kell definiálni (a neve természetesen változtatható):

```
void GLAPIENTRY messageCallback(GLenum source,
                                GLenum type,
                                GLuint id,
                                GLenum severity,
                                GLsizei length,
                                const GLchar *message,
                                const void *userParam)
```

Ezekből a `type` és a `message` paraméterek értéke fontos. Ezek tartalmazzák az üzenet típusát (info, figyelmeztetés, hiba, stb.) és üzenetét. Hiba esetén az előbbo értéke `GL_DEBUG_TYPE_ERROR`, az utóbbit érdemes kiírni a konzolra.

Miután létrehoztuk a kiíró függvényünket, a glDebugMessageCallback függvénnyel beállíthatjuk azt. Ennek a második paramétere egy általunk meghatározott átadandó adatcsomag, ezt figyelmen kívül hagyjatok az esetek nagy részében (hasznos, ha mondjuk szeretnénk

dinamikusan állítani a kimenetek alsó szintjét, mondjuk egy debug módban bekapcsolni az info típusúakat is).

```
glDebugMessageCallback(messageCallback, nullptr);
```

Ezután hasznos üzeneteket fogunk kapni a kimeneten pontos leírással.

Egyéb fontos dolgok

Lépések a 3D-s rajzolás előtt

Amennyiben 3d-ben akarunk rajzolni, az egyetlen fontos dolog a depth buffer bekapcsolása és tisztítása. A bekapcsolását a program elején végezzük:

```
glEnable(GL_DEPTH_TEST);
```

A tisztítását pedig a képernyő tisztításával együtt a `GL_DEPTH_BUFFER_BIT` segítségével. A `BIT` jelentése ebben és a `GL_COLOR_BUFFER_BIT` értékekben arra utal, hogy ezek a konstansok csupán 1 bitet vesznek fel, így a bitenkénti OR (|) operátorral mindkettő egyszerre elvégezhető:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Feltakarítás

Érdemes a programunk végén letakarítanunk minden felhasznált adatot a grafikus kártyáról. Ezekhez a glDelete kezdetű függvényeket kell felhasználni, amik a glGen kezdetű függvényekhez hasonlóan működnek. Példa VBO-kra:

```
glDeleteBuffers(1, vbos);
```

Ha létrehozunk osztályokat, amik mondjuk egyes modelljeinket kezelik, akkor ezt a destruktorukban érdemes elvégezni.

Shaderek létrehozása stringekben

A grafika házihoz egy fájlt szabad csak beadni és tilos minden fájl IO, így a shadereket stringekben kell definiálni. Ehhez nagy segítséget nyújtanak a C++11 nyers stringjei. Ezekkel a következő módon nézne ki egy shader:

```
const char *const vertexShader = R"(
    // A shader kódja
)";
```

Ezzel elkerülhetjük, hogy minden sor végére „\n” karaktert kelljen rakni manuálisan.

Példakód

Az alábbi kód egy háromszöget hoz létre egy kék-fehér sakktábla textúrával. Az „`onInitialization`” függvény a framework része, az első dolog, ami lefut.

```
#include "framework.h"

// A vertex shader forráskódja
const char *const vertexShader = R"(
    #version 330 core
    precision highp float;

    layout(location = 0) in vec2 pos;
    layout(location = 1) in vec2 uv;

    out vec2 vertexUV;

    void main() {
        gl_Position = vec4(pos, 0, 1);
        vertexUV = uv;
    }
)";

// A fragment shader forráskódja
const char *const fragmentShader = R"(
    #version 330 core
    precision highp float;

    in vec2 vertexUV;

    uniform sampler2D sampler;

    out vec4 fragmentColor;

    void main() {
        fragmentColor = texture(sampler, vertexUV);
    }
)";

unsigned int createShader(GLenum type, const char *const source) {
    unsigned int shader = glCreateShader(type);
    glShaderSource(shader, 1, &source, nullptr);
    glCompileShader(shader);

    int status;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
    if (!status) {
        int length;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length);
        char log[length + 1];
        glGetShaderInfoLog(shader, length, nullptr, log);
        printf("%s\n", log);
    }
    return shader;
```

```
}

void onInitialization() {
    ////////////////////
    //// VAO és VBO-k ////
    ////////////////////

    // A vertexek 2d koordinátái
    float positions[] = {
            0, 0.5f,
            -0.5f, -0.5f,
            0.5f, -0.5f
    };
    // A vertexek UV koordinátái
    float uvs[] = {
            0.5f, 0,
            0, 1,
            1, 1
    };

    // A vertex array legenerálása a háromszöghöz
    unsigned int vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // A VBO-k legenerálása
    unsigned int vbos[2];
    glGenBuffers(2, vbos);

    // A vertex koordináták feltöltése a bufferbe
    glBindBuffer(GL_ARRAY_BUFFER, vbos[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions,
                 GL_STATIC_DRAW);
    // A kötési infó betöltése a VAO-ba
    // A "0" megfelel a pos változó layoutjában meghatározott helynek
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, nullptr);

    // Az UV koordináták betöltése a bufferbe
    glBindBuffer(GL_ARRAY_BUFFER, vbos[1]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(uvs), uvs, GL_STATIC_DRAW);
    // A kötési infó betöltése a VAO-ba
    // Az "1" megfelel a pos változó layoutjában meghatározott helynek
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, nullptr);

    //////////////////
    //// SHADEREK ////
    //////////////////

    // Shaderek feltöltése
    unsigned int program = glCreateProgram();
    // A shaderek létrehozásához csináltam egy segédfüggvényt, ez feljebb van
    unsigned int vshader = createShader(GL_VERTEX_SHADER, vertexShader);
    unsigned int fshader = createShader(GL_FRAGMENT_SHADER, fragmentShader);
    glAttachShader(program, vshader);
    glAttachShader(program, fshader);
```

```
// A kimeneti színt állítsuk be a fragmentColor-ra
glBindFragDataLocation(program, 0, "fragmentColor");

glLinkProgram(program);
glUseProgram(program);
int samplerLocation = glGetUniformLocation(program, "sampler");

// A shaderekre ezután nincs szükség
glDeleteShader(vshader);
glDeleteShader(fshader);

//////////////////
//// TEXTÚRA ////
//////////////////

// Ez az egész lecserélhető a framework segítségével a Texture objektum
// létrehozására

// Procedurálisan fogom generálni a textúra adatát, egy kék-fehér
// sakktáblát szeretnék elérni
// 100 * 100 pixel, minden négyzet 10 pixel széles
// 100 * 100 * 4 komponens
unsigned char textureData[100 * 100 * 4];
for (int x = 0; x < 100; x++) {
    for (int y = 0; y < 100; y++) {
        // Egy index az első komponensére annak a pixelnek,
        // ami az (x, y) koordinátákban van
        int index = (x * 100 + y) * 4;

        // Matematikai módja, hogy eldöntsük, hogy milyen négyzetben
        // vagyunk éppen. Ha összeadjuk a négyzetek x és y koordinátáját,
        // akkor a sötét négyzetek és világos négyzetek paritása el fog
        // térni
        if (((x / 10) + (y / 10)) % 2) {
            // Kék szín
            textureData[index + 0] = 0;     // R
            textureData[index + 1] = 0;     // G
            textureData[index + 2] = 255;   // B
            textureData[index + 3] = 255;   // A
        } else {
            // Fehér szín
            textureData[index + 0] = 255;   // R
            textureData[index + 1] = 255;   // G
            textureData[index + 2] = 255;   // B
            textureData[index + 3] = 255;   // A
        }

    }
}

unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 100,
             100, 0, GL_RGBA, GL_UNSIGNED_BYTE, textureData);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    ////////////////////
    //// KIRAJZOLÁS ////
    ////////////////////

    glBindVertexArray(vao);
    glUseProgram(program);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
    // A "0" megfelel a textúra slot-nak, amiben a textúránk benne van
    glUniform1i(samplerLocation, 0);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    // Kötelező, hogy megjelenjen valami a képernyőn
    glutSwapBuffers();
    unsigned int error;
    while ((error = glGetError()) != GL_NO_ERROR) {
        printf("%s\n", gluErrorString(error));
    }
}
```