SZAKDOLGOZAT

Fekete Ábel Zsombor

Debrecen

2010

Debreceni Egyetem

Informatikai Kar

MÁSODRENDŰ FELÜLETEK ÁBRÁZOLÁSA

Témavezető:
Dr. Schwarcz Tibor
Egyetemi adjunktus

Készítette:
Fekete Ábel Zsombor
Programtervező Informatika

Debrecen

2010

1. Tartalomjegyzék

1. Tartalomjegyzék 3
2 Bevezetés 5
2.1 A témaválasztás oka 5
2.2 A témaválasztás célja 5
2.3 A szakdolgozat célja 6
2.4 Jelen dokumentum tartalma 6
3 Alkalmazott technológiák 7
3.1 OpenGL 7
3.1.1 Mi az OpenGL 7
3.1.2 Az OpenGL használata 8
3.1.3 Az OpenGL használata Windows környezetben 9
3.2 JOGL 10
3.2.1 A JOGL és az OpenGL viszonya 10
3.2.2 A JOGL működése 11
3.3 A Java Swing 13
3.3.1 Használt Swing eszközök 13
3.4 „Kidobott” technológiák 14
3.4.1 Windows API 15
3.4.2 Natív OpenGL 16
3.4.3 CSGL 16
4 Másodrendű felületek 17
4.1 Másodrendű felületek megadásának módja 17
4.2 Másodrendű felületek fajtái 19
4.1.1 Ellipszoid 19
4.1.2 Egypalástú Hiperboloid 20
4.1.3 Kétpalástú Hiperboloid 21
4.1.4 Elliptikus Paraboloid 22
4.1.5 Hiperbolikus Paraboloid 22
4.1.6 Kúp 23
4.1.7 További megjelenített másodrendű felületek 24

5 Felületek számítógépes megjelenítése 25
5.1 A felületek definiáló pontjainak számítása 25
5.2 Felület létrehozása pontokból 25
5.3 A Triangle Strip 27
5.4 Megvilágítás 28
5.4.1 A megvilágítás fajtái 29
5.4.1 A vertexnormák meghatározása 31
5.4.2 A megvilágítás megadása OpenGL-ben 33
6 A felhasználói felület 35
7. A program ismert hibái, hiányosságai, fejlesztési lehetőségek 36
8 Összefoglalás 37
9 Irodalomjegyzék 39

2 Bevezetés

2.1 A témaválasztás oka

Tanulmányaim során sok, érdekes témával találkoztam, ezek közül számtalanban el lehet mélyülni egy szakdolgozat szintjéig. Eszközöket kaptam a kezembe, melyekkel számolni, elemezni, programozni lehet, módszereket, amik megmondják, hogy hogyan kell hatékonyan elérni a kitűzött célt, legyen az bonyolult, vagy egyszerű. Ezzel szemben vannak olyan témák, amiket én személy szerint fontosnak és hasznosnak tartok, mégse találkoztam elégszer velük tanulmányaim során. Egy ilyen téma a grafika is. Egyetemi tanulmányaim során nem töltött be túl nagy szerepet a grafika, hiszen csak két olyan tantárgyam volt, amelyik grafikával foglalkozott. Eleinte nem tudtam felmérni a téma jelentőségét, bár azt mindig sejtettem, hogy nem hiába hatalmas üzletág a videokártyák gyártása és terjesztése, mégse kötött le úgy, ahogy egykor a Mesterséges Intelligencia, vagy az Adatszerkezetek és Algoritmusok tanulása. Ezen változtatott az elmúlt év, amikor felvettem a második grafikával foglalkozó tárgyamat. A félév során egyre jobban érdekelni kezdett a téma, és egyre jobban megtetszett a munka a témában. Ennek eredményeképp határoztam úgy, hogy érdemes lenne a félév keretein kívül is foglalkozni a grafikával, és így jutottam oda, hogy ilyen témában szeretném a szakdolgozatomat írni.

2.2 A témaválasztás célja

Az okoknál már említettem, hogy kevésnek tartom az egyetemi, főleg a kötelező egyetemi, grafikus órák számát. Célom az volt, hogy szakdolgozatom során olyan eszközt hozzak létre, amely segítheti az egyetemi grafikai oktatást, felhasználható órákon, szemléltetésre, a diákok segítésére, remélve, hogy ezzel több diák érez rá a számítógépes grafika szépségeire, és kevesebben rettennek el a bonyolult képletektől. Remélem, hogy a munkám valamilyen szinten segítheti a grafikai oktatást, és egy lehetséges irányt mutat a korszerű technológiák alkalmazására.

2.3 A szakdolgozat célja

A szakdolgozat célja egy, akár az oktatásban is használható program létrehozása, mely másodrendű felületeket jelenít meg, hatékony, korszerű eszközök felhasználásával. Célom volt a háromdimenziós objektumok megjelenítésének videokártya szintű támogatása, valamint felhasználóbarát kezelőfelület létrehozása a program vezérléséhez. Emellett mindezt korszerű, könnyen kezelhető programozási környezetben létrehozni, az újrafelhasználhatóság és az átláthatóság miatt. Hogy ezeket a célokat elérjem, a következő eszközöket használtam:

- OpenGL: Megfelelő eszköz a videokártya szintű megjelenítésre, robosztus, sok nagyvállalat is használja. „The Industry Standard for High Performance Graphics”.
- JOGL: OpenGL csomagolás Java alá, segítségével java alól elérhető az OGL teljes funkcionalitása, így hardvertámogatott grafika megjelenítése Java környezetben
- Java: Korszerű, átlátható programozási környezetet nyújt, segítségével könnyen létrehozható a felhasználói felület, és a hardvertámogatott grafika megjelenítéséhez szükséges elemek.

2.4 Jelen dokumentum tartalma

A dokumentum tartalmazza a munkám során szerzett tapasztalataimat, a megvalósítás módját, a problémákat, amikbe ütköztem. Bemutat néhány alternatív megoldást, amikkel foglalkoztam, és olyan próbálkozásaimat, melyek nem jártak sikerrel. Leírom az alkalmazott módszereket, eszközöket, melyek szükségesek voltak a program megvalósítása során.

3 Alkalmazott technológiák

3.1 OpenGL

Az OpenGL nagy teljesítményű hardvertámogatott grafikus megjelenítést és számolást tesz lehetővé. Eszközök széles skáláját tartalmazza, melyek garantálják robosztusságát és multifunkcionalitását, az egyszerű pontok, vonalak és sokszögek ábrázolásától a nézőpont és perspektíva kezelésen át egészen a megvilágítási, és a fejlett shader technológiák programozásáig.

3.1.1 Mi az OpenGL

Az OpenGL jelentése Open Graphics Library, nyílt grafikus könyvtár. Az OpenGL egy API (Application Programming Interface, Alkalmazásfejlesztési Felület) a háromdimenziós grafika megjelenítésére. Szűkebb értelemben az OpenGL-t használjuk, hogy „háromszögeket rajzoljunk a képernyőre". A mai világban ez azt jelenti, hogy az OpenGL-en keresztül mondjuk meg a videokártyának, hogy mit és hogyan rajzoljon a képernyőre. Egyéb, akár grafikához tartozó funkcionalitása nincs az API-nak. Nem tud fájlformátumokat kezelni, így nem nyit meg képeket, vagy modellező programok által használt modelldefiníciós fájlokat (obj, max, maja). Nem animálja a jeleneteket, és nem kezel felhasználói kommunikációt sem, valamint az ablakkezelés sem az API feladata. Ugyanakkor léteznek külső, OpenGL-re épülő eszközök, melyek a fentebb leírt szolgáltatások közül többet is nyújthatnak (ilyen például a GLUT)
Az OpenGL API nem módszerek implementációja, egy nyílt specifikáció, amit mindenki ingyenesen használhat grafikus tartalom létrehozására (Szemben bizonyos ISO sztenderd specifikációkkal, melyek nem érhetőek el ingyenesen). Egy felületet definiál grafikus tartalom létrehozására, és tartalmazza az elvárt viselkedést, ha a programozó a felület eszközeit használja.
A grafika megjelenítése mindenképpen a videokártya feladata, így az OpenGL-t támogató videokártyák vezérlőprogramja implementálja az OpenGL specifikációt. Mac OS esetén ez az Apple saját implementációját jelenti, Windows esetén pedig a nagy videokártya gyártók (ATI, nVidia, Intel) a saját implementációjukat használják a videokártya vezérlőprogramján keresztül. Továbbá létezik az OpenGL-nek nyílt

forráskódú implementációja is (Mesa3D). Ezen kívül a Windows újabb verziói egy Direct3D wrapper (csomagolás) formájában implementálják az OpenGL bizonyos verzióit, bár a teljesítmény ebben az esetben nem kielégítő, így mindenképp érdemes egy OpenGL-t implementáló videokártya vezérlő beszerzése.

3.1.2 Az OpenGL használata

Bár a videokártya driverek tartalmazzák az OpenGL implementációt, és ezt az OpenGL-t alkalmazó programok tudják használni, saját program írásához ismernünk kell a specifikációt. Ez a specifikáció ingyenes, és letölthető az OpenGL oldaláról header és library fájlok formájában, valamint a legtöbb fordító tartalmazza, egyes részei pedig az operáció rendszer részét képezhetik (például Windows esetében a gl.h és az opengl32.lib). Egy program, mely OpenGL által kínált eszközöket akar használni, ezeket a könyvtárakat kell használja, hogy hozzáférjen az OpenGL implementációkhoz. Windows operációs rendszeren a következő módon működik az OpenGL hívása: egy OpenGL alkalmazás fordítása során az alkalmazásba linkelődik az opengl32.dll. Amikor ezt az alkalmazást futtatjuk, az opengl32.dll betöltődik a memóriába, és a Windows registry-ben felkeresi a tényleges OpenGL implementációt. Amennyiben talál, azt betölti (például nVidia driver: nvoglv32.dll) Továbbá az OpenGL könyvtárak, amiket a Windows beépítetten tartalmaz 1995 óta változatlanok, és valószínűleg nem fognak változni a jövőben, így csak 1.1-es verziójú OpenGL specifikációt tartalmaznak, míg a Linux specifikáció 1.2-es verziót támogat. Így amennyiben magasabb verziószámú OpenGL eszközöket kívánunk használni, szükség lesz a kiterjesztett gl könyvtárakra (glext.h, valamint Windows alatt a wglext.h). Ezen kívül, mivel az operációs rendszer alapvetően csak a régi specifikációt tartalmazza, az új használatához futási időben kell lekérni az új verziók alá implementált eljárások címét.

```
include <GL/gl.h>
include <GL/glext.h>
include <GL/wglext.h>

extern PFNGLACTIVETEXTUREPROC glActiveTexture;
PFNGLACTIVETEXTUREPROC glActiveTexture;
.
.
glActiveTexture = (PFNGLACTIVETEXTUREPROC) glGetProcAddress("glActiveTexture");
```

A fentebb látható kódrészlet, szemlélteti az aktuális videokártya által támogatott OpenGL eljárás (jelen esetben a glActiveTexture) címének lekérését. Ez a módszer azonban körülményes nagyszámú eljárás használatánál, így léteznek könyvtárak, amelyek elvégzik az eljárásmutatók beszerzését helyettünk. Ilyen például a GLEW vagy a GLEE. Jelenleg az OpenGL 1.1 eszközök jó része elavultnak számít az OpenGL 3.0 óta, mivel ebben a verzióban számos alapvető változás történt a specifikációban a fejlettebb eszközök (shaderek, textúra és vertex tömbök) könnyebb használhatósága miatt. Ezért aki 3.0, vagy későbbi verzióban leírt eszközöket kíván használni, az nem támaszkodhat a régebbi verziókra, teljes egészében az új specifikáció szerint kell dolgozzon.

3.1.3 Az OpenGL használata Windows környezetben

Munkám során sokat tanultam a Windows ablak és eseménykezeléséről, ugyanis sokáig a programot csak alapvető Windows API segítségével, és OpenGL használatával szerettem volna elkészíteni. Bár a Windows API használata nem tartozik a szakdolgozathoz, és a témáról részletes leírást találhatunk a msdn.microsoft.com oldalon, mégis egy pár dolog erősen köthető az OpenGL környezet létrehozásához. Mindenek előtt szükség van egy Render Kontextusra (RC, Rendering Context), mely az OpenGL hozzákapcsolja az Eszköz Kontextushoz (DC, Device Context). Ez Eszköz Kontextus a megjelenítendő ablakot kapcsolja a Grafikus Eszköz Felülethez (GDI, Graphics Device Interface). A GDI mindig egy olyan fizikai eszközt ír le, amivel megjeleníteni lehet, legyen az egy nyomtató, vagy egy képernyő, így minden megjelenítő eszközt hasonlóan lehet kezelni, míg a GDI alatt a tényleges eszközök vezérlőprogramjai dolgoznak.

```
HGLRC hRC=NULL;
HDC hDC=NULL;
.
.
hDC=GetDC(hWnd); //Eszköz Kontextus lekérése
hRC=wglCreateContext(hDC); //Render Kontextus létrehozása
wglMakeCurrent(hDC,hRC); //Render Kontextus aktiválása
```

A hWnd változó egy „fogantyú” az ablakhoz (Handle to Window), melyet az adott ablak létrehozásakor kapunk meg. Ezek után használható az OpenGL számtalan

eszköze, melyekkel a Render Kontextus által kezelt felület tartalmát lehet befolyásolni. Például egy háromszög létrehozása a következő kóddal:

```
glBegin(GL_TRIANGLES);
glVertex3f( 0.0f, 1.0f, 0.0f);
glVertex3f(-1.0f,-1.0f, 0.0f);
glVertex3f( 1.0f,-1.0f, 0.0f);
glEnd();
```

Itt ejtenék szót a vertex fogalmáról. A vertex egy térbeli objektumot koordinátáival definiáló pontja. Ez az objektum lehet egy egyszerű pont, vonal végpontja, egy sokszög egyik sarka, így a vertexek segítségével definiálhatunk minden megjelenítendő objektumot. Azt, hogy az adott vertex épp milyen típusú objektumot definiál, a glBegin() függvény paramétere határozza meg.

3.2 JOGL

A JOGL egy projekt, melynek keretein belül elérhető a JSR321, egy OpenGL csomagolás (wrapper) Java alá. A JOGL specifikus könyvtárak ingyenesen beszerezhetők a https://jogl.dev.java.net/ oldalról. Az innen letöltött könyvtárak tartalmazzák a javax.media.opengl csomagot, melyben minden szükséges OpenGL eszköz megtalálható a Java alapú OpenGL programozásához.

3.2.1 A JOGL és az OpenGL viszonya

Mivel a Java nem támogatja a mutatókat, melyeket sok OpenGL funkció paraméterezéséhez kell használni, így a mutatókat tömbökkel helyettesíti. Tartalmaz minden funkciót, melyet az OpenGL biztosít, ugyanakkor átveszi a Render Kontextus kezelésének a helyét, melyet ezután Java komponensekkel, ablakokkal, framekkel, panelekkel lehet megvalósítani. Így a fentebb látható háromszöget megjelenítő kód JOGL alatt a következőre módosul:

```
GL gl = drawable.getGL();

gl.glBegin(GL.GL_TRIANGLES);
gl.glVertex3f( 0.0f, 1.0f, 0.0f);
gl.glVertex3f(-1.0f,-1.0f, 0.0f);
gl.glVertex3f( 1.0f,-1.0f, 0.0f);
```

```
gl.glEnd();
```

Ahol a drawable az aktuális Render Kontextusunk, melyet megkapunk automatikusan a rajzolás kezdetén. Fontos tulajdonsága még a JOGL-nek, hogy jól integrálható az alapvető Java komponensekbe, öröklődés útján definiálja a rajzolható felületeket már meglevő Java osztályokból, szemben például a LWJGL-el, melynél az ablakkezelés előre bedrótozott módon valósul meg, így az alkalmazásunkban csak azt rajzolhatunk, amit OpenGL-el tudunk, ezt azt jelenti, hogy nem használhatjuk a Java Swing konténereket és vezérlőket sem.

3.2.2 A JOGL működése

A JOGL JSR321 Java Native Interface (JNI) hívásokkal valósítja meg az OpenGL funkciók elérését, tehát csak olyan rendszeren fog működni, amely eleve támogatja az OpenGL-t. A megjelenítéshez szükséges egy rajzoló osztály, mely implementálja a JOGL által definiált GLEventListener interfészt. Az interfész a következő absztrakt metódusokat tartalmazza, melyeket a rajzoló osztálynak implementálni kell:

- init (GLDrawable drawable): Az OpenGL kontextus inicializálásakor hívódik meg, ebben elvégezhetünk olyan műveleteket, melyekkel előzetes beállításokat végezhetünk el az OpenGL viselkedésére vonatkozóan. Paramétereként megkapjuk az OpenGL kontextust. Egynél többször is meghívódhat, amennyiben a rajzolt felületet megszüntetjük, majd újrakreáljuk, úgyhogy lehetőleg mellékhatásmentesen kell implementálni.

- display (GLDrawable drawable): Ebben a metódusban végezhetünk el minden rajzolással kapcsolatos teendőt. Ez egy kép, jelenet kirajzolását jelenti, ez a metódus folyamatosan meg fog hívódni más JOGL komponenseken keresztül.

- reshape (GLDrawable drawable, int i, int x, int width, int height): Akkor hívódik meg, amikor az ablak mérete, vagy pozíciója változik. Az ezzel kapcsolatos változásokat (perspektíva, méret változás) itt kell lekezelni, melyre szintén léteznek OpenGL eszközök.

- displayChanged (GLDrawable drawable, boolean modeChanged, boolean deviceChanged): Akkor hívódik meg, ha az OpenGL kontextust megjelenítő eszköz, vagy az eszköz megjelenítési beállításai megváltoznak. A modeChanged paraméter igaz, ha a megjelenítési mód megváltozott (például a képernyő felbontása, színmélysége), a deviceChanged paraméter pedig igaz, ha a megjelenítő eszköz megváltozott (például több képernyős környezetben az egyik monitorról a másikra lett húzva az ablak). Ugyanakkor a GLEventListener dokumentációja szerint nem szükséges valódi implementációt megadni ennek a metódusnak, mert ezt a referencia OpenGL implementációk sem tartalmazzák.

Szükség van továbbá egy felületre, amelyre rajzolhatunk, ez lehet GLCanvas, vagy GLJPanel. Erre a komponensre kell meghívni az addGLEventListener metódust, paraméterként annak az osztálynak egy példányával, mely implementálja a GLEventListener interfészt. Így az rajzoló osztályunk példánya érzékelheti, hogy milyen GLEventListener által specifikált műveleteket kérnek tőle. Ezt a kérést egy Animator, vagy FPSAnimator osztály intézi, mely periodikusan előidézi minden hozzá kapcsolódó megjelenítési felületre (GLCanvas-ra vagy GLJPanelre) a display eseményt, ami azt eredményezi, hogy a hozzájuk kapcsolt rajzoló osztályok display() metódusa meghívódik. Az Animator és FPSAnimator példányai háttérben futó szálak, így hogy elkerüljük a processzoridő szükségtelen felhasználását, érdemes az FPSAnimator-t használni, amely csak egy másodpercben adott számú display eseményt generál. Természetesen a másodpercenként generált display események száma csak egy közelítés, és nem garantált, ugyanis a külön szálként futó FPSAnimator működése függ a szálkezeléstől is. Az Animator osztály se küld két display eseményt közvetlenül egymás után, szintén processzoridő kímélés miatt, amennyiben azt szeretnénk, hogy a lehető leggyakrabban jelenítsen meg a rajzoló osztályunk, az Animator példányára meg kell hívnunk a setRunAsFastAsPossible metódust true paraméterrel.

```
Renderer r = new Renderer();
GLJPanel glDisplay = new javax.media.opengl.GLJPanel();

glDisplay.addGLEventListener(r);
FPSAnimator animator = new FPSAnimator( glDisplay, 40);
```

3.3 A Java Swing

Kényelmes, felhasználóbarát felület létrehozása, és a könnyen kezelhetőség volt a cél, amikor a Java Swing-et választottam. Ma egy felhasználónak sem idegenek a Swing által használt vezérlők, melyeket főként az egér segítségével kezelhetünk, ellenben sokan idegenkednek attól, ha egy alkalmazást csak billentyűkkel, vagy billentyűkombinációkkal lehet vezérelni, akármennyire nyilvánvalóan a felhasználó tudatára hozzuk, hogy az adott funkciókat milyen billentyűkombinációkkal lehet elérni, legyen az bármilyen alkalmazás, az alkalmazás használatát tanuló felhasználó jobban szeret az egérrel kapcsolatba lépni a felhasználói felülettel. Ezek az eszközök a Java-ban alapvetően Look&Feel minta alapján működnek, ami azt jelenti, hogy a vizuális megjelenésükből egyértelműen következik a funkció.

3.3.1 Használt Swing eszközök

- JPanel: Alapvető tároló, a többi komponens ebben helyezkedik el.
- GLJpanel: Nem alapvető Swing eszköz, ugyanakkor a JPanel-ből öröklődéssel létrehozott tároló, mely lehetővé teszi, hogy tartalmát hardveres támogatással, OpenGL segítségével hozzuk létre. A NetBeans egyszerű, hatékony felületet biztosít ablakok tervezéséhez, a külső eszközöket is hasonlóan lehet elhelyezni, mint az alapvető Swing eszközöket, amennyiben alapvető Swing eszközökből öröklődnek. A GLJPanel megtalálható a jogl.jar állományban, amely a JOGL egy alapvető könyvtára.
- JLabel: Általános célú, nem módosítható szöveget jeleníthetünk meg egy tároló tetszőleges pontján.
- JCheckBox: Magas szintű vezérlő, általában egy logikai érték beállítására szolgál, mivel a vezérlőnek 2 állása van. Kényelmesen kezelhető Action beállításával:

```
axisCheckBox.setAction(actionMap.get("setShowAxis"));
.
.
```

```
@Action
public void setShowAxis() { r.setShowAxis(!r.isShowAxis()); }
```

- JButton: Működése egyértelmű, ha rákattintunk, történik valami. Action-el történő implementálása még magától értetődőbb.

- JComboBox: Helytakarékos megoldás több exkluzív, választható elem megjelenítésére. Alapállapotban csak az épp kiválasztott elem látszik, viszont a hozzá kapcsolt legördülő menüből kiválaszthatunk bármikor egy másikat. Magas szintű elem, kézenfekvő az ActionPerformed eseményét elkapni, amely akkor következik be, amikor megváltozik a kiválasztott elem (szerkeszthető JComboBox esetében, amikor a szerkesztés véget ér). Nem javallott alacsony szintű eseményeket elkapni a Look&Feel megtartása miatt, ugyanis a felhasználó azt várja, hogy akkor történjen valami, amikor egy új elemet kiválaszt.

- JSpinner: Két részből álló vezérlő. Az első rész a vezérlő által kezelt aktuális értéket jeleníti meg, mely lehet lista, dátum vagy egész, valós szám, míg a másik résszel „növelhetjük” vagy „csökkenthetjük” az értéket. Akkor érdemes használni, ha a kezelt értékek a felhasználó szempontjából egyértelműen következnek egymás után (például számok, hónapok). Ha az aktuális értéktől távoli értéket szeretnénk kiválasztani, az értek szerkeszthető.

Ezeken kívül számtalan Swing eszköz kínálkozik felhasználóbarát felület létrehozására, melyet NetBeans ablakszerkesztő segítségével könnyedén alkalmazhatunk.

3.4 „Kidobott” technológiák

A projekt során nem egyszer futottam zsákutcába, részben amiatt, mert a technológia alkalmazását túl költségesnek vélem idő szempontjából, részben megfelelő, up-to-date dokumentáció hiánya miatt. Ettől függetlenül ezekből a mellékvágányokból is tanultam, és összességében hasznosnak bizonyultak a projekt előmenetelében.

3.4.1 Windows API

Nagyon jól dokumentált, általános célú ablakok létrehozására használható Windows környezetben. Eseménykezelés, vezérlés, bármit meg lehet valósítani, amit a Windowsban egy program megvalósíthat. A Java alapú technológiákkal szemben előnye, hogy nincs szüksége Java Virtual Machine-ra a futáshoz. Hátránya, hogy nehezebben lehet felhasználóbarát felületet létrehozni, valamint a felület mellé az OpenGL hardvertámogatott megjelenítését létrehozni, valamint, hogy sokkal nehezebben kezelhető, mint egy kényelmes NetBeans felületen megtervezni az ablakot, aminek a létrehozását a Java valósítja meg. Ezzel szemben a Windows API-ban egy ablakot létrehozni a következő lépésekkel lehet:

- Windows Class létrehozása. A Windows class attribútumok egy gyűjteménye, mely mintául szolgál az ablak létrehozásához. Minden ablakhoz tartozik egy class, de ugyanaz a class több ablakhoz is tartozhat.

```
#include <windows.h>

const wchar_t CLASS_NAME[]  = L"Sample Window Class";

WNDCLASS wc = { };

wc.lpfnWndProc   = WindowProc;
wc.hInstance     = hInstance;
wc.lpszClassName = CLASS_NAME;
```

- Windows class regisztrálása. Kötelező lépés, csak ezután lehet ablakot létrehozni az adott class alapján.

```
RegisterClass(&wc);
```

- Ablak létrehozása. Az itt adjuk meg, hogy melyik class-hoz tartozzon az ablak, méghozzá nem a programbeli változóján keresztül, hanem a regisztrált nevével.

```
HWND hwnd = CreateWindowEx(
        0,                              // Optional window styles.
        CLASS_NAME,                     // Window class
        L"Learn to Program Windows",    // Window text
        WS_OVERLAPPEDWINDOW,            // Window style
```

```
        // Size and position
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

        NULL,       // Parent window
        NULL,       // Menu
        hInstance,  // Instance handle
        NULL        // Additional application data
        );

ShowWindow(hwnd, nCmdShow);
```

Ezen kívül az ablakba érkező üzenetek kezelését is meg kell oldani, ez a Windows Message Loop implementálását jelenti.

3.4.2 Natív OpenGL

Mivel Java technológiát használtam, így kénytelen voltam egy Java alapú OpenGL wrapper rendszert használni, ugyanakkor az OpenGL tanulmányozásával töltött idő nem volt felesleges, mivel a JOGL OpenGL wrapper működése néhány apróbb dologtól eltekintve ugyanaz, mint a natív változaté, ennek oka, hogy a JOGL JNI (Java Native Interface) segítségével valósítja meg az OpenGL funkciókat, az eleve implementált OpenGL funkciókra támaszkodva. Továbbá elkerülhető a bonyolult Render Kontextus kezelése a JOGL-el, leegyszerűsítve az hardvertámogatott grafikus ablak létrehozását.

3.4.3 CSGL

C# alapú OpenGL wrapper. Mikor úgy döntöttem, hogy nem Windows API segítségével szeretném a programomat megvalósítani, felmerült egy C# alapú OpenGL wrapper ötlete, a CSGL viszont nem bizonyult megfelelőnek, egyrészt a hiányos dokumentáció miatt, másrész mivel a projekt fejlesztése 2003-ban lezárult. Később, miután már a JOGL mellett döntöttem, találtam egy nagyobb támogatottságnak örvendő C# OpenGL wrappert, a Tao Framework-öt, mellyel lehet, hogy hasonló hatékonysággal meg tudtam volna valósítani a projektet, mint JOGL segítségével.

4 Másodrendű felületek

A másodrendű felületek természetes kiterjesztései a háromdimenziós térben a síkban már jól ismert alakzatoknak: ellipsziseknek, paraboláknak és hiperboláknak. Létrehozhatóak ezen alakzatok forgatásával a térben, egyenletük jól leírható matematikai módszerekkel. Másodrendű felületnek nevezünk minden olyan felületet, melynek bármely síkmetszete kúpszelet. Általános egyenlete:

$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Exz + 2Fyz + 2Gx + 2Hy + 2Iz + J = 0$

Ezek közül van sok elfajuló, és képzetes felület is.

4.1 Másodrendű felületek megadásának módja

A geometriai objektumokat meg lehet adni matematikailag pontos egyenlettel, mely akkor teljesül a vizsgált pont koordinátáit az egyenletbe helyettesítve, ha a vizsgált pontot az objektum tartalmazza. Ilyen a sokak által ismert kör egyenlete:

$(x - u)^2 + (y - v)^2 = r^2$.

A másodrendű felületeket is meg lehet adni hasonló egyenlettel. Például a térben (0, 0, 0) x, y, z koordinátájú középponttal rendelkező ellipszoid egyenlete:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$

Ahol az a, b és c a konkrét ellipszoidra jellemző paraméterek. Ugyanakkor ez a hozzáállás a számítógépes grafikában az ábrázoláshoz nem kézenfekvő. Lehetne ugyan olyan módon kirajzolni egy alakzatot a fenti egyenlet alapján, mely szerint:

- Végigmegyünk a tér minden (x, y, z) pontján (bizonyos tartományon és lépésközzel, hogy elkerüljük a tér kontinuum számosságú pontja feldolgozását)
- Megnézzük az éppen aktuális pontra, hogy, bizonyos tűréshatáron belül, teljesíti-e az egyenletet (a tűréshatár a fentebb meghatározott lépésköztől függ)
- Amennyiben teljesíti, az adott pont az alakzat része, így meg kell jeleníteni

Ez azonban roppantul erőforrás igényes, továbbá numerikus problémákat vethet fel, melyek a számítógépen ábrázolható valós számok korlátozott számából és pontosságából

adódnak, és az alakzat megvilágítását is problémásabbá teszi, mint a következő módszer. Bevett gyakorlat a számítógép grafikában, hogy az egyenlettel leírható alakzatokat nem az egyenletük alapján, hanem úgynevezett vektorparaméteres megadási móddal rajzolnak ki.

x = u + r * cos(t)

y = v + r * sin (t)

0<=t<=2*π

A fenti vektorparaméteres egyenlet (Parametric equation vagy Parametric form) ugyanannak a körnek a pontjait írja le, mint amit a fejezet elején egyenlettel adtam meg. Így az egyenletbe a paramétertartományon belüli t-ket behelyettesítve megkapom a kör x és y pontjait, minél több ponttal szeretném a körömet kirajzolni, annál több különböző értéket kell behelyettesítenem a tartományból.

A fentebb látható ellipszoid egyenletét át lehet alakítani egyszerűen vektorparaméteres egyenletté, olyan módon, hogy az egyenletből kifejezzük a z koordinátát, így a következő egyenlethez jutunk:

$$x = tx$$

$$y = ty$$

$$z = \sqrt{c^2 - \left(\frac{tx^2}{a^2} + \frac{ty^2}{b^2}\right) * c^2}$$

Ugyanakkor ez vizuálisan nem fog kielégítő eredményt adni, ugyanis a négyzetgyök csak pozitív eredményt ad, így hiányozni fog az ellipszoid egyik fele. Amennyiben a z koordinátákat negatív előjellel véve ki szeretnénk rajzolni a másik felét, a probléma továbbra se oldódik meg, mivel az ábrázolás nulla z koordináta körül hiányozni fog. Természetesen a kapott pontok adatainak megfelelő feldolgozásával megoldható egy megfelelőt ellipszoid kirajzolása. További problémát jelent a paramétertartományok meghatározása, melyek az a, b és c paraméterek függvényei kell legyenek. Egy minden tengelyen korlátos felület megjelenítésétől elvárható, hogy az egész alakzat látható legyen. Sokkal egyszerűbb megoldás, ha gömbkoordinátás megjelenítést használunk:

$$x = a * \sin(\phi) * \cos(\theta)$$

$$y = b * \sin(\phi) * \cos(\theta)$$

$$b = c * \cos(\phi)$$

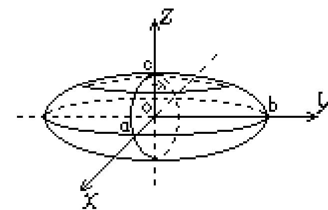
0 <= θ <= 2*π,   0 <= ϕ <=π

A felületeket megjelenítő programban, néhány esettől eltekintve, ilyen megadási módot használtam.

4.2 Másodrendű felületek fajtái

Áttekintjük a másodrendű néhány fontosabb másodrendű felületet, tulajdonságait, síkmetszeteit, egyenletét, különös tekintettel az ábrázolás szempontjából lényeges, valós másodrendű felületekre, melyeket a szakdolgozatként készített programban is megjelenítettem.

4.1.1 Ellipszoid



Az ellipszoid képlete:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$

Ahol a, b, c > 0

Ami vektorparaméteres megadási móddal, gömbkoordinátákkal a következő:

$x = a * \sin(\phi) * \cos(\theta)$

$y = b * \sin(\phi) * \cos(\theta)$

$b = c * \cos(\phi)$

$0 <= \theta <= 2*\pi, \ 0 <= \phi <= \pi$

Aszerint, hogy a, b és c paraméter közül melyek egyenlők, vagy különbözők, megkülönböztetünk háromtengelyű ellipszoidot, forgási ellipszoidot és gömböt. Az XY, XZ és YZ koordinátasíkokkal párhuzamos metszetei egyaránt ellipszisek, melyet a következő módon láthatunk be: (a módszer alkalmazható a többi másodrendű felületre is)

Az ellipszoid metszete az XY síkkal párhuzamos síkokkal: $\begin{cases} z = \lambda \\ \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \end{cases}$

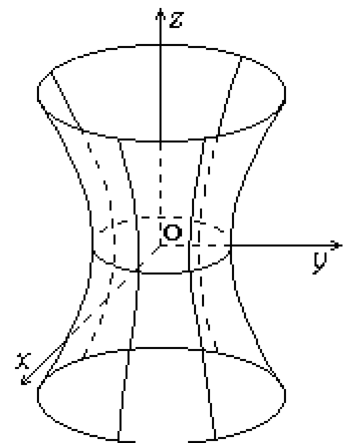
A metszet egyenlete: $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 - \frac{\lambda^2}{c^2}$

Az egyenlet jobb oldala az ellipszis valós pontjaira pozitív, mivel ha z=λ>c, akkor az

egyenletnek nincs megoldása (ezt mutatja az is, hogy a bal oldal biztosan pozitív).

Osszuk el az egyenletet a $\left(\sqrt{1-\frac{\lambda^2}{c^2}}\right)^2$ kifejezéssel.

Ekkor az egyenlet a következő alakú lesz: $\frac{x^2}{a'^2}+\frac{y^2}{b'^2}=1$, ahol a' = a· $\sqrt{1-\frac{\lambda^2}{c^2}}$ és b' = b· $\sqrt{1-\frac{\lambda^2}{c^2}}$. Továbbá $\frac{a'}{b'}=\frac{a}{b}$, azaz a metszet hasonló az ellipszoid egyenletéből z=0 esetén kapott $\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$ egyenlettel, mely pontosan az XY síkkal való metszés eredménye.



4.1.2 Egypalástú Hiperboloid

Az egypalástú hiperboloid képlete:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=1$$

Ahol a, b, c > 0

Vektorparaméteres megadási móddal, gömbkoordinátákkal a következő:

$x = a * cosh(\phi) * sin(\theta)$

$y = b * cosh(\phi) * cos(\theta)$

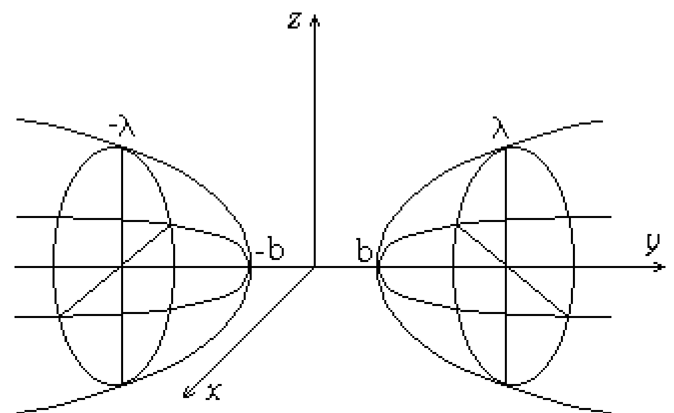
$z = c * sinh(\phi)$

$-\pi <= \theta <= \pi, \quad -\pi/2 <= \phi <= \pi/2$

Az egypalástú hiperboloid két irányban a végtelenbe nyúlik. Síkmetszetei: az XY síkkal párhuzamos metszetei ellipszisek, a=b esetben körök, ez esetben a hiperboloid egypalástú forgáshiperboloid, míg az XZ és az YZ síkokkal párhuzamos metszetei hiperbolák. Érdekes tulajdonsága, hogy az XZ és YZ síkokkal párhuzamos metszetei lehetnek a Z tengely felé nyitott parabolák, vagy az XZ síkkal párhuzamos esetben az X tengely felé nyitott parabolák, az YZ síkkal párhuzamos esetben az Y tengely felé nyitott parabolák. Vegyük az XZ síkkal párhuzamos esetet: amennyiben y < b és y > -b, a metszet az X tengely irányába nyitott parabola lesz. Ha y > b vagy y < -b, úgy a Z tengely irányába lesz nyitott a metszet. Elfajuló esetben, ha y = b vagy y = -b, a

hiperbola két egymást metsző egyenes lesz. A jelenséget jól szemlélteti a http://www.math.umn.edu/~rogness/quadrics/hyper1.shtml oldalon található interaktív ábra. A felületet nem metsző (Z) tengelyen keresztülmenő síkok a felületből hiperbolákat metszenek ki; ezek aszimptotái egy kúpfelületet alkotnak, az aszimptotikus kúpot. Egy általános helyzetű sík a felületet ellipszisben, hiperbolában vagy parabolában metszi aszerint, amint ezzel a kúppal ellipszis, hiperbola vagy parabola metszetet ad. A kúpmetszetekről lásd a 4.1.6-os fejezetben.

4.1.3 Kétpalástú Hiperboloid



A kétpalástú hiperboloid képlete:

$$-\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=1$$

Ahol a, b, c > 0

Vektorparaméteres megadási móddal, gömbkoordinátákkal a következő:

$x = a * sinh(\phi) * sin(\theta)$

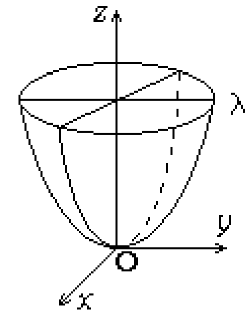
$y = b * sinh(\phi) * cos(\theta)$

$z = c * cosh(\phi)$

$-\pi <= \theta <= \pi, \quad -\pi/2 <= \phi <= \pi/2$

A kétpalástú hiperboloid két különálló részből áll, melyek két irányban a végtelenbe nyúlnak. Síkmetszetei: az XY síkkal párhuzamos metszetei ellipszisek, amennyiben z > c vagy z < -c, ellenkező esetben pontok (z = c vagy z = -c), vagy képzetes ellipszisek. Ha a=b, ezek a metszetei körök (vagy képzetes körök) ez esetben a hiperboloid kétpalástú forgáshiperboloid, míg az XZ és az YZ síkokkal párhuzamos metszetei hiperbolák, amelyek az egypalástú hiperbolaiddal szemben csak Z tengely irányába lehetnek nyitottak. A valós (Z) tengelyen keresztülmenő síkok hiperbolákban metszik felületet, melyek aszimptotái itt is a hiperboloid aszimptotikus kúpjának alkotói. Minden sík, mely a kétköpenyű hiperboloidot metszi, ezzel ugyanolyan nemű metszési görbét ad, mint az aszimptotikus kúppal. A kúpmetszetekről lásd a 4.1.6-os fejezetben.

4.1.4 Elliptikus Paraboloid



Az elliptikus paraboloid képlete:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = z$$

Ahol a, b > 0

Vektorparaméteres megadási móddal, gömbkoordinátákkal a következő:

$x = a * \sqrt{\phi} * sin(\theta)$

$y = b * \sqrt{\phi} * cos(\theta)$

$z = \phi$

$0 <= \theta <= 2*\pi, \ 0 <= \phi <= \pi$

Az elliptikus paraboloid egyik végén határtalan. Síkmetszetei: az XY síkkal párhuzamos metszetei ellipszisek (a = b esetben körök), míg az XZ és az YZ síkokkal párhuzamos metszetei ugyanolyan irányban nyitott parabolák. A Z tengelyt nem tartalmazó minden sík ellipszisben metszi.

4.1.5 Hiperbolikus Paraboloid

A hiperbolikus paraboloid képlete:

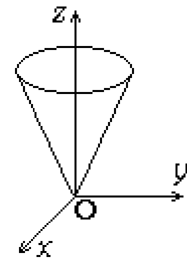
$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = z$$

Ahol a, b > 0

A hiperbolikus paraboloid megjelenítése során nem volt szükségem vektorparaméteres egyenletre. A hiperbolikus paraboloid minden irányban határtalan felület. Az XZ és az YZ tengelyekkel párhuzamos tengelyek különböző irányba nyitott parabolákat metszenek ki a hiperbolikus paraboloidból, igaz ez minden további olyan síkra, amely tartalmazza a Z tengelyt. A Z tengelyt nem tartalmazó síkok (köztük az XY sík, és ezzel párhuzamos síkok is) hiperbolákat metszenek ki a hiperbolikus paraboloidból, mely XY sík (azaz z = 0) esetében egy elfajuló hiperbola, azaz két egymást metsző egyenes. Ugyanez a helyzet minden más érintősíkkal is. Az XY tengellyel párhuzamos metszetei z > 0 esetben X tengely irányában nyitottak, míg z < 0 esetben Y tengely irányában nyitottak. Ez az

egyedüli másodrendű felület, amelyet forgásfelületként sosem lehet előállítani, ez következik abból is, hogy síkmetszetei között nincs zárt görbe.

4.1.6 Kúp



A kúp képlete:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=0$$

Ahol a, b, c > 0

Vektorparaméteres megadási móddal, gömbkoordinátákkal a következő:

$x = a * t * sin(\theta)$

$y = b * t * cos(\theta)$

$z = c * t$

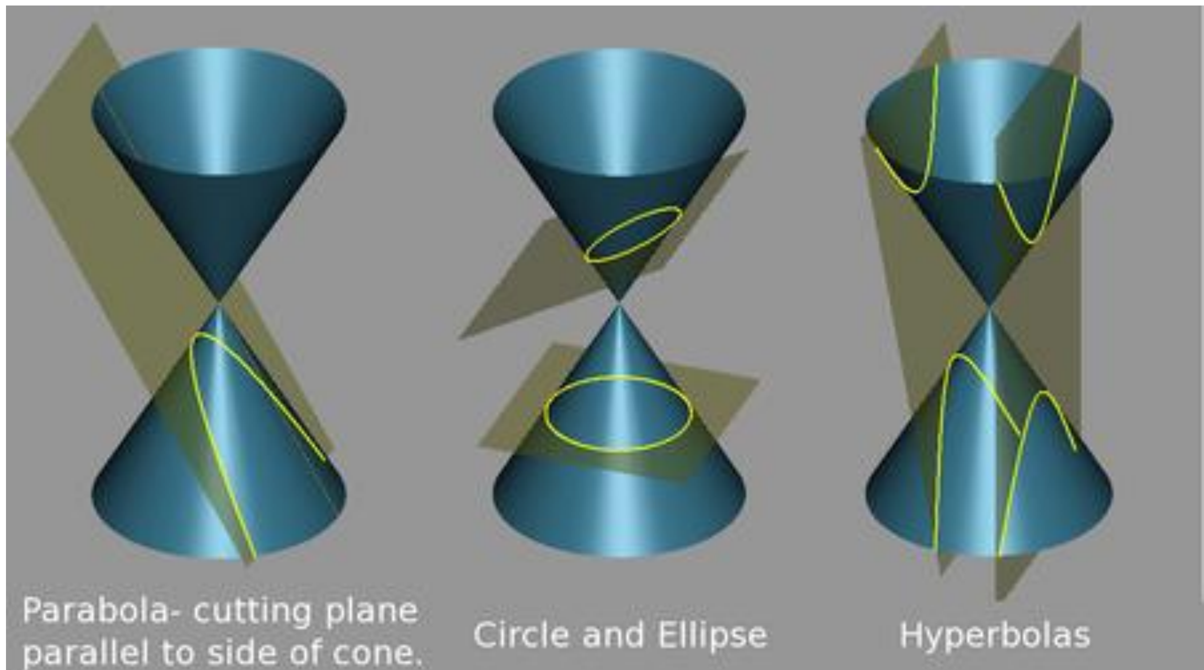
$-\pi <= \theta <= \pi, \quad -p <= t <= p, \quad p > 0$

Ahol a p tetszőlegesen választható, ugyanis csak azt határozza meg, hogy a z tengely mentén milyen hosszan jelenítsük meg a kúpot.

A kúp a Z tengely mentén határtalan felület. A kúp fontos felület, mert metszeteiből előállítható az ellipszis, a parabola és a hiperbola is, ez az úgynevezett kúpszeletek (conics). Az XY síkkal párhuzamos metszetei ellipszisek (a = b esetben körök, z=0 esetben egyetlen pont). XZ és YZ síkokkal párhuzamos metszetei hiperbolák, vagy, ha a metszet keresztülmegy a kúp csúcsán (x = 0, vagy y = 0), akkor két egymást metsző egyenes. A kúpot metsző tetszőleges sík kúpszeleteket alkot, amelyek a következőek lehetnek:

- Ellipszis (vagy kör), ha a metszet zárt görbe
- Parabola, ha a metsző sík párhuzamos a kúp alkotójára. Elfajuló esetben, ha a metsző sík keresztülhalad a kúp csúcsán, egy egyenest kapunk.
- Hiperbola, ha a metszet nyílt görbe, és nem párhuzamos a kúp alkotójával. Ebben az esetben a metszet keresztülhalad a kúp mindkét felén.

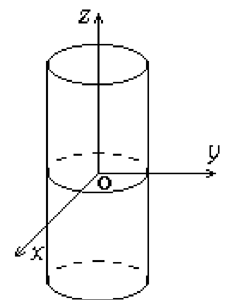
A kúpszeleteknek sosincs inflexiós pontjuk.



4.1.7 További megjelenített másodrendű felületek

Az előző fejezetekben felsoroltakon kívül megjelenítettem még a következő felületeket:

- Elliptikus Henger: $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$



- Hiperbolikus Henger: $\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$
- Parabolikus Henger: $y^2 = 2\text{p}x$

5 Felületek számítógépes megjelenítése

A szükséges eszközök birtokában a 3.2.2-es fejezetben leírtakhoz hasonlóan létrehoztam NetBeans fejlesztői környezetben egy felületet a Java által kezelt ablakon, mely képes hardvertámogatott grafika megjelenítésére. Ezen a felületen kell OpenGL specifikációt követve megjelenítenem a másodrendű felületeket a 4.1-es fejezetben leírtak alapján.

5.1 A felületek definiáló pontjainak számítása

A vektorparaméteres egyenletek alapján könnyedén létrehozhatjuk a felületek pontjait, hiszen csak a paramétereket kell végigfuttatni a tartományokon, és a vektorparaméteres egyenletekbe behelyettesíteni őket. Lássuk az ellipszoid esetében a display() metódus tartalmát egyszerűsítve, amennyiben csak a pontokat kívánjuk megjeleníteni.

```
gl.glBegin(GL.GL_POINTS);
for(double i=0; i<=2*Math.PI; i+=xStep)
{
  for(double j=0; j<=Math.PI; j+=zStep)
  {
    gl.glVertex3d(a*Math.sin(j)*Math.cos(i),
                  c*Math.cos(j),
                  b*Math.sin(j)*Math.sin(i));
  }
}
gl.glEnd();
```

Mint látható, az y és a z koordináta fel van cserélve, ahogy a programban mindenhol, ennek oka, hogy a megjelenítés hasonló legyen a megszokott háromdimenziós koordinátarendszerhez, ahol az x a vízszintes, z a függőleges koordináta, és y a mélységkoordináta.

5.2 Felület létrehozása pontokból

Megfelelő számú pont elég jól definiálhat egy felületet, feladatunk, hogy ezeket a pontokat úgy rendszerezzük, hogy, a grafikában megszokott módon, az általuk képzett háromszögek (jelen esetben használhatók négyszögek is) megadják a teljes felületet. Erre több lehetőség van, az egyik szerint létrehozzuk a felület pontjait, azokat tároljuk, majd a tárolt pontokat

feldolgozva létrehozzuk a felületet meghatározó háromszögeket. Kézenfekvő tárolási mód egy kétdimenziós tömb, mivel az i változó szerint a felület „oszlopait”, míg a j szerint a „sorait” hozzuk létre. Így a tárolt pontok kétdimenziós tömbjéből az (i,j),(i+1,j), (i+1, j+1) indexű valamint a (i,j),(i+1,j+1), (i, j+1) indexű ponthármasok a felület egy-egy háromszögét határozzák meg minden i, j-re. Természetesen az indextartományok helyességére vigyázni kell a tömb bejárásakor. Egy másik módszer szerint egyből a pontok generálásánál létrehozni a háromszögeket a paraméterek csúsztatásával.

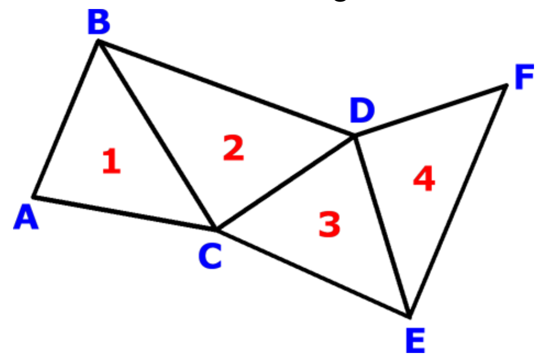
```
gl.glBegin(GL.GL_TRIANGLES);
for(double i=0; i<=2*Math.PI; i+=xStep)
{
  for(double j=0; j<=Math.PI; j+=zStep)
  {
    x=i;
    z=j;
    gl.glVertex3d(a*Math.sin(z)*Math.cos(x),
                  c*Math.cos(z),
                  b*Math.sin(z)*Math.sin(x));
    x=i;
    z=j+zStep;
    gl.glVertex3d(a*Math.sin(z)*Math.cos(x),
                  c*Math.cos(z),
                  b*Math.sin(z)*Math.sin(x));
    x=i+xStep;
    z=j+zStep;
    gl.glVertex3d(a*Math.sin(z)*Math.cos(x),
                  c*Math.cos(z),
                  b*Math.sin(z)*Math.sin(x));
    //…A másik háromszög hasonlóan létrehozható
  }
}
gl.glEnd();
```

Ez a módszer viszont ellenjavallott, ha a pontok kiszámítása erőforrás-igényes, mert így a paramétertartományok szélét kivéve minden pontot hatszor számolunk ki. Megfigyelhető, hogy a kód elején GL_TRIANGLES szerepel, azaz háromszögeket adunk meg a vertexekkel, így a vertexek egymás után hármasával csoportosítva egy háromszöget képeznek. További tudnivaló, hogy amennyiben szükségünk lesz valamilyen okból a háromszögek normáira (például a megvilágításhoz), figyelni kell, hogy konzisztensen, óra járásával megegyező (vagy azzal ellentétes) sorrendben határozzuk meg a háromszögek

pontjait. Létezik ugyanakkor egy ennél hatékonyabb módszer is, melyet az egymáshoz kapcsolódó háromszögek (vagy négyszögek) leírására találtak ki, ez a triangle strip.

5.3 A Triangle Strip

A triangle stip kifejezetten az egymáshoz kapcsolódó háromszögek leírására használt. Működése a következő: Az első háromszöget a triangle strip első három pontja adja. Minden további háromszöget további egy pont, valamint az előző háromszög utolsó két pontja határoz meg. Előnye, hogy a legtöbb videokártyán jól optimalizált a megjelenítése, a háromszögek körüljárási irányát automatikusan konzisztensen határozza meg, valamit kevesebb feldolgozást igényel. Triangle strip segítségével, ha a felületünk pontjai kétdimenziós tömbben vannak tárolva, minden sorra egy triangle stripet kell definiálni, miközben bejárjuk a sort, aminek a pontjai az adott sor adott pontja, valamint a következő sor adott pontja lesznek.



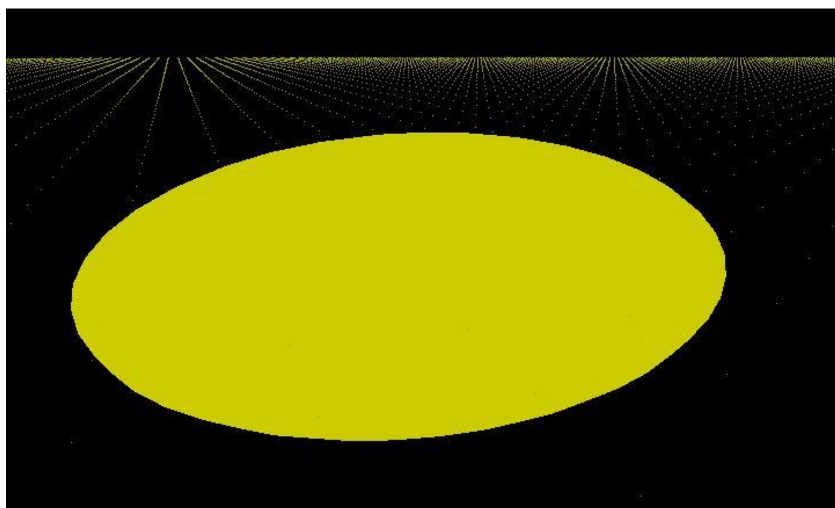
Tárolás nélküli, közvetlen feldolgozásnál ez a következőképp néz ki:

```
gl.glBegin(GL.GL_TRIANGLE_STRIP);
for(double i=0; i<=2*Math.PI; i+=xStep)
{
  for(double j=0; j<=Math.PI; j+=zStep)
  {
    x=i;
    z=j;
    gl.glVertex3d(a*Math.sin(z)*Math.cos(x),
                  c*Math.cos(z),
                  b*Math.sin(z)*Math.sin(x));
    x=i+xStep;
    z=j;
    gl.glVertex3d(a*Math.sin(z)*Math.cos(x),
                  c*Math.cos(z),
                  b*Math.sin(z)*Math.sin(x));
  }
  gl.glEnd();
  gl.glBegin(GL.GL_TRIANGLE_STRIP)
}
gl.glEnd();
```

A külső ciklus végén új triangle stripet kell kezdeni, különben a megjelenítése során minden „sor” vége össze lesz kötve a következő „sor” elejével. Ezzel a módszerrel a közvetlen feldolgozás hatékonysága is javítható, mert így az egyes pontokat (a paramétertartományok szélei kivételével) csak kétszer kell kiszámolni. Valamint a pontok tárolásának, és a közvetlen feldolgozásának a hatékony elemeit ötvözve, triangle strip segítségével könnyedén növelhetünk a hatékonyságon, mint tárhely, mind számítás szempontjából a következő módon: Első körben kiszámoljuk az első és a második sor pontjait, melyeket triangle strip-el egyből megjelenítünk. A második sor pontjait eltároljuk egy ideiglenes, egydimenziós tömbben. Ezek után a letárolt pontokkal, és az éppen következő sor pontjaival jelenítünk meg egy új triangle strip-et, majd a tárolt pontokat felülírjuk az éppen kiszámolt sor pontjaival. Ezzel a módszerrel hatékonyan létrehozható vektorparaméteres egyenletek segítségével bármely másodrendű (és más, vektorparaméteres egyenlettel rendelkező) felületet jól közelítő háromszögek sokasága.

5.4 Megvilágítás

Egy ténylegesen létező, térbeli objektumot azért látunk térben, mert két szemünk van, és a közöttük levő távolság miatt különböző képet érzékelnek szemeink. Ezt a különbséget az agyunk a tárgyak térbeli elhelyezkedésének meghatározására használja, minél nagyobb a különbség a két szemünk által érzékelt képben, annál közelebb van a megfigyelt tárgy. Ezen alapulnak a modern térhatású filmek, melyeknél polarizált lencsék segítségével különböző képeket juttatnak a néző két szemébe. Egy sík monitoron, ahol nincs lehetőség az objektum távolságától függően különböző képeket vetíteni a szembe, a térhatás érzetét a megvilágítás kelti. Íme, egy Ellipszoid megvilágítás nélkül:



Mivel ez vizuálisan nem kielégítő, így kénytelenek vagyunk megvilágítás alapú árnyalást használni.

5.4.1 A megvilágítás fajtái

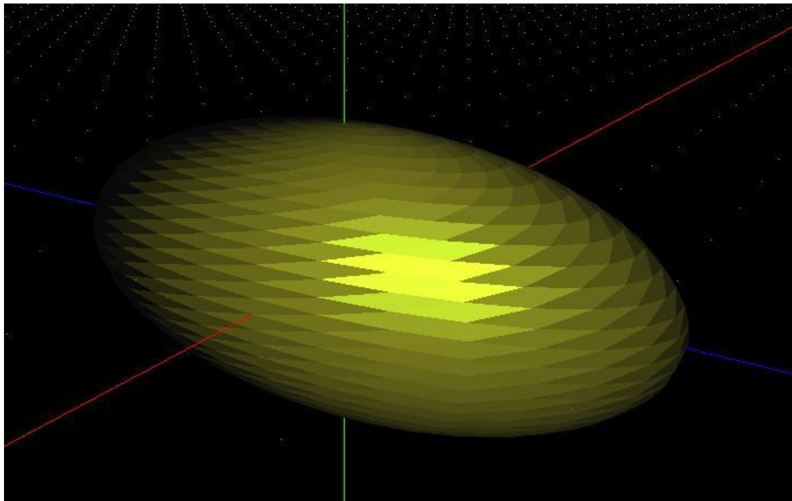
Több különböző megvilágítási ér árnyalási mód van, ezek közül csak néhányat írok le.

- Ambiens, vagy állandó fény. Ezzel a megvilágítási móddal minden objektum a saját fényével, minden irányba egyenletesen világít. Ilyen módon van megvilágítva a fentebb látható ellipszoid. Mint látható, térhatás létrehozására nem alkalmas, az objektum alapvető színét lehet vele meghatározni. Természetesen a valódi fény nem így működik, hiszen az a tárgy, ami a saját fényével világít, az egy fényforrás. A legjobban azt a helyzetet közelíti, amikor a fény egy tárgyra minden irányból azonos erősséggel és szögben vetődik.

- Diffúz, vagy szórt fény. Ennél a megvilágítási móddal az adott irányból érkező fény ütközik a felülettel, és onnan minden irányban egyenlően szétszóródik. A szétszórt fény mennyisége ugyanakkor függ a fénysugárnak a felülettel bezárt szögétől, ami azt jelenti, hogy a felületre merőlegesen érkező fénysugárból sokkal több visszaverődik a felületről, mint egy 45, vagy 15 fokos szögben beesőből, így a merőlegesen megvilágított felület világosabb lesz. Tipikus példája a matt felületeknek. Ennek segítségével már megfelelő térhatású objektumok létrehozhatók.

- Spekuláris, vagy visszavert fény. Ennek során a fényforrásból érkező fény a felületről egyenesen visszapattan, majd a megfigyelő szemébe jut. A felület adott pontjának fényessége a megfigyelő a felület pontja által meghatározott egyenes, valamint a felület adott pontjáról visszapattanó fény által meghatározott egyenes között bezárt szögtől függ. Tipikus esete a fényes felületeknek, melyeken megcsillan a fényforrás. A diffúz fénnyel együtt alkalmazva meglepően jó térhatás hozható létre.

- Konstans árnyalás. A diffúz és a spekuláris fény meghatározásához egyaránt szükség van a fényforrás és a felület által bezárt szögre. Konstans árnyalás



esetén ez a szög az egyes háromszögek minden pontjára ugyanannyi, melyet a fényforrás és a felület által meghatározott egyenes, valamint a felület normája által bezárt szög ad. Görbe felületek esetén az eredmény még nagyszámú, kisméretű háromszög mellett is töredezett hatású felület lesz, viszont síkokkal határolt felületek megjelenítésére alkalmas. Az OpenGL alapvetően nem felületnormákkal számol, hanem vertexnormákkal, ebben az esetben a felületnorma a felületet határoló vertexek normáinak átlaga lesz.

```
gl.glShadeModel(GL.GL_FLAT); //Konstans árnyalás
```

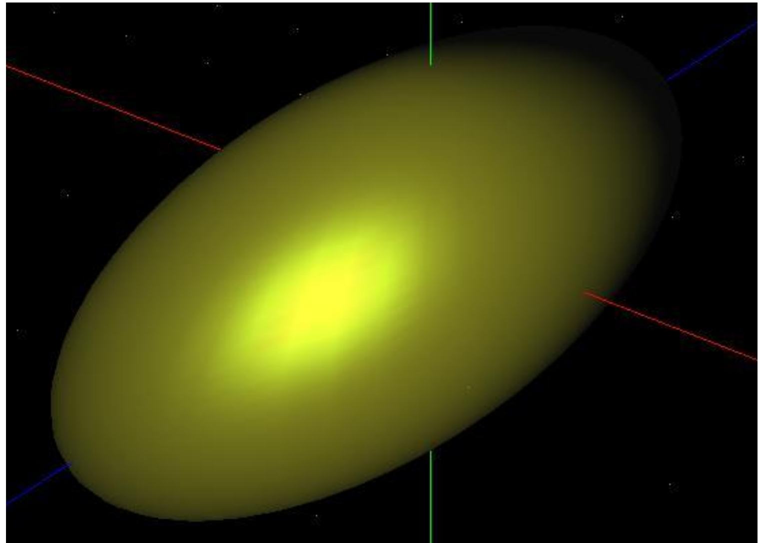
- Gouraud árnyalás. Ehhez az árnyaláshoz szükség van a vertexnormákra. Működése: a háromszögek mindhárom sarkában kiszámolja, hogy az adott fényviszonyok mellett milyen színű kell legyen az adott pont. Itt a fényforrás és a vertex által meghatározott egyenes, és a vertexnorma által bezárt szög számít a felület színének kiszámolásához. A kapott színeket ezután a háromszög felületén interpolálja. Diffúz fény megjelenítésére jó eredményt ad, viszont a spekuláris fény alacsony háromszögszám mellett nem az elvárt eredményt nyújtja, mivel a visszaverődés által okozott csillanás az egész háromszög felületén megjelenik. Az OpenGL általam használt verziója alapvetően ezt támogatja, míg a 3.0 utáni verziókon shader programozás útján bármilyen árnyalási modell megvalósítható.

```
gl.glShadeModel(GL.GL_SMOOTH); //Gouraud árnyalás
```

- Phong árnyalás. Alapvetően hasonlít a Gouraud árnyaláshoz, ugyanakkor nem a vertexekben megkapott színeket interpolálja, hanem magukat a vertexnormákat a felület minden pontján, melyekre azután minden pontban kiszámolja a

hozzájuk tartozó színt a fényviszonyok alapján. Jóval erőforrás-igényesebb, mint a Gouraud árnyalás, ugyanakkor jól kezeli a spekuláris fényt, még viszonylag alacsony háromszögszám mellett is.

Diffúz és spekuláris fénymodellek, valamint Gouraud árnyalás mellett így néz ki egy ellipszoid a programomban:



5.4.1 A vertexnormák meghatározása

Mind az általam használt Gouraud, mint a Phong árnyalási modellhez szükség van a háromszögeket meghatározó pontok normáira, azaz, a felületet az adott pontban érintő sík normájára. A kérdés tulajdonképpen a következő: Mi az adott pont iránya? Felületi normák egyszerűen számolhatók, egy felületi norma a felület síkjának normája, mely kiszámolható a síkon levő két vektor vektoriális szorzataként:

$c_1 = a_2b_3 - a_3b_2$

$c_2 = a_3b_1 - a_1b_3$

$c_3 = a_1b_2 - a_2b_1$

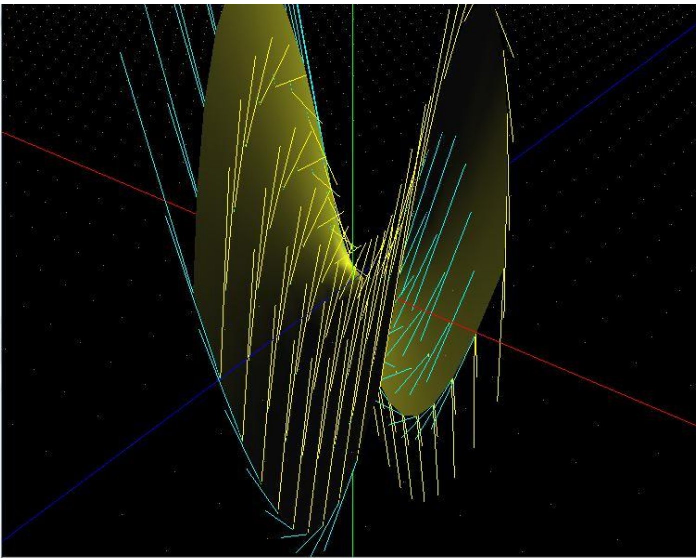
Ahol az alsó 1, 2, 3 indexek rendre a vektorok x, y, z koordinátáit jelentik.

Bármilyen felületre alkalmazható, általános módszer, amennyiben ismerjük az adott vertex-et tartalmazó minden felületet: Az adott vertex normája a vertexet tartalmazó felületek normáinak az átlaga. Ez elég jól közelíti a felületet a vertexben érintő sík normáját, főleg, ha egyszerű átlag helyett az átlagot súlyozzuk a felület kiterjedésével (területével). Jelen esetben viszont létezik jobb megoldás is. Mivel a másodrendű felületeket matematikailag leírható képletek alapján jelenítjük meg, ezekből a képletekből kiszámítható a felületet adott pontban érintő sík normája. Ilyen számításra különösen alkalmas a vektorparaméteres egyenletekkel megadott felület, ugyanis a vektorparaméteres egyenletet parciálisan deriválva az egyes paraméterekre,

megkaphatjuk a felületet az adott pontban, adott paraméter mentén érintő vektort. A két paraméterre külön deriválva így két érintő vektort kapunk minden pontban, melyek vektoriális szorzatából megkapható a felületet a pontban érintő sík normája, amely a vertex normája lesz.

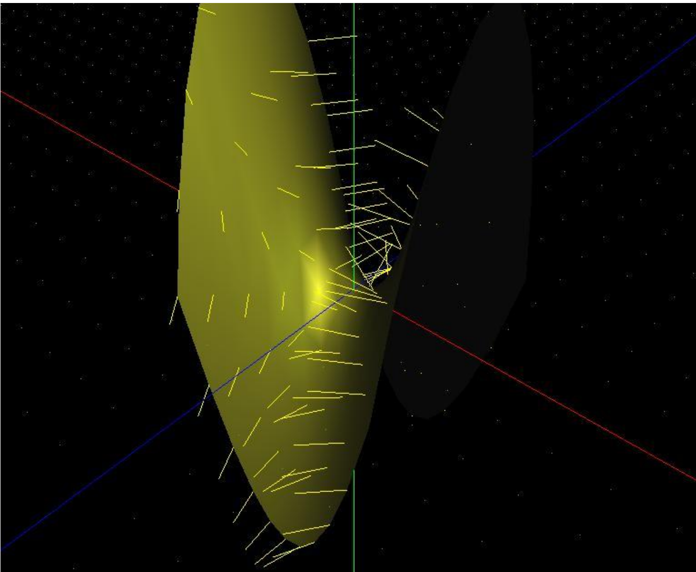
```
gl.glBegin(GL.GL_TRIANGLE_STRIP);
for(double i=0; i<=2*Math.PI; i+=xStep)
{
  for(double j=0; j<=Math.PI; j+=zStep)
  {
    x=i;
    z=j;
    //z szerinti derivált:
    Vector3d v1 = new Vector3d(a*Math.cos(z)*Math.cos(x),
                               -c*Math.sin(z),
                               b*Math.cos(z)*Math.sin(x));

    //x szerinti derivált
    Vector3d v2 = new Vector3d(-a*Math.cos(z)*Math.sin(x),
                               0,
                               b*Math.sin(z)*Math.cos(x));

    //vektoriális szorzat
    Vector3d vn = v1.vektorialisSzorzat(v2);
    gl.glNormal3d(vn.x,
                  vn.y,
                  vn.z);

    gl.glVertex3d(a*Math.sin(z)*Math.cos(x),
                  c*Math.cos(z),
                  b*Math.sin(z)*Math.sin(x));
  }
  gl.glEnd();
  gl.glBegin(GL.GL_TRIANGLE_STRIP)
}
gl.glEnd();
```

Ahol a gl.glNormal3d() az utána definiált vertex normáját adja meg. Emellett figyelni kell a normaként megadott vektorok hosszára: egységnyi hosszúak kell legyenek OpenGL szabvány szerint.



Felületi parciális deriváltak, valamint felületi normák az adott pontokban láthatóak a képeken.

5.4.2 A megvilágítás megadása OpenGL-ben



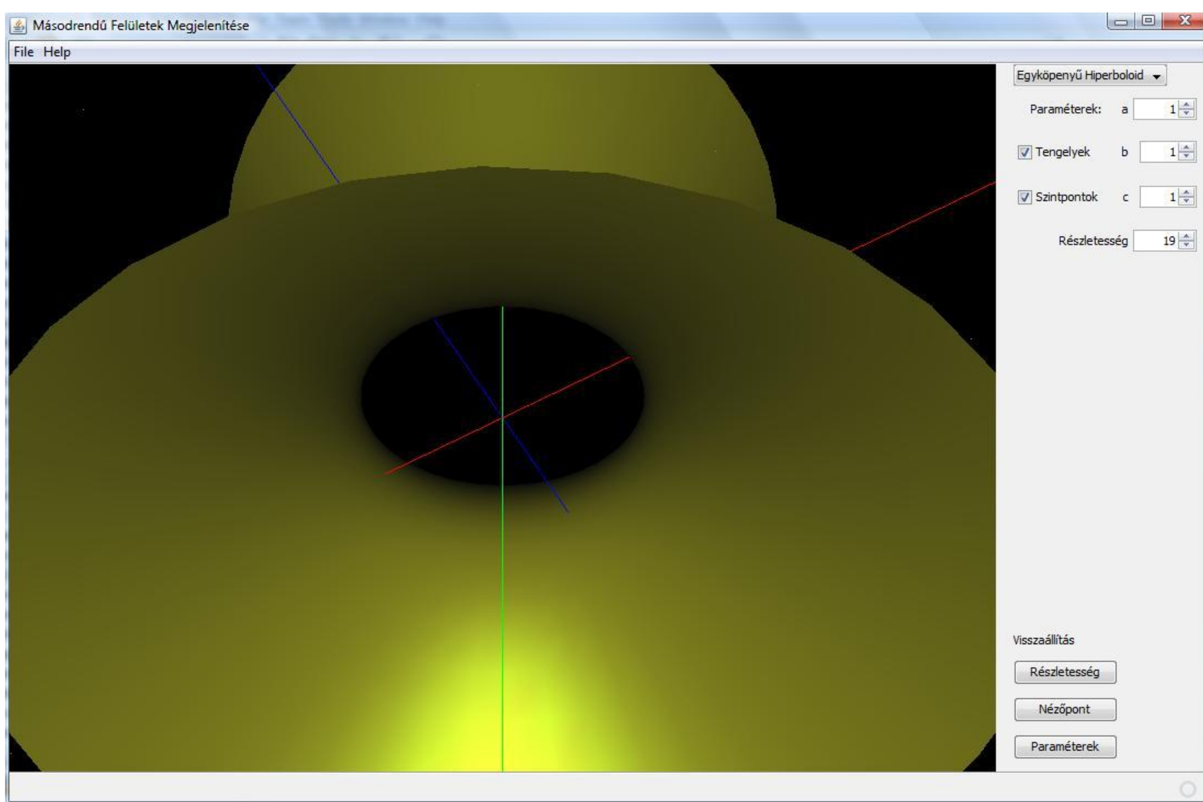
Az OpenGL legfeljebb 8 fényforrás definiálását teszi lehetővé, azaz egy felületet maximum 8 fény világíthat meg. Lehetőségünk van a fényforrás pozíciójának a beállítására, a szórt valamint a visszavert fény színének a meghatározására, ezen kívül a megvilágított felület tulajdonságait is befolyásolhatjuk, hogy viszonyuljon a rá eső fényhez. a fény áradhat egy forrásból minden irányba, vagy adott irányba.

```
gl.glEnable(GL.GL_LIGHTING); // Fények engedélyezése
gl.glEnable(GL.GL_LIGHT1);   // 1-es fényforrás engedélyezése

float LightDiffuse[]= { 0.6f, 0.6f, 0.2f, 1.0f }; //Fény színei
float LightSpecular[]= { 0.7f, 1.0f, 0.35f, 1.0f };
float LightPosition[]= { 0.0f, 0.0f, 0.0f, 1.0f }; // Fény pozíciója

float colorYellowish[]  = {0.8f,0.8f,0.4f,1.0f};

//1-es fény diffúz színének beállítása
gl.glLightfv(GL.GL_LIGHT1, GL.GL_DIFFUSE, LightDiffuse, 0);
//1-es fény spekuláris színének beállítása
gl.glLightfv(GL.GL_LIGHT1, GL.GL_SPECULAR, LightSpecular, 0);
//1-es fény poziciójának beállítása
gl.glLightfv(GL.GL_LIGHT1, GL.GL_POSITION,LightPosition, 0);

//Anyagok tulajdonsága
gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_DIFFUSE, colorYellowish, 0);
gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_SPECULAR, colorYellowish, 0);
gl.glMateriali(GL.GL_FRONT_AND_BACK, GL.GL_SHININESS, 30);
```

```
//Sökszögek hátuljának megvilágítása
gl.glLightModeli(GL.GL_LIGHT_MODEL_TWO_SIDE, GL.GL_TRUE);
```

6 A felhasználói felület

A felhasználói felületet teljes mértékben a NetBeans beépített felülettervezőjével hoztam létre. Kezelése magától értetődő, mindenki számára jól ismert, és használt komponenseket tartalmaz. Bal oldalt található a panel, ahol a felületek megjelennek. Egérvonszolással a nézőpont forgatható, görgővel ránagyíthatunk, vagy eltávolodhatunk a jelenettől. Jobb oldalt



találhatóak a vezérlők. Legfelül a legördülő menüből kiválasztható a megjelenítendő felület egérrel, vagy billentyűzettel egyaránt. Lentebb bal oldalt találhatóak a térbeli navigálást segítő objektumok be/kikapcsoló gombja, Checkbox formában. Ettől jobbra a paraméterek beállításáért felelős spinnerek, melyek értékét közvetlenül billentyűzetről tetszőleges pontossággal, vagy a fel, le gombok segítségével egytized pontossággal állíthatjuk. A paraméterek továbbá állíthatóak a Q, W, E, A, S, D gombok segítségével, amennyiben a fókusz az ablak megjelenítő részén van. A megjelenítő rész fókusza visszanyerhető, ha egérrel belekattintunk. Ezek alatt a megjelenített felület részletességét befolyásoló spinner, melynek értékei csak egész számok lehetnek. A vezérlő rész alján, bal oldalt találhatóak a nézőpontot, paramétereket valamint a részletességet alapállapotba visszaállító gombok.

7. A program ismert hibái, hiányosságai, fejlesztési lehetőségek

- Bizonyos felületeknél, feltehetőleg numerikus hiba miatt 0 x és z koordinátákon (OpenGL szerinti koordináták) a fények megjelenítése nem megfelelő
- Ugyanezen koordináták közelében az Ellipszoid vertexnormái eltérést mutatnak az elvárttól, mely apró torzulásként jelenik meg az ellipszoid felületét ezen a helyen megvilágító fényben. Lehetséges, hogy összefügg az előző problémával, ugyanakkor lehetséges, hogy elszámoltam valamit a felületi parciális deriváltakkal, vagy a vektoriális szorzattal.
- A felületek pontjaiból képzett háromszögek létrehozása nem az 5.3-as fejezetben tárgyalt optimálisabb megoldással valósul meg, így a triangle stripek pontjainak számítása közben a legtöbb pontot kétszer számolja ki.
- Szintén sok számítás megspórolható, ha csak akkor számoljuk újra a felületet, ha változott valamelyik paraméter, vagy a részletesség, máskülönben egy eltárolt verziót jelenítünk meg. Jelenleg minden display() hívásnál minden pont újraszámolódik
- Magasabb verziójú OpenGL támogatás megvalósítása
- Szélesebb körű vezérlési lehetőség: Fények színének, irányának, számának beállítása, megfelelő rendszer egyszerre több felület megjelenítésére paramétereinek állítására
- Információ a megjelenített felületről, oktatási céllal
- Teljes képernyő támogatás

8 Összefoglalás

A fentebb leírt technológiák segítségével sikeresen megvalósítottam egy olyan programot, mely felhasználóbarát felülettel rendelkezik, hardvertámogatottan képes grafikus tartalmat, jelen esetben másodrendű felületeket megjeleníteni, valamint jól általánosítható. Sikerült egy olyan módszert és eszközegyüttest találnom, mellyel bárki korszerű könnyen és hatékonyan hardvertámogatott részt tud elhelyezni egy ablakban. Ebben segítségemre voltak a következő eszközök:

- NetBeans fejlesztői környezet
- Java Swing felület
- OpenGL API
- JOGL Java OpenGL wrapper, mely JNI-vel éri el az OpenGL funkciókat

NetBeans segítségével felhasználóbarát Java Swing felületet hoztam létre, melybe könnyen integrálható a Swing komponensek örököltetésével definiált JOGL GLJPanel, mely a hardvertámogatott grafikus tartalom megjelenítéséért felelős. Ennek a tartalmát egy GLEventListener interfészt implementáló Osztály egy példánya frissíti, az Animator osztály példánya által meghatározott ütemben.
A következő másodrendű felületek megjelenítését valósítottam meg:

- Ellipszoid
- Egypalástú hiperboloid
- Kétpalástú hiperboloid
- Elliptikus paraboloid
- Hiperbolikus paraboloid
- Kúp
- Elliptikus henger

- Hiperbolikus henger
- Parabolikus henger

A megjelenítést legtöbb esetben a felületek gömbkoordinátás vektorparaméteres egyenletével valósítottam meg. A vektorparaméteres egyenletekből kapott pontokból triangle strip struktúrával határoztam meg a felületet közelítő háromszögeket
A felületek térhatása miatt diffúz és spekuláris fényt használtam, az OpenGL általam használt verziója által támogatott Gouraud árnyalás mellet.
Az árnyaláshoz szükséges vertexnormákat a paraméteres egyenletekből nyertem, parciálisan deriválva a paraméterek szerint, majd a nyert vektorokat vektoriálisan szorozva.
A felhasználói felületen a grafikus tartalom megjelenítésére szolgáló panel mellett helyet kaptak a könnyen kezelhető vezérlők, melyekkel hatékonyan és felhasználóbarát módon lehet a programot kezelni.
Programom nem tökéletes, de megvalósítja a célokat, amiért készítettem.

9 Irodalomjegyzék

Krekó Béla: Lineáris Algebra – Másodrendű felületekhez kapcsolódó részek.

http://www.opengl.org/ - OpenGL dokumentációk

http://nehe.gamedev.net/ - OpenGL példakódok

http://www.sjbaker.org/steve/omniv/opengl_lighting.html - OpenGL példakódok

http://www.falloutsoftware.com/tutorials/gl/ - OpenGL példakódok

http://glprogramming.com/red/ - OpenGL példakódok

http://jerome.jouvie.free.fr/index.php - OpenGL példakódok

http://java.sun.com/javase/6/docs/api/ - Java dokumentációk

http://msdn.microsoft.com/hu-hu/library/default(en-us).aspx –WinAPI és más dokumentációk

https://jogl.dev.java.net/ - JOGL dokumentációk és példakódok

http://www.cs.umd.edu/~meesh/kmconroy/JOGLTutorial/ - JOGL példakódok

http://www.javaworld.com/javaworld/jw-02-2005/jw-0221-jogl.html - JOGL példakódok

http://www.math.umn.edu/~rogness/quadrics/index.shtml - Másodrendű felületek

http://abrgeom.uw.hu/segedanyagok/feluletek.pdf - Másodrendű felületek

http://www.math.bme.hu/~simonk/a2/masodrendu_feluletek_b.pdf - Másodrendű felületek

http://www.cs.elte.hu/geometry/kissgy/geobsc1-9.pdf - Másodrendű felületek

http://math.ubbcluj.ro/~ildiko.mezei/feladatlapok/affin/szarmaztatott%20es%20masodrendu%20%20feluletek.doc – Másodrendű felületek

http://wmi.math.u-szeged.hu/wmi/ui/init/index.html - Deriválás

http://wikipedia.org/ - Számtalan cikk, többek között: Phong Shading, Triangle Strip, Kúpszelet, Aszimptota, Circle, Koordinátageometria, Vektoriális szorzat, Quadratic, Spherical coordinate system, Hyperboloid, Paraboloid, Ellipsoid,

Köszönöm témavezetőmnek, Dr. Schwarcz.Tibornak, a témáért, mely lekötött.

Köszönöm szüleimnek, türelmükért és támogatásukért.

Köszönöm évfolyamtársaimnak és barátaimnak, tudásukért és kitartásukért.

Köszönöm a Wikipedia-nak, mert megváltoztatta a világ tudásról alkotott képét.