Rodica Baciu

Programarea aplicaţiilor grafice 3D în OpenGL

-Îndrumar de laborator-

Editura Techno Media
Sibiu, 2010

PREFAŢĂ

Îndrumarul de laborator cuprinde lucrările de laborator care însoţesc cursul "Programarea aplicaţiilor grafice 3D" care face parte din planul de învăţământ pentru studiile de licenţă de la specializarea "Calculatoare şi Automatizări" din cadrul Facultăţii de Inginerie, Universitatea "Lucian Blaga" din Sibiu. La specializarea noastră am ales interfaţa grafică OpenGL, ca exemplificare şi ca aplicaţie a cursului " Programarea aplicaţiilor grafice 3D" în urmă cu 10 ani. Timpul a demonstrat că am ales bine, în marile universităţi din lume se studiază acum tot OpenGL în cadrul disciplinei "Computer Graphics". Alcătuirea îndrumarului este rodul şlefuirii, în decursul mai multor ani, al lucrărilor de laborator pentru această disciplină.

Pentru simplitatea aplicaţiilor dar şi pentru a nu împovăra aplicaţiile cu alt cod (pentru crearea ferestrelor, pentru interfaţa cu utilizatorul, etc.) decât cel de OpenGL s-a ales interfaţa GLAUX care de la versiunea 5.0 până la versiunea 2005 a însoţit mediul Visual C. Această interfaţă este uşor de utilizat şi nu necesită o instalare suplimentară instalării mediului Visual C. Experienţa a arătat că după ce studenţii învaţă să utilizeze OpenGL cu uşurinţă îl folosesc şi în alte medii de programare (Java, BorlandC, ş.a.) şi cu alte interfeţe (GLUT, GLUI).

Pentru fiecare laborator studenţii primesc un set de programe scrise în C şi care folosesc interfaţa API - OpenGL. După ce studenţii examinează aceste aplicaţii trebuie să le modifice conform cu cerinţele din îndrumar. Unele lucrări de laborator sunt dedicate unei singure teme, dar altele cuprind mai multe teme care nu sunt neapărat legate între ele. Îndrumarul cuprinde

temele esenţiale de OpenGL, fără pretenţia de a acoperi toate funcţiile pe care le oferă interfaţa. Cei care doresc să aprofundeze cunoştinţele de OpenGL au în acest îndrumar o bază de cunoştinţe bine selectată.

Conf.dr.ing. Rodica Baciu
Sibiu, 2010

CUPRINS

LABORATOR 1 - Introducere în aplicaţii grafice. Limbajul de modelare al realităţii virtuale VRML (Virtual Reality Modelling Language).

1.1 Suport teoretic

Limbajul VRML este un limbaj textual pentru descrierea scenelor 3D. Diferenţa dintre VRML şi un limbaj de programare (C++ spre exemplu) este că în loc să descrie cum acţionează calculatorul pentru o serie de comenzi specifice, el descrie aspectul unei scene 3D. Limbaje ca C++ sau Basic descriu în detalii operaţiile pe care trebuie să le facă microprocesorul pentru a realiza o sarcină. Limbaje precum VRML nu descriu **cum** se face "ceva" ci **ce** este "ceva". În VRML acest "ceva" este o scenă 3D.

Descrierea scenei se adresează unui software care este capabil să o interpreteze şi apoi să o afişeze pe ecran. Acest software poate fi o aplicaţie off-line dar poate fi şi un plug-in care se ataşează unui browser. Pentru realizarea şi vizualizarea scenelor VRML s-au conceput aplicaţii independente dar s-au şi extins aplicaţiile existente (cum este 3D Studio Max) cu posibilitatea de a exporta scenele 3D şi în format VRML. Spre deosebire de o scenă 3D realizată în alte formate de fişier, fişierele VRML (*.wrl) se comportă precum fişierele HTML pentru text – adică permit hyperlink-uri. Bazele limbajului au fost puse în anul 1994. De atunci a evoluat de la o versiune la alta. Mai mult, s-a realizat pentru VRML o interfaţă Java Script care permite realizarea de aplicaţii interactive. Fişierele *.wrl pot fi plasate oriunde pe Web, pot fi accesate din browser-e care

au ataşate extensii (plug-in) pentru VRML, şi prin hyperlink-uri se poate trece de la un fişier la altul.

Formatul fişierelor VRML este un format text care poate fi editat în orice editor ASCII. Structura fişierelor este ierarhică. În noduri se plasează construcţia geometrică a obiectelor şi starea acestora. Modificarea stării afectează felul în care sunt afişate construcţiile geometrice ulterioare. Spre exemplu un nod de translaţie afectează poziţia în care se redă o sferă descrisă printr-un nod ulterior. Un nod material afectează aspectul sferei, dacă sfera este descrisă într-un nod descendent nodului material.

1.2 Desfăşurarea lucrării

Pas 1. Se instalează un viewer de VRML. Pentru aceasta se utilizează fie fişierul **`VRML2C.exe`** fie **`cortvrml.exe`**.

Pas 2. Se vizualizează diverse scene VRML din biblioteca VIEWPNT. Se constată că la deschiderea unui fişier VRML din browser, în mod automat se ataşează browser-ului două bare de butoane aferente viewer-ului. Se experimentează aceste butoane pentru parcurgerea scenei. Să se încerce clarificarea următoarelor probleme:

Modelarea obiectelor este realizată off-line iar vizualizarea on-line. Unde se realizează modelarea obiectelor şi unde se realizează vizualizarea obiectelor?

În ce constă operaţia de modelare a obiectelor?

Ce operaţii presupune vizualizarea obiectelor?

Ce înţelegeţi prin poziţia camerei şi cum credeţi că se modelează?

Ce înţelegeţi prin iluminare şi cum credeţi că se modelează?

Ce înțelegeţi prin ascunderea suprafeţelor şi cum credeţi că se modelează?

Pas 3. Meniul scurt al mouse-ului permite realizarea unor setări pentru afişarea scenei. Folosind aceste setări:

- să se afişeze obiectele wireframe şi apoi din nou cu atribut de umplere (fill).
- să se afişeze obiectele cu umbrire poligonală constantă şi apoi cu umbrire Gouraud sau Phong.

Să se răspundă la următoarele întrebări:

Din ce primitive grafice sunt modelate obiectele?

De ce foarte multe dintre obiecte sunt modelate din triunghiuri?

Care este deosebirea dintre cele două tipuri de umbrire?

Pas 4. Să se afişeze fişierul **`cub.wrl`** în format wireframe şi în format solid. Folosind un editor de texte ASCII (Notepad-ul eventual) să se vizualizeze descrierea scenei.

Liniile care încep cu # sunt linii de comentariu cu excepţia primei linii care este un fel de semnătură a fişierelor VRML. Fără această linie fişierele nu sunt recunoscute de viewer.

```
#VRML V2.0 utf8
# Construieşte un cub prin fete indexate
```

Nodul rădăcină este nodul **`shape`**. Subnodurile sale sunt **`appearance`** şi **`geometry`**. În nodul **`appearance`** s-ar putea da coeficienţii de reflexie ai materialului pentru a defini culoarea obiectului. Deoarece nu sunt date valori, sunt considerate valorile implicite şi cubul va fi alb cu feţe mai mult sau mai puţin umbrite care apar în diferite nuanţe de gri. În nodul **`geometry`** este o descriere a cubului ca obiect modelat

poligonal. În subnodul **point** sunt date vârfurile cubului prin coordonatele x, y, z ale acestora. Deşi nu apare în mod explicit, fiecare vârf are asociat un index începând de la 0. În subnodul **coordIndex** sunt date feţele prin furnizarea indecşilor vârfurilor. Dacă ordinea specificării vârfurilor în tabloul **point** nu este semnificativă, ordinea vârfurilor în tabloul **coordIndex** este esenţială. În general, în grafică, ordinea specificării vârfurilor unui poligon arată dacă acesta este un poligon faţă sau un poligon spate. Poligoanele faţă sunt cele iluminate. Ordinea furnizării feţelor nu este semnificativă.

```
geometry IndexedFaceSet {
  coord Coordinate {
     point [
     # Coordonatele de deasupra cubului
        -1.0 1.0 1.0,
        1.0 1.0 1.0,
        1.0 1.0 -1.0,
        -1.0 1.0 -1.0,
     # Coordonatele de jos ale cubului
        -1.0 -1.0 1.0,
        1.0 -1.0 1.0,
        1.0 -1.0 -1.0,
        -1.0 -1.0 -1.0
     ]
  }
  coordIndex [
     0, 1, 2, 3, -1
     7, 6, 5, 4, -1,
     0, 4, 5, 1, -1,
     1, 5, 6, 2, -1,
     2, 6, 7, 3, -1,
     3, 7, 4, 0
  ]
}
```

Aplicaţia 1. Să se schimbe ordinea de furnizare a vârfurilor pentru una dintre feţele cubului. Apoi să se vizualizeze fişierul din browser. Ce s-a întâmplat?

Aplicaţia 2. Să se modeleze o piramidă cu baza pătrat.

Aplicaţia 3. Să se modeleze o prismă triunghiulară.

Pas 5. Să se afişeze fişierul **`cilindru.wrl`** în format wireframe şi în format solid. Folosind un editor de texte ASCII să se vizualizeze descrierea scenei. Se constată că singura deosebire faţă de fişierul anterior este nodul **`geometry`**. Deşi cilindrul este modelat tot poligonal nu a mai fost necesară descrierea prin vârfuri şi feţe deoarece obiectul este modelat intern. Limbajul are câteva asemenea primitive care pot fi utilizate (**`Cone, Cube, Sphere, Cylinder`**).

```
geometry Cylinder
   {
     height 2.0
     radius 1.5
   }
```

Pas 6. Să se afişeze fişierul **`cupola.wrl`** în format wireframe şi în format solid. Folosind un editor de texte ASCII să se vizualizeze descrierea scenei. Spre deosebire de fişierele anterioare, nodul rădăcină este nodul **`Group`** care are ca subnoduri, printre altele, şi nodul **`Shape`**. S-a procedat aşa, deoarece nu s-au descris doar obiectele ci s-au făcut şi setări pentru culoarea fundalului (nodul **`Background`**) şi pentru sursa de lumină (nodul **`DirectionalLight`**). Deoarece s-au modelat două obiecte, nodul **`appearance`** al primului obiect, a

fost etichetat (folosind DEF) ca nod **`Brown`** şi utilizat apoi şi la modelarea celui de al doilea obiect (folosind **`USE`**). Apoi nodul **`Shape`** al celui de al doilea obiect apare ca subnod al unui nod Transform. Aceasta deoarece altfel cele două obiecte ar avea aceeaşi poziţie. Translatarea conului s-a făcut cu 2 unităţi (adică atât cât este înălţimea cilindrului) pe axa y.

```
# Acoperisul cupolei
Transform {
   translation 0.0 2.0 0.0
   children [
      Shape {
         appearance USE Brown
         geometry Cone {
            height 2.0
            bottomRadius 2.5
            }
      }
   ]
}
```

Pas 7. Să se urmărească şi celelalte două fişiere (`asterix.wrl` şi `plus_trid.wrl`).

Aplicaţia 4. Să se modeleze un cub care are muchiile formate din cilindrii albaştrii şi în vârfuri are câte o sferă roşie.

De reţinut:

1. Limbajul VRML descrie scenele.
2. Fişierele VRML conţin descrierea scenelor iar viewer-ele realizează reprezentarea pe baza acestor descrieri.
3. În modelarea poligonală contează ordinea specificării vârfurilor.

4. Triangularizarea poligoanelor elimină necesitatea impunerii unor condiţii în modelarea poligonală:
    a. Toate vârfurile unui poligon trebuie să aparţină aceluiaşi plan.
    b. Poligoanele trebuie să fie convexe – ceea ce simplifică algoritmii de umplere.
    c. Poligoanele nu pot fi răsucite.

LABORATOR 2 - Introducere în OpenGL

2.1 Suport teoretic

OpenGL este o API (interfaţă pentru programarea aplicaţiilor grafice) care poate fi implementată atât hardware cât şi software. Varianta hardware presupune executarea funcţiilor OpenGL dintr-o aplicaţie de către acceleratorul grafic OpenGL existent pe placa grafică. Varianta software presupune executarea de către microprocesor a procedurilor corespunzătoare fiecărui apel OpenGL. Interfaţa OpenGL este scalabilă astfel că a fost îmbunătăţită continuu (versiunile 1.1, 1.2, 1.3, 1.4) sub conducerea colectivului ARB (Architecture Review Board).

OpenGL este o interfaţă independentă de hardware. Independenţa este realizată prin neincluderea în ea a funcţiilor dependente de hardware. Aceasta presupune că sistemele de operare sunt cele care au fost adaptate pentru a suporta OpenGL. Biblioteca OpenGL este implementată pe diverse platforme hardware (Intel, Macintosh) şi pe diverse sisteme de operare (Windows, Unix), singura condiţie fiind de a rula într-o fereastră. Funcţiile OpenGL pot fi apelate din aproape orice mediu şi limbaj de programare (Visual C, Borland C, Java, Fortran, Basic, etc.). OpenGL stă la baza unor aplicaţii grafice mai complexe pentru modelarea scenelor 3D (spre exemplu 3D Studio Max).

Fereastra în care pot fi rulate aplicaţii OpenGL nu este una obişnuită. Deci nu în orice fereastră pot fi afişate scene 3D construite cu OpenGL. Modul de specificare al ferestrei OpenGL este diferit funcţie de mediul de programare utilizat.

Pentru câteva medii găsiţi modul de realizare în [1][1]. Pentru a veni în sprijinul programatorilor s-au creat câteva interfeţe care pe lângă crearea în mod simplu a ferestrelor pun la dispoziţie funcţii pentru interacţiunea cu periferia (mouse, tastatură). Biblioteca GLAUX, de la Microsoft poate fi utilizată din Visual C. Noi vom utiliza această bibliotecă la laborator, având în vedere că se instalează odată cu instalarea Visual Studio, şi că este uşor de utilizat. Biblioteca GLUT este mai complexă, are şi funcţii pentru crearea meniurilor, şi în plus poate fi utilizată şi sub Windows şi sub Unix. Cea mai complexă este însă biblioteca GLUI, bazată pe GLUT, care permite crearea unor interfeţe complexe (butoane separate pentru transformări geometrice, ferestre separate pentru comenzi, meniuri, casete de dialog).

OpenGL nu conţine comenzi de nivel înalt pentru descrierea obiectelor 3D. Asemenea comenzi ar permite specificarea unor forme relativ complicate cum ar fi: automobile, părţi ale corpului omenesc, avioane sau molecule. Cu OpenGL trebuie să construieşti modelul dorit folosind un mic set de primitive grafice: puncte, linii şi poligoane. Dar pentru a veni în sprijinul utilizatorilor OpenGL s-au construit biblioteci suplimentare (GLU, GLAUX, GLUT) care pun la dispoziţie funcţii pentru afişarea unor obiecte 3D mai complicate. Aceste funcţii folosesc de fapt primitivele OpenGL. Spre exemplu biblioteca GLU permite afişarea cvadricelor şi a curbelor şi suprafeţelor spline. Bibliotecile GLAUX şi GLUT permit afişarea unui set de obiecte (ceainic, cilindru, sferă, paralelipiped, icosaedru, etc.). Prototipurile funcţiilor respective pot fi găsite în fişierele header (`glu.h, glaux.h, glut.h`).

Aplicaţia *pătrat.c* este o aplicaţie simplă pe care o putem considera un template pentru aplicaţiile care vor urma. Ea

[1] Baciu, R. Programarea aplicaţiilor grafice 3D cu OpenGL, Editura Albastră, Cluj-Napoca, 2005

folosește funcțiile GLAUX pentru crearea ferestrelor, a interacțiunii cu tastatura și cu mouse-ul.

2.2 Desfășurarea lucrării

Aplicația `pătrat.c` afișează un pătrat pe care îl translatează pe axa x la apăsarea săgeților stânga, dreapta.

2.2.1 Fazele obținerii executabilului

a) Dacă se utilizează Visual Studio 5.0 sau 6.0:

- se compilează sursa (fișierul `patrat.c`) și se creează

```
Build
    └→ Compile
```

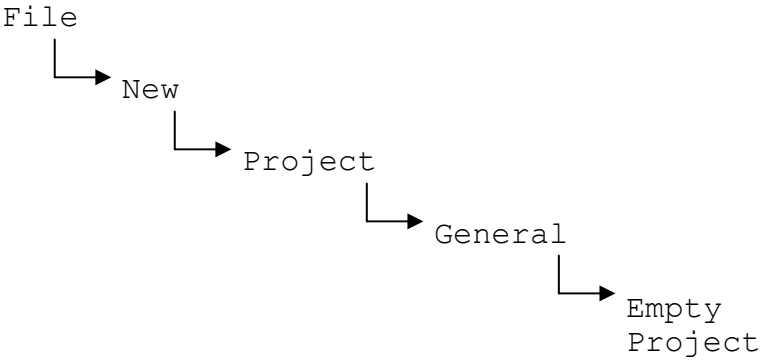
  proiectul:

- se adaugă proiectului bibliotecile `opengl32.lib`, `glu32.lib`, `glaux.lib` astfel:

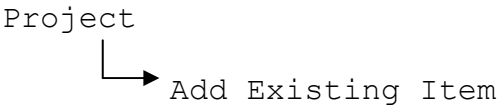
```
Project
    └→ Settings
        └→ Link
            └→ fișierele
               de mai sus
```

a) Dacă se utilizează Visual Studio 2008:

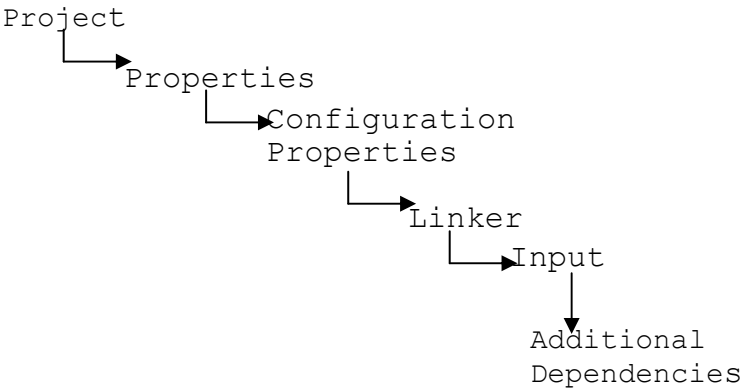
- se creează proiectul



- Se complectează numele proiectului după ce s-a ales directorul în care se lucrează.
- Se adaugă sursele (`pătrat.c` şi `glos.h`).



- Se adaugă bibliotecile:



2.2.2. Comentarea codului sursă

Funcţia **`main(...)`** *iniţializează şi deschide o fereastră pe ecran* cu ajutorul următoarelor rutine GLAUX (rutinele GLAUX încep cu **aux**):

`auxInitDisplayMode()` - creează o fereastră în modelul de culoare RGB cu un singur (AUX_SINGLE) buffer de culoare (AUX_RGB) asociat (OpenGL conţine un buffer de cadru care la rândul său conţine buffer-ul (ele) de culoare, buffer-ul şablon, buffer-ul de acumulare, buffer-ul de adâncime). Deoarece dimensiunea buffer-elor depinde de dimensiunea ferestrei este normal ca la definirea ferestrei să se specifice ce conţine buffer-ul de cadru.

`auxInitPosition()` – setează poziţia ferestrei pe ecran şi dimensiunea ferestrei;

`auxInitWindow()` – deschide fereastra pe ecran, setează butonul ESC pentru ieşirea din program şi asociază un şir pentru titlul ferestrei;

Pentru *tratarea evenimentelor de la tastatură* sunt apelate următoarele funcţii GLAUX:

`auxKeyFunc()` – asociază tastelor AUX_LEFT (←) şi AUX_RIGHT(→) funcţiile de tip callback **`MutaStanga`**, **`MutaDreapta`** definite în codul sursă. De asemenea înregistrează în sistemul de operare aceste funcţii. Sistemul de operare le va apela ori de câte ori sunt apăsate tastele respective.

auxReshapeFunc() – înregistrează în sistemul de operare funcţia de tip callback **myReshape** care indică ce acţiuni să aibă loc în cazul în care fereastra este redimensionată. Funcţia callback apelată – **myReshape()**- va fi discutată ulterior pentru o mai bună înţelegere.

auxMainLoop() - înregistrează în sistemul de operare funcţia **display()** care va fi apelată ori de câte ori se va redesena conţinutul ferestrei şi apoi o apelează.

display()- este funcţia ce conţine codul OpenGL pentru desenarea scenei 3D. De obicei, prima funcţie apelată este **glClear(GL_COLOR_BUFFER_BIT),** care şterge bufferul de culoare. Ultima funcţie apelată este **glFlush()** care "îi spune lui OpenGL:
— Nu mai aştepta alte comenzi, ai toate informaţiile pentru a reprezenta scena"

Funcţia **glLoadIdentity()** - iniţializează matricea curentă (care poate fi matricea de modelare-vizualizare sau matricea de proiecţie sau matricea de texturare) ca matrice de identitate. Acest pas este necesar după ce se comută pe matricea de modelare-vizualizare (cu **glMatrixMode()**) deoarece toate comenzile de transformare multiplică matricea curentă cu o anumită matrice şi apoi rezultatele se trec în matricea curentă. Dacă nu este iniţializată matricea de modelare-vizualizare cu matricea identitate se riscă pornirea cu o matrice de modelare-vizualizare al cărei conţinut nu este controlat.

În funcţia **display()**, urmează apoi definirea culorilor şi a coordonatelor vârfurilor pătratului cu ajutorul rutinelor

glColor3f(), **glVertex2f()** cuprinse între funcţiile **glBegin() şi glEnd()**.

glFlush() - rutina nu aşteaptă ca desenul să fie complet, ci forţează execuţia funcţiilor OpenGL anterioare.

Aplicaţia 1. Să se adauge aplicaţiei translaţia SUS-JOS a pătratului cu ajutorul săgeţilor sus-jos ale tastaturii.

Observaţie: Constantele simbolice asociate cu codurile tastelor se găsesc în fişierul `glaux.h`.

Aplicaţia 2. Să se rotească obiectul folosind butoanele mouse-ului. (Se va utiliza rutina **glRotatef**`(alfa, 0, 0, 1)`, unde „alfa" specifică unghiul de rotaţie, ceilalţi trei parametrii specifică axa de rotaţie care trece întotdeauna prin origine şi prin vârful ale cărei coordonate sunt date ca parametrii).
Funcţia GLAUX care înregistrează în sistemul de operare funcţia apelată la acţionarea mouse-ului va fi:

```
auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, rot_z_up);      // înregistrează funcţia callback rot_z_up
auxMouseFunc (AUX_RIGHTBUTTON, AUX_MOUSEDOWN, rot_z_down);   // înregistrează funcţia callback rot_z_down
```

Se vor programa corespunzător funcţiile **rot_z_up** şi **rot_z_down**.

În cazul în care se foloseşte Visual Studio 2008 sau 2005 funcţia **rot_z_up** va avea prototipul:

```
void CALLBACK rot_z_up(AUX_EVENTREC * event)
```

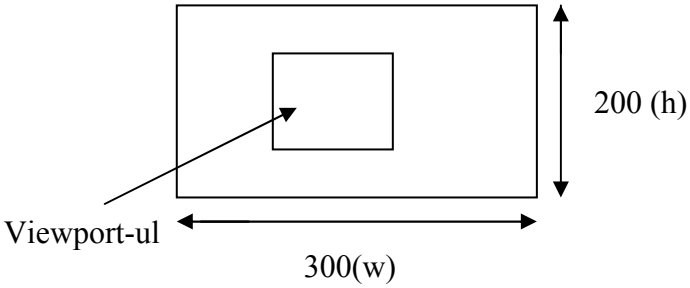
2.2.3. Funcţia myReshape()

Funcţia **`myReshape()`** are un rol foarte important în cazul în care fereastra este redimensionată. Ea apare în aplicaţie în două forme, pentru transformarea anizotropică (comentată), respectiv transformarea izotropică. În această funcţie se defineşte volumul de vizualizare şi viewport-ul. De asemenea se stabilesc dimensiunile volumului de vizualizare în unităţi logice. Unităţile folosite la desenare vor fi unităţi logice. Dacă funcţia **`myReshape()`** lipseşte din aplicaţie se va lucra implicit în pixeli. Trebuie evitat acest lucru, deoarece în acest caz volumul de vizualizare nu are adâncime-ceea ce înseamnă că nu veţi putea afişa obiecte 3D, doar obiecte 2D. Transformarea de viewport controlează conversia coordonatelor logice în coordonate ale ecranului (pixeli). **Transformarea izotropică** presupune că maparea de la unităţi logice la unităţi de dispozitiv se realizează astfel încât dimensiunea în pixeli a unităţilor logice este egală pe cele două axe (x şi y). În cazul transformării anizotropice dimensiunea în pixeli a unităţilor logice este diferită pe cele două axe. Acesta este motivul pentru care un pătrat, spre exemplu, care are laturile egale în unităţi logice va apare pe ecran ca dreptunghi.

Funcţia **`myReshape()`** din această aplicaţie conţine următoarele rutine:

- **`glViewport( 0, 0, w, h)`** – **desemnează dimensiunea viewportului (**indică originea, înălţimea şi lăţimea (în pixeli) a zonei din fereastră în care se afişează scena). Viewport-ul 2D reprezintă porţiunea din fereastră pe care se proiectează scena.

Exemplu:

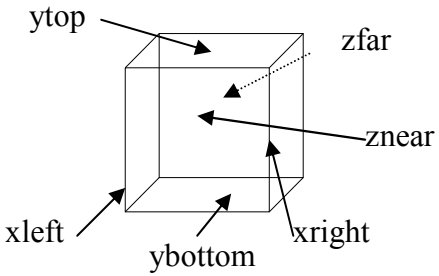


Dacă fereastra îşi măreşte dimensiunea, viewportul trebuie să-şi schimbe dimensiunea în mod corespunzător. În aplicaţia din laborator viewport-ul ocupă întreaga fereastră.

- **`glMatrixMode(GL_PROJECTION)`** – se comută pe matricea curentă de proiecţie. Toate funcţiile următoare, care afectează matricea curentă, vor afecta matricea de proiecţie.

- **`glMatrixMode(GL_MODELVIEW)`** – indică faptul că funcţiile ulterioare care lucrează cu matrice vor afecta matricea curentă de modelare-vizualizare şi nu matricea de proiecţie;

- **`glOrtho()`** – delimitează volumul de vizualizare corespunzător proiecţiei paralele ortografice, corespunzător transformării anizotropice, respectiv izotropice. Parametrii funcţiei sunt `xleft`, `xright`, `ybottom`, `ytop`, `znear`, `zfar`.



Funcţia myReshape() comentată corespunde unei transformări anizotropice, cealaltă funcţie **myReshape()** corespunde unei transformări izotropice.

Pentru a obţine transformarea izotropică se compară `w` cu `h`.
Dacă w>h, stabilim ca bază înălţimea şi stabilim numărul unităţilor logice pe orizontală în funcţie de dimensiunea în pixeli a unei unităţi logice rezultate pe verticală.

$$1 \text{ UL (pe înălţime)} = \frac{h}{320} \text{ pixeli;}$$

Numărul unităţilor logice pe lăţime = $w/(\frac{h}{320}) = 320 * \frac{w}{h}$ unităţi logice se vor fixa pe orizontală.

Aplicaţia 3. Să se ruleze aplicaţia cu fiecare din cele două funcţii **myReshape()** precum şi fără funcţia **myReshape()** (în pixeli). Să se afişeze un obiect 3D (spre exemplu o sferă cu **auxWireSphere()**). Ce se constată?

Aplicaţia 4. Să se fixeze un vârf al patrulaterului în punctul de coordonate (0, 0). Celelalte vârfuri să nu-şi schimbe poziţia. Forma pătratului se va schimba dar important este că ştim unde este originea ferestrei. Să se rescrie funcţia **myReshape()** astfel încât punctul de coordonate logice (0, 0) să nu se mai afle în

centrul ferestrei ci în colţul stânga-jos al ferestrei. Dimensiunea ferestrei să fie tot de 320 unităţi logice iar transformarea să fie tot izotropică.

De reţinut:

Rolul funcţiei **`myReshape()`**;

1. Stabilirea dimensiunii volumului de vizualizare;
2. Stabilirea tipului transformării:
   - izotropică
   - anizotropică;
3. Stabilirea dimensiunii unităţilor logice şi a numărului de unităţi logice;
4. Stabilirea poziţiei originii sistemului de coordonate logic.

LABORATOR 3 - Primitive OpenGL

3.1 Suport teoretic

Primitivele de desenare ale bibliotecii OpenGL (punct, linie, triunghi, patrulater, poligon) pot fi reprezentate cu diferite atribute: culoare, stil linie, grosime linie, etc. Există atribute specifice punctelor, liniilor sau poligoanelor. Fiecare setare poate fi realizată independent, fără a afecta celelalte setări (deşi multe setări pot interacţiona pentru a determina ceea ce se redă în framebuffer). Se specifică toate setările corespunzătoare unei primitive, apoi se specifică primitiva, iar OpenGL va reprezenta primitiva cu atributele corespunzătoare.

Spre exemplu:

```
glPointSize(5);  // dimensiunea pixelului 5
glColor3f(1.0, 0, 0); // culoare roşie
glBegin(GL_POINTS);
      glVertex2f(0.0, 0.0);
glEnd();
```

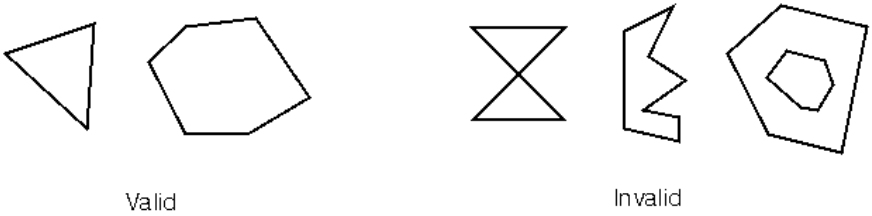
Acest cod va desena un punct roşu, în originea sistemului de coordonate şi de dimensiune `5x5` pixeli.

Primitivele sunt descrise într-un corp **`glBegin()/glEnd()`** şi sunt specificate prin parametrul funcţiei **`glBegin()`**. Corpul conţine unul sau mai multe vârfuri (vertex-uri). Un vârf defineşte un punct, capetele unui segment de linie sau un „colţ" al unui poligon (locul unde se întâlnesc două muchii). Datele asociate unui vârf sunt: coordonate carteziene, culoare, normale,

coordonate de texturare. Corespunzător, în corpul **glBegin()/glEnd()** vor avea efect doar funcţiile **glVertex3f()**, **glColor3f()**, **glNorma3f()**, şi **glTexCoord()**. Fiecare vârf este procesat independent, în ordine şi în acelaşi mod. Singura excepţie de la regulă este cazul în care un grup de vârfuri trebuie decupate faţă de volumul de vizualizare. În această situaţie datele asociate vârfurilor pot fi modificate şi se creează noi vârfuri.

Restricţii pentru vârfurile poligoanelor:

- ordinea în care se dau vârfurile este sensul trigonometric;
- toate vârfurile trebuie să fie în acelaşi plan;
- poligoanele concave şi figurile răsucite nu sunt recunoscute.



Primitivele şi vârfurile lor se vor specifica într-un corp **glBegin()/glEnd();**

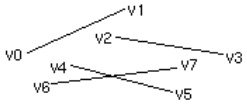
```
glBegin();
      .
      .
glEnd();
```

Parametrul funcţiei **glBegin()** este numele primitivei care se reprezintă şi care poate fi:

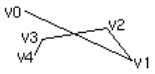
1. Pentru primitive fără atribut de umplere:
   - `GL_POINTS`
   - `GL_LINES`
   - `GL_LINE_STRIP`
   - `GL_LINE_LOOP`
2. Pentru primitive cu atribut de umplere:
   - `GL_TRIANGLES`
   - `GL_TRIANGLE_STRIP`
   - `GL_TRIANGLE_FAN`
   - `GL_QUADS`
   - `GL_QUAD_STRIP`
   - `GL_POLYGON`

Explicitarea primitivelor enumerate mai sus (vârfurile sunt date în sensul acelor de ceasornic):

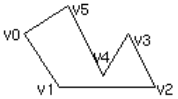
GL_POINTS



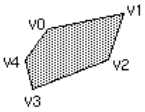
GL_LINES



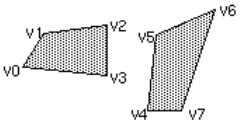
GL_LINE_STRIP



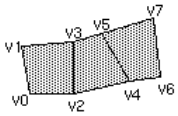
GL_LINE_LOOP



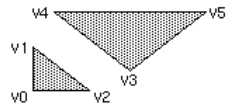
GL_POLYGON



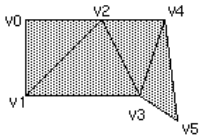
GL_QUADS



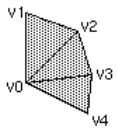
GL_QUAD_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

În corpul **glBegin()/glEnd()** pot fi utilizate funcţiile:

- **glVertex#()** – setează coordonatele carteziene;
- **glNormal#()** – indică coordonatele normalei;
- **glColor#()** – setează culoarea de desenare;
- **glTextCoord#( )** – setează coordonatele pentru texturare.

Caracterul „#” desemnează numărul parametrilor funcţiilor şi tipul acestora.

Exemplu: **glVertex2f(x, y)** – se dau 2 coordonate de tip float;

glVertex3d(x,y,z) – se dau 3 coordonate de tip double;

glVertex4v – se dă un pointer spre un tablou ce conţine 4 elemente (coordonate omogene).

De reţinut că OpenGL lucrează cu punctele specificate în coordonate omogene [x y z w].

Atributul de umplere

Funcţia **glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)** setează modul de redare al poligoanelor (`GL_FILL` – cu atribut de umplere, `GL_POINTS` – conturat cu puncte, `GL_LINE` – wireframe). Primul parametru arată căror poligoane li se aplică setarea:

- poligoanelor faţă: `GL_FRONT`;
- poligoanelor spate: `GL_BACK`;
- poligoanelor faţă-spate: `GL_FRONT_AND_BACK`.

Observaţie: Nu încercaţi să utilizaţi funcţia în corpul **glBegin()/glEnd()**! Nu va avea nici un efect.

3.2 Desfăşurarea lucrării

Utilizând primitivele bibliotecii OpenGL modelaţi următoarele corpuri:

Aplicaţia 1. Desenaţi un cerc folosind succesiv ca primitive punctul, linia şi poligonul (se folosesc 360 de puncte). Se va folosi sursa din laboratorul anterior (`pătrat.c`). În locul pătratului, tot într-un corp **`glBegin()/glEnd()`** se va reprezenta cercul.

Exemplu:

Aplicaţia 2. Modelaţi un cilindru (prin centrul cercului trece axa y, iar cilindrul este înfăşurat de dreptunghiuri). Se va utiliza primitiva `GL_QUAD_STRIP`. Se va lucra pe sursa din laboratorul anterior (`pătrat.c`). Se va dimensiona corespunzător volumul de vizualizare, pe axa z.

Aplicaţia 3. Modelaţi un con (vârful conului este un punct aflat pe una din axe). Se va utiliza primitiva `GL_TRIANGLE_FAN`.

Aplicaţia 4. Desenaţi un paralelipiped folosindu-vă de cilindru (din cele 360 de puncte se aleg numai 4 puncte aflate la 90 de grade unul faţă de celălalt).

De reţinut:

1. Orice funcţie (cu excepţia celor specificate) utilizată în corpul **`glBegin()/glEnd()`** nu are nici un efect.
2. Atributul de umplere implicit este setat pe GL_FILL dar îl puteţi modifica.
3. Valorile coordonatelor specificate prin funcţia **`glVertex()`** trebuie să se încadreze în domeniul unităţilor logice, fixat în funcţia **`myReshape()`**.
4. La desenarea obiectelor 3D trebuie avut în vedere că volumul de vizualizare trebuie dimensionat corect pe axa z. Altfel obiectele vor fi decupate faţă de volumul de vizualizare existent şi nu vor fi complet vizibile
5. Indiferent de numărul coordonatelor unui vârf, intern el este specificat prin 4 valori.

LABORATOR 4 - Cvadrice din biblioteca GLU

4.1 Suport teoretic

Aşa cum s-a arătat în laboratorul 2, folosind funcţiile din biblioteca GLAUX se pot construi, în mod simplu, câteva obiecte 3D. Dezavantajul, în acest caz, este că nu se oferă suficientă flexibilitate. Spre exemplu, pentru un cilindru se poate specifica doar raza şi înălţimea. Cilindru este construit prin patrulatere care-l înfăşoară dar numărul acestora nu poate fi specificat. De aceea, pentru anumite aplicaţii, obiectele construite cu GLAUX pot părea prea colţuroase din cauza numărului mic de poligoane care le alcătuiesc. Funcţiile pentru construirea de cvadrice, din biblioteca GLU sunt mai flexibile. Utilizatorul poate specifica, indirect, numărul de poligoane. Suprafeţele cvadrice sunt suprafeţe definite printr-o ecuaţie de gradul al doilea în x, y, z.

Pentru a crea un obiect de tip cvadrică, se utilizează funcţia **`gluNewQuadric().`** Pentru distrugerea obiectului creat, atunci când acesta nu mai este necesar, se foloseşte funcţia **`gluDeleteQuadric().`** Pentru a specifica alte valori decât cele implicite, în legătură cu stilul de redare al acestor suprafeţe se utilizează funcţiile următoare:

– **`gluQuadricDrawStyle():`** stilul de desenare al cvadricelor - cu puncte, linii sau poligoane.

– **`gluQuadricNormals():`** dacă se vor genera normale la suprafeţe, iar dacă da, se specifică dacă vor fi specificate relativ

la vârfuri sau la suprafeţe. Funcţia se foloseşte când se lucrează cu obiecte iluminate.

– **`gluQuadricTexture():`** dacă se generează coordonate pentru texturare. Funcţia se foloseşte când se texturează obiectele.

– **`gluQuadricOrientation():`** care feţe vor fi considerate exterioare şi care interioare.

După ce s-a specificat stilul de redare, se apelează funcţiile corespunzătoare tipului de cvadrică ce se reprezintă: **`gluSphere(), gluCylinder(), gluDisk()`** sau **`gluPartialDisk().`** Dacă se doreşte mărirea sau micşorarea obiectului, este recomandat să se specifice noile dimensiuni decât să se folosească funcţia de scalare **`glScalef()`**. Motivul este creşterea vitezei de reprezentare. Pentru o redare a iluminării cât mai precisă se recomandă valori ridicate pentru parametrii `loops` şi `stacks`. În continuare se vor da prototipurile tuturor acestor funcţii.

Funcţii pentru controlarea obiectelor

```
GLUquadricObj* gluNewQuadric (void);
```

- Funcţia permite crearea unei cvadrice.

```
void  gluDeleteQuadric (GLUquadricObj *state);
```

- Funcţia permite ştergerea unei cvadrice.

Funcţii pentru modificarea felului în care sunt desenate cvadricele

```
void gluQuadricNormals(GLUquadricObj
*quadObject, Glenum normals);
```

Parametrul `normals` poate lua următoarele valori:

`GLU_NONE`	Nu se generează normale pentru iluminare
`GLU_FLAT`	Normalele sunt generate pentru fiecare faţă, rezultând o iluminare poligonală constantă pe poligoane
`GLU_SMOOTH`	Normalele sunt generate pentru fiecare vârf, rezultând o iluminare GOURAUD.

```
void gluQuadricTexture(GLUquadricObj
*quadObject, GLboolean textureCoords);
```

- Funcţia activează `(GL_TRUE)` sau dezactivează `GL_FALSE)` generarea coordonatelor de texturare.

```
void gluQuadricOrientation (GLUquadric
*quadObject, GLenum orientation);
```

- Funcţia controlează direcţia normalelor pentru iluminare. Acestea pot avea orientarea spre exteriorul obiectelor `(GLU_OUTSIDE)` sau spre interiorul acestora `(GLU_INSIDE)`.

```
void gluQuadricDrawStyle (GLUquadricObj
*quadObject, GLenum drawStyle);
```

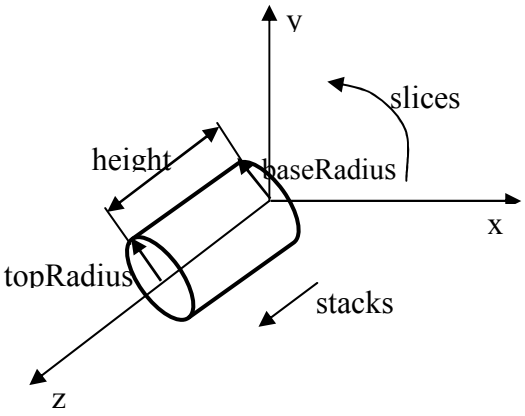
- Funcţia specifică tipul primitivelor OpenGL utilizate pentru reprezentarea cvadricelor. Parametrul `drawStyle` poate lua următoarele valori:

`GLU_FILL`	Cvadricele au atribut de umplere şi pentru generarea lor sunt utilizate poligoane.
`GLU_LINE`	Cvadricele sunt reprezentate wireframe utilizând primitivele pentru reprezentarea liniilor

`GLU_SILHOUETTE`	Cvadricele sunt reprezentate utilizând primitivele pentru linii; sunt desenate doar muchiile exterioare.
`GLU_POINT`	Cvadricele sunt reprezentate utilizând ca primitivă punctul

Funcţii pentru specificarea tipului cvadricei

`void gluCylinder(GLUquadricObj *qobj, GLdouble baseRadius, GLdouble topRadius,GLdouble height, GLint slices, GLint stacks);`



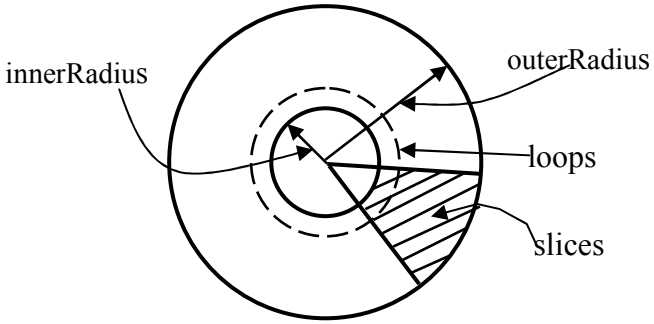
- Funcţia generează un cilindru având centrul bazei în originea sistemului de axe. Funcţia poate fi utilizată şi pentru generarea conurilor, prin specificarea uneia dintre cele două raze ca fiind 0.

- `Slices` reprezintă numărul poligoanelor care se generează în jurul cilindrului.

- `Stacks` reprezintă numărul poligoanelor generate în lungul cilindrului. Această valoare va creşte atunci când se doreşte o iluminare mai bună.

```
void gluDisk(GLUquadricObj *qobj, GLdouble
innerRadius, GLdouble outerRadius, GLint
slices, GLint loops);
```

- Discurile sunt forme plane care au forma unui CD (sau dischetă). Funcţia poate fi utilizată şi pentru generarea cercurilor, poligoanelor.



-`innerRadius` reprezintă raza interioară;
-`outerRadius` reprezintă raza exterioară.
-`slices` reprezintă numărul sectoarelor generate.
-`loops` reprezintă numărul cercurilor concentrice generate la reprezentarea discurilor.

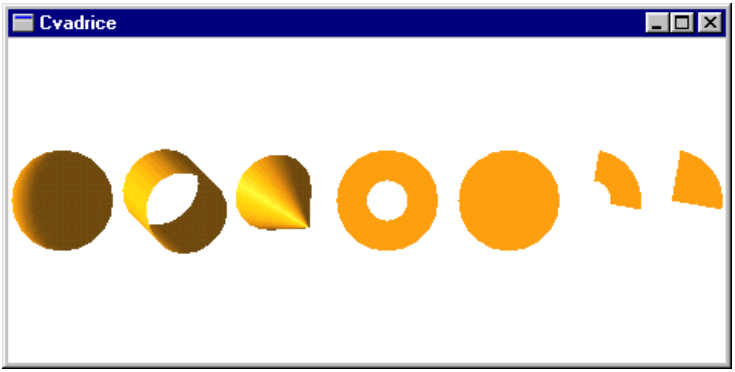
```
void gluPartialDisk(GLUquadricObj  *qobj,
GLdouble innerRadius,  GLdouble outerRadius,
Glint slices, GLint loops, GLdouble startAngle,
GLdouble sweepAngle);
```

- Funcţia este utilizată pentru reprezentarea unui sector dintr-un disc. Pentru aceasta se specifică valorile unghiurilor între care se încadrează sectorul. Funcţia poate fi utilizată şi pentru generarea unui sector dintr-un cerc.

```
void gluSphere(GLUquadricObj *qobj, Gldouble
radius, GLint slices, GLint stacks);
```

- Funcţia reprezintă o sferă.

- `slices` determină numărul longitudinilor generate,
- `stacks` pe cel al latitudinilor generate la reprezentarea sferei.



4.2 Desfăşurarea lucrării

Folosind funcţiile pentru reprezentarea cvadricelor reprezentaţi obiectele următoare: sferă, cilindru, con, disc, cerc, sector de disc, sector de cerc, cub, prismă hexagonală, prismă triunghiulară, piramidă cu baza pătrat, trunchi de piramidă, hexagon, pentagon, triunghi, etc.

Pentru aceasta se va avea în vedere că, în mod implicit, obiectele sunt modelate în originea sistemului de coordonate. Pentru a fi vizibile toate obiectele în aceeaşi fereastră se va face

o translaţie înaintea reprezentării fiecărui obiect. Pentru ca parametrii translaţiei să fie relativi la originea sistemului de coordonate, fiecare obiect va fi reprezentat într-un corp de tipul următor:

```
glPushMatrix();
glTranslatef(x, y, z);
//reprezentare obiect
glPopMatrix();
```

Aplicaţia 1. Să se reprezinte o sferă. Se va utiliza fişierul `pătrat.c` din al doilea laborator. În locul pătratului se va reprezenta sfera. Atenţie la dimensiunile volumului de vizualizare!

Aplicaţia 2. Să se adauge, rând pe rând, toate obiectele specificate anterior.

De reţinut:

1. Corpul **`glBegin()/glEnd()`** nu mai este necesar. El a fost utilizat intern la scrierea funcţiilor GLU pentru reprezentarea obiectelor.
2. Pentru ca obiectele să fie vizibile ele trebuie să aibă altă culoare decât cea a fundalului.
3. Dimensiunile obiectelor trebuie să se încadreze în volumul de vizualizare.
4. Este necesar un singur pointer la un obiect de tipul cvadrică şi folosindu-l se pot genera o mulţime de obiecte.
5. Folosind funcţia pentru cilindru puteţi crea şi un con, un paralelipiped, un cub, diferite prisme şi diferite piramide sau trunchiuri de piramide.

6. Folosind un disc puteţi crea un cerc şi toate tipurile de poligoane regulate.

LABORATOR 5 - Curbe parametrice şi suprafeţe parametrice

5.1 Suport teoretic

Curbele parametrice se reprezintă pe baza unui polinom în *u* - *P(u)* - de un anumit grad. Pentru fiecare valoare a lui *u* se obţine alt punct pe curbă.

În OpenGL pot fi reprezentate curbe Bézier. Utilizând biblioteca GLU se pot reprezenta şi curbe spline.

Pentru fiecare clasă de curbe se furnizează punctele de control pentru definirea formei curbei, puncte ce aproximează forma curbei. Pentru fiecare punct de pe curbă se determină valorile *x(u), y(u), z(u).* Cu cât numărul de puncte de control este mai mare cu atât gradul polinomului, în *u*, este mai mare, iar lucrul cu un asemenea polinom este greoi. De aceea se optează pentru segmentarea curbei (se preferă lucrul cu curbe de gradul 3).

În OpenGL, pentru calcularea punctelor de pe curbă se folosesc evaluatori (funcţii de evaluare). Aceştia utilizează punctele de control şi domeniul lui *u* pentru care vrem să calculeze punctele de pe curbă. Punctele determinate pe curbă sunt unite prin linii. Calitatea reprezentării creşte dacă numărul punctelor determinate este mai mare.

Curbe Bézier

Caracteristici:

- primul şi al doilea punct de control, unite, formează o dreaptă tangentă la curbă;
- ultimul şi penultimul punct de control, unite, formează o dreaptă tangentă la curbă;
- curba este înscrisă în poligonul caracteristic (poligon convex format de punctele de control).
- curba va fi atrasă în zona în care sunt definite mai multe puncte de control;
- curba trece totdeauna prin primul şi ultimul punct de control;
- parametrul *u* variază în intervalul [0, 1].

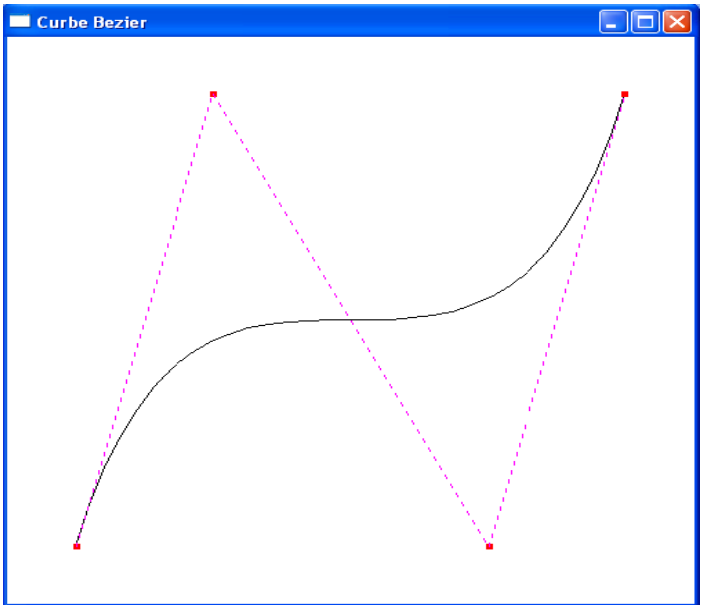


Fig. 1

Curbe spline

În cazul acestor curbe relaţiile permit modelarea unor curbe de forme mai complicate, deci mai multă flexibilitate. Oricât de multe puncte de control se utilizează nu creşte gradul polinomului. Gradul curbei este dat de un parametru suplimentar „t". Curba va fi fragmentată în segmente de curbă, dar acest lucru este transparent pentru utilizator. Pentru realizarea unei curbe se folosesc următoarele relaţii:

- grad: `t-1`;
- număr de puncte de control: `n+1`;
- număr de segmente de curbă: `n-t+2`,
- `u`$\in$`[0, n-t+2]`;
- numărul valorilor nodale (r_j): `n+t+1` (acestea se pun într-un tablou `knots` în aplicaţia din laborator);

$$r_j = \begin{cases} 0, & \text{pentru } j<t; \\ j-t+1, & \text{pentru } t \le j \le n; \\ n-t+2, & \text{pentru } j>n. \end{cases}$$

- număr de subintervale: `n+t`.

Pe baza acestor relaţii rezultă următoarea tehnică de stabilire a valorilor nodale. După ce se calculează domeniul lui *u*, primele *t* valori pentru valorile nodale vor fi 0, următoarele valori 1, 2, s.a.m.d. până se ajunge la valoarea maximă a lui *u* care se repetă de *t* ori.

Exemplu:

Pentru reprezentarea unei curbe după un polinom de grad 3 avem relaţiile:

- grad = 3 $\Rightarrow$ t=4;
- n = 5 $\Rightarrow$ 6 puncte de control: P_0, P_1, P_2, P_3, P_4, P_5.
- u$\in$ [0, 3];
- 3 segmente: Q_0 , Q_1 , Q_2 ;

- 9 subintervale;
- 10 valori nodale: - $r_0 = r_1 = r_2 = r_3 = 0$,
  $r_4 = 1$,
  $r_5 = 2$,
  $r_6 = r_7 = r_8 = r_9 = 3$.

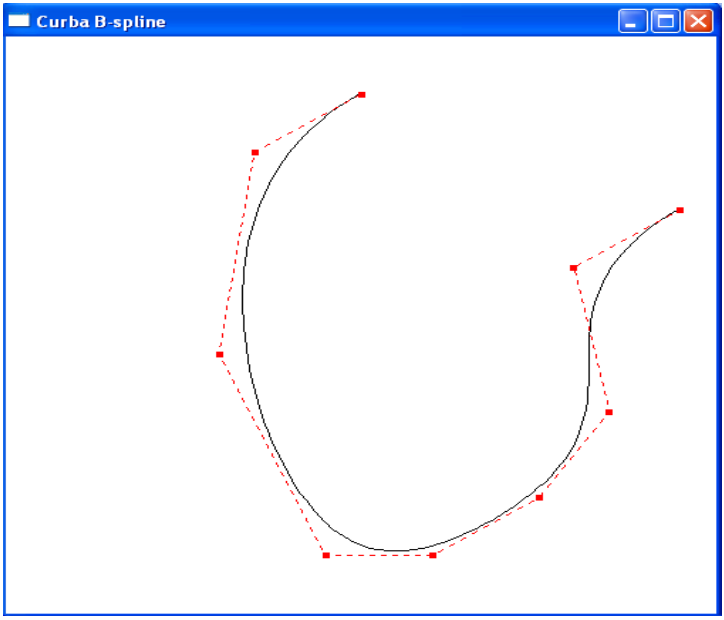


Fig. 2

Suprafeţe parametrice

Dacă punctele de control care descriu o curbă în *u* variază după curbe în *s*, ceea ce descriu este o suprafaţă *S(u, s)*. Dacă se menţine *u* constant şi se variază *s* se obţine o curbă. Dacă se menţine *s* constant şi se variază *u* se obţine de asemenea o curbă.

Pentru diferite valori pentru *u* şi *s* se obţin puncte pe suprafaţa *P(u, s)*. Pentru definirea fiecărei suprafeţe parametrice cubice (de grad 3) sunt necesare 16 puncte de control.

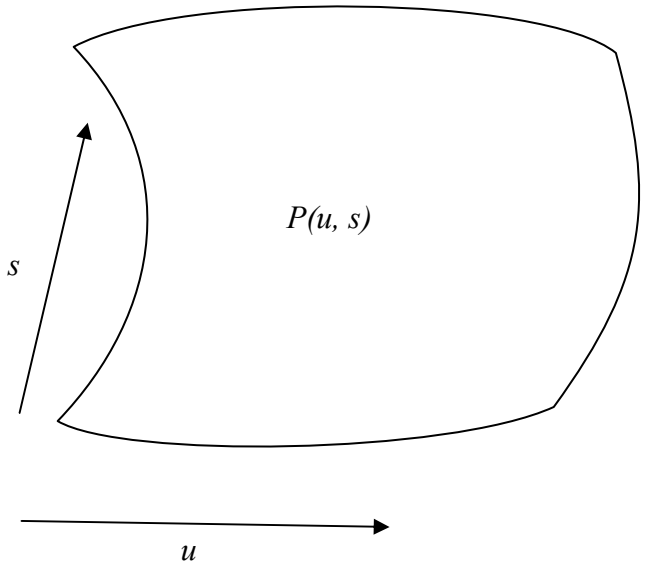


Fig. 3

5.2 Desfăşurarea lucrării

Aplicaţia 1. Se dă sursa `curbe_Bezier.c`. Cunoscând caracteristicile curbelor Bézier să se modeleze o curbă de forma unei sinusoide (**Fig. 4**) din două segmente de curbă. În punctul P să existe continuitate geometrică de ordinul 1.
Definiţie: Continuitatea geometrică de ordinul 1 înseamnă că tangentele în punctul P la fiecare din cele două curbe sunt coliniare.

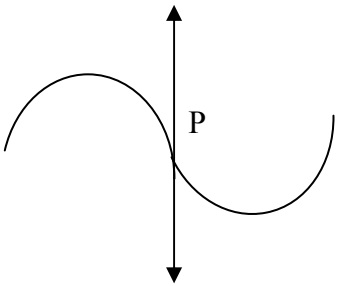


Fig.4

Aplicaţia 2. Se dă sursa `Curba_spline.c`. Programul utilizează biblioteca GLU pentru redarea unei curbe spline. În program funcţia afişează o curbă B-spline: obiectul este de tip **`GLUnurbsObj`** şi este creat dinamic în memoria liberă prin apelul funcţiei **`gluNewNurbsRenderer().`**
Funcţia **`display()`** desenează curba respectivă ţinând cont de vectorul de puncte de control, de numărul de noduri, de intervalul de valori dintre două puncte de control consecutive, de ordinul curbei, de tipul evaluatorului folosit (exemplu: `GL_MAP1_VERTEX_3`).
Să se modeleze o curbă de forma literei „e” folosind curbele spline.

Suprafeţe Bézier şi spline

Aplicaţia 3. Realizaţi o formă de ou format din două suprafeţe. Fiecare dintre cele două suprafeţe va avea forma de căpiţă. Se vor folosi o dată suprafeţele Bézier, iar apoi suprafeţele spline. Pentru a obţine forma de căpiţă se va proceda în felul următor.

1. Cele 16 puncte de control se stabilesc astfel încât să formeze o reţea de puncte aflate pe acelaşi plan, şi

care controlează un pătrat. Punctele se furnizează de jos în sus şi de la stânga la dreapta, câte patru puncte pe un rând. Se păstrează distanţe egale între puncte.

2. Se vizualizează suprafaţa şi se verifică dacă s-a obţinut un pătrat.
3. Punctele încercuite în figură sunt mutate apoi într-un plan paralel cu pătratul, pentru a "bomba" suprafaţa.
4. Punctele din chenare se deplasează spre exterior pentru a obţine o formă cât mai apropiată de forma unui cerc.

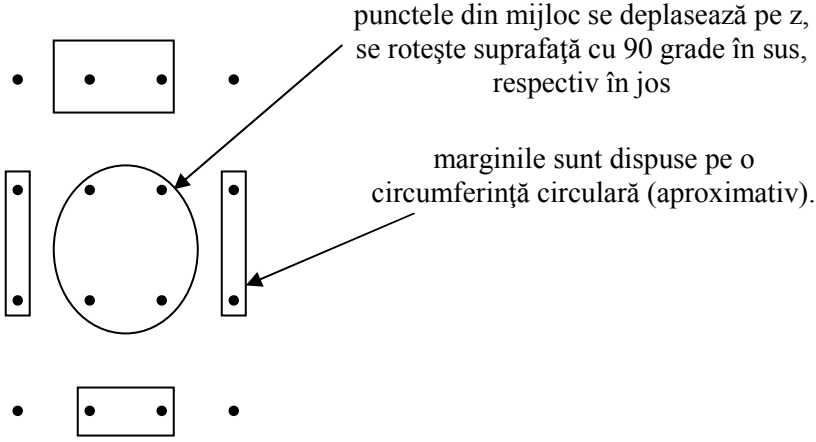


Fig.5

De reţinut:

1. Înainte de a desena o curbă sau o suprafaţă trebuie stabilite atributele acesteia.
2. Dacă se desenează mai multe curbe sau suprafeţe, setările acestora trebuie stabilite înainte de fiecare reprezentare.

3. Funcţia **`myinit()`**, care poate primi orice nume, nu este o funcţie obligatorie într-o aplicaţie OpenGL. Ea scade însă timpul de execuţie al aplicaţiei şi de aceea este bine să avem o asemenea funcţie. Ea grupează toate setările care se fac o singură dată la începutul aplicaţiei. Aceste setări ar putea fi, foarte bine, grupate la începutul funcţiei **`display()`**, însă în acest caz s-ar executa în mod repetat ori de câte ori s-ar executa funcţia **`display()`**, adică foarte des. Dacă însă o setare făcută în **`myinit()`** se modifică în mod repetat în timpul aplicaţiei atunci ea trebuie sa fie mutată în funcţia **`display()`**.
4. În OpenGL setările se fac fie cu funcţia **`glEnable()/glDisable(),`** fie cu funcţii specifice, cum ar fi funcţia **`glMap()`**, în cazul curbelor şi al suprafeţelor.
5. Aceeaşi suprafaţă, controlată de anumite puncte de control, se poate desena în poziţii diferite dacă înainte de desenare se aplică diferite transformări geometrice.

LABORATOR 6 - Compunerea transformărilor geometrice în OpenGL

6.1 Suport teoretic

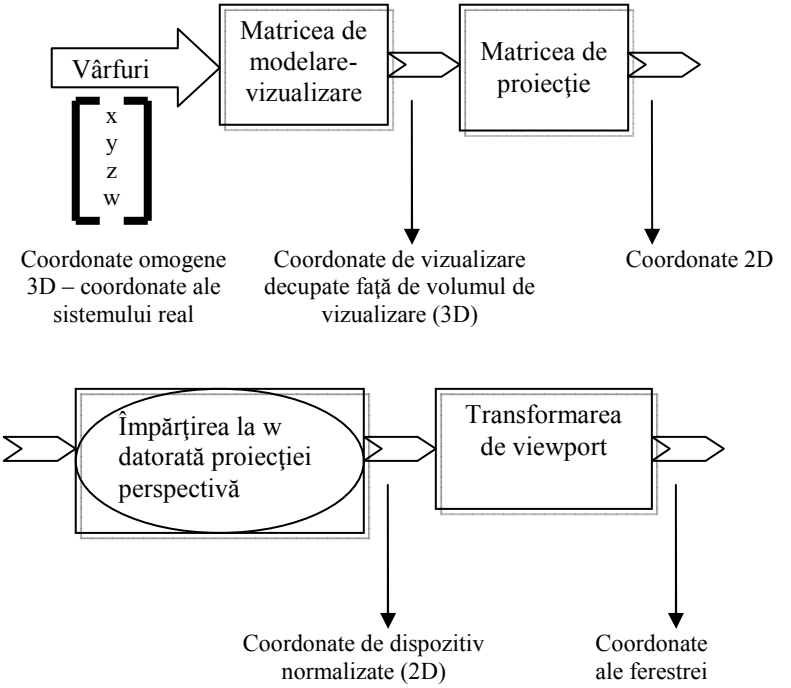
Într-o aplicaţie grafică un obiect poate trece prin 3 tipuri de transformări şi anume transformări de vizualizare, de modelare şi de proiecţie. Procesul de vizualizare al unei scene este similar procesului de fotografiere cu o cameră video. Transformarea de modelare presupune plasarea obiectelor în scenă. Transformarea de proiecţie presupune obţinerea, din modelul 3D, a scenei unui model 2D care este posibil de reprezentat pe dispozitivele de afişare 2D de care dispunem în prezent. Transformările geometrice intervin în toate aceste 3 categorii de transformări pe care le-am explicat pe scurt.

Funcţiile pentru transformări geometrice (de care dispune OpenGL) sunt:

- **`glRotate#`**`(alfa, x, y, z)`- rotaţie cu unghiul `alfa` faţă de o axă care trece prin originea sistemului de coordonate sau faţă de axele sistemului de coordonate;
- **`glTranslate#`**`(Tx, Ty, Tz)`- translaţie cu vectorul (`Tx, Ty, Tz`);
- **`glScale#`**`(Sx, Sy, Sz)`- scalare faţă de originea sistemului de coordonate.

Transformarea de vizualizare trebuie să preceadă transformările de modelare în codul programului, dar se poate

specifica proiecţia şi transformările de vizualizare în orice punct din cod, înainte ca desenarea să se producă. Schema următoare arată ordinea în care aceste operaţii au loc în OpenGL:



Pentru specificarea unei transformări se foloseşte o matrice `4x4 (M)` care multiplică coordonatele fiecărui vârf (v) din scenă, obţinând astfel coordonatele vârfului (v') după aplicarea transformării specificate de matricea M.

$$v' = v * M$$

În OpenGL relaţia se transpune:

[x′ y′ z′ w′] = [x y z w] $*$ M **/ T**

// T-transpus

$$[x' \quad y' \quad z' \quad w']^T = M^T * [x \quad y \quad z \quad w]^T$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = M^T * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

! OpenGL lucrează cu matricele transformărilor transpuse. Deci foloseşte M^T în loc de M.

Presupunem că se aplică o rotaţie (M1), o translaţie (M2) şi apoi o altă rotaţie (M3). Deoarece M este compusă din 3 transformări ($M = M_1 * M_2 * M_3$) în urma transpunerii se obţine:

$$M^T = M_3^T * M_2^T * M_1^T .$$

! Deci, transformările trebuie apelate în ordine inversă: rotaţia finală, translaţia şi apoi prima rotaţie, chiar dacă ordinea logică a transformărilor este alta.

Reţine!, vârfurile au întotdeauna 4 coordonate, deşi în multe cazuri `w` este 1 şi pentru obiectele `2D` coordonata `z` este `0`.

OpenGL lucrează cu 3 matrice curente: matricea de modelare-vizualizare, matricea de proiecţie, matricea de texturare, dar numai una dintre ele este activă la un moment dat. Transformările apelate se vor aplica matricei active în punctul respectiv din cod. Pentru comutarea între matricele curente se foloseşte rutina **`glMatrixMode()`** cu unul dintre parametrii:

```
GL_PROJECTION,
GL_MODELVIEW,
GL_TEXTURE.
```

Pentru salvarea unei matrice curente se utilizează operaţiile cu stiva:

`glPushMatrix()`- copiază matricea curentă şi o adaugă în vârful stivei, iar

`glPopMatrix()`- actualizează matricea curentă cu matricea din vârful stivei.

Transformarea de proiecţie, aşa cum îi spune şi numele, determină cum sunt proiectate obiectele pe ecran. Există 2 tipuri de proiecţie: proiecţia perspectivă, care corespunde tipului de proiecţie realizată pe retina umană (exemplu: obiectele care se află în depărtare se văd mici) şi proiecţia ortografică ce afişează obiectele pe ecran fără a modifica dimensiunile. Acest ultim tip se utilizează în proiectarea asistată unde imaginea finală trebuie să reflecte dimensiunile exacte ale obiectelor, paralelismul laturilor şi nu aspectul lor. Arhitecţii utilizează în proiectare proiecţia paralelă iar proiecţia perspectivă o utilizează doar pentru a arăta imaginea anumitor clădiri sau spaţii interioare văzute din diferite puncte de vizualizare.

În OpenGL, proiecţia perspectivă se specifică prin:

- **gluPerspective**(alfa, $\frac{w}{h}$, near, far)

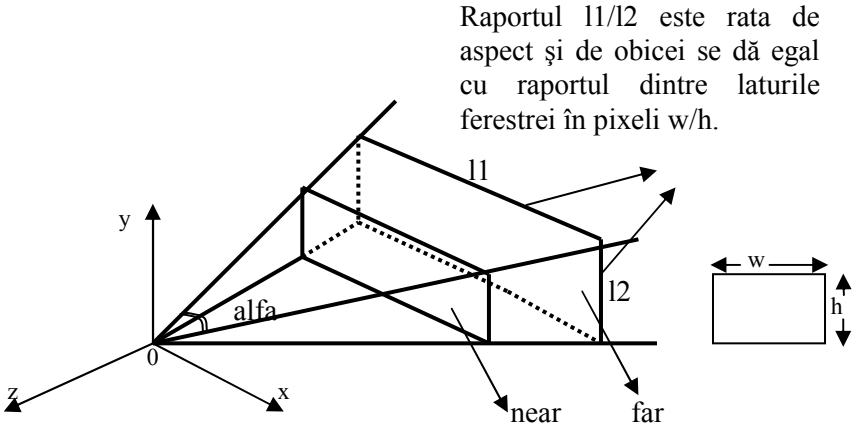
  unde:

  -alfa este unghiul din vârful piramidei de vizualizare;

  -$\frac{w}{h}$ reprezintă rata de aspect;

  -near indică planul apropiat;

  -far planul îndepărtat.



Valorile near şi far se dau întotdeauna pozitive. OpenGL le va negativa deoarece în OpenGL centrul proiecţiei este totdeauna în originea sistemului de coordonate şi volumul

de vizualizare se formează în semispaţiul cu `z` negativ. În schema de mai jos se poate observa cum arată volumul de vizualizare în cazul proiecţiei perspectivă.

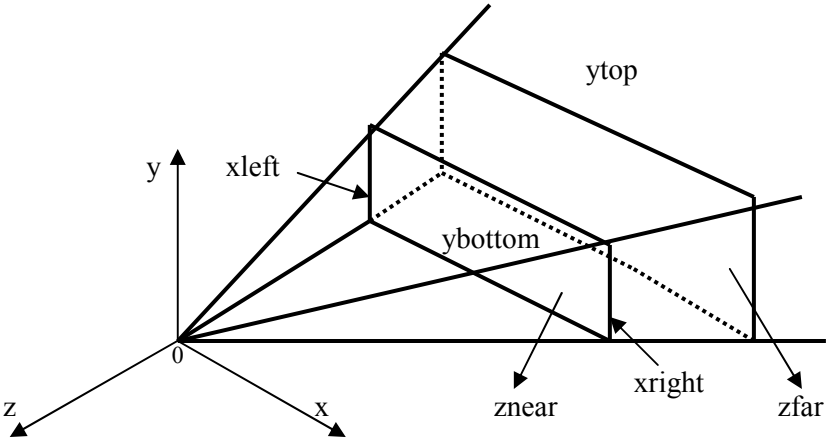
Exemplu:

```
gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
```

În urma acestui apel, OpenGL construieşte matricea de proiecţie care va fi folosită în fluxul OpenGL pentru transformarea coordonatelor 3D în coordonate 2D.

Transformarea de proiecţie se realizează parcurgând următorii paşi: se foloseşte `glMatrixMode()` cu argumentul `GL_PROJECTION`. Aceasta indică faptul că matricea curentă specifică transformarea de proiecţie. Operaţiile cu matrice care urmează vor afecta matricea de proiecţie. De obicei se apelează funcţia `glLoadIdentity()` pentru iniţializarea cu matricea identitate şi apoi una dintre matricele de proiecţie. Este apelată apoi din nou funcţia **`glMatrixMode()`** cu parametrul `GL_MODELVIEW`. Aceasta indică faptul că transformările următoare afectează matricea de modelare-vizualizare. O altă funcţie pentru specificarea proiecţiei perspective este funcţia **`glFrustum()`**.

- **`glFrustum`**`(xleft, xright, ybottom, ytop, znear, zfar);`

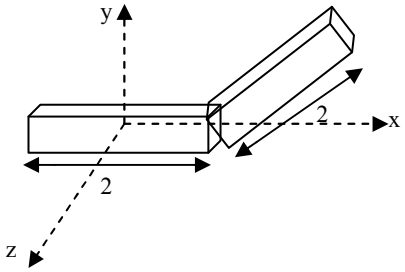


6.2 Desfăşurarea lucrării

Utilizând transformările geometrice corespunzătoare realizaţi următoarele aplicaţii:

Aplicatia 1.

Se dă sursa `robot.c` . În aplicaţie este realizată rotaţia



sus-jos a braţului şi antebraţului robotului. Se explică în continuare cum anume s-au realizat transformările.

Ordinea logică a transformărilor pentru rotaţia sus-jos în încheieturile umărului şi braţului este următoarea:

```
/* rotaţia braţului în umăr */
        3. Tx (1);          // translaţie cu o unitate pe
                               axa x a axei de rotaţie
                               astfel încât aceasta să se
                               suprapună peste axa z;
        2. Rz(umăr);        // rotaţie în umăr faţă de
                               axa z;
        1. Tx (-1);         // se duce axa de rotaţie în
                               poziţia originală;
        4. Desenare braţ;
/* rotaţia în încheietura umărului */
        7.
            7.1 Tx(2);      // se plasează antebraţul în
                               continuarea braţului;
            7.2 Tx(-1);     // se translatează axa de
                               rotaţie peste axa z;
        6. Rz(cot);         // rotaţie în încheietura
                               cotului faţă de axa z;
        5. Tx(1);           // se translatează axa de
                               rotaţie în poziţia originală;
        8. Desenare antebraţ.
```

Numerotarea conţine ordinea specificării operaţiilor în OpenGL, având în vedere ceea ce s-a explicat privind ordinea apelării transformărilor geometrice în OpenGL.

Să se urmărească pe cod implementarea aplicaţiei. În încheieturile umărului şi cotului să se **adauge** rotaţia faţă-spate a braţului respectiv a antebraţului.

Aplicaţia 2.

Se dă sursa `planet.c`.

Ordinea transformărilor în sursa originală:

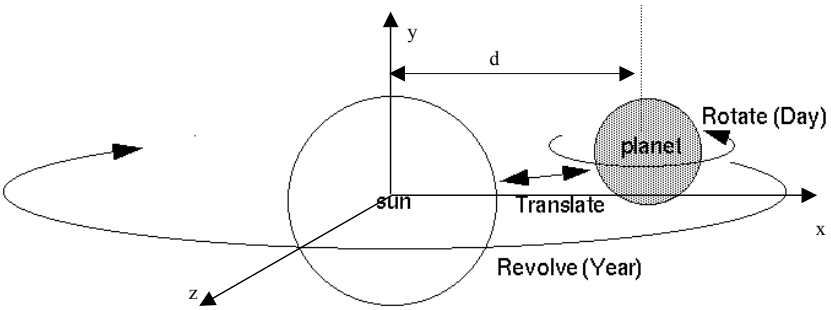
1. Desenează soare

Se anulează deoarece înmulţite dau I (matricea identitate).

Tx(d);	// se deplasează planeta la distanţa d faţă de Soare;
Tx(-d);	// se mută axa de rotaţie astfel încât să se suprapună peste axa y;
4. Ry(day)	// rotaţie diurnă faţă de axa y (rotaţia planetei în jurul axei proprii);
3. Tx (d)	// se mută axa de rotaţie în poziţia originală;
2. Ry(year)	// rotaţia planetei în jurul Soarelui;

5. Desenează planetă.

1. Să **se adauge** un satelit planetei, care se va roti în jurul planetei, în jurul axei proprii dar şi odată cu planeta în jurul Soarelui.
2. Să se rotească fiecare corp astfel încât axa polilor să fie paralelă cu axa y.
3. Să se adauge o rotaţie globală (care să afecteze întreaga scenă) cu 30 de grade faţă de axa care trece prin origine şi prin punctul (1,1,1).



De reţinut:

1. Funcţia de rotaţie **`glRotate()`** poate fi utilizată doar în cazul axelor care trec prin origine!
2. Pentru rotaţia faţă de axe care nu trec prin origine trebuie compuse transformările!
3. OpenGL foloseşte matricele transpuse şi în consecinţă ordinea transformărilor este inversată!

LABORATOR 7 - Ascunderea suprafeţelor. Eliminarea feţelor în funcţie de orientare. Animarea obiectelor. Reuniunea dintre două corpuri.

7.1 Ascunderea suprafeţelor

Obiectele 3D sunt ordonate în adâncime după z. Pentru a se reprezenta corect obiectele în funcţie de adâncime trebuie activat algoritmul de ascundere a suprafeţelor. OpenGL utilizează algoritmul z-buffer (metoda buffer-ului de adâncime). Pentru a se putea lucra cu algoritmul de ascundere:

1. Buffer-ul de cadru trebuie să conţină şi buffer-ul de adâncime. În funcţia **`auxInitDisplayMode()`**, la crearea ferestrei se va specifica şi opţiunea **`AUX_DEPTH`**.

2. Funcţia **`glEnable(GL_DEPTH_TEST)`** activează testul de adâncime.

3. La ştergerea buffer-elor aferente ferestrei se va şterge şi buffer-ul de adâncime.

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

Pentru vizualizarea modului de ascundere a suprafeţelor se dă sursa `TEST_ADANCIME.c`. Programul afişează două triunghiuri care se ascund reciproc. Să se analizeze poziţia în spaţiu a triunghiurilor, pe baza coordonatelor vârfurilor.

Aplicaţia 1. Să se facă reprezentarea fără activarea testului de adâncime. În acest caz să se schimbe ordinea specificării triunghiurilor. Mai contează ordinea specificării triunghiurilor în cazul în care se activează testul de adâncime?

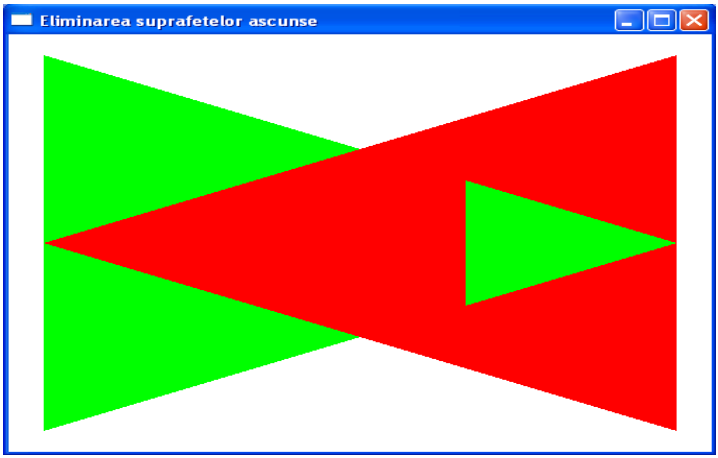


Fig. 1

7.2 Eliminarea feţelor în funcţie de orientare

OpenGL permite ca să fie eliminate (tăiate) feţele. Fiecare poligon are o faţă faţă (`FRONT`) şi o faţă spate (`BACK`) . În mod explicit, feţele `FRONT` sunt cele pentru care vârfurile sunt specificate în sens invers rotirii acelor ceasornicului. Orientarea implicită poate fi schimbată din `GL_CCW` (counter clock wise) în `GL_CW` (clock wise).

Orientarea implicită a vârfurilor se schimbă cu **`glFrontFace()`** având parametrii - `GL_CW` şi `GL_CCW`.

glFrontFace(GL_CW) // orientare GL_CW ca orientare directă;
glFrontFace(GL_CCW) // orientare GL_CCW ca orientare directă.

Pentru eliminarea feţelor se folosesc funcţiile:

glEnable(GL_CULL_FACE)
//activează eliminarea feţelor;
glCullFace(GL_BACK)
//sunt eliminate feţele spate;
glCullFace(GL_FRONT)
//sunt eliminate feţele faţă.

Aplicaţia 2. Se dă sursa `FETE_ELIMINATE.c`. Programul afişează suprafeţele fără activarea eliminării feţelor. Cu orientarea directă implicită se vor schimba feţele care se elimină, cele faţă sau cele spate, folosind funcţia **glCullFace()**.

Fig.2

Să se modifice sursa astfel încât să permită vizualizarea ambelor poligoane chiar dacă se activează tăierea feţelor cu funcţia **`glCullFace()`**. Nu se va modifica modelarea dreptunghiurilor.

7.3 Animarea obiectelor

Varianta 1:

Pentru rotirea unui obiect se înregistrează în sistemul de operare o funcţie de tip `CALLBACK` - **`IdleFunction`**`()` şi care va fi apelată de acesta când aplicaţia este în aşteptare.

```
auxIdleFunc(IdleFunction);
```

Funcţia **`IdleFunction`** se va programa. Un exemplu este dat în continuare.

```
void CALLBACK IdleFunction(void)

        {
        glRotatef(30,1,1,1);
        display();
        Sleep(300);
        }
```

Această variantă este cea recomandată pentru programarea animaţiei deoarece permite interacţiunea utilizatorului cu aplicaţia fără blocaje.

Varianta 2:

```
void CALLBACK display (void)
{
for( i=0;i<=500;i++)
     {
          glClear(GL_COLOR_BUFFER_BIT|
                    GL_DEPTH_BUFFER_BIT);
          glPushMatrix();
               glRotatef(i,1,1,1);
               auxWireCube(100);
          glPopMatrix();
          Sleep(70);
          glFlush();
     }
}
```

Varianta 2 nu este indicată deoarece blochează aplicaţia până la terminarea buclei.

Aplicaţia 3. Se dă sursa `cub.c`. Realizaţi următoarele :
- activaţi testul de ascundere;
- tăiaţi feţele faţă;
- descoperiţi eroarea de modelare a cubului şi corectaţi-o;
- faceţi o funcţie care să includă porţiunea de cod cu modelarea cubului;
- rotiţi cubul folosind ambele variante.

7.4 Reuniunea a două corpuri

Pentru a se realiza reuniunea dintre două corpuri care au zone comune în spaţiu, este suficient să se activeze ascunderea suprafeţelor.

Aplicaţia 4. În aplicaţia 3 se va adăuga şi o sferă care se suprapune peste cub. Se utilizează funcţiile **`auxWireSphere()`** sau **`auxSolidSphere(0.6)`** pentru desenarea sferei. Se activează algoritmul de ascundere. Se vor reprezenta obiectele fără activarea tăierii feţelor, cu activarea tăierii feţelor şi tăind feţele faţă. Se va vizualiza scena din spate. Pentru aceasta se va modifica comparaţia executată de testul de ascundere, utilizând funcţia **`glDepthFunc()`**. În mod implicit, comparaţia se face folosind **`GL_LESS`**, ceea ce înseamnă că vor trece testul de adâncime doar acele fragmente care au adâncimea mai mică decât adâncimea maximă (1).

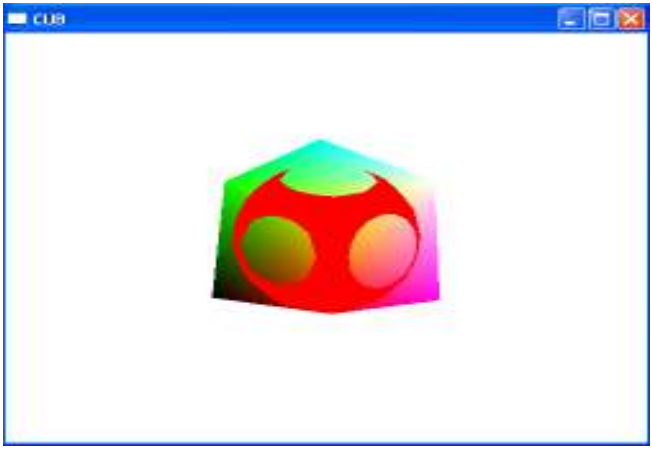


Fig. 3

Indicaţie: pentru ca imaginea să nu clipească se folosesc două buffere de culoare, unul pentru scriere şi unul pentru afişare în memoria video. În funcţia de iniţializare a ferestrei se va pune opţiunea `AUX_DOUBLE` în loc de `AUX_SINGLE`.

Pentru comutarea celor două buffere, se utilizează funcţia **`auxSwapBuffers(),`** la sfârşitul rutinei **`Display(),`** în locul funcţiei **`glFlush().`**

De reţinut:

1. La modelarea obiectelor este bine să fie utilizată aceeaşi orientarea a vârfurilor pentru toate feţele unui obiect.
2. Pentru activarea testului de ascundere trebuie respectaţi toţi cei trei paşi specificaţi.

LABORRATOR 8 - Decuparea faţă de un plan de decupare. Primitive raster în OpenGL. Liste de display. Fonturi în OpenGL.

8.1 Decuparea faţă de un plan de decupare

Un algoritm de decupare tridimensională identifică şi salvează pentru reprezentare toate obiectele aflate de o parte a planului de decupare. Aceste obiecte vor fi apoi proiectate pe planul de vizualizare.

Decuparea poate fi realizată extinzând algoritmii de decupare a liniilor sau a suprafeţelor. Se utilizează ecuaţia planului. În felul acesta se poate testa poziţia relativă a unei linii faţă de planul de decupare şi se pot localiza punctele de intersecţie.

Prin substituirea în ecuaţia planului a coordonatelor punctului extrem al unei drepte se poate verifica dacă acest punct se află în interiorul sau în exteriorul planului. Un punct `(x, y, z)` se află în exteriorul planului de parametrii `A, B, C, D` dacă se verifică inegalitatea:

$$A * x + B * y + C * z - D > 0.$$

În cazul în care punctul se află în interiorul planului se verifică inegalitatea:

$$A * x + B * y + C * z - D < 0.$$

În situaţia segmentelor de dreaptă pentru care ambele extremităţi se află în exteriorul planului, dreapta va fi eliminată. Dacă ambele extremităţi ale unui segment de dreaptă se află în semispaţiul interior, segmentul va fi salvat pentru proiecţie. Punctul de intersecţie al unei drepte cu un plan se determină ca fiind punctul de coordonate `(xl, yl, zl)` care verifică atât ecuaţia dreptei cât şi egalitatea următoare:

$$A * xl + B * yl + C * zl - D = 0.$$

OpenGL dispune de funcţii pentru specificarea parametrilor planului de decupare şi pentru activarea planului de decupare. La un moment dat pot fi activate cel mult 6 plane de decupare. Aceste plane de decupare sunt simbolizate prin constantele simbolice `GL_CLIP_PLANE0, GL_CLIP_PLANE1, etc.`

`glClipPlane` `(GL_CLIP_PLANE0, eqn0);` //se asociază planul `GL_CLIP_PLANE0` cu coeficienţii `A, B, C, D,` furnizaţi în tabloul `eqn0`.

`glEnable` `(GL_CLIP_PLANE0);`//se activează planul `GL_CLIP_PLANE0.`

Un plan de decupare împarte spaţiul în două semispaţii. Dacă ecuaţia `eqn0` conţine coeficienţii `A, B, C, D,` se vor vedea obiectele din unul din cele două semispaţii.

Fig.1

Dacă se înmulţesc coeficienţii cu `-1` se vor vedea obiectele din celălalt semispaţiu.

$$A * x + B * y + C * z - D = 0. \quad / (-1)$$
$$-A * x - B * y - C * z + D = 0.$$

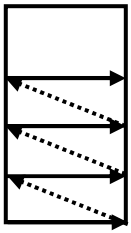
Aplicaţia 1. Se foloseşte sursa `Decupare.c`.

Să se decupeze sfera cu un plan orizontal care taie sfera în jumătate. Să se vizualizeze calota superioară apoi calota inferioară asferei.

8.2 Primitive raster

Până acum s-au utilizat doar primitivele vectoriale din OpenGL. OpenGL este însă o bibliotecă grafică ce acoperă atât necesităţile graficii vectoriale cât şi cele ale graficii punctuale. Noi vom prezenta doar câteva funcţii utilizate pentru reprezentarea bitmap-urilor monocrome, dar biblioteca este mult mai extinsă în funcţii pentru grafica punctuală. Bitmap-urile le vom utiliza pentru a putea afişa text în ferestrele OpenGL.

Într-un tablou de tip `Glubyte` sunt specificaţi octeţii în hexazecimal. Fiecărui pixel îi corespunde un bit.



Ordinea în care se specifică pixelii în tablou este cea din figură. Deci se începe cu pixelul din stânga-jos şi se continuă spre dreapta iar apoi se trece la rândul de deasupra.

Funcţia **`glRasterPos2i()`** furnizează poziţia punctului raster curent în unităţi logice. Acesta este punctul în care vrem să plasăm originea bitmap-ului.

Reprezentarea bitmap-ului se face cu funcţia **`glBitmap()`**. Parametrii funcţiei sunt:

- lăţimea şi înălţimea bitmap-ului în pixeli,
- poziţia originii,
- deplasarea poziţiei raster curente după desenarea bitmap-ului,
- tabloul care conţine imaginea bitmap-ului.

După desenarea bitmap-ului este actualizată poziţia raster curentă.

OBSERVAŢIE!

Tabloul trebuie să aibă un număr par de octeţi pe lăţime. Spre exemplu, dacă lăţimea bitmap-ului este de 13 pixeli se vor utiliza 2 octeţi pe lăţime. Biţii de la 14 la 16 vor fi complectaţi (padaţi) cu 0. În funcţia **`glBitmap()`** se vor specifica dimensiunile reale ale bitmap-ului dar tabloul va fi dimensionat cu un număr par de octeţi în care biţii ce nu aparţin bitmap-ului sunt complectaţi cu 0.

Aplicaţia 2.

În aplicaţia `Bitmap_E.c` este reprezentată o primitivă raster care modelează caracterul `E`. Să se reprezinte caracterul `A`.

Fig. 2

8.3 Liste de display

Listele de display permit mărirea vitezei de afişare în situaţia în care un obiect se afişează de mai multe ori într-o scenă. Apelurile OpenGL dintr-o listă de display vor fi salvate cu valoarea calculată a parametrilor, din momentul executării listei de display. La apelarea listei se vor folosi funcţiile cu parametrii calculaţi. Apelarea se face pe baza unui index (număr) asociat listei. Listele de display măresc viteza de afişare în situaţia în care se repetă afişarea aceloraşi obiecte. Listele de display nu sunt proceduri. Apelarea unei proceduri presupune executarea codului procedurii de fiecare dată, dar pentru parametrii diferiţi. Listele de display fac o preprocesare astfel încât anumite calcule de rasterizare nu mai sunt repetate de OpenGL la fiecare reprezentare a obiectului.

Aplicaţia 3. Aplicaţia `Liste1.c` arată modul de definire şi de executare al listelor de display. În aplicaţia `Liste2.c` se arată felul în care starea OpenGL modificată de liste nu afectează exteriorul listei. Să se identifice funcţiile pentru definirea listelor şi funcţiile pentru apelarea listelor de display.

Să se compare cele 2 aplicaţii.

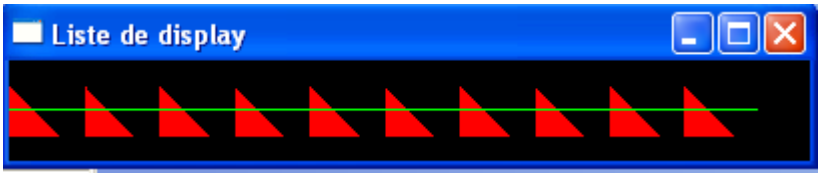


Fig. 3

8.4 Fonturi

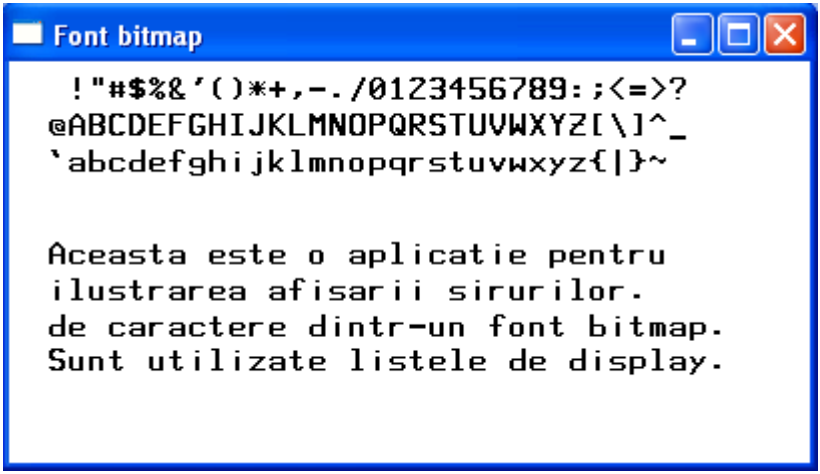


Fig. 4

LABORRATOR 8 - Decuparea faţă de un plan de decupare. Primitive raster în OpenGL. Liste de display. Fonturi în OpenGL.

Fontul este un set de caractere cu aspect grafic asemănător. Folosind listele de display şi primitivele raster, în aplicaţia `Font.c` s-a construit un font. S-a construit o listă de display pentru fiecare caracter. Indexul fiecărei liste s-a dat ca fiind chiar codul ASCII al caracterului respectiv. Aceasta a permis scrierea unei funcţii **`PrintString()`** pentru afişarea unui şir de caractere. Funcţia primeşte ca parametru şirul de afişat. Va apela toate listele de display care au indexul egal cu codul ASCII al caracterelor din şir.

Aplicaţia 4. Să se reprezinte o sferă. Sub sferă să se afişeze textul „Aceasta este o sferă”.

De reţinut:

1. În OpenGL singura posibilitate de a afişa un text este de a afişa fiecare caracter ca un bitmap.
2. OpenGL nu este doar o bibliotecă de grafică vectorială ci deţine şi un set de funcţii pentru grafica punctuală!

LABORATOR 9 - Iluminarea suprafeţelor

9.1 Suport teoretic

OpenGL permite iluminarea obiectelor într-o scenă pentru a crea diferite efecte. Suprafeţele, datorită fenomenului de reflexie difuză împrăştie lumina în mod egal în toate direcţiile. Acesta este motivul pentru care în cazul reflexiei difuze nu contează poziţia punctului de vizualizare. În plus, anumite materiale, în special cele foarte lucioase, prezintă şi fenomenul de reflexie speculară. În acest caz contează poziţia punctului de vizualizare. Reflexia speculară şi reflexia difuză diferă şi prin faptul că în primul caz lumina este reflectată în totalitate iar în al doilea caz, în funcţie de culoarea materialului, se absorb o parte din unde.

În OpenGL, deoarece se foloseşte modelul de culoare RGB, lumina este împărţită în trei componente: componenta roşie, componenta verde şi cea albastră. Astfel, culoarea sursei de lumină este caracterizată de „cantităţile" de lumină roşie, verde şi albastră pe care le emite, iar materialul suprafeţelor este caracterizat de procentajul componentelor roşu, verde şi albastru care sunt reflectate în diferite direcţii.

Intensitatea reflectată are trei componente:

- componenta de reflexie ambientală;
- componenta de reflexie difuză;
- componenta de reflexie speculară.

Lumina ambientală iluminează în mod egal toate obiectele din scenă. Ea este utilă atunci când nu există nici o sursă de lumină, sau atunci când anumite zone ale corpurilor nu sunt vizibile deoarece sunt ascunse faţă de sursa de lumină. Din cauză că lumina ambientală are aceeaşi intensitate în orice punct

al unui obiect (deci nu există dependenţă de geometria obiectului), obiectele iluminate doar cu lumină ambientală par plate.

Intensitatea luminoasă într-un punct se calculează după formula:

$$I_r = k_a \cdot I_a + \frac{1}{d + d_0}\left[k_{rd} \cdot I_d \left(\vec{N} \cdot \vec{L}\right) + k_s \cdot I_s \left(\vec{V} \cdot \vec{R}\right)^n\right]$$

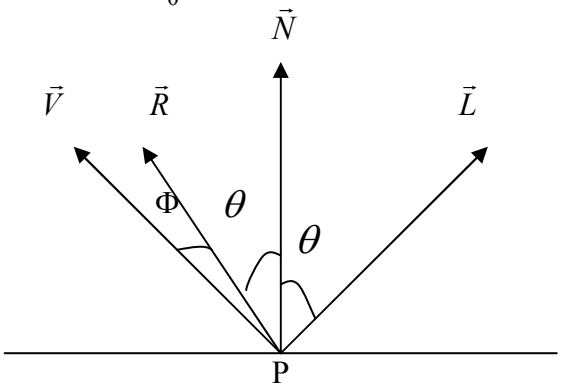


Fig. 1

unde : - I_r este intensitatea reflectată calculată într-un punct;

- I_a, I_{rd}, I_s reprezintă componentele intensităţii luminii ambientale, difuze şi speculare;

- $k_{a,} k_{rd}, k_s$ reprezintă coeficienţii de reflexie ambientală, difuză, speculară;

- d reprezintă distanţa punctului P faţă de sursă, şi se obţine din poziţia sursei (x, y, z, w); $w = 1$ indică sursă poziţională, $w = 0$ indică sursă direcţională aflată la infinit;

- $\vec{N}$ reprezintă perpendiculara în punctul P, pe suprafaţă;
- $\vec{L}$ reprezintă direcţia sursei de lumină;
- $\vec{R}$ reprezintă direcţia razei reflectate;
- $\vec{V}$ reprezintă direcţia punctului de vizualizare;
- n este o constantă de material şi are valori ridicate pentru suprafeţe lucioase şi valori mici pentru suprafeţe mate.

Primul termen al formulei modelează calculului intensităţii luminii ambientale. Se poate constata că nu depinde de geometria obiectului şi că are o valoare constantă pe suprafaţa aceluiaşi obiect.

Al doilea termen modelează intensitatea luminii difuze. Se poate constata dependenţa (prin normala N) de geometria obiectului. Reflexia difuză este cea care dă zone mai iluminate şi mai umbrite pe suprafaţa aceluiaşi obiect, zone care depind de poziţia sursei de lumină faţă de poziţia punctului în care se face calculul intensităţii luminoase.

Ultimul termen reprezintă reflexia speculară, cea care creează aspectul de obiect lucios respectiv mat.

Pentru o mai bună înţelegere a tuturor mărimilor necesare modelării iluminării unui obiect se dă tabelul:

	Material	Sursă	Modelare corp
GL_AMBIENT →	k_{aR}, k_{aG}, k_{aB}	I_{aR}, I_{aG}, I_{aB}	
GL DIFFUSE →	k_{dR}, k_{dG}, k_{dB}	I_{dR}, I_{dG}, I_{dB}, $\vec{L}$, d	$\vec{N}$
GL SPECULAR →	$k_{sR}, k_{sG}, k_{sB}, n$	I_{sR}, I_{sG}, I_{sB}, d	

Fig. 2

Parametrii din tabel pentru sursa de lumină se specifică prin funcţia **glLight()**. Parametrii de material din tabel se specifică prin funcţia **glMaterial().**

OpenGL are 8 surse de lumină, dintre care doar sursa 0 are setări implicite, celorlalte le dăm proprietăţile cu **glLight()**. În general, sursa de lumină nu se colorează ci doar materialul folosind funcţia **glMaterial()**. În general, culoarea obiectului se stabileşte din coeficienţii de reflexie difuză.

Activarea iluminării se face cu funcţiile:

```
glEnable(GL_LIGHTING); //activare iluminare
glEnable(GL_LIGHT 0); // activare sursa 0.
```

Când este activată iluminarea, colorarea obiectelor nu se mai poate face folosind funcţia **glColor()**. Pentru a putea furniza coeficientul de reflexie difuză folosind funcţia **glColor()**, se fac următoarele 2 setări:

```
- glColorMaterial(GL_FRONT,GL_DIFFUSE);
// funcţia glColor() furnizează coeficientul de
reflexie difuză
- glEnable(GL_COLOR_MATERIAL);
// se activează setarea culorilor prin glColor().
```

Pentru redarea iluminării, OpenGL are nevoie de normalele la suprafeţe pe care le furnizăm cu **glNormal3f**(x, y, z). Normalele se specifică pentru fiecare vârf în corpul **glBegin()/glEnd()** în care se modelează obiectele. În cazul în care utilizatorul modelează propriul obiect iluminat, de exemplu un triunghi, trebuie să se furnizeze înaintea fiecărui vârf normala asociată vârfului respectiv. Dacă se furnizează doar o normală înaintea tuturor funcţiilor **glVertex()** se consideră că toate vârfurile au

asociată aceeaşi normală. Dacă valorile normalei sunt furnizate aleator se poate ajunge la situaţia în care apare iluminată acea parte a obiectului care este opusă sursei de lumină. Pentru suprafeţele spline sau Beziér, OpenGL determină automat normalele dacă facem setarea:

```
glEnable(GL_AUTO_NORMAL);
```

9.2 Desfăşurarea lucrării

Aplicaţia 1.

Se dă sursa `teaamb.c`. Programul afişează trei ceşti de ceai, cu aceeaşi sursă de lumină, dar cu diferite valori pentru coeficienţii de reflexie pentru lumina ambientală.

Cerinţe:

- Să se elimine sursa şi să rămână doar iluminarea dată de lumina ambientală;
- Să se elimine lumina ambientală şi să se lase doar sursa;
- Să se modifice poziţia sursei de lumină;
- Să se modifice culoarea obiectelor prin modificarea coeficientului de reflexie difuză.

Fig. 3

Aplicaţia 2.

Se dă sursa `cone.c`. Programul demonstrează utilizarea iluminării în OpenGL. Se desenează un con şi o sferă cu material de culoarea gri. Obiectele sunt iluminate de o singură sursă de lumină.

Coloraţi conul şi sfera distinct.

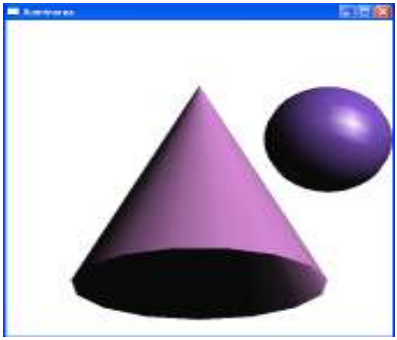


Fig. 4

Aplicaţia 3.

Se dă sursa `material.c`. Programul demonstrează utilizarea modelului de iluminare OpenGL pentru setarea caracteristicilor de material. Sunt desenate mai multe sfere cu caracteristici de material diferite. În prima coloană sferele nu au reflexie specular, în a doua coloană sferele au reflexie specular dar sunt mate, în a treia coloană este reflexie specular pentru materiale lucioase, iar în a patra coloană obiectele sunt emisive. De asemenea, în prima linie obiectele sunt iluminate doar de sursă, în a doua linie există pe lângă lumina sursei şi o lumină ambientală, iar în linia a treia lumina ambientală este colorată (nu este albă).

Să se urmărească setările de material corespunzătoare fiecărei sfere.

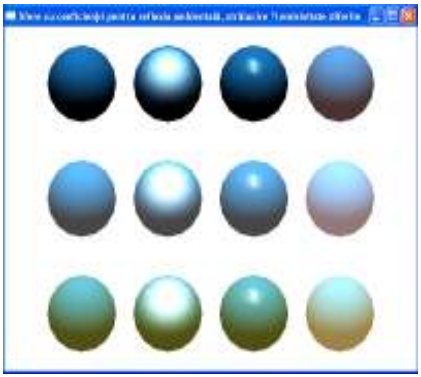


Fig. 5

Aplicaţia 4.

Se dă sursa `colormat.c`. După iniţializare programul va fi în modul ColorMaterial. Apăsarea tastelor R, G, B va conduce la modificarea culorilor pentru reflexia difuză prin modificarea parametrilor funcţiei **`glColor()`**. Fiind făcută setarea pentru furnizarea coeficienţilor de reflexie difuză prin **`glColor()`** culoarea se modifică deşi obiectul este iluminat.

În aplicaţia 2 să se facă modificările necesare astfel încât culoarea conului şi a sferei să poată fi modificată din funcţia **`glColor()`**.

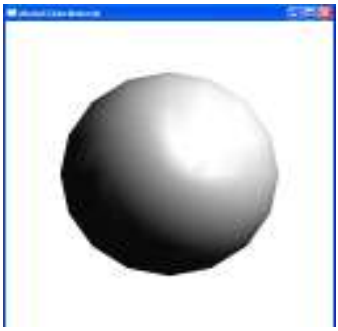


Fig. 6

Aplicaţia 5.

Se dă sursa `movelight.c`. Programul demonstrează modul de modificare a poziţiei sursei de lumină utilizând funcţiile de transformare geometrică (translaţie sau rotaţie). Poziţia sursei de lumină este setată cu funcţia **`glLight()`** după apelarea transformării geometrice. Transformările geometrice apelate înainte de setarea poziţiei sursei afectează poziţia sursei de lumină corespunzător. Poziţia de vizualizare nu este modificată. Se desenează un tor cu material de culoare gri. Obiectul este iluminat de o singură sursă de lumină.

Interacţiune: prin apăsarea butonului stâng al mouse-ului transformarea de modelare se modifică prin înmulţirea cu matricea de rotaţie cu 30 grade. Scena este redesenată cu torul în noua poziţie. Cubul reprezintă poziţia sursei de lumină. Rotaţia este izolată pentru tor, nu şi pentru cub deoarece cubul reprezentând sursa trebuie să fie afectat de modificarea poziţiei sursei de lumină. Să se rotească sursa în jurul axei y.

Aplicaţia 5'.

În aplicaţia 1 să se rotească sursa în jurul ceainicelor. Să se seteze sursa o dată ca sursă poziţională şi apoi ca sursă direcţională. Să se compare.

Fig. 7

Aplicaţia 6.

Iluminaţi cubul din laboratorul 5 prin determinarea în mod corespunzător a normalelor la suprafeţele acestuia. Normalele în vârfuri se determină ca medie aritmetică a normalelor poligoanelor care se întâlnesc în vârful respectiv.

$$N = \frac{N_1 + N_2 + N_3}{3}$$

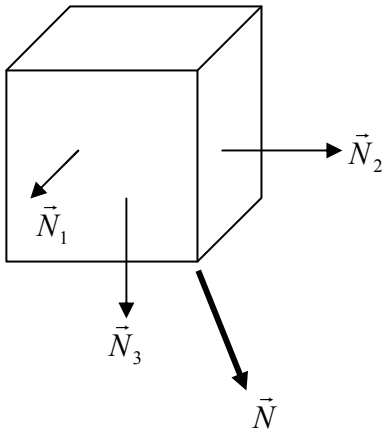


Fig. 8

Dacă sunt executate corect calculele, la modificarea poziţiei sursei, efectul trebuie să fie vizibil în felul în care este iluminat cubul.

De reţinut:

1. Funcţia **`glColor()`** nu mai are efect atunci când se activează iluminarea.
2. Intensitatea luminii ambientale trebuie setată la valori mici pentru a nu afecta rezultatul iluminării difuze!
3. Dacă nu se specifică normalele, obiectele sunt incorect iluminate.

LABORATOR 10 - Texturarea suprafeţelor

10.1 Texturarea

Texturarea suprafeţelor reprezintă un subiect complex în grafică şi permite reprezentarea realistă a suprafeţelor. Prin texturare se poate aplica unei suprafeţe un material care are aspectul materialului real. Pentru aceasta se foloseşte un bitmap care poate fi obţinut prin fotografierea suprafeţei reale urmate de scanarea pozei. Nu este însă vorba de o simplă aplicare a unui bitmap pe o suprafaţă rectangulară. Calculele de texturare permit aplicarea bitmap-ului pe orice formă de poligon dar şi pe suprafeţe curbate.

Există trei tipuri de texturare:

- texturare 1D
- texturare 2D
- texturare 3D.

Texturile sunt tablouri de date - de exemplu, culoare (RGB), intensitate (în cazul imaginilor monocrome), culoare şi transparenţă (RGBA). Aceste date sunt elemente de textură denumite `texeli`.

Un `texel` poate fi format din unul, două, trei, patru elemente în funcţie de textură (exemplu: pentru textura de tip RGB texel-ul are 3 elemente, pentru textura de tip RGBA sunt necesare 4 elemente, ultimul reprezentând transparenţa - alfa).

Un element de textură îl putem repera pe baza coordonatelor de texturare `(s, t, r, q)`.

Pentru texturarea unei suprafeţe se parcurg următorii *paşi*:

1. Se încarcă un tablou cu imaginea texturii, unidimensional sau bidimensional; acesta este fie generat intern, fie încărcat dintr-un fişier BMP. Componentele se memorează după regulile de la bitmap-uri.

2. Pe baza tabloului se construieşte textura în memoria OpenGL. În memoria OpenGL pot fi salvate mai multe texturi.

   glTextImage2D();
   glTextImage3D(), având ca parametrii dimensiunea texturii, tabloul cu texelii, nivelul de texturare, tipul texturii.

3. Caracterizarea texturii:

   - tipul de wrap-are (înfăşurare);
   - moduri de filtrare a texturii;
   - mipmaping;
   - modurile de texturare.

4. Activarea texturării:

```
glEnable(GL_TEXTURE_1D);
glEnable(GL_TEXTURE_2D);
glEnable(GL_TEXTURE_3D);
```

   Dacă avem o singură textură, aceasta este realizată în **myinit()**. Dacă sunt mai multe texturi, acestea sunt utilizate înaintea fiecărui obiect desenat (astfel se leagă textura cu obiectul).

5. Reprezentarea obiectelor:
   - dacă obiectele sunt modelate intern, atunci trebuie date coordonate de texturare asociate cu vertex-urile respective;
   - Se foloseşte funcţia **`glTexCoord`**`()`.
   - dacă obiectele sunt luate dintr-o bibliotecă (`GLAUX, GLUT, GLU`), atunci coordonatele de texturare sunt generate automat.

Este obligatoriu ca dimensiunile texturilor să fie puteri ale lui 2.:

$$width = 2^n + 2 \cdot b$$

$$height = 2^n$$

10.2 Wrapping

Wrapping-ul (înfăşurare) se referă la felul cum se plasează textura pe poligon, adică este întinsă pe tot poligonul, sau o aplicăm pe fiecare direcţie de 2, 3…n ori.

`GL_CLAMP` - texelii din marginea texturii sunt extinşi;
`GL_REPEAT` – se repetă texturarea.

Nu este suficient a se furniza modul de wrapping ci mai este necesară şi furnizarea în mod corespunzător a coordonatelor de texturare.

10.3 Modele de filtrare

În OpenGL există două filtre: filtrul de maximizare şi filtrul de minimizare.

Filtrele pot fi:

GL_NEAREST- se alege texel-ul cel mai apropiat pentru un anumit pixel;
GL_LINEAR- se face o interpolare liniară a texel-ilor.

Filtrele următoare se folosesc doar ca filtre de minimizare în modul mipmap:

```
GL_LINEAR_MIPMAP_NEAREST
GL_LINEAR_MIPMAP_LINEAR
GL_NEAREST_MIPMAP_LINEAR
GL_NEAREST_MIPMAP_NEAREST
```

Filtrele se specifică în felul următor:

– filtru de minimizare NEAREST:

```
glTexParameterf(GL_TEXTURE_2D,
   GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

– filtru de maximizare LINEAR:

```
glTexParameterf(GL_TEXTURE_1D,
   GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

10.4 Mipmap

Numărul de texeli nu este întotdeauna egal cu numărul de pixeli ai figurii care se texturează şi atunci furnizăm texturi pe mai multe nivele. Deci aceeaşi textură dar din ce în ce mai puţin detaliată. În felul acesta atunci când sunt reprezentate puncte îndepărtate ale unui obiect sunt alese nivele mai ridicate de texturare care nu au prea multe detalii.

Exemplu:
```
level 0   64x64
level 1   32x32
level 2   16x16
level 3   8x8
level 4   4x4
level 5   2x2
level 6   1x1
```

În **figura 1** este texturat un poligon folosind tehnica mipmaping. OpenGL delimitează zonele pentru aplicarea fiecărui nivel de texturare. Texturile pe fiecare nivel au în acest caz culori solide pentru a se vedea felul în care OpenGL a schimbat nivelul de texturare.

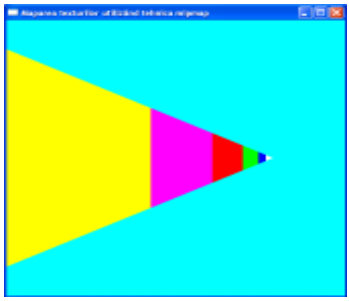


Fig. 1

10.5. Moduri de texturare

OpenGL dispune de trei moduri de texturare:

`GL_DECAL` – nu se ţine seama de iluminare şi de culoarea existentă în buffer-ul de culoare;

`GL_MODULATE` – se ţine cont de intensitate, culorile texelilor sunt alterate;

`GL_BLEND` – se amestecă culoarea din buffer cu cea a texturii.

10.6. Desfăşurarea lucrării

Aplicaţia 1. Aplicaţiile `GL_CLAMP`, `GL_CLAMP1`, `GL_CLAMP2`, `REPEAT` demonstrează modul de folosire al wrappingului. Să se urmărească în fiecare aplicaţie felul în care se setează modul de wrapping şi felul în care se dau coordonatele de texturare:

−Aplicaţia `checker.c`: mapează o imagine - tabla de şah - pe două dreptunghiuri. Acest program blochează (GL_CLAMP) texturarea dacă coordonatele texturii (s, t) depăşesc valorile 0 respectiv 1.

−Aplicaţia `chess.c`: reia programul `cheker.c` dar cu valori pentru coordonatele de texturare cuprinse între 0 şi 3 în condiţiile în care se păstrează parametrul GL_CLAMP.

−Aplicaţia `chess.c`: reia programul "checker.c", pe o direcţie se repetă texturarea iar pe cealaltă nu.

−Aplicaţia `checker2.c`: texturează cu imaginea unei table de şah două dreptunghiuri. Este repetată textura dacă coordonatele de texturare depăşesc domeniul 0 respectiv 1.

Aplicaţia 2. Programul `mipmap.c` demonstrează folosirea tehnicii mipmap. Să se urmărească modificările care apar în acest caz.

Aplicaţia 3. Programul `texgen.c.` mapează o textură pe un ceainic cu generarea automată a coordonatelor de texturare. Este construită o textură 1D cu 3 octeţi pentru fiecare texel, din 32 texeli 5 sunt roşii şi 27 sunt verzi. Textura este redată ca haşuri pe ceainic.

Se foloseşte modul de texturare `GL_MODULATE` astfel că ceainicul este şi iluminat şi texturat.

a) Să se schimbe orientarea dungilor pe ceainic schimbând planul faţă de care se direcţionează textura.
b) Să se modifice generarea texturii astfel încât aceasta să nu mai fie solidară cu obiectul.
c) Să se modifice culorile ceainicului.
d) Să se aplice o rotaţie ceainicului.
e) Să se înlocuiască ceainicul cu o sferă, un cub, etc.

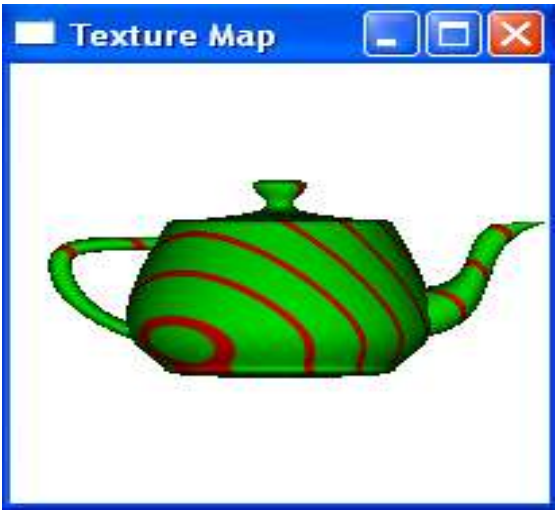


Fig. 2

Aplicaţia 4. În aplicaţia `incarcare_textura.c` să se urmărească felul în care se aplică texturile din fişiere *.bmp. Să se textureze ceainicul din aplicaţia 3 cu una din cele două texturi folosite în aplicaţia 4.

De reţinut:

1. Funcţia **`glTexCoord()`** este una din cele 4 funcţii OpenGL care au efect în corpul **`glBegin()/glEnd()`**. Celelalte 3, studiate până acum sunt: **`glVertex#()`**, **`glColor()`**, **`glNormal#()`**!
2. În cazul obiectelor cărora vrem să le aplicăm textură şi iluminare să nu se uite că obiectele trebuie să aibă definite normalele. Orientarea normalelor este dependentă de orientarea vârfurilor.
3. Reprezentarea obiectelor texturate consumă din puterea de calcul şi încetineşte afişarea scenelor.

LABORATOR 11 - Transparenţa

11.1 Suport teoretic

Până acum s-a utilizat în aplicaţii cea de a patra componentă de culoare `-alfa-` dar s-a dat de fiecare dată valoarea 1 şi nu s-a discutat în mod special felul în care această valoare poate fi utilizată. Valorile `alfa` sunt furnizate cu funcţia **`glColor#()`** care specifică culoarea curentă, cu funcţia **`glClearColor()`** care specifică culoarea de ştergere şi atunci când se specifică anumiţi parametrii de iluminare cum ar fi intensitatea surselor de lumină cu funcţia **`glLight#()`** sau proprietăţi de material cu funcţia **`glMaterial#()`**. Nu s-a arătat însă până acum care este felul în care valoarea `alfa` afectează ceea ce se desenează. Dacă este activat blending-ul (amestecare) atunci culoarea fragmentului care se procesează va fi combinată cu valoarea pixelilor memoraţi deja în buffer-ul de cadru. Combinarea apare după ce scena care trebuie reprezentată a fost rasterizată şi convertită în fragmente, dar înainte ca pixelii finali să fie înscrişi în buffer-ul de cadru. Valorile `alfa` pot fi de asemenea utilizate în testul `alfa` pentru a accepta sau respinge un fragment în funcţie de valoarea sa `alfa` (**testul alfa**).

Dacă nu se face operaţia de amestecare (blending) atunci fiecare nou fragment se suprascrie peste valorile culorii existente deja în buffer-ul de cadru, ca şi cum fragmentul ar fi opac. Folosind combinarea (blending) se poate controla cât anume din culoarea existentă se va combina cu noua valoare a fragmentului. În acest fel valoarea `alfa` poate fi utilizată pentru

a crea fragmente transparente, care lasă să se vadă ceva din ceea ce s-a memorat anterior pentru pixelul respectiv.

11.1.1 Factorii sursă şi destinaţie

Fragmentul şi pixelul au fiecare un factor care controlează contribuţia lor la culoarea finală a pixelului: factorul sursă, care este utilizat pentru a scala culoarea fragmentului care vine, şi factorul de amestecare destinaţie, care scalează pixelii citiţi din buffer-ul de cadru. Procesul de combinare a culorii fragmentului procesat (sursă) cu a culorii pixelului aflat deja în buffer-ul de cadru (destinaţie) se face în două etape (figura 1). Mai întâi trebuie specificat cum se calculează factorii sursă şi destinaţie. Aceşti factori au patru componente (pentru R,G,B,A) care sunt multiplicaţi cu fiecare componentă sursă şi destinaţie. Rezultatele multiplicărilor sunt adunate. Matematic această operaţie poate fi exprimată astfel:

```
   (SrRs+DrRd, SgGs+DgGd, SbBs+DbBd,
SaAs+DaAd)
```

unde:

cvadruplul `(Sr, Sg, Sb, Sa)` reprezintă factorii pentru sursă

cvadruplul `(Dr, Dg, Db, Da)` reprezintă factorii pentru destinaţie

cvadruplul `(Rs, Gs, Bs, As)` reprezintă culoarea fragmentului sursă

cvadruplul `(Rd, Gd, Bd, Ad)` reprezintă culoarea fragmentului destinaţie

Să vedem acum care sunt funcţiile care sunt utilizate pentru amestecarea culorilor. În primul rând trebuie activată amestecarea folosind:

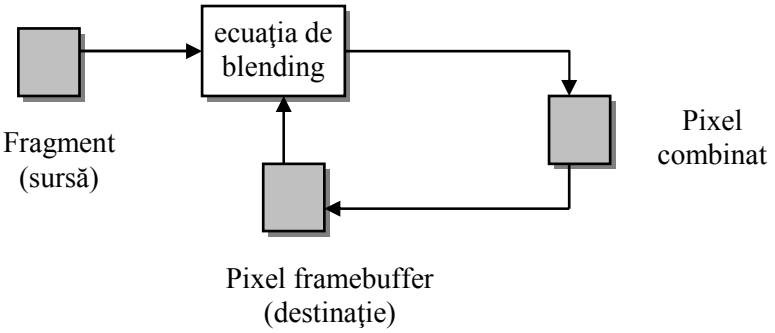
glEnable(GL_BLEND);

Dezactivarea se face cu:

glDisable(GL_BLEND).

Factorii pentru sursă şi pentru destinaţie sunt furnizaţi folosind funcţia **glBlendFunc()**:

void glBlendFunc(GLenum sfactor, GLenum dfactor);



Argumentul `sfactor` arată cum se calculează factorul de amestecare al sursei; argumentul `dfactor` arată cum se calculează factorul de amestecare al destinaţiei. Se impune ca factorii de amestecare să fie cuprinşi în intervalul [0, 1]. În acest fel după combinare culoarea fragmentului se încadrează în domeniul [0, 1]. În tabelul 1 se dau valorile care pot fi luate de argumentele `sfactor` şi `dfactor`.

Tabelul 1 - Factorii de amestecare sursă şi destinaţie

Parametrul	Relevanţa	Calcularea factorului de amestec
GL_ZERO	sursă sau destinaţie	(0, 0, 0, 0)
GL_ONE	sursă sau destinaţie	(1, 1, 1, 1)
GL_DST_COLOR	sursă	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destinaţie	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_ DST_ COLOR	sursă	(1, 1, 1, 1)- (R_d, G_d, B_d, A_d)
GL_ONE_MINUS_ SRC_COLOR	destinaţie	(1, 1, 1, 1)- (R_s, G_s, B_s, A_s)
GL_SRC_ALPHA	sursă sau destinaţie	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_ SRC_ALPHA	sursă sau destinaţie	(1, 1, 1, 1)- (A_s, A_s, A_s, A_s)
GL_DST_ALPHA	sursă sau destinaţie	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_ DST_ALPHA	sursă sau destinaţie	(1, 1, 1, 1)- (A_d, A_d, A_d, A_d)
GL_SRC_ALPHA_ SATURATE	sursă	(f, f, f, f); f=min(A_s, 1-A_d)

11.1.2 Exemple de utilizare a amestecului

1. O modalitate de a desena o imagine care compune jumătate dintr-o imagine şi jumătate din altă imagine, egal amestecate, este de a seta factorul sursă ca `GL_ONE` şi de a desena prima imagine, apoi a seta factorii sursă şi destinaţie ca `GL_SRC_ALPHA` şi a desena cea de a doua imagine cu valoarea alfa de 0.5. Dacă imaginea finală trebuie să amestece în proporţie de 0.75 prima imagine şi 0.25 cea de a

doua imagine, se va desena prima imagine ca mai înainte şi cea de a doua cu valoarea `alfa` de 0.25 dar cu factorul sursă setat `GL_SRC_ALPHA` şi cel destinaţie `GL_ONE_MINUS_SRC_ALPHA`. Această pereche de factori reprezintă probabil cea mai obişnuită utilizare a operaţiei de amestecare.

2. Pentru a amesteca în mod egal trei imagini, factorul destinaţie se setează ca `GL_ONE` şi factorul sursă ca `GL_SRC_ALPHA`. Se desenează fiecare dintre imagini cu o valoare `alfa` egală cu 0.3333333. Cu această tehnică, fiecare imagine are doar o treime din intensitatea originală, ceea ce este vizibil când imaginile nu se suprapun.

3. Să presupunem că se creează o aplicaţie de tipul aplicaţiei "Paint" din Windows şi se doreşte obţinerea unei pensule care adaugă în mod gradat culoare astfel că la fiecare atingere a pensulei se amestecă puţin mai multă culoare cu ceea ce există deja în imagine (de pildă, 10 procente culoare cu 90 procente din imagine la fiecare trecere). Pentru a se realiza aceasta se va desena imaginea pensulei cu procentul `alfa` de 10 şi se va utiliza factorul sursă `GL_SRC_ALPHA` şi factorul destinaţie `GL_ONE_MINUS_SRC_ALPHA` (se poate crea o pensulă care adaugă mai multă culoare în centrul ei şi mai puţină culoare în margine, folosind valori diferite ale lui `alfa`). Pentru a implementa o pensulă care şterge (gumă de şters) se va seta culoarea pensulei la culoarea fundalului.

4. Funcţiile de amestecare care utilizează culorile sursă şi destinaţie – `GL_DST_COLOR` sau `GL_ONE_MINUS_DST_COLOR` pentru factorul sursă şi `GL_SRC_COLOR` sau `GL_ONE_MINUS_SRC_COLOR`

pentru factorii destinaţie − permit în mod efectiv modularea fiecărei componente de culoare în mod individual. Această operaţie este echivalentă cu aplicarea unui filtru simplu − spre exemplu, multiplicarea componentei roşu cu 80 procente, a componentei verzi cu 40 procente, şi a componentei albastre cu 72 procente va simula vizualizarea scenei printr-un filtru fotografic care blochează 20% din lumina roşie, 60% din lumina verde şi 28% din lumina albastră.

5. Presupunem că doriţi o imagine compusă din suprafeţe transparente, care se ascund una pe cealaltă şi toate acoperind un fundal netransparent. Presupunem că suprafaţa cea mai din spate transmite 80% din lumina din spatele ei, următoarea transmite 40% şi cea mai apropiată transmite 90%. Pentru realizarea acestei imagini se desenează mai întâi fundalul cu factori impliciţi pentru sursă şi destinaţie iar apoi se modifică factorul sursă la `GL_SRC_ALPHA` şi cel destinaţie la `GL_ONE_MINUS_SRC_ALPHA`. Apoi se desenează suprafaţa transparentă cea mai îndepărtată cu valoarea alfa de 0.2, suprafaţa mijlocie cu valoarea 0.6 şi în final suprafaţa cea mai apropiată cu valoarea 0.1.

Aplicaţia 1.

În exemplul `amestec.c` se vor amesteca culorile roşu şi verde iar apoi verde şi roşu. Obiectele care vor primi aceste culori sunt nişte simple primitive OpenGL-dreptunghiuri. Imaginea finală va conţine patru pătrate diferit colorate. Pătratele din colţul stânga-jos şi dreapta-sus vor fi desenate de două ori, de fiecare dată cu alta culoare. Se vor folosi cele două culori specificate, roşu şi verde dar în ordine diferită. Alfa va avea de fiecare dată aceeaşi valoare şi anume 0.75. Factorii de

amestecare vor fi: pentru sursă `GL_SRC_ALPHA`, pentru destinaţie `GL_ONE_MINUS_SRC_ALPHA`.

Să se explice de ce diferă culorile pătratului stânga-jos de cel dreapta-sus. Să se calculeze conform formulei 1 care sunt culorile finale ale acestor pătrate?

Folosind această aplicaţie să se verifice efectul celorlalte combinaţii posibile exemplificate în tabel.

11.1.3 Eliminarea suprafeţelor ascunse şi transparenţa

Aşa cum s-a văzut din exemplul anterior, ordinea desenării obiectelor ale căror culori se amestecă are efect asupra culorii finale. Atunci când se desenează obiecte transparente, pot fi obţinute aspecte diferite în funcţie de desenarea obiectelor dinspre spate spre faţă sau dinspre faţă spre spate. Pentru determinarea ordinii corecte trebuie ţinut seama şi de buffer-ul de adâncime. Buffer-ul de adâncime (z-buffer-ul) este utilizat pentru eliminarea suprafeţelor ascunse. El memorează distanţa dintre punctul de vizualizare şi porţiunea ocupată de un anumit pixel; atunci când o altă culoare candidează pentru pixelul respectiv, va fi înscrisă doar dacă obiectul căreia îi aparţine este mai aproape de punctul de vizualizare, caz în care valoarea adâncimii va fi memorată în buffer-ul de adâncime. Dacă algoritmul buffer-ului de adâncime este activat, porţiunile ascunse ale suprafeţelor nu sunt desenate motiv pentru care nu sunt utilizate la amestecare.

În mod obişnuit, se doreşte ca să se redea atât obiecte opace cât şi obiecte transparente în aceeaşi scenă, şi se doreşte de asemenea ca algoritmul de ascundere a suprafeţelor să fie activat astfel încât obiectele aflate în spatele obiectelor opace să fie eliminate. Dacă un obiect opac ascunde fie un obiect transparent fie un obiect opac se doreşte ca obiectul aflat la distanţa cea mai mare să fie eliminat. Dacă însă un obiect transparent se află în

faţa altor obiecte se doreşte ca să se utilizeze amestecarea culorilor pentru simularea transparenţei. În general, pentru situaţia statică, chiar fără activarea algoritmului de ascundere, se poate stabili ordinea corectă de desenare ca aspectul să fie cel dorit. Dacă însă punctul de vizualizare îşi modifică poziţia sau dacă obiectele se mişcă problemele se complică.

Soluţia este de a se activa algoritmul de ascundere z-buffer şi buffer-ul de adâncime să fie făcut read-only atunci când se desenează obiectele transparente. Mai întâi se desenează toate obiectele opace, cu algoritmul z-buffer activat şi cu buffer-ul de adâncime în stare normală (poate fi scris/citit). Apoi valorile din z-buffer vor fi păstrate prin setarea buffer-ului de adâncime ca read-only. Se desenează apoi obiectele transparente. Deoarece algoritmul z-buffer este activat în continuare, dacă obiectele transparente se află în spatele obiectelor opace nu vor fi desenate, adâncimea lor fiind mai mare decât cea înscrisă în z-buffer. Dacă însă obiectele transparente se află în faţa obiectelor opace, ele nu vor elimina obiectele opace deoarece scrierea în z-buffer este blocată. Culoarea lor va fi însă amestecată cu a obiectelor opace, simulându-se astfel transparenţa.

Pentru a controla starea buffer-ului de adâncime se va utiliza funcţia **`glDepthMask()`**; dacă se transmite `GL_FALSE`, ca argument, buffer-ul z devine read-only iar dacă se transmite `GL_TRUE` se revine în starea normală.

Aplicaţia 2:

Să se deseneze două obiecte din biblioteca GLAUX (un cub şi o sferă) aflate unul în faţa celuilalt. Mai exact, cubul se află în faţa sferei. Sfera este opacă şi se desenează mai întâi, cu algoritmul z-buffer activat. Apoi se desenează cubul transparent, dar cu inactivarea scrierii în z-buffer. Culoarea cubului este amestecată cu cea a sferei. La apăsarea butonului stâng al mouse-ului se vizualizează scena din spate. În aceste condiţii, obiectul aflat mai în spate este cubul iar sfera se află mai în faţă.

LABORATOR 12 – Aplicaţie finală

Să se modeleze un leagăn. Se vor utiliza ca primitive cilindrul şi paralelipipedul. Se va folosi biblioteca GLAUX sau biblioteca GLU. Fiecare element al leagănului va fi colorat distinct. Scena va fi iluminată. Leagănul se va roti faţă-spate la apăsarea tastelor sau automat. Se va aplica scenei o rotaţie globală astfel încât scena să fie văzută de sus.

BIBLIOGRAFIE

1. Baciu, R., Volovici,D., *Sisteme de prelucrare grafică*, Editura Microinformatica, Cluj-Napoca 1999.
2. Baciu, R., *Programarea aplicaţiilor grafice 3D cu OpenGL*, Editura Albastră, Cluj-Napoca, 2005.
3. Ionescu, F., *Grafica în realitatea virtuală*, Editura Tehnică, Bucureşti 2000.
4. Neider, J., Davis, T., Woo, M., *OpenGL Programming Guide*, Addison-Wesley, Menlo Park, 1993.
5. OpenGL Architecture Rewiew Board, *OpenGL Reference Manual*, Addison-Wesley, Menlo Park, 1993.