UNIVERSITATEA DIN BACĂU
FACULTATEA DE INGINERIE

Culea George **Găbureanu Cătălin**

PRELUCRARE GRAFICĂ

Note de curs - laborator

Editura Alma Mater Bacău
2007

1 ELEMENTE INTRODUCTIVE

1.1 Sistemul grafic OpenGL

Grafica cu calculatorul (în special grafica 3D şi în particular grafica interactivă 3D) îşi caută propriul drum într-un număr în creştere de aplicaţii, de la programe de grafică simple pentru calculatoare personale până la software de modelare şi vizualizare sofisticat pentru staţii grafice şi supercalculatoare. Deoarece a crescut interesul pentru grafica cu calculatorul, a crescut de asemenea şi dorinţa de a fi capabil să scrii aplicaţii care pot rula pe o varietate de platforme cu un domeniu de capacităţi grafice. Un standard grafic uşurează această sarcină prin eliminarea nevoii de a scrie un driver grafic distinct pentru fiecare platformă pe care rulează aplicaţia.

Pentru a fi viabil, un standard grafic propus pentru aplicaţii interactive 3D trebuie să satisfacă câteva criterii. El trebuie să fie implementabil pe platforme cu capacităţi grafice diferite fără compromiterea performanţelor grafice. El trebuie să asigure o interfaţă naturală care permite unui programator să descrie operaţiile de redare într-un mod concis. în final, interfaţa trebuie să fie suficient de flexibilă pentru a găzdui extensii astfel încât dacă noi operaţii grafice devin semnificative sau disponibile în noile subsisteme grafice, aceste operaţii să poată fi asigurate fără fragmentarea interfeţei.

OpenGL (Open Graphics Library) îndeplineşte aceste criterii prin asigurarea unei interfeţe simple, directe pentru operaţiile fundamentale de redare grafică 3D. El permite primitive grafice de bază cum ar fi punctele, segmentele de dreaptă, poligoanele şi imaginile, precum şi operaţii de redare de bază cum ar fi transformările afine şi proiective şi calculele de iluminare. El permite de asemenea operaţii de redare avansată cum ar fi maparea texturilor şi redarea cu antialiasing.

OpenGL este o interfaţă software pentru plăcile grafice. OpenGL se bazează pe IrisGL dezvoltat de Silicon Graphics (SGI). Scopul iniţial al dezvoltării IrisGL a fost de a dezvolta o interfaţă programabilă pentru staţiile grafice SGI. Această interfaţă programabilă s-a intenţionat iniţial a fi independentă de hardware şi a îndeplini cerinţele speciale ale programării grafice 3D. Ulterior IrisGL a devenit interesantă şi pentru alte staţii.

În 1992 a luat fiinţă OpenGL Architecture Review Board (ARB) printre ai cărui membrii se află principalii producători de staţii grafice cum ar fi SGI, Sun, Hewlett-Packard, Microsoft, Evans&Sutherland, IBM, Intergraph. Site-ul ARB OpenGL poate fi găsit la www.opengl.org. OpenGL a devenit un standard industrial, disponibil din anul 1992, pe baza specificaţiilor realizate de acest consorţiu independent. Scopul ARB este de a controla dezvoltarea OpenGL, şi de a introduce noi funcţionalităţi în versiunile următoare. OpenGL evoluează în mod continuu permiţând ca inovaţiile la nivelul hardware-ului să fie accesibile dezvoltatorilor de aplicaţii prin mecanismul de extensii

OpenGL Adăugările la specificații (prin extensii) sunt bine controlate de consorțiul ARE şi actualizările propuse sunt anunțate din timp pentru a permite dezvoltatorilor să adopte modificările. Atunci când extensiile sunt larg acceptate, ele sunt luate în considerare pentru includerea în nucleul de bază OpenGL. Este asigurată compatibilitatea cu dezvoltările anterioare ale bibliotecii astfel că aplicațiile mai vechi vor rula pe acceleratoarele hardware cu drivere OpenGL mai noi.

Utilizatorii finali ai OpenGL, furnizorii independenți de software şi alți realizatori de aplicații bazate pe APLuI OpenGL pot utiliza biblioteca fără cerințe de licență.

OpenGL Performance Characterisation Committee, o altă organizație independentă, creează şi menține benchmark-urile OpenGL şi publică rezultatele acestor benchmark-uri pe site-ul www.specbenc.org/gpc/opc.static/index.htinl.

OpenGL este portabilă, fiind disponibilă pe o varietate de sisteme cum ar fi PC, Macintosh, Silicon Graphics, UNIX, Linux, Irix, Solaris, HP-UX OpenGL rulează pe fiecare din principalele sisteme de operare incluzând MacOS, OS/2, UNIX, Windows95, Windows NT, Linux, OPENStep, Python şi BeOS. Lucrează de asemenea cu fiecare sistem de ferestre principal, incluzând Presentation Manager, Win32 şi X/Window System. OpenGL este practic disponibilă pe aproape orice calculator care suportă un monitor grafic, o implementare OpenGL poate fi în mod eficient găzduită la aproape orice nivel al hardware-ului grafic, de la memoria video de bază la cele mai sofisticate subsisteme grafice. OpenGL poate fi apelat din limbajele de programare C, C++, Java, FORTRAN şi Ada şi oferă independență completă față de topologiile şi protocoalele de rețea.

OpenGL este de asemenea scalabilă deoarece poate rula pe o varietate de calculatoare, de la cele personale până la stații de lucru şi supercalculatoare Aceasta se realizează prin mecanismul OpenGL de cerere a capacităților hardware.

OpenGL este bine structurat având o arhitectură intuitivă şi comenzi logice (în număr de câteva sute). Utilizând comenzile OpenGL se pot scrie aplicații având câteva linii de cod spre deosebire de programele realizate utilizând alte biblioteci. Una din caracteristicile forte ale interfeței OpenGL este că interfața sa este uşor de utilizat de către începători fiind în acelaşi timp suficient de puternică pentru a satisface cerințele unor aplicații profesioniste indiferent că acestea sunt simulatoare de zbor, animații, aplicații de proiectare asistată sau vizualizări ştiințifice. Driverele OpenGL încapsulează informații despre substratul hardware, eliberând dezvoltatorul aplicației de necesitatea de a scrie aplicațiile pentru anumite caracteristici hardware.

1.2 Sisteme grafice pentru grafica 3D

Pentru grafica 3D sunt disponibile câteva sisteme. Un sistem relativ bine cunoscut este PHIGS (Programmer's Hierarchical Interactive Graphics System). Bazat pe GKS (Graphics Kernal Systems), PHIGS este standard ANSI. PHIGS asigură modalitatea de a manipula şi a desena obiecte 3D prin încapsularea descrierii obiectelor şi a atributelor într-o listă de display care este apoi referită când obiectul este afişat sau manipulat. Un avantaj al listei de display este că un obiect complex este descris doar o dată chiar dacă este afişat de mai multe ori. Aceasta este important în special dacă obiectul de afişat trebuie transmis de-

a lungul unui canal de bandă joasă (cum ar fi rețeaua). Un dezavantaj al listei de display este că poate cere un efort considerabil pentru respecificarea obiectului dacă el este modificat continuu ca rezultat al interacțiunii cu utilizatorul. O altă dificultate cu PHIGS este lipsa suportului caracteristic de redare avansată cum ar fi suportul pentru maparea texturilor.

PEX extinde sistemul de ferestre X (standard pentru stațiile UNIX) pentru a include posibilitatea de a manipula şi desena obiecte 3D (PEXlib este API care asigură protocolul PEX). Original bazată pe PHIGS, PEX permite redarea în mod imediat, ceea ce înseamnă că obiectele pot fi afişate după ce sunt descrise, fără a fi necesar ca mai întâi să se complecteze o listă de display. PEX de obicei nu permite caracteristici avansate de redare, şi este disponibil doar pentru utilizatorii X. în linii mari, metodele de descriere a obiectelor grafice, pentru a fi redate utilizând PEX sunt similare celor OpenGL.

Ca OpenGL şi PEXlib, RenderMan este o API care asigură o modalitate de redare a obiectelor geometrice. Spre deosebire de aceste interfețe, oricum, RenderMan asigură un limbaj de programare (denumit shading language) pentru descrierea felului cum aceste obiecte trebuie să apară Ia desenare. Această programabilitate permite generarea imaginilor care arată foarte real, dar este de nepracticat pentru implementarea celor mai multe acceleratoare grafice, făcând din RenderMan o alegere slabă pentru grafica interactivă 3D.

În final, există API-uri care asigură accesul la redarea 3D ca un rezultat al metodelor pentru descrierea obiectelor grafice de nivel ridicat. Liderii acestora sunt HOOPS şi IRIS Inventor. Obiectele asigurate de aceste interfețe sunt de obicei mai complexe decât primitivele geometrice simple descrise cu API-uri cum ar fi PEXlib sau OpenGL; ele pot comprima nu doar geometria dar de asemenea şi informații despre cum sunt ele desenate şi cum reacționează la intrările utilizatorului. HOOPS şi Inventor eliberează programatorul de descrierea plicticoasă a operațiilor de desenare individuale, dar accesul simplu la obiecte complexe în general însemnând pierderea controlului fin asupra redării (sau cel puțin făcând un asemenea control dificil). În orice caz, OpenGL poate asigura o bază bună pe care să se construiască asemenea API-uri de nivel ridicat.

Caracteristicile de standardizare, stabilitate, fiabilitate, portabilitate, uşurință în utilizare şi buna documentare fac din OpenGL un produs preferat înaintea altor biblioteci pentru realizarea vizualizărilor ştiințifice, a mediilor virtuale, aplicațiilor CAD/CAM/CAE, imagistică medicală, jocuri, etc.

Cei mai renumiți dezvoltatori de software utilizează OpenGL ca suport pentru realizarea unor API de nivel mai înalt. Spre exemplu, Openlnventor asigură o interfață cu utilizatorul care permite uşurința realizării aplicațiilor OpenGL. Iris Performer extinde funcționalitatea OpenGL şi permite caracteristici suplimentare create pentru cerințele unei rate de reîmprospătare a imaginii mai mari necesare în simulări vizuale şi realitatea virtuală. OpenGL Optimizer este un toolkit pentru interacțiunea, modificarea în timp real şi redarea unor modele complexe cum ar fi cele din aplicațiile CAD/CAM.

Lista aplicațiilor realizate având la bază OpenGL este mare şi ea cuprinde aplicații de modelare şi animație 3D (Maya, truSpace, 3D Studio Max, etc.), aplicații CAD/CAM (CATIA, 3D Studio Viz, Pro/ENGINEER, I-DEAS, etc), simulări vizuale şi realitate virtuală (Visualisation Data Explorer, WorldToolKit, Designer, Workbranch, etc.), playere VRML {Cosmo World, RenderSoft VRML Editor, etc.), jocuri (Quake2, X-Plane, Unreal, etc.).

Pentru a ține pasul cu inovările la nivelul hardware-ului, fără însă a forța programatorii să lucreze în limbaj de asamblare, soluția oferită de firmele din domeniul grafic este un limbaj de nivel înalt pentru OpenGL - OpenGL Shading Language (sau GLslang), independent de hardware, uşor de utilizat, suficient de puternic pentru a trece testul timpului şi care va reduce drastic nevoia de extensii. Continuând tradiția de compatibilitate "backwards", pe care au avut-o toate cele patru versiuni OpenGL, OpenGL 2.0 va fi un superset al lui OpenGL 1.4, astfel că aplicațiile mai vechi vor rula pe acceleratoarele hardware cu drivere OpenGL 2.0 fără modificări.

1.3 Caracteristici OpenGL

Deşi specificația OpenGL defineşte un anumit flux de procesare grafică, furnizorii de platforme au libertatea de a realiza o implementare OpenGL particulară pentru a îndeplini obiective de performanță şi de cost al sistemului unice. Apelurile individuale OpenGL pot fie executate de hardware dedicat, pot rula ca rutine software pe sisteme standard CPU, sau pot fi imlementate ca o combinație de rutine hardware şi software. Dezvoltatorii de aplicații au garanția unor rezultate de afişare consistente indiferent de implementarea OpenGL.

OpenGL permite dezvoltatorilor de software accesul la primitive geometrice şi imagine, liste de display, transformări de modelare, iluminare şi texturare, antialiasing, blending şi multe alte facilități.

Din punctul de vedere al programatorului, OpenGL reprezintă un set de comenzi care permit specificarea obiectelor geometrice în două sau trei dimensiuni, împreună cu comenzi care controlează felul în care aceste obiecte sunt rasterizate în buffer-ul cadru (framebuffer). Pentru cele mai multe din aceste comenzi, OpenGL asigură o interfață cu efect imediat, în sensul că specificarea unui obiect determină desenarea sa.

OpenGL conține o mare cantitate de informații de stare. Această stare controlează modul în care sunt desenate obiectele în framebuffer. OpenGL se află totdeauna într-o stare definită, setată prin variabile de condiție; aceasta înseamnă că OpenGL este o maşina de stare. Un exemplu de variabilă de condiție este culoarea curentă cu care sunt redate (desenate) primitivele individuale. Aceasta este setată utilizând comanda glcolor() şi apoi culoarea setată se aplică tuturor obiectelor care se desenează, până când este utilizată o nouă comandă de modificare a culorii. La un anumit moment, o parte din această stare - chiar şi conținutul unei texturi şi al memoriei video - este disponibilă în mod direct utilizatorului, care poate utiliza comenzi pentru a obține valori asociate cu diferite informații de stare. O parte a informațiilor de stare sunt vizibile, însă, doar prin efectul pe care îl au asupra a ceea ce se desenează.

OpenGL permite de asemenea aplicații de vizualizare cu imagini 2D tratate ca tipuri de primitive care pot fi manipulate la fel ca şi obiectele geometrice 3D. OpenGL dispune de unmai 10 primitive geometrice, şi orice obiect care se desenează în OpenGL este compus din aceste primitive. Setul de instrucțiuni din bibliotecă conține câteva sute de comenzi, toate fiind prefixate de gl. OpenGL asigură controlul direct asupra operațiilor fundamentale de grafică 3D şi 2D. Aceste operații includ specificarea unor

parametrii cum ar fi matricele transformărilor, coeficienții ecuațiilor de iluminare, operatori pentru actualizarea pixelilor. El nu asigură o modalitate pentru descrierea sau modelarea obiectelor geometrice complexe (cum ar fi cilindrul, cubul, sfera, etc.). Altfel spus, OpenGL asigură mecanismele pentru a descrie cum sunt redate obiectele geometrice complexe şi nu mecanismele de a descrie obiectele complexe însele.

Toate corpurile complexe trebuie să fie construite de dezvoltatorul aplicației 3D pe baza primitivelor simple - puncte, linii, poligoane. Pentru a simplifica puțin lucrurile pentru dezvoltatorii de aplicații, experții în grafica 3D au dezvoltat câteva biblioteci dintre care cele mai importante sunt GLU (OpenGL Utility Library), GLUT (OpenGL Utility Toolkit) sau echivalentul său Microsoft -GLAUX. GLU simplifică lucrurile pentru crearea calculelor de proiecție şi pentru construirea suprafețelor complexe, reprezentând printre altele curbe şi suprafețe NURBS (Non-uniform-rational-B-splines). GLUT este un utilitar independent de sistem pentru manevrarea în mod simplu a ferestrelor OpenGL şi pentru furnizarea dezvoltatorului de aplicații de rutine pentru controlarea evenimentelor externe provenite de la utilizator prin mouse sau tastatură.

Oricine dorește să devină expert în grafica 3D trebuie să se familiarizeze cu câteva noțiuni fundamentale de algebră şi geometrie analitică. Altfel utilizarea comenzilor OpenGL se face mecanic fără o profundă înțelegere a mecanismelor interne. Aceste noțiuni sunt calculul vectorial (produs scalar, produs vectorial), calcul matricial (înmulțirea matricelor, matrice identitate), transformări geometrice. Sunt de asemenea necesare câteva cunoştințe din domeniul opticii. Cei interesați pot să îşi reîmprospăteze aceste cunoştințe şi din cărți care prezintă fundamentele graficii cu calculatorul.

Un program tipic care utilizează OpenGL începe cu deschiderea unei ferestre în framebuffer-ul în care programul va desena. Apoi, se apelează funcții pentru alocarea unui context GL şi asocierea sa cu fereastra. Odată ce contextul OpenGL este alocat, programatorul este liber să dea comenzi OpenGL. Unele comenzi sunt utilizate pentru desenarea obiectelor geometrice simple (cum ar fi puncte, segmente de dreaptă şi poligoane), în timp ce altele au efect asupra redării acestor primitive inclusiv a felului cum sunt iluminate, colorate şi a modului în care spațiul modelului utilizatorului este mapat la ecranul bidimensional. Sunt şi comenzi care controlează efectiv framebuffer-ul, cum ar fi citirea şi scrierea pixelilor.

Din punctul de vedere al implementatorului, OpenGL este un set de comenzi care au efect asupra felului în care operează hardware-ul grafic. Dacă hardware-ul constă doar dintr-un framebuffer adresabil, atunci comenzile OpenGL trebuie să fie implementate în întregime software, de CPU-ul calculatorului gazdă. Tipice pentru acest moment sunt însă plăcile grafice care conțin acceleratoare grafice variind de la cele cu un subsistem de redare capabil să redea linii şi poligoane 2D până la procesoare în virgulă mobilă sofisticate capabile de transformări şi calcule asupra datelor geometrice. Sarcina implementatorului este de a asigura interfața software CPU astfel încât pentru fiecare comandă OpenGL să se dividă sarcinile între CPU şi placa grafică. Pentru a se obține un optim al performanței în executarea comenzilor OpenGL, această diviziune trebuie să fie în concordanță cu placa grafică disponibilă.

2 BAZELE PROGRAMĂRII ÎN OPENGL

2.1 Arhitectura OpenGL

OpenGL desenează primitive într-o memorie video, subiectul a numeroase moduri selectabile. O primitivă poate fi un punct, segment de dreaptă, poligon sau bitmap. Fiecare mod poate fi modificat independent; setarea unuia nu afectează setarea altora (deşi pot interacţiona în mai multe moduri pentru a determina ceea ce se produce în final în memoria video). Modurile sunt setate, primitivele specificate şi celelalte operaţii OpenGL sunt descrise prin intermediul comenzilor, în forma apelurilor de funcţii sau proceduri.

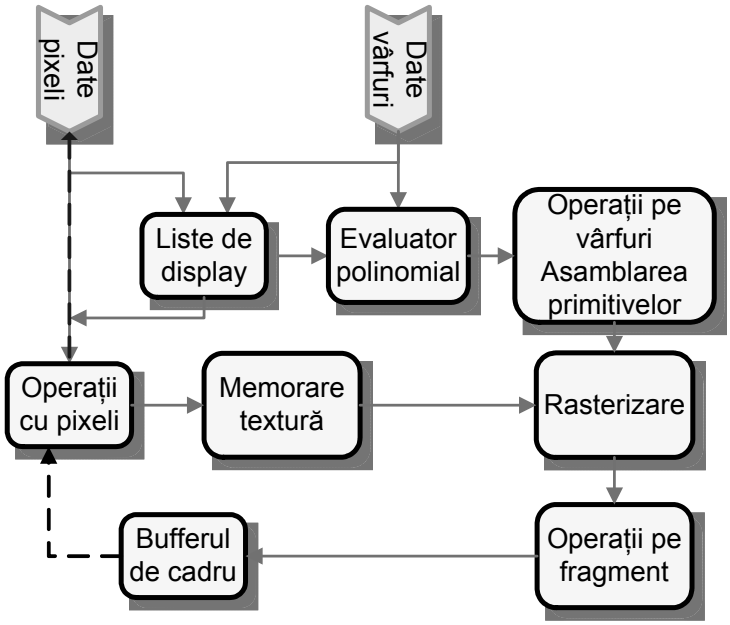


Figura 2.1 Schema fluxului de procesare OpenGL

Figura 2.1 arată schema fluxului de procesare OpenGL. Modelul de interpretare a comenzilor de către OpenGL este client-server. Aceasta înseamnă că programul (client) dă comenzile şi aceste comenzi sunt procesate şi interpretate de OpenGL (server). Server-ul poate sau nu să opereze pe acelaşi calculator cu client-ul.

Multe dintre comenzi pot fi acumulate în liste de display pentru o procesare ulterioară. În cazul în care se lucrează cu procesare imediată (deci fără liste de display) comenzile sunt transmise prin fluxul de procesare OpenGL. Comenzile nu sunt altceva decât apeluri de funcţii şi proceduri OpenGL.

Primul stadiu de procesare asigură o modalitate eficientă pentru aproximarea curbelor şi a suprafeţelor curbe prin evaluarea funcţiilor polinomiale ale valorilor de la intrare. Acest stadiu este parcurs doar de acele comenzi utilizate pentru reprezentarea curbelor şi a suprafeţelor Bezier şi spline. Următorul nivel operează asupra primitivelor geometrice descrise prin coordonatele vârfurilor: puncte, segmente de dreaptă şi poligoane. în acest stadiu vârfurile sunt transformate şi iluminate, şi primitivele sunt decupate faţă de volumul de vizualizare pentru a fi pregătite pentru nivelul următor - rasterizarea.

Rasterizarea converteşte o primitivă proiectată, scalată la viewport într-o serie de fragmente. Fiecare fragment comprimă pentru o locaţie a unui pixel din memoria video - culoarea, coordonatele de textură şi adâncimea (z). Rasterizarea produce o serie de adrese şi de valori pentru memoria video utilizând descrierea 2D a unui punct, segment de dreaptă, sau poligon. Fiecare fragment astfel produs alimentează nivelul următor care asigură operaţii asupra fragmentelor individuale înainte ca ele să modifice memoria video. Aceste operaţii includ actualizări condiţionale în memoria video pe baza noilor valori sau a valorilor de adâncime memorate anterior (pentru efectuarea testului de ascundere), amestecarea culorilor fragmentelor cu culorile memorate, precum şi mascarea şi celelalte operaţii logice asupra valorilor din fragment.

Când este rasterizat un segment de dreaptă sau un poligon, aceste date asociate sunt interpolate de-a lungul primitivei pentru a obţine o valoare pentru fiecare fragment.

Rasterizarea fiecărui tip de primitivă este controlată de un grup corespunzător de parametrii (atribute de redare a primitivelor). Un atribut de lăţime afectează rasterizarea punctului şi un altul afectează rasterizarea segmentelor de dreaptă. Suplimentar, se poate specifica o secvenţă stipple (stilul liniei -linie punctata, întreruptă, etc.) pentru segmentele de dreaptă, şi un model de haşură pentru poligoane.

Antialiasing-ul poate fi activat sau dezactivat individual pentru fiecare tip de primitivă. Când este activat, o valoare acoperitoare este calculată pentru fiecare fragment ce descrie porţiunea acelui fragment care este acoperit de primitiva proiectată. Această valoare acoperitoare este utilizată după ce texturarea a fost terminată pentru modificarea valorii alfa a fragmentului (în modul RGBA) sau valorii color index (în modul index).

Procesarea pixelilor şi a imaginilor trece peste secţiunea de procesare a vârfurilor din flux pentru a transmite un bloc de fragmente în mod direct prin blocul de rasterizare spre blocul operaţiilor pe fragmente individuale, determinând eventual ca un bloc de pixeli să fie scris direct în memoria video. Valorile pot fi de asemenea citite din memoria video sau copiate dintr-o porţiune a memoriei video în alta. Aceste transferuri pot include unele tipuri de decodificări şi codificări.

Se poate constata că există două fluxuri de date. Fluxul din partea de sus a schemei este pentru primitivele bazate pe vertex-uri. Fluxul din partea de jos este pentru primitive bazate pe pixeli - primitive imagine. Se poate spune că texturarea combină cele două tipuri de primitive.

Aşa cum s-a mai arătat, la modul general, sunt două operaţii principale care se pot face utilizând OpenGL:

- Se desenează ceva;
- Se modifică starea (aspectul) a ceea ce se desenează.

În ceea ce priveşte obiectele pe care le putem desena cu OpenGL, şi acestea sunt de două tipuri:

- Primitive geometrice;

- Primitive imagine.

Altfel spus, grafica pe care o putem realiza utilizând OpenGL este atât grafică vectorială cât şi grafică punctuală. Primitivele geometrice pe care le poate reda OpenGL sunt puncte, linii şi poligoane. Primitivele imagine sunt bitmap-uri şi imagini grafice (adică pixeli care se pot extrage dintr-o imagine JPEG după ce s-a citit această imagine în program). Suplimentar, OpenGL prin maparea texturilor uneşte primitivele geometrice cu cele imagine.

O altă operație comună care se face asupra ambelor tipuri de primitive este setarea stării. Setarea stării este procesul de inițializare al datelor interne, utilizate de OpenGL pentru redarea primitivelor. Setarea poate fi o operație simplă cum ar fi stabilirea dimensiunii şi culorii unui punct desenat dar şi o operație mai complicată cum ar fi inițializarea nivelelor multiple pentru maparea texturilor.

Trebuie subliniat că deşi primitivele geometrice pe care le poate reda OpenGL nu sunt spațiale, în sensul că ele pot fi redate şi în plan, totuşi OpenGL este o bibliotecă de grafică 3D. Ceea ce este deosebit în felul în care se desenează un punct într-o bibliotecă 2D, şi felul în care se desenează un punct într-o bibliotecă 3D sunt coordonatele acestui punct. Bibliotecii 2D i se furnizează coordonate 2D, pe când bibliotecii 3D i se furnizează coordonate 3D şi prin mecanismele proiecției se face transformarea din sistemul de coordonate 3D în 2D urmând ca apoi primitivele să fie redate pe dispozitivul de afişare.

Efectul comenzilor OpenGL asupra memoriei video este fundamental controlat de sistemul de ferestre care alocă resurse de memorie video. Sistemul de ferestre este cel care determină care porțiuni ale memoriei video pot fi accesate de OpenGL la un anumit moment de timp şi tot el este cel care îi comunică lui OpenGL cum sunt structurate acele porțiuni. în mod similar, afişarea conținutului memoriei video pe un tub CRT nu este controlată de OpenGL (incluzând transformarea valorilor individuale din memoria video prin asemenea tehnici, cum ar fi corecția gamma). Configurarea memoriei video are loc în exteriorul OpenGL, în conjuncție cu sistemul de ferestre; inițializarea unui context OpenGL are loc când sistemul de ferestre alocă o fereastră pentru redare OpenGL. Suplimentar, OpenGL nu are facilități de obținere a intrărilor de la utilizator, deoarece este de aşteptat ca sistemul de ferestre sub care rulează OpenGL să asigure asemenea facilități. Aceste considerente fac de fapt OpenGL independent de sistemul de ferestre.

2.2 Descriere generală OpenGL, GLU şi GLAUX

2.2.1 OpenGL Utility Library (GLU)

Un principiu de bază în proiectarea interfeței OpenGL a fost de a se asigura portabilitatea unui program fără a se specifica cât de înalt să fie nivelul la care pot fi descrise obiectele grafice. Ca rezultat, interfața de bază OpenGL nu permite redarea unor obiecte geometrice, care sunt asociate în mod tradițional cu standardele grafice. Spre exemplu, implementarea OpenGL nu redă poligoane concave. Un motiv ar fi că algoritmii pentru redarea (umplerea) poligoanelor concave sunt mai complecşi decât cei pentru redarea poligoanelor convexe în particular, dacă se redă un poligon concav mai mult de o dată este mai eficient să fie mai întâi descompus în poligoane convexe (sau triunghiuri) şi apoi să se deseneze poligoanele convexe.

Un algoritm de descompunere a poligoanelor concave este asigurat ca parte a bibliotecii GLU, care este asigurată pentru fiecare implementare OpenGL. GLU asigură de asemenea o interfață, care se bazează pe evaluatorii polinomiali OpenGL, pentru descrierea şi afişarea curbelor şi a suprafețelor NURBS (cu posibilitatea de segmentare spațială), precum şi o modalitate de reprezentare a cvadricelor (sferelor, conurilor, şi a cilindrilor). GLU este utilă atât pentru redarea unor obiecte geometrice utile cât şi pentru exemplificarea modelului de construire a unei biblioteci care se bazează pe OpenGL pentru redarea în memoria video.

Pentru a nu exista o imagine confuză asupra a ceea ce reprezintă OpenGL trebuie subliniat că OpenGL nu este un limbaj de programare ci aşa cum s-a mai arătat o API (interfață pentru programarea aplicațiilor). Atunci când ne referim la o aplicație OpenGL, înțelegem că aplicația respectivă a fost scrisă într-un limbaj de programare (cum ar fi C) şi că apelează funcții OpenGL. Nu este obligatoriu ca o aplicație să utilizeze doar pachetul de funcții OpenGL pentru desenare. Simultan pot fi utilizate mai multe biblioteci grafice. Spre exemplu se pot utiliza funcțiile OpenGL pentru partea de reprezentare 3D iar pentru partea de grafică ce ține de interfața aplicației să se utilizeze funcțiile grafice specifice mediului cu care se lucrează (spre exemplu interfața GDI a Windows-ului).

Ca orice API, funcțiile OpenGL pot fi apelate după convențiile de apel din C. Deci apelarea bibliotecii din limbajul C nu constituie o problemă. Apelarea din C++ a funcțiilor API, se face în acelaşi mod ca din limbajul C, cu considerații minore. Funcțiile OpenGL pot fi de asemenea apelate din limbaje de tipul Visual Basic - care pot apela funcții C.

Deşi OpenGL este o interfață API puternică ce conține peste 300 funcții ea nu are nici măcar o singură funcție pentru managementul ferestrelor şi a ecranului. De asemenea nu are funcții pentru manevrarea evenimentelor de la mouse şi tastatură. Motivul este clar - independența de platformă. Crearea şi deschiderea unei ferestre se realizează în mod diferit în diferitele sisteme de operare.

Pentru a putea lucra, programele OpenGL necesită însă o interfață grafică bazată pe ferestre. Deoarece OpenGL este independent de platformă, este nevoie de o modalitate de a integra OpenGL în fiecare sistem grafic bazat pe ferestre. Fiecare sistem grafic bazat pe ferestre care suportă OpenGL are funcții suplimentare API pentru controlarea ferestrelor OpenGL, manevrarea culorilor şi a altor caracteristici. Aceste API-uri suplimentare sunt dependente de platformă.

2.2.2 Biblioteci disponibile

Pentru crearea ferestrelor dar şi pentru alte operaţii necesare în realizarea aplicaţiilor grafice OpenGL, s-au creat biblioteci suplimentare cum ar fi GLAUX sau GLUT (OpenGL Utility Toolkit). Trebuie spus că funcţiile puse la dispoziţie de aceste două biblioteci sunt similare şi că un utilizator care a învăţat utilizarea uneia dintre aceste biblioteci va utiliza cu uşurinţă şi cealaltă bibliotecă.

Pentru simplitatea programelor care exemplifică OpenGL, noi vom utiliza biblioteca GLAUX, care simplifică interacţiunea cu sistemul de ferestre şi cu mouse-ul sau tastatura. Unul dintre motivele pentru care am ales această bibliotecă este faptul că programele sunt implementate în Visual C (versiunea 6.0) şi că acest mediu conţine biblioteca GLAUX precum şi o serie de aplicaţii pentru exemplificarea funcţiilor OpenGL care se bazează pe biblioteca GLAUX. GLAUX este o bibliotecă care face scrierea programelor OpenGL, în partea legată de sistemul de ferestre mult mai uşoară. OpenGL este independent de sistemul de ferestre şi de sistemul de operare. în felul acesta, partea din aplicaţia realizată cu OpenGL, care face redarea este de asemenea independentă de platformă. Oricum pentru ca OpenGL să poată face redarea are nevoie de o fereastră în care să deseneze. în general această fereastră este controlată de sistemul de operare bazat pe ferestre cu care se lucrează.

Pentru a putea integra OpenGL în diferitele sisteme grafice bazate pe ferestre sunt utilizate biblioteci suplimentare pentru modificarea unei ferestre native într-una capabilă OpenGL. Fiecare sistem de gestionare a ferestrelor are propria sa bibliotecă, unică şi funcţiile care fac acest lucru. Iată câteva astfel de biblioteci:

- GLX pentru sistemul X Windows, obişnuit pe platformele Unix;
- AGL pentru Apple Macintosh;
- WGL pentru Miocrosoft Windows.

Pentru a simplifica programarea şi dependenţa de sistemul de ferestre noi vom utiliza biblioteca GLAUX. GLAUX este un toolkit pentru realizarea simplă a aplicaţiilor OpenGL. Biblioteca GLAUX simplifică procesul creării ferestrelor, al lucrului cu evenimente în sistemul de ferestre şi manevrarea animaţilor.

În general aplicaţiile care necesită mai mult (adică butoane, meniuri, bare de scroll, etc.) în realizarea interfeţei cu utilizatorul, vor utiliza o bibliotecă proiectată pentru a realiza aceste caracteristici cum ar fi Motif sau Win32 API.

Aplicaţiile prototip sau cele care nu necesită toate caracteristicile unei interfeţe grafice cu utilizatorul complete, pot însă lucra cu GLAUX datorită modelului său de programare simplificat şi datorită independenţei de sistemul de ferestre.

2.3 GLAUX

Iniţial biblioteca GLAUX a fost creată ca un toolkit pentru a permite învăţarea OpenGL fără a intra în detaliile unui anumit sistem de operare sau de interfaţă cu utilizatorul., Pentru aceasta GLAUX conţine funcţii pentru crearea ferestrelor şi pentru citirea intrărilor de la mouse şi de la tastatură. Intern aceste funcţii utilizează funcţiile API ale mediului în care se lucrează. Modul de apelare al funcţiilor GLAUX rămâne însă acelaşi pentru toate platformele. Deşi are doar câteva funcţii pentru crearea ferestrelor,

biblioteca GLAUX scuteşte utilizatorul de sarcina de a utiliza funcţiile Windows API care realizează acest lucru. Deşi nu face parte din specificaţia OpenGL, biblioteca GLAUX este implementată pentru fiecare platformă pentru care se implementează OpenGL. Windows-ul nu face excepţie de la acest lucru, şi biblioteca GLAUX este inclusă free în Win32 SDK de la Microsoft. Dacă mediul de programare în care se lucrează nu conţine biblioteca GLAUX ea poate fi obţinută free de la MicrosoftWin32 SDK.

În afara funcţiilor pentru ferestre şi pentru manevrarea evenimentelor de la tastatură biblioteca GLAUX conţine o serie de funcţii pentru desenarea unor obiecte 3D: sfera, cubul, torul şi chiar un ceainic, etc.

Programele scrise cu GLAUX-ul pot fi mutate, prin recompilare pe diverse medii. Suplimentar acestor funcţii principale biblioteca GLAUX implementează câteva funcţii pentru a permite operaţii specifice sistemului cum ar fi inversarea buffer-elor şi încărcarea imaginilor. Utilizarea lor poate face însă programele neportabile.

Toate exemplele şi aplicaţiile din această carte sunt scrise în C. Pentru C, există câteva elemente necesare pe care trebuie să le facă o aplicaţie:

- Fişierele header aşa cum se ştie, conţin prototipurile tuturor funcţiilor apelate, parametrii acestora, definirea valorilor constante. Aplicaţiile OpenGL trebuie să includă fişierele header OpenGL, GLU şi GLAUX (gl.h, glu.h, glaux.h).
- Proiectele aplicaţiei trebuie să includă cele trei biblioteci care vor fi legate (la linkeditare) de aplicaţie (opengl32.1ib, glu32.1ib şi glaux.lib).
- Bibliotecile sunt dependente de implementarea OpenGL pentru sistemul de operare pe care se lucrează. Fiecare sistem de operare are propriile sale biblioteci. Pentru sistemul Unix, biblioteca

OpenGL este de obicei denumită libGL.so şi pentru Microsoft Windows este denumită opengl32. lib.

Funcţiile celor trei biblioteci pot fi recunoscute după prefixele lor: gl pentru OpenGL, glu pentru GLU şi aux pentru GLAUX. Tipurile enumerare (enumerated types) sunt definiţii OpenGL pentru tipurile de bază (adică float, double, int, etc.) utilizate de program pentru definirea variabilelor. Pentru a se simplifica independenţa de platformă a programelor OpenGL, se defineşte un set complet de tipuri enumerated types. Este indicat să se utilizeze aceste tipuri pentru a se simplifica transferarea programelor pe alte sisteme de operare.

În continuare se dă structura de bază care va fi utilizată într-o aplicaţie.

- Se configurează şi se deschide fereastra
- Se iniţializează starea OpenGL
- Se înregistrează funcţiile callback
    - Redare
    - Redimensionare
    - Intrări: tastatură, mouse, etc.
- Bucla de procesare a evenimentelor de intrare.

În general, aceştia sunt paşii într-o aplicaţie OpenGL, paşi pe care îi detaliem în continuare.

1) Se alege tipul de fereastră necesar pentru aplicaţie şi se iniţializează fereastra.

2) Se iniţializează starea OpenGL care nu este necesar a fi modificată în fiecare porţiune din program. Asemenea operaţii pot fi setarea culorii background-ului, poziţia surselor de lumină, setări pentru texturare.

3) Se înregistrează funcțiile callback utilizate de aplicații. Funcțiile callback sunt rutine scrise de programator pe care GLAUX-ul le apelează la apariția anumitor evenimente, cum ar fi reîmprospătarea ferestrei, mişcarea mouse-ului de către utilizator. Cea mai importantă funcție callback este cea de redare a scenei.

4) Se introduce bucla principală de procesare a evenimentelor. Aici aplicația recepționează evenimentele, şi se programează când anume sunt apelate funcțiile callback.

Exemplu

Structura unui program simplu se exemplifică în continuare;

Programul afişează un pătrat pe care îl translatează pe axa x la apăsarea săgeților stânga, dreapta.

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void myinit(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK MutaStanga(void);
void CALLBACK MutaDreapta(void);
static GLfloat x=0;
void myinit (void) {
   glClearColor(1.0, 1.0,1.0, 1.0);
}

void CALLBACK MutaStanga(void)
{
        x=x-10;
}
void CALLBACK MutaDreapta(void)
{
        x=x+10;
}
void CALLBACK display (void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   glLoadIdentity ();
        glTranslatef(x, 0.0, 0.0);
        glBegin(GL_QUADS);
                glBegin(GL_QUADS);
                glColor3f (1.0, 0.0, 0.0);
                glVertex2f(100.0,100.0);
```

```
                glColor3f (0.0, 1.0, 0.0);
                glVertex2f(150.0,100.0);
                glColor3f (0.0, 0.0, 1.0);
                glVertex2f(150.0,150.0);
                glColor3f (1.0, 1.0, 0.0);
                glVertex2f(100.0,150.0);
        glEnd();
  glFlush();
}

/*void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  if (!h) return;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho (-160.0, 160.0, -160.0,
    160.0, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
}*/

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  if (!h) return;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
  glOrtho (-160.0, 160.0, -160.0*(GLfloat)h/(GLfloat)w,
    160.0*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
  else
  glOrtho (-160.0*(GLfloat)w/(GLfloat)h,
    160.0*(GLfloat)w/(GLfloat)h, -160.0, 160.0, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
  auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
  auxInitPosition (0, 0, 300, 200);
  auxInitWindow ("Un patrat care se translateaza pe axa x");
  myinit ();
        auxKeyFunc (AUX_LEFT, MutaStanga);
        auxKeyFunc (AUX_RIGHT, MutaDreapta);
  auxReshapeFunc (myReshape);
  auxMainLoop(display);
  return(0);
```

```
}
```

Marea majoritate a programelor din această carte se bazează pe acest model. Pentru a obține fișierul executabil plecând de la această sursă, în mediul Visual C 6.0, parcurgeți pașii următori:

- Se editează programul.
- Se compilează programul (Build/ Compile). La compilare se va întreba dacă se doreşte crearea workspace-ului. Se va răspunde cu yes.

Figura 2.2

- Se introduce în proiect bibliotecile opengl32.1ib, glu32.1ib, glaux. lib (Project / Settings / Link / Project Options)
- Se construieşte fişierul executabil şi se rulează aplicația (Build/Execute...)

Programul afişează pe un background alb un pătrat care la apăsarea săgeților stânga/dreapta poate fi deplasat (figura 2.2).

Se poate remarca în codul aplicației utilizarea atât a funcțiilor GLAUX încep cu aux) cât şi a celor OpenGL (încep cu gl).

Win32 permite crearea unei ferestre grafice dintr-o aplicație în mod consolă. Aceste detalii sunt acoperite de biblioteca GLAUX, creată tocmai pentru a ascunde aceste detalii de platformă. Spre exemplu aplicația următoare nu face altceva decât să creeze o fereastră şi să aştepte apoi introducerea unei taste.

```
/*  Programul afişează o fereastră şi aşteaptă introducerea unei taste după care se închide fereastra */
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <conio.h>

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
```

```
    auxInitPosition (50, 50, 200, 200);
    auxInitWindow ("Deschiderea unei ferestre intr-o aplicatie consola");
        cprintf("Apasati o tasta pentru inchiderea ferestrei\n");
        getch();
    return(0);
}
```

Fereastra va fi afişată şi la introducerea unei taste în fereastra consolă se va închide aplicația (figura 2.3)

Figura 2.3

Să vedem cum este scrisă funcția principală a aplicației: main(). Funcțiile auxlnitDisplayMode(), auxinitPosition() şj auxInitWindow() fac parte din pasul de configurare a ferestrei. Folosind funcția auxinitDisplayMode() se specifică diverse informații privitoare la buffer-ele utilizate de fereastra aplicației - dacă se foloseşte modelul de culoare RGBA sau index, dacă se folosesc unul sau două buffer-e de culoare, dacă fereastra are sau nu asociate buffer-e de adâncime, buffer şablon sau/şi buffer de acumulare.

void auxInitDisplayMode(GLbitfield mask);

Argumentul mask este un SAU la nivel de bit între AUX_RGBA (modelul de culoare RGBA), AUX_INDEX (modelul de culoare index). AUX_SINGLE (un buffer de culoare), AUX_DOUBLE (două buffer-e de culoare). AUX_DEPTH (buffer de adâncime), AUX_STENCIL (buffer şablon). AUX_ACCUM (buffer de acumulare). Valorile implicite sunt pentru modelul de culoare index, cu un singur buffer de culoare.

În exemplul de mai sus, ferestrei i se va asocia un singur buffer de culoare şi modelul de culoare va fi RGB. Poziția ferestrei pe ecran: colțul stânga-sus al ferestrei va fi poziționat la pixelul (0,0) al ecranului. Dimensiunea ferestrei este de 300 pixeli pe lățime şi de 200 pixeli pe înălțime.

Prototipul funcției care specifică aceşti parametrii este;

auxInitPosition (GLint x, Glint y, Glsizei width, Glsizei heigth);

Se remarcă faptul că poziția colțului stânga-sus este specificată în coordonatele ecranului. Sistemul de coordonate al ecranului este figurat de cele două axe de coordonate. Funcția auxInitWindows() creează fereastra şi specifică titlul afişat pe bara de titlu. Dacă aplicația s-ar rezuma doar la funcția main() care ar include doar funcțiile comentate până în acest moment rezultatul ar fi afişarea pe ecran a unei ferestre de

culoare neagră care ar şi dispărea imediat fără a se mai întâmpla nimic altceva. în exemplu, se apelează în continuare funcţia myinit () care conţine setările proprii fiecărei aplicaţii, în general este vorba de acele setări care se fac o singură dată în program. în cazul programului nostru în funcţia myinit() s-a setat culoarea de ştergere a ecranului, ştergere care se realizează odată cu apelarea funcţiei glClear(GL_COLOR_BUFFER_BlT) din funcţia display. Următoarele 4 funcţii înregistrează rutinele callback, rutine care se definesc în cadrul programului. Este vorba de înregistrarea rutinelor MutaStanga(), MutaDreapta(), my/Reshape() şi display(). Funcţia auxMainLoop() pe lângă faptul că înregistrează funcţia callback display() este şi bucla de procesare a evenimentelor, care interpretează evenimentele şi apelează rutinele callback scrise de programator.

2.3.1 Funcţiile callback GLAUX

GLAUX-ul utilizează mecanismul callback pentru a realiza procesarea evenimentelor. Utilizând acest mecanism se simplifică procesarea evenimentelor pentru dezvoltatorul aplicaţiei. Comparând cu programarea tradiţională de manevrare a evenimentelor, în care autorul trebuia să recepţioneze şi să proceseze fiecare eveniment, şi să apeleze acţiunile necesare, mecanismul callback simplifică procesul prin definirea acţiunilor care sunt suportate, şi manevrarea automată a intrărilor dinspre utilizator. Tot ce trebuie să facă programatorul este să scrie codul pentru ceea ce se întâmplă când are loc evenimentul.

GLAUX permite mai multe tipuri de funcţii callback, incluzând:

- auxMainLoop() - apelată când trebuie să fie reîmprospătaţi pixelii din fereastră.
- auxReshapeFunc () - apelată când fereastra îşi modifică dimensiunea.
- auxKeyFunc () - apelată când se apasă o tastă la tastatură.
- auxMouseFunc () - apelată când utilizatorul apasă sau relaxează butonul mouse-ului
- auxidleFunc() - apelată când nu se întâmplă nimic altceva. Funcţia este foarte utilă în animaţii.

Funcţia callback pentru redare

În rutina principală a exemplului anterior apare următorul apel:

auxMainLoop(display);

Tot ceea ce se desenează se scrie în funcţia display () (în cazul nostru) înregistrată de funcţia auxMainLoop(). Funcţia callback display() este una dintre cele mai importante funcţii callback. Funcţia display() este apelată atunci când este necesar a se reîmprospăta conţinutul ferestrei. Acesta este motivul pentru care tot ceea ce se desenează trebuie programat aici. Pe scurt, funcţia display(), din aplicaţia noastră, redă un pătrat. Deoarece pentru fiecare vârf al pătratului s-a specificat altă culoare şi deoarece în mod implicit în OpenGL este setat modelul de umbrire Gouraud, care interpolează intensităţile vârfurilor, pătratul va fi desenat cu o combinaţie a culorilor specificate pentru fiecare vârf Se va studia fiecare funcţie OpenGL detaliat, în capitolele următoare.

Funcţia glFlush() determină ca orice funcţie OpenGL neexecutată până în acest moment să fie executată în momentul apelului glFlush. în această situaţie se află, în

cazul nostru, toate apelurile din display(). Intern OpenGL utilizează un flux de redare care procesează comenzile în mod secvențial. Deseori comenzile OpenGL sunt reținute până când server-ul OpenGL procesează mai multe comenzi deodată. Desenarea este accelerată deoarece hardware-ul grafic lent este deseori accesat mai puțin pentru un set dat de instrucțiuni de desenare.

Se va exemplifica funcția auxKeyFunc() care în funcția principală a programului a fost apelată sub forma:

auxKeyFunc (AUX_LEFT, MutaStanga);

Funcția auxKeyFunc() asociază apăsarea tastei - săgeată stânga - cu funcția MutaStanga(). Funcția MutaStanga() din această aplicație este:

void CALLBACK MutaStanga(void)

{ x=x-10;)

Rezultatul este că la apăsarea tastei <—, variabila x va fi modificată, în consecință, se va redesena fereastra iar funcția de translație glTranslatef() din display() va determina deplasarea pătratului spre stânga cu 10 unități.

Exemplul de mai sus arată o modalitate de tratare a evenimentelor de la utilizator. Cele două proceduri înregistrate în acest caz tratează intrările de la tastatură: săgeată stânga ←, săgeată dreapta →. GLAUX permite intrări de la utilizator prin mai multe dispozitive de intrare cum ar fi tastatura, mouse-ul, etc.

Funcția pentru redimensionarea ferestrei

Funcția callback myReshape() este apelată atunci când se redimensionează fereastra. Funcția este înregistrată de funcția:

auxReshapeFunc (myReshape);

Să presupunem că nu am include în aplicație aceste funcții. Rezultatul este arătat în figura 2.6. în figura 2.7 s-a redimensionat fereastra. Se poate însă constata că pozițiile în pixeli ale pătratului nu s-au modificat. Trebuie specificat că în fereastra OpenGL pixelul (0, 0) se află în colțul stânga jos al ferestrei şi orientarea axelor este cea din figura 2.4, în cazul în care nu se defineşte funcția MyReshape().

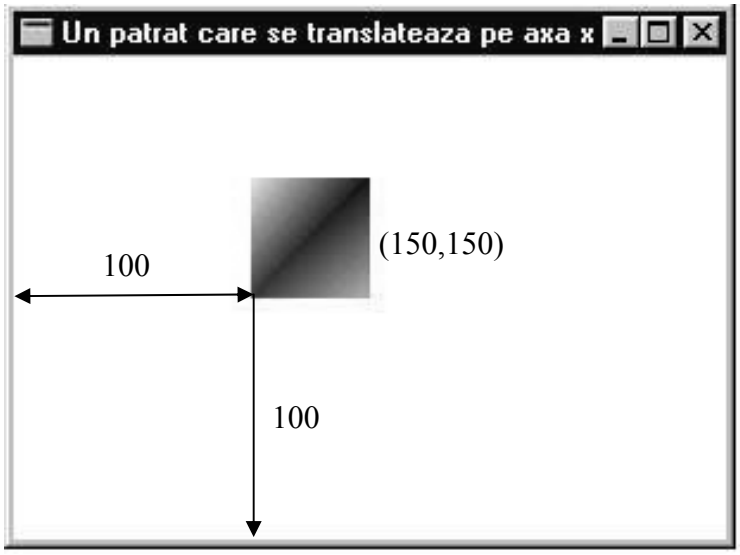


Figura 2.4 Un pătrat care se translatează pe axa x

Rolul funcției myReshape este pentru redimensionarea ferestrei. La aplicarea funcției conținutul ferestrei este redesenat, luând în considerare noile dimensiuni. Sunt cazuri în care la redimensionarea ferestrei se doreşte ca desenul să nu-şi modifice dimensiunile ci să fie decupat în cazul micşorării ferestrei şi afişat la dimensiunea reală în cazul lărgirii acesteia. în alte cazuri, cum este de altfel şi cazul aplicației noastre ne dorim ca la micşorarea ferestrei să fie proporțional modificate şi dimensiunile desenului.

Funcția auxIdIeFunc

Biblioteca GLAUX asigură o funcție care permite realizarea animaților. Această funcție, auxidleFunc(ldleFunction) înregistrează funcția idleFunction() care este apelată în mod continuu în timp ce programul este în aşteptare, mai puțin în situațiile în care fereastra este mutată sau redimensionată. Iată un exemplu de scriere a funcției idleFunction().

```
void CALLBACK IdleFunction (void)
{     t+=dt;
display(); }
```

Funcția actualizează o variabilă şi apoi apelează funcția de desenare. Animația necesită abilitatea de a desena o secvență de imagini. Funcția auxidleFunc() este mecanismul pentru realizarea animaților. Programatorul înregistrează prin această funcție o rutină care actualizează variabilele de mişcare (de obicei variabile globale care controlează cum se mişcă obiectele) şi apoi cere ca scena să fie actualizată. Funcția IdleFunction() cere ca funcția înregistrată prin auxMainLoop()-display(), să fie apelată cât de repede posibil. Aceasta este de preferat apelării în mod direct a rutinei de redare, deoarece este posibil ca utilizatorul aplicației să fi interacționat cu aplicația şi să fie necesar ca să fie procesate evenimentele de intrare.

Exemplu:

Exemplul următor (figura 2.5) arată un exemplu de utilizare a funcției auxidleFunc().

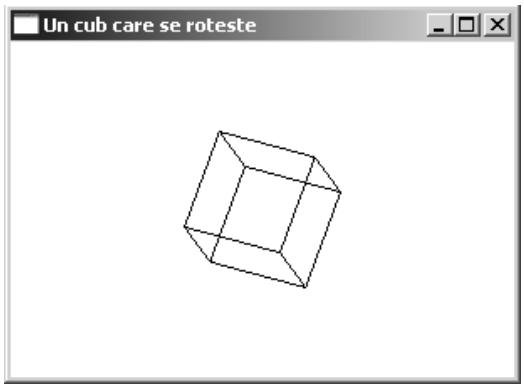


Figura 2.5

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
```

```
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK IdleFunction(void);

void myinit (void) {
   glClearColor(1.0, 1.0,1.0, 1.0);
         glColor3f(0.0,0.0,0.0);
}

void CALLBACK display (void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   auxWireCube(100);
         glFlush();
}

void CALLBACK IdleFunction(void)
{
         glRotatef(30,1,1,1);
         display();
         Sleep(300);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho (-160.0, 160.0, -160.0*(GLfloat)h/(GLfloat)w,
      160.0*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
   glOrtho (-160.0*(GLfloat)w/(GLfloat)h,
      160.0*(GLfloat)w/(GLfloat)h, -160.0, 160.0, -80.0, 80.0);
   glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (0, 0, 300, 200);
   auxInitWindow ("Un cub care se roteste");
   myinit ();
```

```
        auxReshapeFunc (myReshape);
        auxIdleFunc(IdleFunction);
    auxMainLoop(display);
    return(0);
}
```

Funcții GLAUX pentru desenarea primitivelor 3D

În exemplul anterior s-a constatat că s-a utilizat o funcție GLAUX pentru desenarea unui cub wireframe având latura de 100 de unități - auxWireCube(100). Dimensiunea unei unități în pixeli fiind cea specificată în funcția myReshape(). Biblioteca GLAUX dispune de funcții pentru desenarea şi a altor corpuri 3D. În continuare vom enumera aceste funcții.

- void auxSolidBox(Gldouble width, Gldouble height, Gldouble depth); permite desenarea unui paralelipiped, centrat în origine, pentru care se specifică lățimea, înălțimea şi adâncimea. Paralelipipedul are atribut de umplere.
- void auxWireBox(Gldouble width, Gldouble height, Gldouble depth); similar cu auxSolidBox() dar wireframe (doar cu atribut de contur).
- void auxSolidCube(Gldouble width); permite desenarea unui cub, centrat în origine, pentru care se specifică latura. Cubul are atribut de umplere.
- void auxWireCube(Gldouble width); similar cu auxSolidCube() dar wireframe (doar cu atribut de contur).
- void auxSolidTetrahedron(Gldouble radius); permite desenarea unui tetraedru (poliedru cu 4 fețe, fețele sunt triunghiulare), centrat în origine, pentru care se specifică raza. Corpul are atribut de umplere.
- void auxWireTetrahedron(Gldouble radius); similar cu auxSolidTetrahedron() dar wireframe (doar cu atribut de contur).
- void auxSolidOctahedron(Gldouble radius); permite desenarea unui octaedru (poliedru cu 8 fețe, fețele sunt triunghiulare), centrat în origine, pentru care se specifică raza. Corpul are atribut de umplere.
- void auxWireOctahedron (Gldouble radius); similar cu auxSolidOctahedron() dar wireframe (doar cu atribut de contur).
- void auxSolidDodecahedron(Gldouble radius); permite desenarea unui dodecaedru (poliedru cu 12 fețe, fețele sunt pentagonale), centrat în origine, pentru care se specifică raza. Corpul are atribut de umplere.
- void auxWireDodecahedron(Gldouble radius); similar cu auxSolidDodecahedron() dar wireframe (doar cu atribut de contur).
- void auxSolidDodecahedron (Gldouble radius); permite desenarea unui icosaedru (poliedru cu 20 fețe, fețele sunt triunghiulare), centrat în origine, pentru care se specifică raza. Corpul are atribut de umplere.
- void auxWireIcosahedron(Gldouble radius); similar cu auxSolidicosahedron() dar wireframe (doar cu atribut de contur).
- void auxSolidCylinder{Gldoubble radius, Gldouble height); permite desenarea unui cilindru, centrat în origine, pentru care se specifică raza bazei şi înălțimea. Cilindrul are atribut de umplere.
- void auxWireCylinder(Gldouble radius, Gldouble height); similar cu auxSolidCylinder() dar wireframe (doar cu atribut de contur).

- void auxSolidCone(Gldouble radius, Gldouble height); permite desenarea unui con, centrat în origine, pentru care se specifică raza bazei şi înălțimea. Conul are atribut de umplere.
- void auxWireCone(Gldouble radius, Gldouble height); similar cu auxSolidConeOdar wireframe (doar cu atribut de contur).
- void auxSolidSphere(Gldouble radius); permite desenarea unei sfere, centrată în origine, pentru care se specifică raza. Corpul are atribut de umplere.
- void auxWireSphere(GldoublG radius); similar cu auxSoildSphere() dar wireframe (doar cu atribut de contur).
- void auxSolidTorus{Gldouble InnerRadius, Gldouble outerRadius); permite desenarea unui tor (forma colacului), centrat în origine. Parametrul innerRadius este raza secțiunii prin tor, iar outerRadius este raza găurii din centrul torului. Corpul are atribut de umplere.
- void auxWireTorus(Gldouble innerRadius, Gldouble outerRadius); similar cu auxSolidTorus() dar wireframe (doar cu atribut de contur).
- void auxSolidTeapot(Gldouble size); permite desenarea unui ceainic, centrat în origine, pentru care se specifică dimensiunea (aproximativ diametrul). Corpul are atribut de umplere.
- void auxWireTeapot (Gldouble size); similar cu auxSolidTorus() dar wireframe (doar cu atribut de contur).

Exemplu:

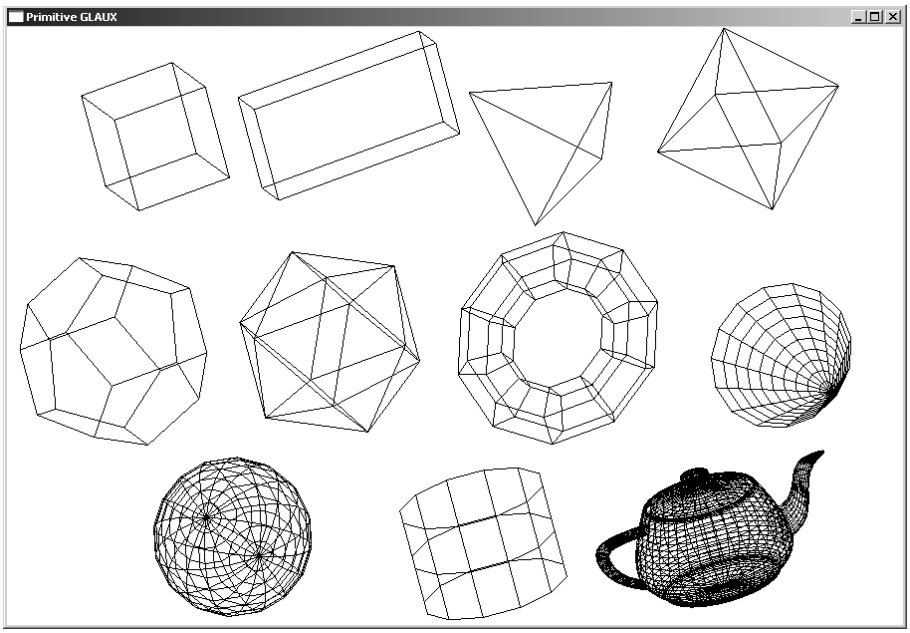


Figura 2.6 (cub, paralelipiped, tetraedru, octaedru, dodecaedru, icosaedru, tor, con, sferă, cilindru, ceainic - reprezentate wireframe)

Programul următor reprezintă wireframe toate primitivele enumerate mai sus (figura 2.6). Pentru poziționarea corpurilor în cadrul ferestrei, înaintea apelării fiecărei

funcţii de desenare s-a apelat o funcţie de translatare. Iniţial s-a aplicat o rotaţie care are efect asupra tuturor corpurilor.

Codul programului este dat în continuare:

```
/* Programul afişează primitivele 3D GLAUX */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

void myinit (void) {
   glClearColor(1.0, 1.0,1.0, 1.0);
        glColor3f(0.0,0.0,0.0);
}

void CALLBACK display (void)
{
   glClear(GL_COLOR_BUFFER_BIT);
        glLoadIdentity();
        glRotatef(30,1,1,1);
   glTranslatef(-250,300,0);
        auxWireCube(100);
        glTranslatef(200,-50,0);
        auxWireBox(200, 100, 50);
        glTranslatef(200,-100,0);
        auxWireTetrahedron(100);
        glTranslatef(200,-50,0);
        auxWireOctahedron(100);
        glTranslatef(-700,0,0);
        auxWireDodecahedron(100);
        glTranslatef(220,-70,0);
        auxWireIcosahedron(100);
        glTranslatef(230,-80,0);
        auxWireTorus(30, 80);
        glTranslatef(220,-100,0);
        auxWireCone(75, 150);
        glTranslatef(-600,20,0);
        auxWireSphere(80);
        glTranslatef(250,-50,0);
        auxWireCylinder(75, 100);
        glTranslatef(230,-120,0);
        auxWireTeapot(80);
```

```
        glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  if (!h) return;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
  glOrtho (-300.0, 300.0, -300.0*(GLfloat)h/(GLfloat)w,
    300.0*(GLfloat)h/(GLfloat)w, -300.0, 300.0);
  else
  glOrtho (-300.0*(GLfloat)w/(GLfloat)h,
    300.0*(GLfloat)w/(GLfloat)h, -300.0, 300.0, -300.0, 300.0);
  glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
  auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
  auxInitPosition (0, 0, 900, 600);
  auxInitWindow ("Primitive GLAUX");
  myinit ();
        auxReshapeFunc (myReshape);
  auxMainLoop(display);
  return(0);
}
```

Funcția auxMouseFunc

Funcția auxMouseFunc() asociază o funcție callback cu o acțiune asupra mouse-ului. Prototipul funcției este:

void auxMouseFunc (int button, int mode, AUXMOUSEPROC func);

- Funcția func va fi apelată atunci când este apăsat sau relaxat butonul button al mouse-ului.
- Parametrul button poate avea una din următoarele valori: AUX_LEFTBUTTON, AUX_MIDDLEBUTTON sau AUX_RIGHTBUTTON.
- Parametrul mode specifică acțiunea asociată cu butonul mouse-ului. Aceasta poate fi AUX_MOUSEDOWN sau AUX_MOUSEUP.

Funcția auxSwapBuffers

Funcția auxSwapBuffers() comută între cele două buffer-e color în timpul desenării cu dublu buffer. Prototipul funcției este:

void auxSwapBuffers(void) :

Funcția este utilizată în special pentru animație. Apelul ei va determina afişarea pe ecran a scenei ascunse. Funcția necesită ca la inițializarea ferestrei cu funcția auxInitDisplayMode() să fie utilizat flag-ul AUX_DOUBLE, pentru a specifica folosirea a două buffere de culoare.

Funcția auxSetOneColor

Funcția auxSetOneColor() setează o culoare în paleta de culori în cazul modelului index de culoare. Prototipul funcției este:

void auxSetOneColor(int index, float red, float green, float blue);

In cazul modelului de culoare index se creează o paletă de culori. O culoare este selectată prin specificarea unui index în paleta de culori. Această funcție asociază cu un anumit index o culoare specificată prin componentele RGB ale culorii.

3 PRIMITIVE GEOMETRICE

În acest capitol se va discuta despre primitivele geometrice de care dispune OpenGL pentru redarea scenelor, precum şi modul în care se controlează starea OpenGL pentru aspectul acestor primitive. Se va discuta apoi despre modul de reprezentare a curbelor şi a suprafețelor curbe utilizând evaluatori OpenGL dar şi funcțiile GLU pentru curbe şi suprafețe spline. Un subcapitol va fi dedicat pentru redarea cvadricelor utilizând funcțiile GLU. In afara tratării acestor primitive vectoriale este inclus un subcapitol special pentru tratarea primitivelor punctuale de care dispune OpenGL (bitmap-uri, imagini).

3.1 Primitive geometrice OpenGL

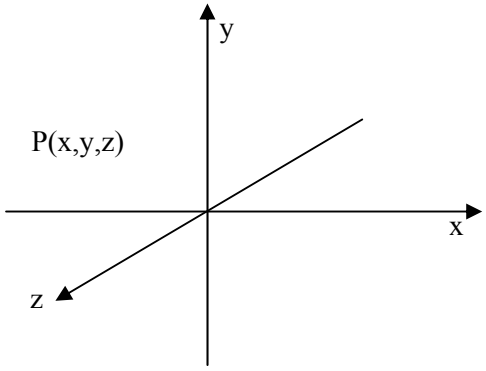


Figura 3.1

Înainte de a începe să discutăm despre felul în care sunt construite obiectele 3D pe baza primitivelor geometrice simple, trebuie să explicăm în detaliu sistemul de coordonate utilizat de OpenGL. Acesta este un sistem cartezian. Axele x şi y formează un plan, ceva de genul suprafeței vizibile a monitorului. Axa z adaugă cea de a treia dimensiune - adâncimea spațială (figura 3.1). Astfel pentru a avea o poziție spațială fixă în acest sistem de coordonate, un punct este necesar să fie specificat prin trei coordonate (x, y, z).

Primitivele geometrice (puncte, segmente de dreaptă, şi poligoane) sunt definite de un grup de unu sau mai multe vârfuri (vertex-uri). Un vârf defineşte un punct, punctul terminal al unei muchii, sau colțul în care se întâlnesc două muchii ale unui poligon. Unui vârf îi sunt asociate date (constând din coordonatele de poziție, culori, normale, şi coordonate de textură) şi fiecare vârf este procesat în mod independent şi în acelaşi fel. Singura excepție de la această regulă este în cazul în care un grup de vârfuri trebuie să fie decupate astfel ca primitiva respectivă să fie cuprinsă în regiunea specificată prin volumul de vizualizare definit de funcția myReshape(). În acest caz datele despre un vârf pot fi modificate şi sunt create noi vârfuri. Tipul decupării depinde de primitiva care este reprezentată de grupul de vârfuri.

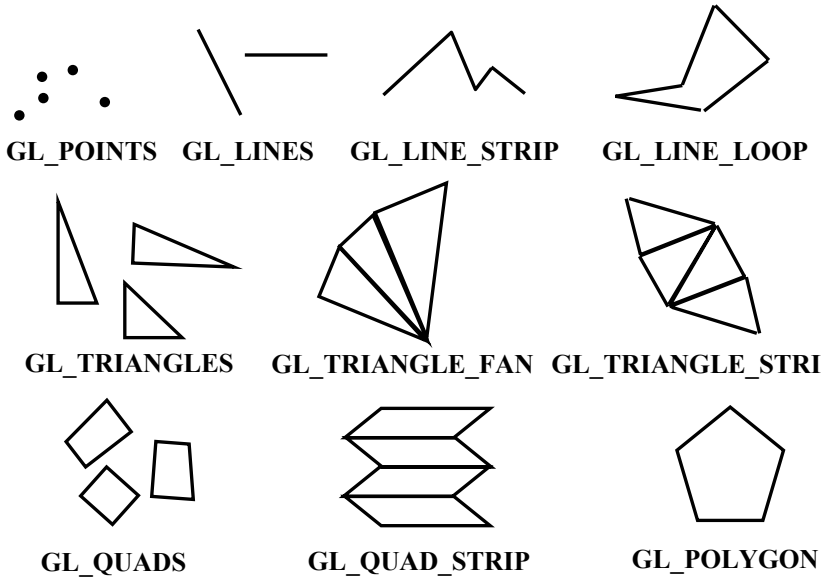


Figura 3.2

Toate primitivele geometrice sunt specificate prin vârfuri (coordonatele vârfurilor). Intern OpenGL operează cu coordonate omogene. Coordonatele omogene sunt de forma (x, y, z, w). Funcțiile OpenGL permit specificarea vârfurilor în coordonate 2D, 3D şi în coordonate omogene.

În funcție de parametrul funcției glBegin() (care poate fi unul din cele 10 nume enumerate în figura 3.2) şi de felul cum sunt organizate vârfurile, OpenGL poate reda oricare din primitivele arătate în figura 3.2.

Exemplu

Pentru ca cele spuse să fie mai clare se va exemplifica printr-o funcție de redare a unui patrulater.

```
void patrulater( GLfloat color[] ) {
```

```
glBegin( GL_QUADS ) ;
glColor3fv( color );
glVertex2f( 0.0, 0.0 );
glVertex2f( 1.0, 0.0 ) ;
glVertex2f( 1.5, 1.118 );
glVertex2f( 0.5, 1.118 ) ;
glEnd(); }
```

Funcția patrulater() determină OpenGL să deseneze un patrulater într-o singură culoare. Patrulaterul este planar deoarece, coordonat z este setată automat la 0 de funcția glvertex2f (). Dacă s-ar fi dorit ca patrulaterul să nu se afle în planul x0y, ci într-un plan oarecare, atunci ar fi fost necesar să fie utilizată o funcție glvertex3#(). În felul acesta s-ar fi furnizat pentru fiecare vârf şi cea de a treia coordonată. Trebuie remarcat de asemenea că numele primitivei geometrice se dă ca parametru al funcției glBegin() şi corpul în care sunt furnizate coordonatele primitivei este delimitat de funcțiile glBegin() şi glEnd().

3.1.1 Formatul comenzilor OpenGL

Fiecare din comenzile care specifică coordonatele vârfului, normalele, culorile sau coordonatele de textură sunt disponibile în câteva formate pentru a se potrivi diferitelor formate de date ale aplicațiilor şi numărului de coordonate (figura 3.3). Datele pot fi de asemenea transmise acestor comenzi fie ca o listă de argumente sau ca un pointer la un tablou ce conține date. Variantele se disting prin sufixele mnemonicelor comenzilor.

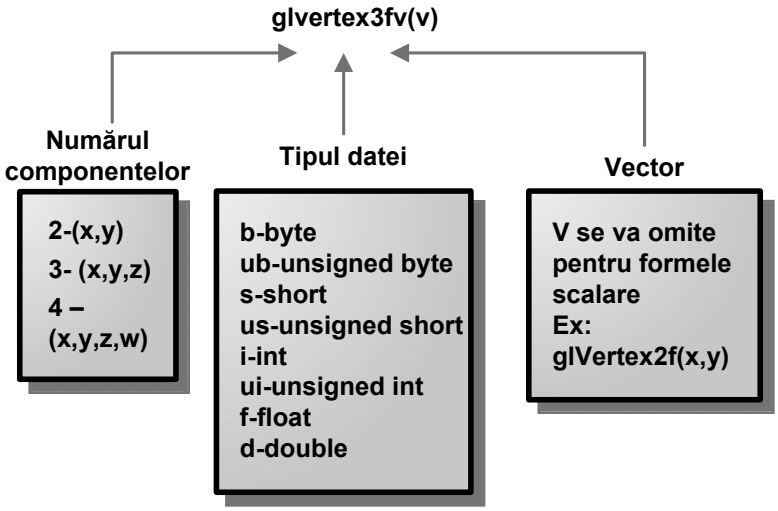


Figura 3.3

Interfața de programare a aplicațiilor OpenGL (OpenGL API) are astfel proiectate apelurile încât să fie acceptate aproape orice tipuri de date de bază, al căror nume este reflectat în numele funcției. Cunoscând felul în care sunt formate numele apelurilor este uşor de determinat care apel va fi utilizat pentru un format particular de

date şi pentru o anume dimensiune. Spre exemplu, coordonatele vârfurilor pentru aproape toate modelele comerciale sunt reprezentate de un vector cu trei componente în virgulă mobilă. În felul acesta, cea mai potrivită comandă este glvertex3fv(coord). în acest caz coord este un vector cu trei componente de tip float. Dacă cele trei componente s-ar furniza ca parametri ai funcției glVertex# atunci forma apelului ar fi givertex3f (x, y, z); unde x, y, z sunt elemente de tip float.

Fiecare vârf poate fi specificat cu două, trei sau patru coordonate (patru coordonate indică coordonate omogene 3D). După cum s-a arătat mai înainte, intern, OpenGL utilizează coordonate omogene 3D pentru specificarea vârfurilor. Pentru acele forme de apel glvertex#() care nu specifică toate coordonatele (adică glvertex2f()), OpenGL va fixa implicit z=0.0 şi w=1.0.

3.1.2 Specificarea primitivelor geometrice OpenGL

În OpenGL cele mai multe obiecte geometrice sunt desenate prin includerea unei mulțimi de coordonate care specifică vârfurile şi opțional normalele, coordonatele de textură şi culorile între perechea de comenzi glBegin()/glEnd():

```
glBegin(tipul_primitivei) ;
        //se specifică coordonatele vârfurilor, normalele,
        //culorile, coordonatele de texturare
glEnd();
```

unde tipul_primitivei determină felul cum sunt combinate vârfurile din blocul respectiv.

Spre exemplu, pentru specificarea unui patrulater cu vârfurile având coordonatele (0,1,0), (1,3,0), (3,4,0),(0,4,0) se poate scrie:

```
glBegin(GL_QUADS)
glVertex3i(0,1,0) ;
glVertex3i(1,3,0) ;
glVertex3i(3,4,0) ;
glVertex3i(0,4, 0) ;
glEnd();
```

Cele zece primitive geometrice care pot fi astfel desenate sunt descrise în figura 3.2. Acest grup particular de primitive permite ca fiecare obiect să poată fi descris printr-o listă de vârfuri, şi permite satisfacerea necesităților fiecărei aplicații grafice.

Exemplu:

```
Glfloat red, greed, blue;
Glfloat coords[3];
glBegin(tipul_primitivei);
for (i=0; i< nvertex; ++i){
        glColor3f( red, green, blue );
        glVertex3fv( coords);
}
glEnd() ;
```

În acest exemplu, nvertex este numărul vârfurilor care se specifică. Acestea sunt grupate după tipul primitivei, specificat ca parametru al funcției glBegin(). Spre exemplu dacă tipul primitivei este GL_QUADS şi nvertex este 12 se vor desena 3 patrulatere. Tipurile posibile pentru tipul primitivei sunt:

GL_POINTS
GL_LINE_STRIP
GL_LINES
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUADS
GL_QUAD_STRIP
GL_POLYGON

În esență sunt doar 5 tipuri de primitive (punct, linie, triunghi, patrulater şi poligon) restul de 5 chiar dacă nu introduc primitive suplimentare prezintă avantajul ca în cazul în care două vârfuri aparținând la două primitive distincte, se suprapun, să fie memorate o singură dată. în felul acesta se câştigă atât timp de procesare cât şi spațiu de memorie.

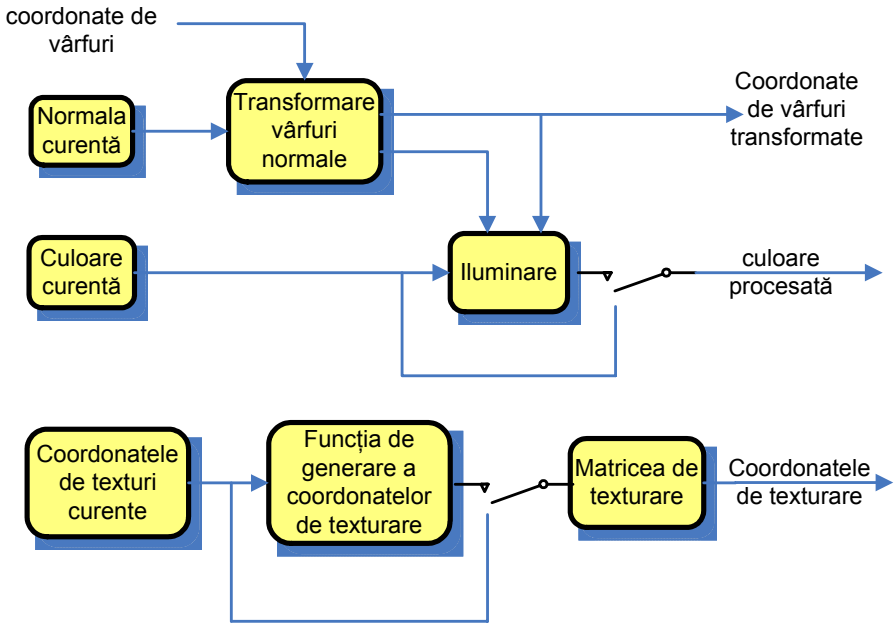


Figura 3.4 Asocierea valorilor curente cu vârfurile

Pentru procesarea unui vârf mai pot fi, suplimentar, utilizate normala curentă, coordonatele texturii curente şi culoarea curentă. OpenGL utilizează normalele pentru calculele de iluminare. Normala curentă este un vector 3D care se poate seta prin specificarea celor trei coordonate. Culoarea poate fi formată din valorile pentru roşu, verde, albastru şi alfa (atunci când se utilizează modelul de culoare RGBA) sau o singură valoare index (atunci când la inițializare se specifică modul index). Una două trei sau patru coordonate de textură determină felul cum imaginea unei texturi se mapează pe o primitivă.

Atunci când este specificat un vârf, culoarea curentă, normala, coordonatele texturii sunt utilizate pentru a obține valori care sunt apoi asociate cu vârful (figura 3.4). însuşi vârful este transformat de matricea de modelare-vizualizare, care este o matrice de 4x4. Această transformare are loc intern prin înmulțirea matricei de modelare vizualizare cu coordonatele vârfului obținându-se coordonatele acestuia în sistemul de vizualizare. Culoarea asociată vârfului se obține fie pe baza calculelor de iluminare, fie dacă iluminarea este dezactivată, pe baza culorii curente. Coordonatele de texturare sunt transmise în mod similar unei funcții de generare a coordonatele de texturare (care poate fi identitate). Coordonatele de texturare rezultate sunt transformate de matricea de texturare (această matrice poate fi utilizată pentru scalarea sau rotirea unei texturi ce se aplică unei primitive).

Reguli pentru construirea poligoanelor

Nu ne referim aici la triunghiuri deşi fac şi ele parte din categoria poligoanelor. Motivele sunt simple: triunghiurile aparțin întotdeauna unui singur plan, triunghiurile sunt întotdeauna convexe.

Regula 1

Poligoanele trebuie să fie totdeauna planare. Aceasta înseamnă că toate vârfurile unui poligon trebuie să aparțină aceluiaşi plan. în figura 3.5 se poate vedea un poligon planar şi altul neplanar (răsucit).

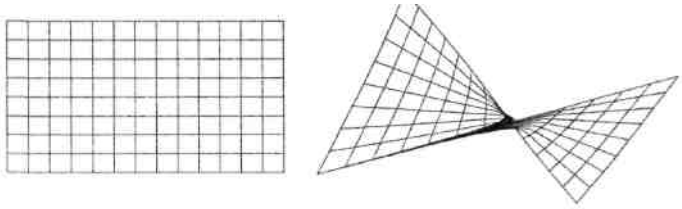


Figura 3.5

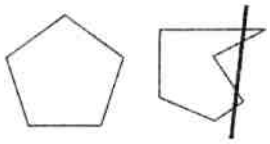


Figura 3.6

Regula 2

Poligoanele trebuie să fie convexe. Pentru a verifica dacă un poligon este convex se face testul următor: dacă oricare linie care traversează poligonul este intersectată în mai mult de două puncte de laturile poligonului acesta nu este convex (figura 3.6).

Regula 3

Muchiile poligoanelor nu se pot intersecta. Spre exemplu poligonul din figura 3.7 nu este un poligon corect în concepția OpenGI

Figura 3.7

Orientarea poligoanelor

În OpenGL ordinea specificării vârfurilor este cea dată de ordinea apelurilor givertex# () în blocul glBegin() /glEnd().

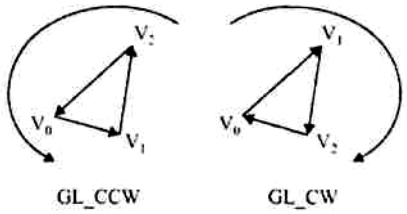


Figura 3.8

Ordinea stabilită a vârfurilor este implicit în sens invers rotirii acelor de ceasornic. Ordinea vârfurilor se specifică în OpenGL prin funcția glFrontFace() care are ca parametrii (figura 3.8):

- GL_CCW pentru sensul trigonometric (invers acelor de ceasornic "counterclockwise") - care este valoarea implicită
- GL_CW pentru sensul rotirii acelor de ceasornic ("clockwise")

O primitivă geometrică are orientare directa ("frontface") daca ordinea specificării vârfurilor coincide cu ordinea stabilită a vârfurilor. Dacă ordinea specificării vârfurilor nu coincide cu ordinea stabilită a vârfurilor atunci primitiva are orientare inversa ("backface")

Ordinea corectă pentru specificarea vârfurilor primitivelor OpenGL

In figura 3.9 este arătată ordinea în care trebuie specificate vârfurile primitivelor OpenGL pentru ca orientarea acestora să fie în sens invers rotirii acelor de ceasornic - adică ordinea directă în OpenGL.

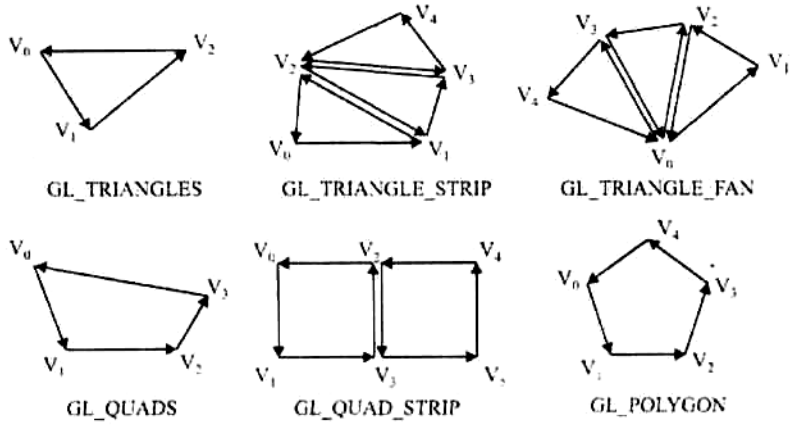


Figura 3.9

Eliminarea poligoanelor în funcție de orientare

În OpenGL se poate realiza eliminarea anumitor suprafețe în funcție de orientarea acestora.

- Validarea eliminării primitivelor geometrice în funcție de orientare:
- glEnable(GL_CULL_FACE);
- Selectarea suprafețelor eliminate:
- glCullFace(GLenum type);

unde type poate fi: GL_BACK - sunt eliminate poligoanele având orientare inversă, GL_FRONT - sunt eliminate poligoanele având orientare directă.

Utilitatea orientării poligoanelor

La construirea unui corp solid toate poligoanele care-1 mărginesc trebuie să aibă aceeaşi orientare. Dacă spre exemplu toate poligoanele au orientare trigonometrică, toate poligoanele vor fi poligoane față şi vor fi vizibile. Se poate stabili ca fețele spate să fie eliminate în situația în care corpul este vizualizat din exterior. Dacă însă corpul este vizualizat dinspre interior atunci se va stabili să fie vizibile doar fețele spate şi să fie eliminate fețele față.

Să presupunem acum că un anumit corp solid a fost consistent construit, deci cu toate poligoanele având aceeaşi orientare. Orientarea

specificată însă nu este aceeaşi cu cea stabilită pentru fețele față. Atunci se poate interveni şi se schimbă orientarea stabilită pentru fețele față.

3.1.3 Atribute ale primitivelor de ieşire

În acest subcapitol ne vom referi la atribute primitivelor de ieşire: dimensiunea punctului, grosimea sau stilul liniei, modelul de umplere pentru primitivele cu atribut de umplere, etc.

Atributul de culoare

Modelele de culoare OpenGL sunt RGBA sau Color Index. Fiecare implementare OpenGL trebuie să permită redarea atât în modelul de culoare RGBA (denumit uneori ca modelul TrueColor) cât şi în modelul index (sau colormap).

Pentru redarea cu modelul RGBA, culorile vârfurilor sunt specificate utilizând apelul glColor#(). Pentru modelul index, indexul culorii fiecărui vârf este specificat cu apelul gllndex# ().

Sistemul de ferestre cere specificarea modelului de culoare al ferestrei. Dacă se utilizează GLUT, atunci modelul de culoare al ferestrei se specifică folosind apelul glutlnitDisplayMode(), cu flag-urile GLUT_RGBA (pentru modelul RGBA) sau GLUT_INDEX (pentru modelul index). Dacă se utilizează biblioteca GLAUX, modelul de culoare se specifică folosind funcția auxinitDisplayMode() cu flag-urile AUX_RGBA sau AUX_INDEX.

Setările de culoare pentru primitive pot fi specificate utilizând fie modelul RGBA fie modelul index. Odată specificată o anumită culoare, va afecta toate primitivele care urmează până la o modificare a culorii. Noua culoare va afecta apoi doar primitivele specificate după modificarea culorii.

În modelul RGB se vor specifica componentele roşu, verde şi albastru ale culorii. Se poate folosi şi varianta RGBA în care se specifică şi o a patra componentă - valoarea alfa. Această a patra componentă este utilizată pentru amestecarea culorilor pentru obiectele care se suprapun. O aplicaţie importantă a amestecării culorilor este simularea efectului de transparenţă. Pentru aceste calcule valoarea alfa corespunde coeficientului de transparenţă.

Pentru a folosi modelul RGB (sau RGBA), se setează simplu culoarea cu funcţia glColor#() dând celor patru parametrii ai culorii valori cuprinse între 0.0 şi 1.0. Pentru specificarea componentelor culorii pot fi utilizate valori întregi, în virgulă mobilă sau alte formate numerice. Ca şi în cazul funcţiei giVertex se vor utiliza ca sufix coduri ca 3, 4, d, f şi i pentru a indica numărul parametrilor şi formatul acestora. De pildă, funcţia următoare specifică componentele de culoare RGB în virgulă mobilă şi setează culoarea curentă ca albastru de intensitate maximă:

glColor3f(0.0, 0.0, 1.0);

Valoarea implicită a culorii este alb (1, 1, 1) şi valoarea implicită pentru alfa este 1. În mod opţional, se pot seta valorile de culoare utilizând tablourile. În acest caz se va adăuga funcţiei un al treilea sufix - v - şi se va furniza un singur argument - pointer-ul la tabloul ce conţine componentele culorii. Utilizând un tablou colorArray care conţine spre exemplu valorile anterioare, se va seta culoarea utilizând apelul:

glColor3fv(colorArray);

Celălalt model pentru setarea culorii, disponibil în OpenGL este modelul index care face referinţe la tabele de culoare. Acest model este utilizat în cazul în care spaţiul de memorare este limitat. în acest model se setează ca şi culoare curentă o culoare din tabelul de culori utilizând funcţia:

gllndex#(index);

unde index specifică o poziţie în tabloul cu culori. Funcţia necesită de asemenea un sufix (d, f, i) care specifică tipul datelor. Un tablou de culori poate fi setat cu funcţia:

glIndexPointer(type, offset, ptr);

unde parametrul type dă tipul datei, parametrul offset, dă spaţierea între valori consecutive de culoare şi parametrul ptr este un pointer spre primul element al tabloului de culori.

A patra componentă de culoare

Componenta alfa pentru o culoare este o măsură a opacităţii fragmentului. Ca şi cu celelalte componente de culoare, domeniul valorilor sale este între 0.0 (care reprezintă complet transparent) şi 1.0 (complet opac). Valorile alfa sunt importante pentru o mulţime de utilizări despre care se va discuta pe larg în capitolul 8 (Efecte vizuale: transparenţa, ceaţa, antialiasing).

Atributele punctului

Pentru punct pot fi setate două atribute: culoare şi dimensiune. Culoarea este controlată de setarea curentă a culorii, iar dimensiunea punctului se setează cu funcţia:

void glPointSize(Glfloat size);

Parametrul size poate fi specificat ca orice valoare pozitivă în virgulă mobilă. Dacă valoarea specificată nu este un întreg, ea va fi rotunjită la cea mai apropiată valoare întreagă (presupunând că antialiasing-ul este dezactivat). Valoarea implicită pentru size este 1.

Atributele liniei

Atributele de bază pentru segmentul de linie dreaptă sunt tipul liniei, grosimea liniei şi culoarea liniei. Alte efecte posibile, cum ar fi tipul peniţă sau tipul pensulă, pot fi implementate cu adaptări ale funcţiilor pentru tipul şi grosimea liniei.

Pentru afişarea unui segment de dreaptă într-o singură culoare este folosită setarea curentă a culorii. O altă posibilitate ar fi ca linia să fie afişată cu culoarea variind de-a lungul liniei. O modalitate de a face acest lucru este de a atribui culori diferite fiecărei extremităţi a unei linii, în loc de a se seta o culoare curentă pentru întreaga linie. De pildă, porţiunea de cod următoare setează o extremitate a liniei verde iar cealaltă galbenă. Rezultatul va fi afişarea unei linii continue în culori care interpolează culorile extremităţilor.

```
glShadeModel(GL_SMOOTH);
glBegin (GL_LINES);
glColor3f (0.0, 1.0, 0.0);
glVertex2i (-5,-5);
glColor3f (1.0, 1.0, 0.0);
glVertex2i (5,5);
glEnd ();
```

În ceea ce priveşte celălalt atribut al linie - tipul - în mod implicit acesta este tipul continuu. În afara liniilor continue se pot utiliza linii întrerupte, punctate sau în diverse combinaţii linie-punct. Spaţiile dintre liniuţe sau puncte, dimensiunea liniuţelor pot fi de asemenea modificate. Stilul liniei este definit de programator - nu există un set de stiluri din care programatorul să-şi aleagă un anume stil.

Şablonul se stabileşte într-un cuvânt de 16 biţi în care fiecare bit are semnificaţia de un pixel. Modelul astfel stabilit se repetă de-a lungul liniei. Dacă se doreşte ca semnificaţia fiecărui bit să fie de 2, 3, ... pixeli se poate realiza o multiplicare cu un factor. Pentru a se putea afişa linii de diferite stiluri trebuie realizată activarea acestui atribut cu funcţia glEnable():

```
glEnable(GL_LINE_STIPPLE);
```

Altfel toate liniile vor fi afişate ca linii continue. Revenirea la linie continuă se face, evident, cu funcţia glDisable().

Un anumit tip de linie este specificat cu funcţia:

```
void glLineStipple(GLint repeatFactor, GLushort pattern);
```

care specifică şablonul după care sunt distribuiţi pixelii la desenarea liniei. Parametrul pattern dă un model, în binar, pe 16 biţi (l=pixel on, 0=pixel off) şi parametrul întreg repeatFactor specifică de câte ori se repetă fiecare bit din model. Modelul este aplicat pixelilor de-a lungul liniei începând cu biţii mai puţin semnificativi. Spre exemplu, apelul:

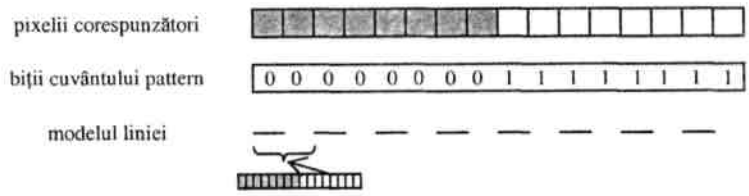
```
glLineStipple(1, 0x00ff);
```

specifică o linie întreruptă în care liniuțele ocupă 8 pixeli iar spațiul dintre liniuțe este tot de 8 pixeli. Deoarece biții mai puțin semnificativi sunt pe on fiecare segment de dreaptă care se desenează va începe cu liniuță şi modelul se va repeta până se ajunge la cealaltă extremitate a segmentului de dreaptă. În cazul în care se desenează o linie frântă modelul nu va fi restartat după fiecare segment de dreaptă, ci va fi aplicat continuu.

Iată spre exemplu cum se poate specifica o linie întreruptă cu intervale egale pentru întrerupere şi linie.

Exemplu:

```
GLint factor=l;
GLushort pattern=0x0(
glEnable(GL_LINE_S
glLineStipple(factor, p
glBegin(GL_LINES);
glVertex2f(20, 30);

glVertex2f(100, 30);
glEnd(
);
```



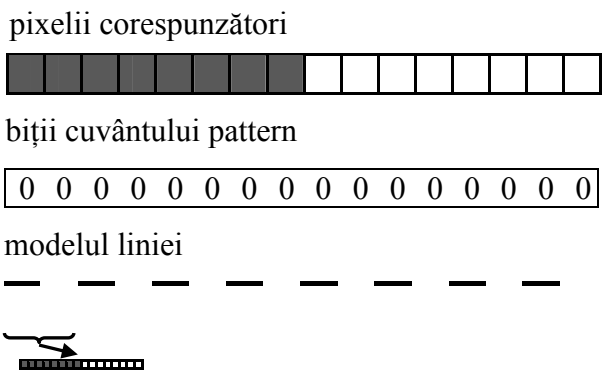


Figura 3.10

Din figura 3.10 se poate constata care este corespondența dintre modelul liniei şi cuvântul pattern utilizat.

Ultimul atribut al liniei - grosimea - se setează cu funcția:

void glLineWidth(Glfloat width);

Valoarea implicită a parametrului width este 1. Valorile în virgulă mobilă sunt rotunjite la întregul cel mai apropiat atunci când nu este activat antialiasing-ul.

Atributele poligoanelor

Pentru poligoane, atributul de bază este stilul de umplere. Poligoanele pot fi umplute într-o singură culoare, cu un anumit model de umplere, sau fără atribut de umplere ("hollow") atunci când se reprezintă doar conturul. Aşa cum s-a arătat un poligon are două fețe: față şi spate care pot fi diferit reprezentate. Aceasta permite reprezentarea secțiunilor prin corpuri. Se pot selecta atribute diferite pentru poligoanele

"față" şi pentru cele "spate", se poate inversa definirea poligoanelor "față" sau "spate" şi se poate elimina afişarea poligoanelor "față" sau "spate".

În mod implicit ambele fețe sunt desenate la fel. Pentru a schimba acest lucru se utilizează funcția glPolygonMode(),

glPolygonMode(GLenum face, GLenum mode)

care stabileşte modul de desenare al poligoanelor (mode: GL_POINT, GL_LINE, GL_FILL) şi tipul fețelor care sunt afectate (face: direct -GL_FRONT, invers - GL_BACK, ambele - GL_FRONT_AND_BACK).

În mod implicit ambele fețe sunt desenate cu atribut de umplere. Se poate spre exemplu, stabili ca partea față să fie desenată plin iar cea spate să fie desenată cu wireframe (fără atribut de umplere).

glPolygonMode(GL_FRONT, GL_FILL);

glPolygonMode(GL_BACK, GL_LINE);

Culoarea primitivelor cu atribut de umplere

Triunghiurile, patrulaterele şi poligoanele sunt primitivele care au atribut de umplere. Acestea pot fi colorate cu o anumită culoare sau pot fi umplute cu un anumit model de umplere. în ceea ce priveşte culoarea, aceasta se poate stabili pentru fiecare vârf în parte sau se utilizează culoarea curentă de desenare. Funcția utilizată este glColor3f(). În funcție de modelul de umbrire, poligoanele sunt umplute cu o culoare compactă sau cu o interpolare a culorii vârfurilor. Modelul de umbrire se stabileşte cu funcția glShadeModel().

void glShadeModel (Glenum mode);

Dacă se stabileşte o umbrire poligonală constantă (parametrul GL_FLAT) atunci culoarea de umplere a poligoanelor va fi culoarea curentă sau culoarea stabilită pentru ultimul vârf al poligonului. Dacă se stabileşte modelul de umbrire Gouroud (GL_SMOOTH) atunci se vor interpola culorile vârfurilor (ca la pătratul reprezentat pentru exemplificare funcțiilor glaux).

Utilizarea şabloanelor de umplere

Aşa cum liniile pot fi desenate având anumite stiluri şi poligoanele pot fi umplute cu anumite modele de umplere. Pentru aceasta programatorul stabileşte şablonul, care se va repeta apoi pe suprafața întregului poligon. Pentru activarea umplerii cu şablon se foloseşte funcția glEnable().

glEnable(GL_POLYGON_STIPPLE);

Modelul de umplere se specifică folosind funcția glPolygonStipple având ca parametru un pointer spre un tablou care conține şablonul.

glPolygonStipple(fillPattern);

Parametrul fillPattern este un pointer la o mască de 32X32 biți. Tabloul fillPattern conține elemente de tip GLubyte. O valoare 1 în mască arată că pixelul corespunzător va fi pe on, şi 0 indică un pixel pe of f. Intensitatea pixelilor este setată în funcție de setarea culorii curente. Spre exemplu

GLubyte fillPattern[]={Oxff, 0x00, Oxff, 0x00, ...};

Biții trebuie să fie specificați începând cu rândul de jos al modelului, şi continuând până la rândul cel mai de sus (al-32-lea). Modelul este apoi aplicat de-a lungul ferestrei curente, umplând poligonul specificat acolo unde modelul se suprapune

cu interiorul poligonului. Odată specificat modelul şi activată umplerea cu şablon se poate trece la desenarea poligonului.

O altă modalitate de umplere a poligoanelor este utilizând texturile. De asemenea, poligoanele pot fi umplute cu culoarea curentă setată. Ca şi în cazul liniilor se pot specifica culori diferite pentru fiecare vârf al unui poligon şi în acest caz culoare interiorului poligonului va fi interpolarea culorilor vârfurilor poligonului. Aşa cum s-a mai arătat poligoanele pot fi desenate în modul wireframe sau doar marcând vârfurile poligonului, depinde de ce parametrii sunt folosiți pentru funcția glPolygonMode():

GL_FILL - cu atribut de umplere
GL_LINE - fără atribut de umplere(wireframe)
GL_POINTS - marcate vârfurile

În OpenGL, funcțiile de redare a poligoanelor pot fi aplicate doar poligoanelor convexe. Pentru un poligon concav, trebuie mai întâi despărțit într-o mulțime de poligoane convexe, de obicei triunghiuri. Pentru a afişa apoi doar poligoanele originale trebuie să putem specifica care linii fac parte din poligoanele originale. Pentru aceasta se foloseşte funcția glEdgeFlag, care arată dacă un vârf este sau nu primul punct al unei laturi din poligonul original. Funcția glEdgeFlag() este utilizată în interiorul perechii glBegin/glEnd.

glEdgeFlag(GL_FALSE);

Această comandă arată că vârful care urmează nu precede o latură a unui poligon original. De asemenea, comanda se aplică tuturor vârfurilor care urmează până la un nou apel al funcției glEdgeFlag. Argumentul GL_TRUE arată că vârful următor inițiază o latură a poligonului original.

Antialiasing

Pentru activarea procedurilor pentru antialiasing se foloseşte apelul:

glEnable();

având ca parametru una din valorile GL_POINT_SMOOTH, GL_LINESMOOTH sau GL_POLYGON_SMOOTH.

Funcții de interogare

Se pot obține valorile pentru setările curente ale atributelor şi pentru diverşi alți parametrii utilizând funcțiile corespunzătoare "Get" cum ar fi glGetlntegerv(), glGetFloatv(), glGetPolygonStipple(). Spre exemplu, se poate afla setarea curentă a culorii cu

glGetFloatv(GL_CURRENT_COLOR, colorArray);

Tabloul colorArray va fi setat în urma acestui apel cu culoarea curentă. Similar se poate afla dimensiunea curentă a punctului (GL_POINT_SIZE) sau a liniei (GL_LINE_WIDTH_RANGE). De asemenea se poate afla dacă anumite proceduri, cum ar fi antialiasing-ul, sunt activate sau dezactivate.

Exemple

În continuare vor fi date câteva exemple ilustrative pentru utilizarea primitivelor geometrice şi a atributelor acestora.

Exemplul 1

```
/*Un cerc din puncte de dimensiune un pixel */
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void myinit(void)
{
   glClearColor(1.0, 1.0, 1.0, 1.0);
   glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
   int i;
         double PI = 3.1415926535;
         double raza =2.0;

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);

   glBegin(GL_POINTS);
   for (i = 0; i < 360; i+=5)
      glVertex2f(raza*cos(PI*i/180.0), raza*sin(PI*i/180.0));
   glEnd();
   glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
      5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
   else
   glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
      5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
```

```
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (0, 0, 200, 200);
   auxInitWindow ("Un cerc din puncte");
   myinit();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
   return(0);
}
```

Observaţii:

Ceea ce se obţine la rularea acestui program este afişat în figura 3.11. Ce trebuie remarcat în acest exemplu este felul cum s-au furnizat coordonatele punctelor în corpul glBegin/glEnd. Într-un ciclu for s-a apelat funcţia glVertex() de 360/5 ori. Coordonatele punctelor de pe cerc s-au dat în coordonate polare. Mai trebuie remarcat că în funcţia myReshape() s-au stabilit 10 unităţi pe fiecare axă şi s-a urmărit ca raza cercului(2) să se încadreze în aceste dimensiuni.

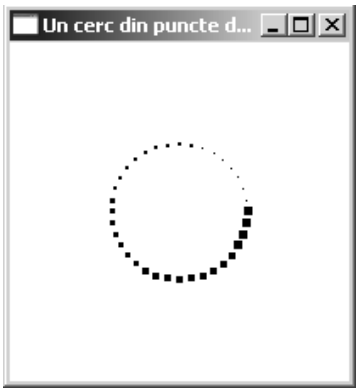


Figura 3.11

Figura 3.12

Exemplul 2

Să modificăm acum acest exemplu astfel încât punctele de pe cerc să aibă dimensiuni crescătoare. Ceea ce se modifică este funcția display(). Rezultatul rulării este arătat în figura 3.12.

```
void CALLBACK display(void) {
int i ;
double  PI  =  3.1415926535;
double  raza =2.0;
GLfloat   size=0.5;
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0,   0.0,   0.0);
for  (i  =  0;  i  <  360;  i+=10)
{ glBegin(GL_POINTS);
glVertex2f(raza*cos(PI*i/180.0),   raza*sin(PI*i/180.0));
glEnd() ;
size+=0.125;       //incrementează dimensiunea punctului
glPointSize(size); }
glFlush() ; }
```

Observaţii:

Ceea ce trebuie remarcat aici este faptul că în corpul glBegin/glEnd nu au efect decât instrucțiunile pentru caracteristicile vârfurilor. Astfel că următorul cod nu ar fi condus la rezultatul aşteptat, deşi nu ar fi apărut eroare la compilare.

```
glBegin(GL_POINTS);
for  (i  =  0;  i  <  360;  i+=5)
glVertex2f(raza*cos(PI*i/180.0), raza*sin(PI*i/180.0));
size+=0 .125;       //incrementează dimensiunea punctului
glPointSize(size);
glEnd() ;
```

Exemplul 3

Exemplul 1 se poate modifica pentru desenarea unui cerc cu linie continuă, prin simpla modificare a parametrului funcției glBegin() în GL_LINE_LOOP (figura 3.13). Dacă se foloseşte parametrul GL_LINE_STRIP atunci nu este unit primul punct cu ultimul punct al liniei frânte din care s-a construit cercul. Acesta este un exemplu despre cum se construieşte o curbă dintr-o linie frântă. Pentru ca linia care formează cercul să fie mai netedă, vârfurile vor fi alese şi mai apropiate. Cu toate că cercul este format dintr-un contur închis, el nu are atribut de umplere. Pentru aceasta este suficient să se înlocuiască parametrul GL_LINE_LOOP, al funcției glBegin() cu parametrul GL_POLYGON. Rezultatul obținut în acest caz este arătat în figura 3.14.

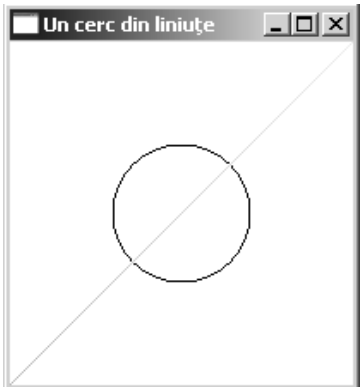


Figura 3.13

Figura 3.14

Exemplul 4

Se reia acum exemplul anterior dar cercul va fi umplut cu un model (figura 3.15). În continuare se poate vedea felul în care a fost declarat tabloul care conține modelul de umplere.

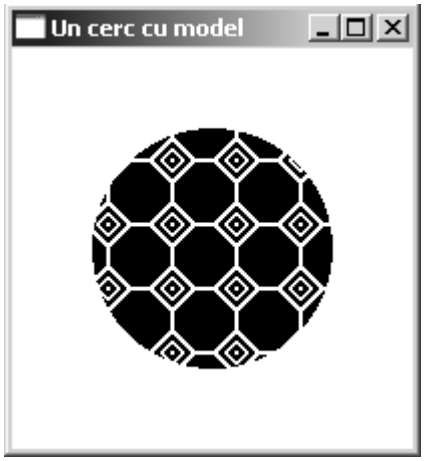


Figura 3.15

Codul programului este următorul:

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void CALLBACK display(void)
{
   int i;
        double PI = 3.1415926535;
        double raza =3.0;

        GLubyte model[] = {  0xFF, 0xFe, 0x7F, 0xFF,
                                                0xFF, 0xFe, 0x7F, 0xFF,
                                                0xFF, 0xFe, 0x7F, 0xFF,
                                                0xFF, 0xFe, 0x7F, 0xFF,
                                                0xFF, 0xFe, 0x7F, 0xFF,
                                                0xFF, 0xFc, 0x3F, 0xFF,
                                                0xFF, 0xF8, 0x1F, 0xFF,
                                                0xFF, 0xF1, 0x8F, 0xFF,
                                                0xFF, 0xe3, 0xc7, 0xFF,
                                                0xFF, 0xc7, 0xe3, 0xFF,
                                                0xFF, 0x8e, 0x71, 0xFF,
                                                0xFF, 0x1c, 0x38, 0xFF,
                                                0xFe, 0x39, 0x9c, 0x7F,
                                                0xFc, 0x73, 0xce, 0x3f,
                                                0xF8, 0xe7, 0xe7, 0x1f,
                                                0x01, 0xce, 0x73, 0x80,
                                               0x01, 0xce, 0x73, 0x80,
                                               0xF8, 0xe7, 0xe7, 0x1f,
                                               0xFc, 0x73, 0xce, 0x3f,
                                               0xFe, 0x39, 0x9c, 0x7F,
                                               0xFF, 0x1c, 0x38, 0xFF,
                                               0xFF, 0x8e, 0x71, 0xFF,
                                               0xFF, 0xc7, 0xe3, 0xFF,
                                               0xFF, 0xe3, 0xc7, 0xFF,
                                               0xFF, 0xF1, 0x8F, 0xFF,
                                               0xFF, 0xF8, 0x1F, 0xFF,
                                               0xFF, 0xFc, 0x3F, 0xFF,
                                               0xFF, 0xFe, 0x7F, 0xFF,
                                               0xFF, 0xFe, 0x7F, 0xFF,
                                               0xFF, 0xFe, 0x7F, 0xFF,
                                               0xFF, 0xFe, 0x7F, 0xFF,
                                               0xFF, 0xFe, 0x7F, 0xFF,
```

```
                                                    };

            glClearColor(1.0, 1.0, 1.0, 1.0);
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(0.0, 0.0, 0.0);

            glEnable (GL_POLYGON_STIPPLE);
      glPolygonStipple (model);

      glBegin(GL_POLYGON);
      for (i = 0; i < 360; i+=5)
         glVertex2f(raza*cos(PI*i/180.0), raza*sin(PI*i/180.0));
      glEnd();
      glFlush();
   }

   void CALLBACK myReshape(GLsizei w, GLsizei h)
   {
      if (!h) return;
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      if (w <= h)
      glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
         5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
      else
      glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
         5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
   }

   int main(int argc, char** argv)
   {
      auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
      auxInitPosition (0, 0, 200, 200);
      auxInitWindow ("Un cerc cu model");
      auxReshapeFunc (myReshape);
      auxMainLoop(display);
      return(0);
   }
```

Exemplul 5

Exemplul următor construieşte un cub din patrulatere. Cubul este animat (rotit), utilizându-se funcţia IdleFunctionO. Indiferent de poziţia cubului în timpul rotaţiei, muchiile spate sunt reprezentate cu linie punctată iar muchiile faţă cu linie continuă.

/*Un cub care se roteşte şi care are muchiile spate desenate cu linie punctată */

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>
void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK IdleFunction(void);
void CALLBACK display(void);
void cub(GLfloat latura);

void myinit(void)
{
   glClearColor(1.0, 1.0, 1.0, 1.0);
   glShadeModel(GL_FLAT);
}

void CALLBACK IdleFunction(void)
{
        glRotatef(30,1,1,1);
        display();
        Sleep(300);
}

void CALLBACK display(void)
{
   GLint factor=10;
        GLushort pattern=0x255;

        glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
        glLineStipple(factor, pattern);

        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        glEnable(GL_CULL_FACE);
        glCullFace(GL_FRONT);
        glEnable(GL_LINE_STIPPLE);
        cub(0.5);

        glCullFace(GL_BACK);
        glDisable(GL_LINE_STIPPLE);
        cub(0.5);

   glFlush();
}
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho(-1.0, 1.0, -1.0*(GLfloat)h/(GLfloat)w,
      1.0*(GLfloat)h/(GLfloat)w, -1.0, 1.0);
   else
   glOrtho(-1.0*(GLfloat)w/(GLfloat)h,
      1.0*(GLfloat)w/(GLfloat)h, -1.0, 1.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (0, 0, 200, 200);
   auxInitWindow ("Un cub");
   myinit();
   auxReshapeFunc (myReshape);
         auxIdleFunc(IdleFunction);
   auxMainLoop(display);
   return(0);
}

void cub(GLfloat latura) //funcția care n cubul
//cu latura de 2Xlatura şi centrat în origine
//ordinea specificării vârfurilor este invers rotirii acelor de ceasornic
{   glBegin(GL_QUAD_STRIP);
          glVertex3f(-latura, latura, latura);
          glVertex3f(-latura, -latura, latura);
          glVertex3f(latura, latura, latura);
          glVertex3f(latura, -latura, latura);
          glVertex3f(latura, latura, -latura);
          glVertex3f(latura, -latura, -latura);
          glVertex3f(-latura, latura,  -latura);
          glVertex3f(-latura, -latura, -latura);
          glVertex3f(-latura, latura, latura);
          glVertex3f(-latura, -latura, latura);
          glEnd();
          glBegin(GL_QUADS);
          glVertex3f(-latura, latura, latura);
          glVertex3f(latura, latura, latura);
          glVertex3f(latura, latura, -latura);
```

```
    glVertex3f(-latura, latura, -latura);
    glEnd();
    glBegin(GL_QUADS);
          glVertex3f(-latura, -latura, latura);
          glVertex3f(-latura, -latura,  -latura);
          glVertex3f(latura, -latura, -latura);
          glVertex3f(latura, -latura, latura);
       glEnd();
}
```

Observaţii:

In figura 3.16 sunt capturate două poziţii diferite în timpul rotaţiei. Se constată că acele muchii care intervin şi în poligoanele faţă şi în cele spate sunt desenate cu ambele tipuri de linii. Pentru a se masca acest lucru s-ar putea alege o grosime mai mare pentru linia continuă.

Să analizăm codul aplicaţiei. în primul rând să constatăm că funcţia cub () păstrează orientarea trigonometrică pentru toate patrulaterele. Dacă se schimbă orientarea directă (funcţia glFrontFace () care este comentată în aplicaţie) atunci poligoanele spate vor fi desenate cu linie continuă şi cele faţă cu linie întreruptă. Deoarece funcţia glPoligonMode() nu ne permite specificarea de tipuri diferite de linii pentru poligoanele faţă şi cele spate, s-a desenat cubul de două ori.

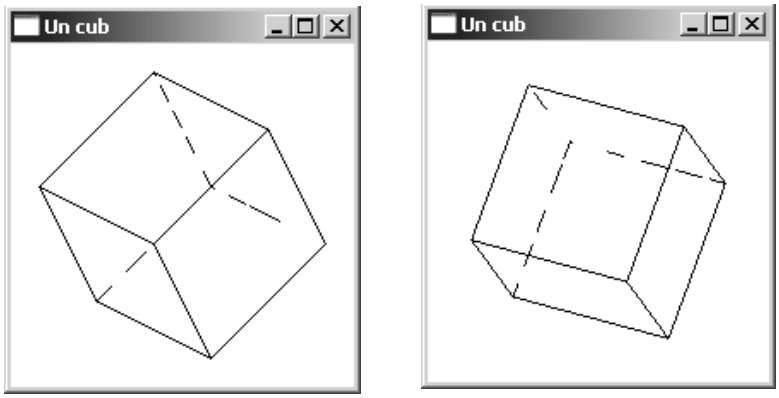


Figura 3.16

Prima dată s-au eliminat poligoanele faţă şi s-a activat linia punctată pentru desenarea poligoanelor spate. A doua oară s-au eliminat poligoanele spate şi s-a dezactivat linia punctată. In ceea ce priveşte linia punctată, ea apare neuniformă, deoarece fiecare muchie cu linie punctată intervine în desenarea a două poligoane.

3.2 Reprezentarea curbelor şi a suprafeţelor curbe

Deoarece curbele nu pot fi descrise cu exactitate prin ecuații lineare, în OpenGL nu există funcții simple pentru desenarea curbelor, așa cum există pentru puncte sau pentru drepte. O modalitate de a desena o curbă este de a o aproxima prin puncte sau prin segmente de dreaptă, cum s-a putut de altfel vedea într-unui din exemplele anterioare care reprezenta cercul cu linie continuă. Reprezentarea cercului este însă o problemă simplă având în vedere că se cunoaşte ecuația sa. O soluție pentru modelarea curbelor ar fi deci de a se memora un set de puncte de pe curbă. Reprezentarea s-ar putea face prin unirea punctelor prin linii în cazul în care acestea sunt suficient de apropiate. Soluția este costisitoare având în vedere numărul mare al punctelor ce ar trebui memorate pentru o curbă. Pentru un set de puncte prin care ar trebui să treacă curba se poate realiza o interpolare utilizând pentru aceasta polinomul Newton. Şi această soluție este costisitoare având în vedere gradul mare al polinomului care rezultă. O altă metodă este de a genera curbele utilizând funcțiile polinomiale Bezier pentru care OpenGL dispune de evaluatori. In acest caz sunt suficiente câteva puncte de control care aproximează forma curbei şi evaluatorii determină puncte succesive pentru curba Bezier respectivă. Reprezentarea se face prin linii. Biblioteca GLU dispune chiar de o interfață pentru reprezentarea curbelor, folosind pentru aceasta ecuațiile NURBS (Non-Uniform Rațional B-Spline). In acest subcapitol se va discuta despre reprezentarea curbelor Bezier şi spline (NURBS). Utilitatea lor în realizarea aplicațiilor grafice este cu atât mai mare cu cât permit programarea interactivă în reprezentarea curbelor. Mutând punctele de control utilizatorul unei aplicații poate modifica forma curbei.

În ceea ce priveşte reprezentarea suprafețelor, problemele şi soluționarea lor sunt asemănătoare cu cele ale curbelor. Pentru reprezentarea unei suprafețe curbe la care nu se cunoaşte o ecuație presupune determinarea unui set de puncte pe suprafață şi aproximarea reprezentării prin triunghiuri. Soluția presupune multă memorie pentru salvarea coordonatelor triunghiurilor şi nu este interactivă. Inginerul francez Pierre Bezier a fost primul care a venit cu o soluție care permite realizarea într-o manieră necostisitoare a unor aplicații interactive, prin inventarea ecuațiilor care-i poartă numele. Interfața OpenGL dispune de posibilitatea de evaluare a polinoamelor Bezier, prin evaluatori. Pentru reprezentarea unei suprafețe Bezier de ordinul 3 este necesară o matrice de 16 puncte care aproximează forma suprafeței. Pe baza acestei matrice de puncte se determină ecuațiile Bezier şi apoi, funcție de gradul de precizie, mai multe sau mai puține puncte de pe suprafață. Ca şi în cazul curbelor, interfața NURBS a bibliotecii GLU pune la dispoziție o metodă de reprezentare a suprafețelor care generalizează curbele spline. Despre aceste metode se va discuta în continuare.

În sfârşit, biblioteca GLU are de asemenea funcții pentru generarea reprezentărilor poligonale ale obiectelor cvadrice. Aceste funcții pot genera de asemenea normalele pentru iluminare şi coordonatele de texturare pentru obiectele cvadrice.

Pentru a se înțelege fundamentul matematic pentru reprezentarea curbelor şi a suprafețelor curbe se recomandă studierea capitolului respectiv din cartea "Sisteme de prelucrare grafică".

3.2.1 Evaluatori

Evaluatorii asigură o modalitate de a specifica puncte de pe o curbă sau de pe o suprafață, utilizând doar punctele de control. Curba sau suprafața pot fi redate apoi cu precizia dorită. Suplimentar, pot fi determinate şi normalele la suprafață. Punctele determinate de evaluatori pot fi utilizate pentru reprezentare în mai multe moduri a unei suprafețe - reprezentarea unui set de puncte de pe suprafață, reprezentarea unei suprafețe wireframe, reprezentarea unei suprafețe cu iluminare şi umbrire. Evaluatorii pot fi utilizați pentru a descrie orice curbă sau suprafață polinomială de orice grad. Interfața GLU care asigură o interfață de nivel ridicat NURBS are la bază tot evaluatorii. Ea conține o mulțime de proceduri complicate, dar redarea finală se realizează tot cu evaluatorii.

Evaluatorii permit specificarea funcțiilor polinomiale de una sau două variabile ale căror valori determină coordonatele vârfurilor primitivelor, coordonatele normalelor, coordonatele texturilor. Pentru oricare din aceste grupuri de valori, poate fi dată o hartă polinomială specificată în termenii de bază Bezier. Odată definite şi activate, hărțile sunt invocate în unul din două moduri posibile. Prima modalitate este de a determina o singură evaluare a fiecărei hărți activate prin specificarea unui punct în domeniul hărții utilizând funcția glEvalCoord(). Această comandă se dă între apelurile glBegin() şi glEnd() astfel că se folosesc primitivele individuale pentru construirea unei porțiuni dintr-o curbă sau o suprafață. A doua metodă este de a specifica o matrice de puncte în spațiu utilizând funcția glEvalMesh(). Fiecare vârf din matricea evaluată este o funcție de polinoamele definite. Funcția glEvalMesh() îşi generează propriile sale primitive şi de aceea nu poate fi plasată între funcțiile glBegin () şi glEnd ().

Interfața pentru evaluatori asigură o bază pentru construirea unui pachet mai general pentru curbe şi suprafețe. Un avantaj al asigurării evaluatorilor în OpenGL în locul unei interfețe NURBS mai complexe este acela că aplicațiile care reprezintă curbe şi suprafețe altele decât NURBS sau care utilizează proprietăți speciale de suprafață au totuşi acces la evaluatori polinomiali eficienți (care pot fi implementați în hardware-ul grafic) fără a fi nevoie să fie convertiți la reprezentări NURBS.

În continuare se va descrie pe scurt suportul teoretic pentru curbele şi suprafețele Bezier, pentru a înțelege mai bine semnificația evaluatorilor.

Evaluatori uni-dimensionali

O curbă parametrică polinomială este descrisă printr-un vector cu trei elemente:

P(u) = [x(u) y(u) z(u)]

Parametrul u ia valori de la 0 la 1. Pentru fiecare valoare a lui u se obține un punct P(u) aparținând curbei.

Utilizând un set de puncte de control se poate obține o curbă aproximativă folosind un set de funcții polinomiale obținute pe baza coordonatelor acestor puncte de control. Vom considera cazul general al curbelor Bezier de orice grad.

Pentru un set de (n+1) puncte de control care definesc vectorii:

Pk = (xk yk zk) k=0, 1, 2, ...n

se poate aproxima polinomul vectorial P(u), care reprezintă trei ecuații parametrice (x(u), y(u), z(u)) pentru curba care trece prin punctele de control pk

$$P(u) = \sum_{k=0}^{n} pk \cdot B_{k,n}(u)$$

k=0

Fiecare polinom $B_{k,n}$ este o funcție polinomială definită ca

$$B_{k,n}(n) = C(n,k) \cdot u^k \cdot (1-u)^{k-1}$$

iar C(n, k) reprezintă coeficienţii binomiali

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

Relaţiile coordonatelor pot fi scrise în formă explicită ca un set de ecuaţii parametrice pentru coordonatele curbelor:

$$x(u) = \sum_{i=0}^{n} x_k B_{k,n}(u)$$

$$y(u) = \sum_{i=0}^{n} y_k B_{k,n}(u)$$

$$z(u) = \sum_{i=0}^{n} z_k B_{k,n}(u)$$

Evaluatori bi-dimensionali

Ecuaţiile parametrice pentru suprafeţe sunt date folosind doi parametrii, u şi v. Poziţia coordonatelor unui punct de pe suprafaţă este reprezentată printr-un vector:

S(u, v)=[x(u, v) y(u, v) z(u,v)] (3.9)

Pentru reprezentarea unor suprafeţe specificate prin punctele de control se folosesc două seturi de curbe Bezier. Funcţiile pentru vectorii parametrici care definesc suprafaţa Bezier se obţin ca produs cartezian al funcţiilor de amestec al celor două seturi de curbe.

$$S(u,v) = \sum_{i=0}^{m} \sum_{k=0}^{n} p_{i,k} \cdot B_{i,m(u)} \cdot B_{k,n(v)}$$

unde pi,k specifică un punct de control din cele (m+l)x(n+l) puncte de control care definesc suprafaţa.

3.2.2 Curbe Bezier

Exemplul 1

In exemplul următor se arată modul de utilizare al evaluatorilor pentru reprezentarea unei curbe Bezier.

```
/* curbe_Bezier.c
Programul utilizează evaluatorii pentru determinarea punctelor de pe curba Bezier */
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
```

```
GLfloat ctrlpoints[4][3] = {
   { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
   {2.0, 4.0, 0.0}, {4.0, -4.0, 0.0}};

void myinit(void)
{
   glClearColor(1.0, 1.0, 1.0, 1.0);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
   glShadeModel(GL_FLAT);
        glLineStipple (1, 0x0F0F);
}

void CALLBACK display(void)
{
   int i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
   glBegin(GL_LINE_STRIP);
   for (i = 0; i <= 30; i++)
      glEvalCoord1f((GLfloat) i/30.0);
   glEnd();
   glPointSize(5.0);
   glColor3f(1.0, 0.0, 0.0);
   glBegin(GL_POINTS);
   for (i = 0; i < 4; i++)
      glVertex3fv(&ctrlpoints[i][0]);
   glEnd();
        glEnable(GL_LINE_STIPPLE);
        glColor3f(1.0, 0.0, 1.0);
        glBegin (GL_LINE_LOOP);
        for (i=0; i<4; i++)
                glVertex3fv(&ctrlpoints[i][0]);
        glEnd();

   glFlush();
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
      5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
   else
```

```
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
        5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("Curbe Bezier");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

Observații:

In figura 3.17 se poate vedea rezultatul rulării programului. Din program se poate vedea că funcția glMaplf() descrie caracteristicile curbei (puncte de control, gradul polinoamelor de amestec, intervalul pentru parametrul w), cu funcția glEnable(GL_MAPl_VERTEX_3) se activează un anumit tip de evaluatori, iar apoi se obțin puncte pe curbă cu funcția glEvalCoordlf().

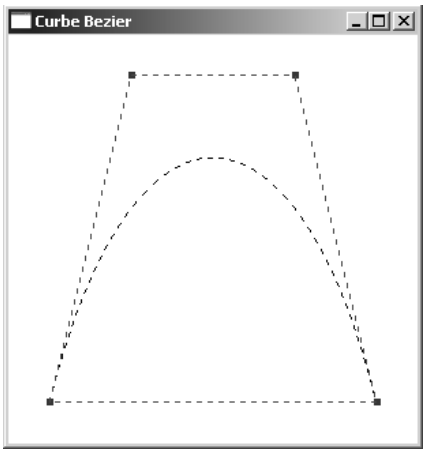


Figura 3.17

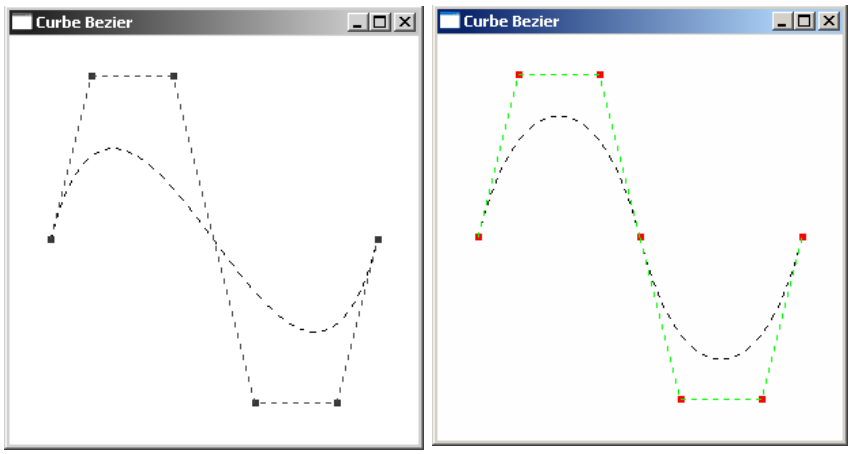


Figura 3.18

Exemplul 2

O problemă care se pune atunci când se lucrează cu curbe Bezier, este realizarea unor curbe care sunt aproximate de mai mult de patru puncte de control. Ecuațiile curbei permit în acest caz specificarea mai multor puncte de control. Dar implicit în cazul acesta creşte şi gradul curbei. Dacă se măreşte doar numărul punctelor de control şi nu se actualizează şi gradul curbei în funcția glMaplf(), rezultatul va fi că se va reprezenta curba doar pentru primele patru puncte de control din tabloul punctelor de control. Exemplul anterior este modificat pentru reprezentarea unei curbe aproximată de 6 puncte de control, (figura 3.18). Se vor da doar funcțiile myinit() şi display() în care s-au făcut modificări față de exemplul anterior.

```
GLfloat ctrlpoints[6][3] = {
   { -4.0, 0.0, 0.0}, { -3.0, 4.0, 0.0}, { -1.0, 4.0, 0.0},
   {1.0, -4.0, 0.0}, {3.0, -4.0, 0.0}, {4.0, 0.0, 0.0}};
void myinit(void)
{
   glClearColor(1.0, 1.0, 1.0, 1.0);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 6, &ctrlpoints[0][0]);
/*funcția defineşte caracteristicile curbei:
-   tipul punctelor de control date in vectorul ctrlpoints, si al
-   datelor de ieşire generate de funcția de evaluare glEvalCoordlf
-   valorile extreme luate de parametrul u (0 si 1 in acest caz)
-   numărul coordonatelor date pentru fiecare punct de control, in tabloul
    ctrlpoints
-   numărul punctelor de control pe baza cărora se va determina ordinul curbei
    (număr puncte-1) -vectorul punctelor de control
*/
glEnable (GL_MAP1_VERTEX_3) ;        //se validează un anumit tip de
evaluare
gl ShadeModel (GL_FLAT) ;          //umbrire constanta pe poligoane
        glLineStipple (1, 0x0F0F);
}
void CALLBACK display(void)
{
   int i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
   glBegin(GL_LINE_STRIP);
   for (i = 0; i <= 30; i++)
   glEvalCoord1f((GLfloat) i/30.0);
   glEnd();
   glPointSize(5.0);
```

```
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POINTS);
    for (i = 0; i < 6; i++)
    glVertex3fv(&ctrlpoints[i][0]);
    glEnd();
    glEnable(GL_LINE_STIPPLE);
    glColor3f(1.0, 0.0, 1.0);
        glBegin (GL_LINE_STRIP);
        for (i=0; i<6; i++)
                glVertex3fv(&ctrlpoints[i][0]);
        glEnd();

    glFlush();
}
```

Exemplul 3

În exemplul următor se desenează o curbă (figura 3.19) care trece prin aceleaşi puncte de control ca şi curba din exemplul anterior. De data aceasta însă se folosesc două curbe Bezier de gardul 3 cu continuitate geometrică (G^1) de ordinul 1 în punctul de contact. Punctul de contact al celor două curbe este un punct de control introdus suplimentar față de exemplul anterior. El este punct terminal pentru prima curbă şi punct inițial pentru cea de a doua curbă. Pentru a se obține continuitate geometrică de ordinul 0 (G°) în punctul de contact este necesar ca ultimul punct de control al primei curbe să coincidă cu primul punct de control al celei de a doua curbe (punctul (0, 0, 0) în exemplul nostru). Pentru a avea continuitate geometrică de ordinul 1 este necesar ca în punctul de contact tangentele la cele două curbe să fie coliniare. O caracteristică a curbelor Bezier este că primele două puncte de control şi ultimele două puncte de control sunt extremitățile unor drepte care sunt tangente la curba Bezier. Condiția de continuitate geometrică de ordinul 1 revine la condiția ca penultimul punct de control al primei curbe, punctul comun şi al doilea punct de control al celei de a doua curbe să fie puncte coliniare. In cazul nostru aceste puncte sunt: {-1.0, 4.0, 0.0}, {0.0, 0.0, 0.0}, {l.0, -4.0, 0.0}. Toate aceste puncte sunt puncte coliniare şi se află pe dreapta de ecuație y = -4x.

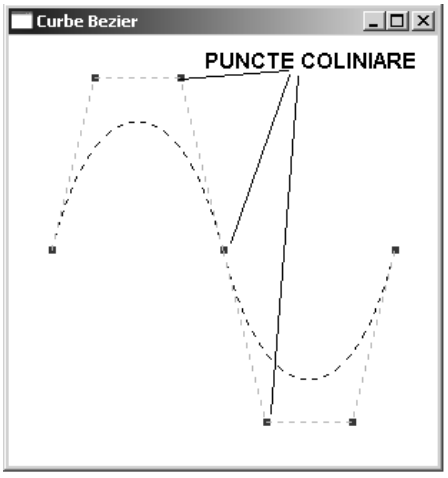


Figura 3.19

Şi pentru acest exemplu se furnizează doar partea de cod modificată față de exemplu anterior.

```
GLfloat ctrlpointsl[4][3] = {
//sunt date coordonatele celor 4 puncte de control pentru prima curbă
{-4.0, 0.0, 0.0},{-3.0, 4.0, 0.0},{-1.0, 4.0, 0.0},{0.0, 0.0, 0.0}};
GLfloat ctrlpoints2[4][3] = {
//sunt date coordonatele celor 4 puncte de control pentru a doua curbă
{0.0, 0.0, 0.0}, {1.0, -4.0, 0.0}, {3.0, -4.0, 0.0},{4.0, 0.0, 0.0}};
void myinit(void) {
glClearColor (1.0, 1.0, 1.0, 1.0); //culoarea background-ului
glShadeModel (GL_FLAT) ; //umbrire constanta pe poligoane
glLineStipple (1, 0x0F0F) ; //stilul liniei punctate }
void CALLBACK display(void) {
int i ;
glClear(GL_COLOR_BUFFER_BIT);
glColor3 f (0 . 0, 0.0, 0.0); //culoarea curenta de desenare
glMaplf(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpointsl[0][0]);
glEnable (GL_MAP1_VERTEX_3) ; //se validează un anumit tip de evaluare
glBegin (GL_LINE_STRIP) ; //se desenează curba prin segmente de dreapta
for (i = 0; i <^= 30; i++)
glEvalCoordlf ( (GLfloat) i/30 . 0) ; //pentru cele 30 vârfuri determinate de
// funcția glEvalCoordlf glEnd() ;
glMaplf (GL_MAP1_VERTEX__3, 0.0, 1.0, 3, 4, &ctrlpoints2[0][0]);
glBegin (GL_LINE_STRIP) ; //se desenează curba prin segmente de dreapta
for (i = 0; i <= 30; i++)
glEvalCoordlf ( (GLf loat) i/30.0) ; //pentru cele 30 vârfuri determinate de
//funcția glEvalCoordlf glEnd() ;
/*Se afişează punctele de control. */
glPointSize (5 . 0) ; //de dimensiune 5
glColor3f (1.0, 0.0, 0.0); //culoare roşie
glBegin (GL_POINTS) ; //pentru prima curba
for (i = 0; i < 4; i + +)
glVertex3fv(&ctrlpointsl[i][0]);
glEnd() ;
glBegin (GL_POINTS) ; //pentru a doua curba
for (i = 0; i < 4; i++)
glVertex3fv(&ctrlpoints2[i] [0] ) ;
glEnd() ;
//se unesc punctele de control
glEnable (GL_LINE_STIPPLE) ; //linie punctată
glColor3f(0.0, 1.0, 0.0);
glBegin (GL_LINE_STRIP) ; //pentru prima curba
```

```
for (i=0; i<4; i++)
glVertex3fv(&ctrlpointsl[i][0]);
glEnd() ;
glBegin (GL_LINE_STRIP) ;        //pentru a doua curba
for (i=0; i<4; i++)
glVertex3fv(&ctrlpoints2[i][0]); glEnd();
glFlush() ;
}
```

3.2.3 Suprafețe Bezier

Următoarele două exemple arată modul de utilizare al evaluatorilor bidimensionali pentru reprezentarea suprafețelor.

Exemplul 1

In figura 3.20 se poate vedea rezultatul rulării acestui exemplu.

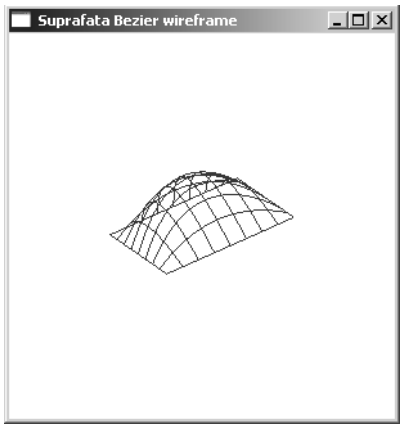


Figura 3.20

```
/* Wire_Bezier.c
Programul realizează o reprezentare Wireframe
pentru o suprafața Bezier, utilizând evaluatorul bidimensional EvalCoord2f
*/

#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
GLfloat ctrlpoints[4][4][3] = {
```

```
    {{-1.5, -1.5, -4.0}, {-0.5, -1.5, -4.0},
    {0.5, -1.5, -4.0}, {1.5, -1.5, -4.0}},
    {{-1.5, -0.5, -4.0}, {-0.5, -0.5, -2.0},
        {0.5, -0.5, -2.0}, {1.5, -0.5, -4.0}},
    {{-1.5, 0.5, -4.0}, {-0.5, 0.5, -2.0},
        {0.5, 0.5, -2.0}, {1.5, 0.5, -4.0}},
    {{-1.5, 1.5, -4.0}, {-0.5, 1.5, -4.0},
        {0.5, 1.5, -4.0}, {1.5, 1.5, -4.0}}
};

void myinit(void)
{
  glClearColor (1.0, 1.0, 1.0, 1.0);        // culoarea background-ului
      /* funcția glMap2f definește caracteristicile suprafeței Bezier:
      - tipul punctelor determinate de funcția glEvalCoord2f
      - intervalul de variație al parametrului u (0 -l in acest caz)
      - intervalul valorilor in tabloul ctrlpoints intre doua puncte de control
        pe direcția u
      - numărul punctelor de control pe direcția u
      - intervalul de variație al parametrului v (0 -l in acest caz)
      - intervalul valorilor in tabloul ctrlpoints intre doua puncte de control
        pe direcția v
      - numărul punctelor de control pe direcția v –
      - tabloul punctelor de control
*/

glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
0, 1, 12, 4, &ctrlpoints[0][0][0]);
glEnable (GL_MAP2_VERTEX_3 ) ;        // validarea tipului de evaluare
// GL_MAP2_VERTEX_3 glMapGrid2 f(20,  0.0,  1.0,  20,  0.0,  1.0);
// intervalele de eşantionare
// a suprafeței pentru parametrii u si v. Fiecare parametru variază între 0 şi 1 şi
sunt  // 20 de intervale de eşantionare. Deci valorile pentru care se calculează puncte
pe    //suprafață sunt pentru u şi v variind cu un pas de 1/20. }
void CALLBACK display(void) {
int i, j; glClear(GL_COLOR_BUFFER_BIT);
// se foloseşte si buffer de refresh si buffer de adâncime
glColor3f (1.0,  0.0,  0.0);  //culoarea curenta
glLoadIdentity() ;      // pentru a nu aplica transformări geometrice
// la fiecare redesenare a ferestrei glRotatef (-85.0,  1.0,  1.0,  1.0);
//rotație in jurul axei (1, 1, 1) glTranslatef (0,  0,  4) ;
// urmează desenarea wireframe a suprafeței
//fiecare patch (8X8 patch-uri) este desenat //dintr-o linie frântă în 30 de
segmente      //de dreapta for  (j  =  0;  j  <=  8; j++)  {
glBegin(GL_LINE_STRIP);
for  (i  =  0;  i  <=  30;  i++)
glEvalCoord2f((GLfloat)i/30.0,   (GLfloat)j/8.0);
```

```
// evaluează un punct pe suprafața pentru valorile u si v ale ale parametrilor
glEnd();
glBegin(GL_LINE_STRIP); for (i = 0; i <= 30; i++)
glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
glEnd(); }
glFlush() ; }

void CALLBACK myReshape(GLsizei w# GLsizei h) {
if (!h) return;
glViewport(0, 0, w, h) ;
glMatrixMode(GL_PROJECTION);
glLoadIdentity(); if (w <= h)
glOrtho(-4.0/4.0,-4.0*(GLfloat)h/(GLfloat)w,
4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
else
glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0,  4.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); }
int main(int  argc,  char**  argv) {
auxInitDlsplayMode(AUX_SINGLE | AUX_RGB);
auxInitPosition (0, 0, 300,300); auxInitwindow ("Suprafața Bezier
wireframe");
myinit();
auxReshapeFunc  (myReshape) ;
auxMainLoop(display);
return(0);
}
```

Observații:

În exemplul de mai sus se poate remarca asemănarea în utilizarea evaluatorilor unidimensionali şi bidimensionali. Pentru a fi mai clar felul în care se declară tabloul punctelor de control se recomandă afişarea acestei suprafețe fără a se aplica cele două transformări geometrice (glRotatef, glTranslatef). De asemenea pentru a se obține rezultate mai bune la reprezentare se recomandă încercarea de a reprezenta suprafața cu mai mult de 8X8 patch-uri.

Exemplul 2

În exemplul următor se reia reprezentarea suprafeței cu punctele de control specificate anterior dar utilizând celălalt tip de evaluator bidimensional glEvalMesh2. Se poate constata, că dacă în exemplul anterior evaluatorul era folosit în corpul glBegin/glEnd, în acest caz nu mai este necesar acest lucru. Motivul este următorul: în primul exemplu evaluatorul furnizează coordonate de vârfuri, iar în al doilea caz se şi generează poligoanele care formează suprafața. Deoarece modul de reprezentare al poligoanelor permite reprezentarea în puncte, linii sau cu umplere (GL_POINT,

GL__LINE, GL_FILL) în acest caz suprafața poate fi reprezentată în toate aceste modalități (figura 3.21).

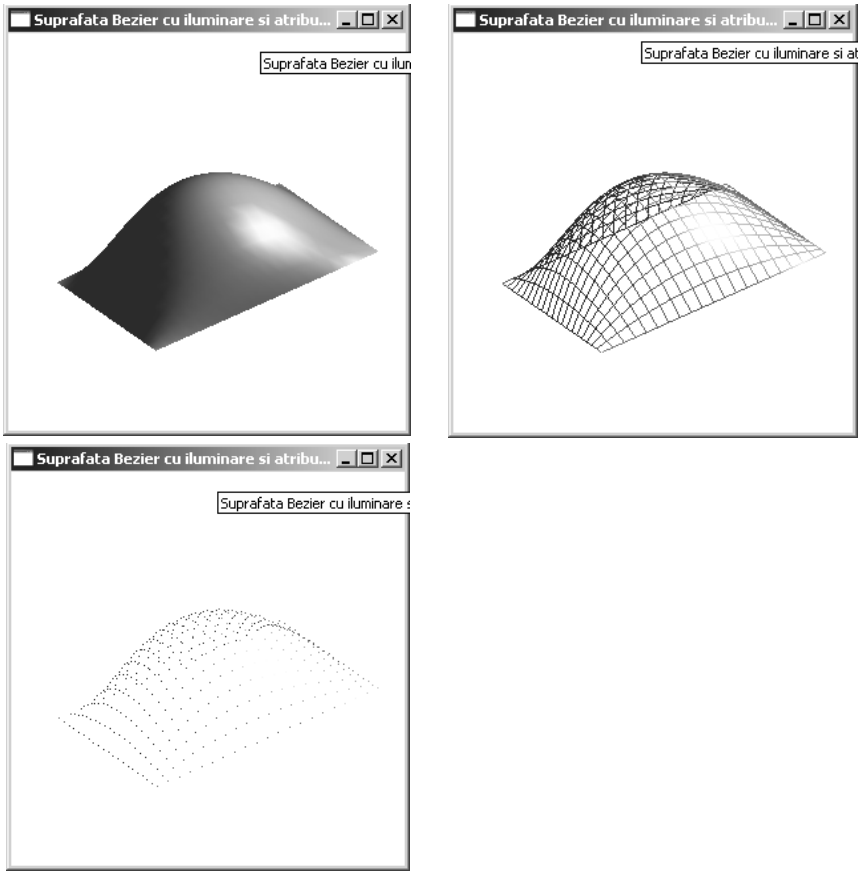


Figura 3.21

```
/* Solid_Bezier.c
Programul afişează o suprafaţa Bezier folosind
evaluatori bidimensionali
*/
#include "glos.h"
#include <GL/gl.h> #include <GL/glu.h> #include <GL/glaux.h>
void myinit(void);
void initlights(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
// tabloul ctrlpoints defineşte cele 16 puncte de control ale suprafeței
GLfloat ctrlpoints[4][4][3] = {
   {{-1.5, -1.5, -4.0}, {-0.5, -1.5, -4.0},
         {0.5, -1.5, -4.0}, {1.5, -1.5, -4.0}},
   {{-1.5, -0.5, -4.0}, {-0.5, -0.5, -2.0},
         {0.5, -0.5, -2.0}, {1.5, -0.5, -4.0}},
   {{-1.5, 0.5, -4.0}, {-0.5, 0.5, -2.0},
         {0.5, 0.5, -2.0}, {1.5, 0.5, -4.0}},
   {{-1.5, 1.5, -4.0}, {-0.5, 1.5, -4.0},
```

```
        {0.5, 1.5, -4.0}, {1.5, 1.5, -4.0}}
};
// setări pentru iluminarea suprafeței
void initlights(void)
{
  GLfloat ambient[] = { 1.0, 0.6, 0.0, 1.0 };
  GLfloat position[] = { 2.0, 2.0, 2.0, 1.0 };
  GLfloat mat_diffuse[] = { 1.0, 0.6, 0.0, 1.0 };
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_POSITION, position);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}

void CALLBACK display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();     //salvare în stivă matrice curentă de modelare
                      //trei transformări de modelare
                //rotația şi translația poziționează suprafața în fereastra de
vizualizare                    // iar scalarea măreşte dimensiunea suprafeței de 1,7 ori
        glRotatef(-85.0, 1.0, 1.0, 1.0);
        glTranslatef(0, 0, 6);
        glScalef(1.7,1.7,1.7);
        glEvalMesh2(GL_POINT, 0, 20, 0, 20);   //specifica modul
// de redare al poligoanelor (GL_FILL, GL_POINT, GL_LINE,
// si intervalele de eşantionare a suprafeței pentru u si v
glPopMatrix() ;//scoate din stivă matricea de modelare
glFlush() ; }
void myinit(void)
{
glClearColor (1.0, 1.0, 1.0, 1.0);  //culoarea background-ului
glEnable (GL_DEPTH_TEST) ;          //se activează ascunderea
suprafețelor
/*      funcția glMap2f defineşte caracteristicile suprafeței Bezier
        - tipul punctelor determinate de funcția glEvalCoord2f
        - intervalul de variație al parametrului u (0 -l in acest caz)
        - intervalul valorilor in tabloul ctrlpoints intre doua puncte de control
          pe direcția u
        - numărul punctelor de control pe direcția u
        - intervalul de variație al parametrului v (0 -l in acest caz)
        - intervalul valorilor in tabloul ctrlpoints intre doua puncte de control
```

```
            pe direcția v
        -   numărul punctelor de control pe direcția v
        -   tabloul punctelor de control
    */

    glMap2f(GL_MAP2_VERTEX_3, 0,  1, 3, 4,
    0,  1, 12, 4,   &ctrlpoints[0][0][0]);
    glEnable (GL_MAP2_VERTEX_3 ) ;          // validarea tipului de evaluare
    //GL_MAP2_VERTEX_3 glEnable(GL_AUTO_NORMAL);
    glEnable (GL_NORMALIZE) ;        // pentru iluminare
    glMapGrid2f (20,   0.0,   1.0,   20,   0.0,   1. 0); //intervalele de eşantionare
    // a suprafeței pentru parametrii u si v
    //20 de intervale pentru fiecare parametru.
    //fiecare parametru variază între 0 şi 1 initlights () ;
    /* doar daca se doreşte reprezentarea cu iluminare */ }

    void CALLBACK myReshape(GLsizei w, GLsizei h) {
    if (!h) return;
    glViewport(0, 0, w, h) ;
    glMatrixMode (GL_PROJECTION) ;
    glLoadIdentity();
    if (w <= h)
    glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
    4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0) ;
    else
    glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
    4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity(); }
    int main(int argc, char** argv) {
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("Suprafața Bezier cu iluminare si atribut de umplere pe
poligoane");   myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0); }
```

Observații:

Ca o concluzie, în primul exemplu suprafața wireframe este reprezentată prin evaluarea unor puncte pe curbe şi prin reprezentarea acestor curbe. In al doilea exemplu evaluatorii determină vertex-urile unor poligoane şi reprezintă aceste poligoane în oricare din modurile posibile pentru poligoane.

3.2.4 Interfața NURBS

Aşa cum am arătat, în cazul curbelor Bezier pentru forme mai complicate de curbe sunt necesare mai multe puncte de control. Pentru a aproxima forma unei curbe pe baza acestor puncte, în exemplele anterioare fie am crescut gradul polinoamelor de amestec, fie am asamblat curba din mai multe bucăți. O soluție pentru construirea curbelor, care oferă o mai mare flexibilitate în modelare este utilizarea interfeței NURBS (Non-Uniform Rațional B-Spline) din biblioteca GLU. Curbele NURBS sunt de fapt o metodă generalizată care permite descrierea matematică atât a curbelor Bezier cât şi a altor curbe. Un alt avantaj al acestor curbe este tipul de continuitate în punctele de control. Aşa cum s-a văzut, în cazul curbelor Bezier ordinul polinoamelor de amestec creşte odată cu numărul punctelor de control. Pentru asamblarea mai multor curbe programatorul este cel care trebuie să aleagă astfel punctele de control ca să se obțină continuitate de ordinul 1 în punctele de control. In cazul curbelor spline, creşterea numărului punctelor de control nu conduce la creşterea gradului polinoamelor de amestec. Gradul polinoamelor este controlat de un alt parametru. Indiferent de numărul punctelor de control, în cazul curbelor spline se asigură continuitate de ordinul 2 în noduri (aceasta înseamnă că în noduri atât derivatele de ordin întâi cât şi derivatele de ordin 2 sunt egale).

Curbele B-spline sunt o clasă de curbe spline foarte utile în aplicațiile grafice. Pentru un set de (n+1) puncte de control ph (k variază de la 0 la n) se pot determina punctele care aparțin unei curbe B-spline aproximată prin punctele de control date, utilizând ecuația parametrică următoare:

$$P(y) = \sum_{i=0}^{n} pk * Nk,t(u)$$

Se observă că gradul polinoamelor de amestec (care este t-1) este independent de numărul punctelor de control (n), ceea ce reprezintă un avantaj al curbelor spline.

Funcțiile de amestec A^, sunt polinoame de gradul (t-l). O metodă pentru definirea polinomului folosit pentru funcțiile de amestec este de a le defini recursiv față de diferitele subintervale din domeniul de variație al parametrului u. Domeniul de variație al parametrului u nu mai este în intervalul [0, 1] ci el este dependent de numărul punctelor de control şi de alegerea făcută pentru t. Astfel u variază în intervalul:

0 - (n-t+2)

Numărul total al segmentelor de curbă va fi deci de (n-t+2). Fiecare din aceste curbe va fi controlată (ca şi formă) de t puncte de control din cele (n+l) puncte de control.

Dacă se consideră (n+t) subintervale, se definesc funcțiile de amestec în mod recursiv:

$$N_{k,1}(u) = \begin{cases} 1, & r_k \le u \prec r_{k+1} \\ 0, & altfel \end{cases}$$

$$N_{k,1}(u) = \frac{u - r_k}{r_{k+t-1} - r_k} N_{t,n-1}(u) + \frac{r_{k+1} - u}{r_{k+1} - r_{k+1}} N_{k+1,t-1}(u)$$

Se observă că polinomul de amestec corespunzător unui punct de control se calculează folosind polinoamele de amestec corespunzătoare acestui punct şi celui

următor dar de grad mai mic cu 1. Deci pentru fiecare punct de control k, se calculează funcțiile de amestec plecând de la t=l. Cu cât t este mai mic se fac mai puține calcule.

Deoarece numitorii, în calculele recursive pot avea valoarea 0, această formulare consideră că se evaluează prin 0 rapoartele 0/0.

Pozițiile definite de xj- pentru subintervalele lui u sunt referite ca valori nodale, iar punctele corespunzătoare lor pe o curbă B-spline sunt numite noduri. Cele (n-t+2) segmente de curbă vor fi cuprinse între aceste noduri. In noduri există continuitate de ordinul 2, asigurată de modul în care sunt definite curbele B-spline.

Spre exemplu, pentru n=6 şi t=4 curbele B-spline uniforme pot fi caracterizate în felul următor:

Tabelul 3.1

numărul punctelor de control:	n+1=7
gradul polinoamelor de amestec	t -1 = 3
numărul segmentelor de curbă	n-t + 2 = 4
intervalul de variație al lui u	[0,4]
numărul nodurilor	5
valorile nodale	0,1,2,3,4
numărul subintervalelor	n + t= 10

Valorile nodale pot fi definite în diverse feluri. O distanțare uniformă a valorilor nodale se obține prin setarea lui rd egal cu j. Trebuie spus .că în cazul în care valorile nodale sunt distanțate uniform curbele B-spline se numesc curbe B-spline uniforme. O altă metodă de definire uniformă a intervalelor este de a seta valorile nodale astfel:

$$rj = \begin{cases} 0 & pentru \quad j < t \\ j - t + 1 & pentru \quad t \le j \le n \\ n - t + 2 & pentru \quad j > n \end{cases}$$

Pentru grad 2 (t=3) folosind cinci puncte de control (n«4), subintervalele definite ca în relațiile (3.14), parametrul u variază în intervalul de la 0 la 3, cu valori nodale de la r0 la r7 având valorile:

r0 = 0
ri = 0
r2 = 0
r3 = 1
r4 = 2
r5 = 3
r6 = 3
r7 = 3

n = 4, t = 3, j∈ [0,7], u∈[0,3]

r0 = 0; ri = 0; r2 = 0; r3 = 1; r4 = 2; r5 = 3; r6 = 3; r7 = 3;

Numărul segmentelor de curbă în acest caz va fi 3: Q1, Q2, Q3. Curba Q1 va fi controlată de punctele P0, P1, P2 . Curba Q2 va fi controlată de punctele P1, P2, P3. Curba Q3 va fi controlată de punctele P2, P3, P4.

Primul punct de control are influență asupra formei curbei spline doar pentru valorile lui u cuprinse în intervalul (0,1). în felul acesta se pot localiza uşor punctele în care apar modificări. Dacă utilizatorul modifică poziția primului punct de control, forma curbei se va modifica în apropierea acestui punct fără a afecta celelalte porțiuni ale curbei. Pentru valorile date pentru n şi t se poate deci realiza următoare schemă:

Tabelul 3.2

u	[0,1]	[1, 2]	[2,3]
Curba	Q1	Q2	Q3
Polinoamele care controlează	N0, 3 N1,3 N2,3	N1, 3 N2,3 N3,3	N2,3 N3,3 N4.3
Punctele care controlează forma curbei	P0 P1 P2	P1 P2 P3	P2 P3 P4

Folosirea curbelor B-spline prezintă avantajul că utilizatorul poate specifica orice număr de puncte de control fără a fi nevoie pentru aceasta să crească gradul curbei. O funcție de grad 3 (funcție cubică în care t=4) poate fi folosită pentru diferite forme de curbe, fără a fi nevoie pentru aceasta de a compune curba din segmente de curbă ca în cazul curbelor Bezier. Pentru a modifica forma curbei poate fi adăugat orice număr de puncte de control. Nu se aleg valori mai mici pentru t deoarece în cazul în care, spre exemplu, t=l (funcții de amestec de grad 0) curbele spline sunt discontinue pe intervalele de variație ale lui u. în cazul în care t=2 (funcții de amestec de grad 1) se obține continuitate de ordinul 0 în noduri (curba nu este netedă). In cazul în care t=3 (funcții de amestec de ordin 2) se obține continuitate de ordinul 1 în punctele de joncțiune. Pentru t=4 (funcții de amestec de grad 3) se obține continuitate de ordinul 2 în noduri.

La fel ca şi în cazul curbelor Bezier, specificarea mai multor puncte de control, în poziții apropiate, va conduce la "atragerea" curbei spre poziția respectivă. De asemenea, pentru obținerea unei curbe închise trebuie să se specifice primul şi ultimul punct de control având aceleaşi coordonate. De asemenea, curbele spline se aşează în interiorul poligonului convex definit de punctele de control.

3.2.5 Curbe NURBS

Exemplul 1

În acest exemplu se vor utiliza aceleaşi puncte de control care s-au utilizat în exemplul 2 de la curbe Bezier. Ceea ce trebuie pus în vederea programatorilor este felul în care se calculează numărul nodurilor şi valorile acestora. De asemenea este important felul în care se stabilesc caracteristicile curbei B-spline. în acest exemplu s-au utilizat 6 puncte de control deci n=5. Deoarece s-a dorit continuitate de ordinul 2 s-au utilizat polinoame de gradul 3 deci t=4. Ca urmare a relațiilor 3.14 numărul intervalelor nodale este n+t=9 (deci 10 noduri de la r0 la r9). Valorile nodurilor calculate conform relațiilor 3.14 sunt:

r0=0; ri=0; r2=0; r3=0; r4=l; r5=2; r6=3; r7=3; r8=3; r9=3

Trebuie subliniat că dacă valorile nodale şi numărul acestora sunt alese aleator, rezultatele nu sunt controlabile. Forma curbei pentru valorile nodale alese ca mai sus, va fi asemănătoare cu cea a curbei Bezier pentru aceleaşi puncte de control (figura 3.22).

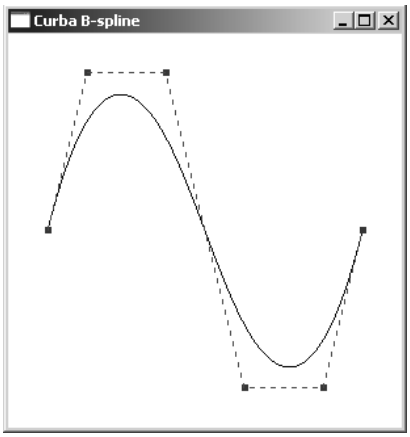


Figura 3.22

```
/* Curba_spline.c
Programul utilizează biblioteca GLUT
pentru redarea unei curbe spline */

#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
GLUnurbsObj *theNurb;  // curba este un obiect de tipul GLUnurbsObj
void myinit(void)
{
   glShadeModel (GL_FLAT);
        glLineStipple (1, 0x0F0F);  //stilul liniei întrerupte
        theNurb = gluNewNurbsRenderer();  // obiectul de tip GLUnurbsObj
   gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 10.0);
```

```
}
/*  Funcția afişează o curba B-spline.
 */
void CALLBACK display(void)
{       int i;
   GLfloat ctrlpoints[6][3] =
        {{ -4.0, 0.0, 0.0}, { -3.0, 4.0, 0.0}, { -1.0, 4.0, 0.0},
   {1.0, -4.0, 0.0}, {3.0, -4.0, 0.0}, {4.0, 0.0, 0.0}};
        GLfloat knots[10] = {0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 3.0, 3.0, 3.0, 3.0};
        glClearColor (1.0, 1.0, 1.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
   glColor3f (0.0, 0.0, 0.0);
   gluBeginCurve(theNurb);
   gluNurbsCurve(theNurb,
      10, knots,
      3,
      &ctrlpoints[0][0],
      4,
      GL_MAP1_VERTEX_3);
   gluEndCurve(theNurb);
   glPointSize(5.0);
   glColor3f(1.0, 0.0, 0.0);
   glBegin(GL_POINTS);
   for (i = 0; i < 6; i++)
      glVertex3fv(&ctrlpoints[i][0]);
   glEnd();
        glEnable(GL_LINE_STIPPLE);
        glColor3f(1.0, 0.0, 1.0);
        glBegin (GL_LINE_STRIP);
        for (i=0; i<6; i++)
        glVertex3fv(&ctrlpoints[i][0]);
        glEnd();
   glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
      5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
   else
   glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
      5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
```

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("Curba B-spline");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

Exemplul 2

Pentru a exemplifica faptul că interfața NURBS permite obținerea de curbe de forme complicate, fără a fi nevoie pentru aceasta să se crească gradul polinoamelor de amestec sau să se calculeze poziția punctelor de control pentru a asigura continuitatea dorită în punctele de contact, vom mai prezenta un exemplu. Se vor da doar funcțiile care au suferit modificări față de exemplul anterior. În figura 3.23 se poate vedea rezultatul rulării programului.

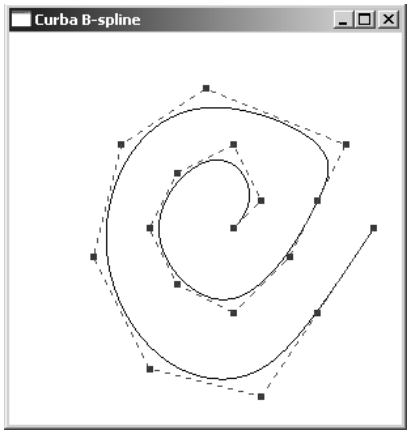


Figura 3.23

```
/* Funcția afișează o curba B-spline. */
void CALLBACK display(void)
{     int i;
// 6 puncte de control n=5
GLfloat  ctrlpoints[17][3]   =
GLfloat ctrlpoints[17][3] =
        {{ 1.0, 0.0, 0.0}, { 2.0, 1.0, 0.0}, { 1.0, 3.0, 0.0},
        { -1.0, 2.0, 0.0}, { -2.0, 0.0, 0.0}, { -1.0, -2.0, 0.0},
        { 1.0, -3.0, 0.0}, { 3.0, -1.0, 0.0}, { 4.0, 1.0, 0.0},
        { 5.0, 3.0, 0.0}, { 0.0, 5.0, 0.0}, { -3.0, 3.0, 0.0},
```

```
        { -4.0, -1.0, 0.0}, { -2.0, -5.0, 0.0}, { 2.0, -6.0, 0.0},
        { 4.0, -3.0, 0.0}, { 6.0, 0.0, 0.0}};
// 21 noduri, n+t=20subintervale, valorile sunt calculate după relația de mai sus
GLfloat knots[21] = {0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
                9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 14.0, 14.0, 14.0};
glClearColor(1.0,1.0,1.0, 1.0); //culoarea background-ului
glClear(GL_COLOR_BUFFER_BIT);
glColor3f  (0.0,  0.0,  0.0);  //culoarea curenta de desenare
// începe corpul de redare al curbei Spline gluBeginCurve(theNurb);
gluNurbsCurve (theNurb,     // pointer obiect NURBS
21,  knots,     //număr noduri, tablou noduri
3 ,      // intervalul de valori dintre doua puncte de control consecutive
&ctrlpoints [ 0 ] [ 0 ] ,          // vector puncte de control
4,       // ordinul curbei, t=4
GL_MAP1_VERTEX_3 ) ;  // tip evaluator
gluEndCurve(theNurb);
/* Se afişează punctele de control. */
glPointSize(5. 0) ;     //de dimensiune 5
glColor3f (1.0,  0.0,  0.0);  //culoare roşie
glBegin(GL_POINTS) ;
for  (i  =  0;  i  <  17;  i++)
glVertex3fv(&ctrlpoints[i][0]);
glEnd() ;
//se unesc punctele de control
glEnable(GL_LINE_STIPPLE) ;
glColor3f(1.0, 0.0, 1.0);
glBegin (GL_LINE_STRIP);
for (i=0; i<17; i + +)
glVertex3fv(&ctrlpoints[i][0]);
glEnd() ;
glFlush() ; }
void CALLBACK myReshape(GLsizei w, GLsizei h) {
if (!h) return;
glViewport(0, 0, w, h) ;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho(-7.0, 7.0, -7.0*(GLfloat)h/(GLfloat)w,
7.0*(GLfloat)h/(GLfloat)w, -7.0, 7.0) ;
else
glOrtho(-7.0*(GLfloat)w/(GLfloat)h,
7.0*(GLfloat)w/(GLfloat)h, -7.0, 7.0, -7.0, 7.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Suprafețe NURBS

Vom relua redarea suprafeței definită prin punctele de control din exemplul de la "Suprafețe Bezier". Modul de utilizare al interfeței NURBS pentru reprezentarea suprafețelor NURBS este foarte asemănător cu cel pentru redarea curbelor NURBS. Trebuie creat un obiect de tipul GLUnurbsObj, utilizând funcția gluNewNurbsRenderer. Apoi se definesc caracteristicile pentru obiectul creat folosind funcția gluNUrbsProperty pentru fiecare atribut definit. Desenarea suprafeței se face în corpul gluBeginSurf ace (theNurb) / gluEndSurface (theNurb) utilizând funcția gluNurbsSurface (gluNurbsCurve pentru curbe NURBS). Ca şi în cazul curbelor trebuie definit tabloul punctelor de control şi tabloul nodurilor. In funcție de numărul punctelor de control (n+1, m+1) şi al ordinului (s sau t) pe fiecare direcție se va stabili şi numărul nodurilor (n+s+l sau m+t+l) pentru fiecare direcție. în cazul exemplului următor valorile sunt următoarele:

Tabelul 3.3

Parametrul	Valoare
n	3
Numărul punctelor de control pe direcția u	4
m	3
Numărul punctelor de control pe direcția v	4
Ordinul s pe direcția u	4
Ordinul t pe direcția v	4
Gradul polinoamelor de amestec pe direcția u (s-1)	3
Gradul polinoamelor de amestec pe direcția v (t-1)	3
Numărul subintervalelor pe direcția u (n+s)	7
Numărul subintervalelor pe direcția v (m+s)	7
Numărul nodurilor pe direcția u (puncte control+ordin=n+s+l)	8
Numărul nodurilor pe direcția v (puncte control+ordin=m+t+l)	8

Pentru calcularea valorilor nodurilor se folosesc aceleaşi relații ca şi în cazul curbelor (3.14). Desigur că se pot reprezenta şi suprafețe de forme mult mai complicate.

Exemplul 1

```
/*   Supf_spline_solid2.c
Afişează o suprafața spline
folosind biblioteca GLUT */

#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
```

```
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

// tabloul ctrlpoints defineşte cele 16 puncte de control ale suprafeței

GLfloat ctrlpoints[4][4][3] = {
   {{-1.5, -1.5, -4.0}, {-0.5, -1.5, -4.0},
   {0.5, -1.5, -4.0}, {1.5, -1.5, -4.0}},
   {{-1.5, -0.5, -4.0}, {-0.5, -0.5, -2.0},
   {0.5, -0.5, -2.0}, {1.5, -0.5, -4.0}},
   {{-1.5, 0.5, -4.0}, {-0.5, 0.5, -2.0},
   {0.5, 0.5, -2.0}, {1.5, 0.5, -4.0}},
   {{-1.5, 1.5, -4.0}, {-0.5, 1.5, -4.0},
   {0.5, 1.5, -4.0}, {1.5, 1.5, -4.0}}
};
GLUnurbsObj *theNurb;
/* Inițializarea buffer-ului de adâncime si a atributelor materialului si sursei de
lumina
Inițializarea suprafeței NURBS */

void myinit(void)
{
 GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_diffuse[] = { 1.0, 0.2, 1.0, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };
   GLfloat light0_position[] = { 1.0, 0.0, -1.0, 0.0 };
   GLfloat light1_position[] = { -1.0, 0.1, 0.0, 0.0 };
   GLfloat lmodel_ambient[] = { 0.3, 0.3, 0.3, 1.0 };
   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
   glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_LIGHT1);
   glDepthFunc(GL_LESS);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_AUTO_NORMAL);
   theNurb = gluNewNurbsRenderer();
   gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
   gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
                  //sau GLU_OUTLINE_POLYGON
}
```

```
void CALLBACK display(void)
{
   GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glRotatef(-85.0, 1.0, 1.0, 1.0);
        glTranslatef(0, 0, 6);
        glScalef(1.7,1.7,1.7);
   gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb,
      8, //numărul nodurilor în direcția u
      knots, // tabloul nodurilor în direcția u
      8, //numărul nodurilor în direcția v
      knots,
      4 * 3,       // offsetul între puncte de control succesive în direcția u în tabloul
                        //ctrlpoints
      3,  // offsetul între puncte de control succesive în direcția v în tabloul
ctrlpoints
      &ctrlpoints[0][0][0], // tabloul punctelor de control
      4, // ordinul curbei s în direcția u, gradul polinoamelor de amestec este s-1
      4, // ordinul curbei t în direcția v, gradul polinoamelor de amestec este t-1
      GL_MAP2_VERTEX_3);
   gluEndSurface(theNurb);
   glPopMatrix();
   glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
      4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
   else
   glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
      4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
```

```
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("Suprafața B-Spline");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

Observații:

Se poate observa că forma suprafeței obținute este aceeaşi cu a suprafeței Bezier (figura 3.24). În figura 3.24 se poate vedea şi reprezentarea wireframe în cazul în care se foloseşte constanta GLU_OUTLINE_POLYGON în locul constantei GLU_FILL în funcția gluNurbsProperty pentru parametrul GLU_DISPLAY_MODE.

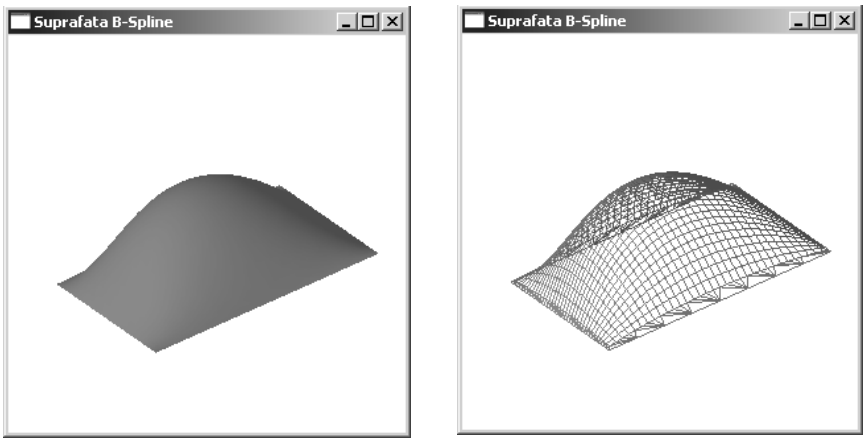


Figura 3.24

Exemplul 2

Se va exemplifica şi o formă mai complicată. Se vor da doar acele funcții care au suferit modificări față de exemplul anterior. Reprezentarea suprafeței se poate vedea în figura 3.25.

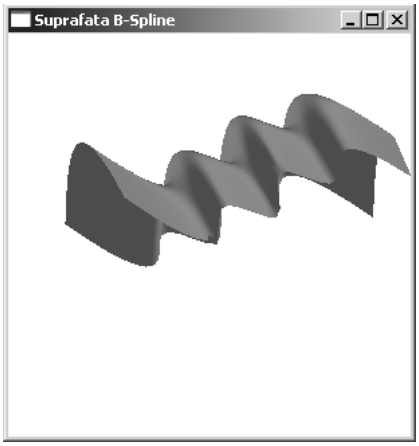


Figura 3.25

```
GLfloat ctrlpoints[4][8][3] = {
   {{-4.5, -1.5, -4.0}, {-3.0, -1.5, -2.0},
         {-1.5, -1.5, -0.0}, {0.0, -1.5, -2.0},
         {1.5, -1.5, -4.0}, {3.0, -1.5, -2.0},
         {4.5, -1.5, -0.0}, {6.0, -1.5, -2.0}},
         {{-4.5, 0.0, -4.0}, {-3.0, 0.0, -2.0},
         {-1.5, 0.0, -0.0}, {0.0, 0.0, -2.0},
         {1.5, 0.0, -4.0}, {3.0, 0.0, -2.0},
         {4.5, 0.0, -0.0}, {6.0, 0.0, -2.0}},
         {{-4.5, 1.5, -4.0}, {-3.0, 1.5, -2.0},
         {-1.5, 1.5, -0.0}, {0.0, 1.5, -2.0},
         {1.5, 1.5, -4.0}, {3.0, 1.5, -2.0},
         {4.5, 1.5, -0.0}, {6.0, 1.5, -2.0}},
         {{-4.5, 3.0, -4.0}, {-3.0, 3.0, -2.0},
         {-1.5, 3.0, -0.0}, {0.0, 3.0, -2.0},
         {1.5, 3.0, -4.0}, {3.0, 3.0, -2.0},
         {4.5, 3.0, -0.0}, {6.0, 3.0, -2.0}},
};
GLUnurbsObj *theNurb;

void CALLBACK display(void)
{        // punctele de control
GLfloat s_knots[12] = {0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 5.0, 5.0 };
GLfloat t_knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
glClearColor(1.0, 1.0, 1.0, 1.0);   //culoarea background-ului
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
// transformări geometrice pentru poziționarea şi
// dimensionarea suprafeței
glRotatef(-85.0, 1.0, 1.0, 1.0);
glTranslatef(0, 0, 6);
glScalef(1.7,1.7,1.7);
gluBeginSurface(theNurb) ; // începe redarea suprafeței
gluNurbsSurface (theNurb, //obiectul NURBS
12 , //numărul nodurilor în direcția u
s_knots, // tabloul nodurilor în direcția u
8, //numărul nodurilor în direcția v
t_knots, // tabloul nodurilor în direcția u
4  *  3, // offsetul între puncte de control succesive în direcția u în tabloul ctrlpoints
3, // offsetul între puncte de control succesive în direcția v în tabloul ctrlpoints
&ctrlpoints [0] [0] [0] ,   //tabloul punctelor de control
4, // ordinul curbei s în direcția u, gradul polinoamelor de amestec este s-1
4 , // ordinul curbei t în direcția v, gradul polinoamelor de amestec este t-1
GL_MAP2_VERTEX_3        );      //      GL_MAP2_VERTEX_3      sau GL_MAP2_COLOR_4
```

```
gluEndSurf ace (theNurb) ; // se termină redarea suprafeței
glPopMatrix();
glFlush() ; *    }
void CALLBACK myReshape(GLsizei w, GLsizei h) {
if (!h) return;
glViewport(O, 0, w, h);
glPopMatrix();
glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho(-7.0, 7.0, -7.0*(GLfloat)h/(GLfloat)w,
      7.0*(GLfloat)h/(GLfloat)w, -7.0, 7.0);
   else
   glOrtho(-7.0*(GLfloat)w/(GLfloat)h,
      7.0*(GLfloat)w/(GLfloat)h, -7.0, 7.0, -7.0, 7.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (0, 0, 300, 300);
   auxInitWindow ("Suprafața B-Spline");
   myinit();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
   return(0);
}
```

3.3 Suprafețe cvadrice

Suprafețele cvadrice sunt suprafețe definite printr-o ecuație de gradul al doilea. Biblioteca GLU dispune de funcții pentru reprezentarea acestui tip de suprafețe (sfere, cilindrii, discuri şi porțiuni de discuri).

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

- gluQuadricNormals() : dacă se vor genera normale la suprafețe, iar dacă da, se specifică dacă vor fi specificate relativ la vârfuri sau la suprafețe.
- gluQuadricTexture(): dacă se generează coordonate pentru texturare.
- gluQuadricOrientation(): care fețe vor fi considerate exterioare şi care interioare.
- gluQuadricDrawStyle() : stilul de desenare al cvadricelor - cu puncte, linii sau poligoane.

După ce s-a specificat stilul de redare, se apelează funcțiile 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 funcția de scalare glScalef(). Motivul este creşterea vitezei de reprezentare, având în vedere, că dacă se utilizează funcția de scalare are loc o renormailzare a normalelor la suprafețe. 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 funcții.

Funcții pentru controlarea obiectelor

GLUquadricObj* gluNewQuadric (void);

- Funcția permite crearea unei cvadrice.

void gluDeleteQuadric (GLUquadricObj *state);

- Funcția permite ştergerea unei cvadrice.

Funcții pentru modificarea felului în care sunt desenate cvadricele

void gluQuadricNormals (GLUquadricObj *quadObject, Glenum normals) ;

- Parametrul normals poate lua următoarele valori:

Tabelul 3.4

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

- voidgluQuadricTexture(GLUquadricObj*quadObject,
  GLboolean textureCoords);
- Funcția activează (GL_TRUE) sau dezactivează (GL_FALSE) generarea coordonatelor de texturare.
- void gluQuadricOrientation (GLUquadric *quadObject, GLenum orientation);
- Funcția controlează direcția normalelor pentru iluminare. Acestea pot avea orientarea spre exteriorul obiectelor (GLU_OUTSIDE) sau spre interiorul acestora (GLU_INSIDE).
- void gluQuadricDrawStyle (GLUquadricObj *quadObject, GLenum

drawStyle);

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

Tabelul 3.5

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

Funcții pentru specificarea tipului cvadricei

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

- Funcția generează un cilindru având centrul bazei în originea sistemului de axe. Funcția 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 (figura 3.26). Această valoare va creşte atunci când se doreşte o iluminare mai bună.

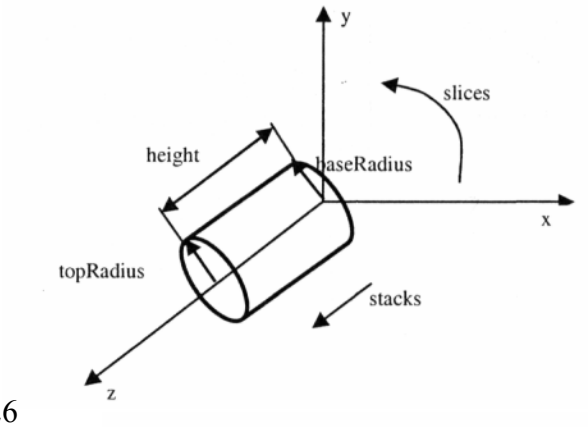


Figura 3.26

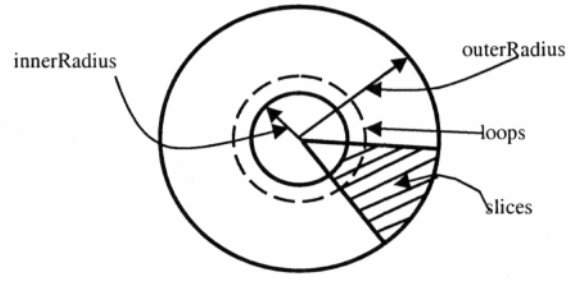


Figura 3.27

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

- Discurile (figura 3.27) sunt forme plane care au forma unui CD (sau dischetă). Funcția poate fi utilizată şi pentru generarea cercurilor.
- 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);

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

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

- Funcția reprezintă o sferă.
- slices determină numărul longitudinilor generate,
- stacks pe cel al latitudinilor generate Ia reprezentarea sferei.

Exemplu:

în exemplul următor se arată felul în care se pot utiliza funcțiile GLU pentru redarea cvadricelor. În figura 3.28 se pot vedea obiectele pe care le desenează programul următor.

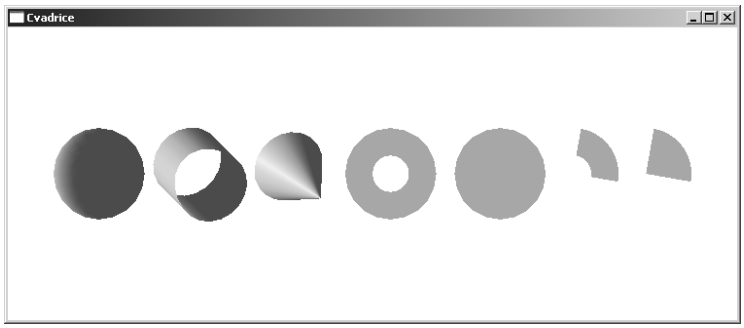


Figura 3.28

/* cvadrice.c

Programul arată modul de utilizare al cvadricelor din GLU pentru desenare de cilindri, conuri, sfere, disk-uri, cercuri, arce de cerc, sectoare de disk. */

```
#include  "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void initlights(void);
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
GLUquadricObj   *  quadObj ;  //obiect de tip cvadrică
// setări pentru iluminarea suprafeței
void initlights(void) {
GLfloat ambient[] = { 1.0, 0.6, 0.0, 1.0 };
   GLfloat position0[] = { -3.0, 0.0, 2.0, 0.0 };
   GLfloat position1[] = { 1.0, 0.0, 1.0, 0.0 };
   GLfloat mat_diffuse[] = { 1.0, 0.6, 0.0, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_LIGHT1);
   glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT0, GL_POSITION, position0);
   glLightfv(GL_LIGHT1, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT1, GL_POSITION, position1);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}
void CALLBACK display(void) {//Şterge ecranul
glClearColor(1.0,   1.0,   1.0,   1.0);
glClear   (GL_COLOR_BUFFER_BIT);
//Stabileşte atributele, comune pentru toate cvadricele.
quadObj   =  gluNewQuadric   () ; //generează cvadrică
gluQuadricDrawStyle   (quadObj,   GLU_FILL) ; //atribut de umplere
gluQuadricNormals (quadObj ,GLU_SMOOTH) ; //iluminare GOURAUD
//reprezintă sfera
glPushMatrix();
glTranslatef (-15.0, 0.0, 0.0);
gluSphere(quadObj, 2.5, 20.0, 20);
glPopMatrix(); //reprezintă cilindru
glPushMatrix () ;
glRotatef(30,   1,   1,   0);
glTranslatef (-11.0, 1.5, 0.0);
gluCylinder(quadObj, 2.0, 2.0, 4.0, 20.0, 20);
glPopMatrix(); //reprezintă conul
glPushMatrix();
glRotatef(30, 1,   1, 0);
glTranslatef (-5.0, 0.8, 0.0);
gluCylinder(quadObj, 2.0, 0.0, 5.0, 20.0, 20);
glPopMatrix(); //reprezintă disk-ul
glPushMatrix();
glTranslatef (1.0, 0.0, 0.0);
gluDisk   (quadObj,   1.0,   2.5,   20,   20);
```

```
glPopMatrix(); //reprezintă cercul
glPushMatrix();
glTranslatef (7.0, 0.0, 0.0);
gluDisk (quadObj, 0.0, 2.5, 20, 20);
glPopMatrix(); //reprezintă sectorul de disk
glPushMatrix();
glTranslatef (11.0, 0.0, 0.0);
gluPartialDisk (quadObj, 1.0, 2.5, 20, 20, 10.0, 90.0);
glPopMatrix(); //reprezintă sectorul de cerc
glPushMatrix();
glTranslatef (15.0, 0.0, 0.0);
gluPartialDisk (quadObj, 0.0, 2.5, 20, 20, 10.0, 90.0);
glPopMatrix(); glFlush() ; }
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho(-8.0, 8.0, -8.0*(GLfloat)h/(GLfloat)w,
      8.0*(GLfloat)h/(GLfloat)w, -8.0, 8.0);
   else
   glOrtho(-8.0*(GLfloat)w/(GLfloat)h,
      8.0*(GLfloat)w/(GLfloat)h, -8.0, 8.0, -8.0, 8.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (10, 10, 750, 300);
   auxInitWindow ("Cvadrice");
   initlights();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
   return(0);
}
```

3.4 Primitive raster

OpenGL nu este doar o interfață completă pentru aplicațiile de grafică vectorială 3D ci este, de asemenea, un foarte bun motor de procesare a imaginilor (utilizate în aplicațiile de grafică punctuală sau raster).

Primitivele conțin explicit informații de culoare pentru fiecare pixel pe care-1 conțin. In Windows se utilizează denumirea de bitmap atât pentru bitmap-urile monocrome cât şi pentru cele color. în OpenGL, spre deosebire de Windows, se face distincție între bitmap-uri şi pixelmap-uri. OpenGL lucrează deci cu două tipuri de primitive raster:

- Bitmap-urile care utilizează un singur bit (0 sau 1) pentru fiecare pixel. Ele sunt utilizate, în principal, ca tablouri 2D de măşti la nivel de bit pentru a determina care pixeli să fie actualizați. Culoarea curentă, setată cu funcția glColor() este utilizată pentru a determina noua culoare a pixelilor corespunzători valorii 1 din bitmap, în timp ce pixelii corespunzători biților 0 sunt transparenți.
- Pixelmap-urile (imagini) sunt blocuri de pixeli (tablouri 2D) cu informații complete de culoare pentru fiecare pixel.

În ceea ce priveşte fişierele de imagini se subliniază că OpenGL nu înțelege formatele de imagini, cum ar fi JPEG, PNG, sau GIF-uri. Pentru ca OpenGL să poată utiliza informațiile conținute în aceste formate de fişiere, fişierul trebuie să fie citit şi decodificat pentru a obține informația de culoare, şi plecând de aici OpenGL poate rasteriza valorile culorilor.

Fluxul de procesare al pixelilor

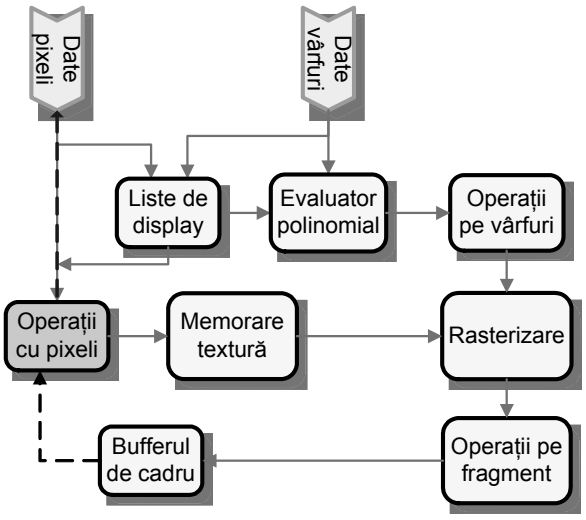


Figura 3.29

În figura 3.29 se dă procesarea în cascadă a primitivelor geometrice şi de asemenea a pixelilor. Pixelii sunt citiți din memoria principală, sunt procesați pentru a se obține formatul intern pe care-1 utilizează OpenGL, care poate include modificări de culoare sau înlocuiri de octeți. După aceasta, este procesat fiecare pixel din imagine prin operațiile pe fragment din ultima secțiune a fluxului de procesare, şi în final rasterizate în buffer-ul de cadru.

Suplimentar redării în buffer-ul de cadru, pixelii pot fi copiați din buffer-ul de cadru înapoi în memoria gazdă, sau transferați în memoria de mapare a texturii.

Pentru performanțe mai bune, reprezentarea internă a tabloului de pixeli va fi în concordanță cu hardware-ul. Spre exemplu, pentru un buffer de cadru de 24 biți, RGB 8-8-8 va fi probabil o potrivire bună, dar RGB 10-10-10 ar fi o potrivire rea.

Atenție:

Pentru valori neimplicite, memorarea şi transferul pixelilor se face foarte încet.

În figura 3.30, pe lângă fluxul de procesarea pixelilor sunt specificate şi funcțiile OpenGL utilizate în fiecare etapă.

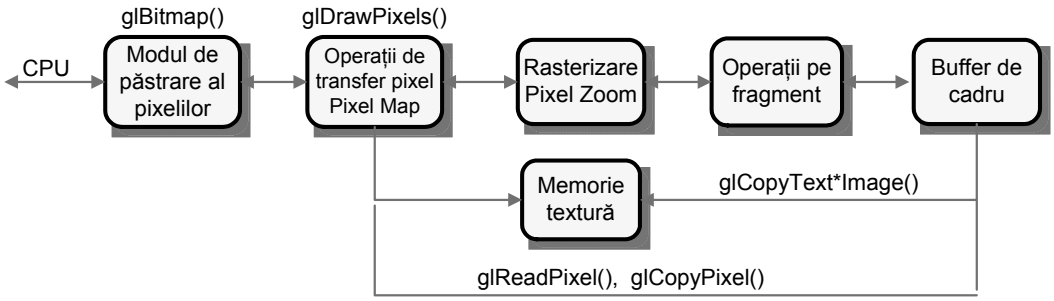


Figura 3.30

Bitmap-urile şi pixmap-urile sunt cele două primitive care nu sunt afectate de operațiile geometrice care au loc în fluxul operațiilor anterior rasterizării.

3.4.1 Reprezentarea imaginilor bitmap

Un bitmap specifică un dreptunghi complectat cu 0 şi 1, şi este mult utilizat pentru descrierea caracterelor care pot fi plasate la o locație proiectată 3D (prin poziția raster curentă). Fiecare 1 din bitmap produce un fragment ale cărui valori asociate sunt cele ale poziției raster curente, în timp ce pentru fiecare 0 nu se produce nici un fragment. Comanda glBitmap() specifică de asemenea offset-urile care controlează cum este plasat bitmap-ul în funcție de poziția raster curentă şi cum poziția raster curentă este modificată după ce se desenează bitmap-ul (astfel determinând pozițiile relative ale bitmap-uilor succesive).

Poziționarea primitivelor raster

glRasterPos3f(x, y, z)

- poziția raster (figura 3.31) este afectată de transformări ca şi poziția geometrică;
- primitiva bitmap nu se afişează dacă poziția raster este în exteriorul viewport-ului.

Imaginile sunt poziționate prin specificarea poziției raster, care mapează colțul stânga jos al unei primitive imagine la un punct din spațiu. Pozițiile raster sunt transformate şi decupate la fel ca vârfurile. Dacă o poziție raster se pierde prin decuparea față de viewport, nu mai este rasterizat nici un fragment din primitiva bitmap.

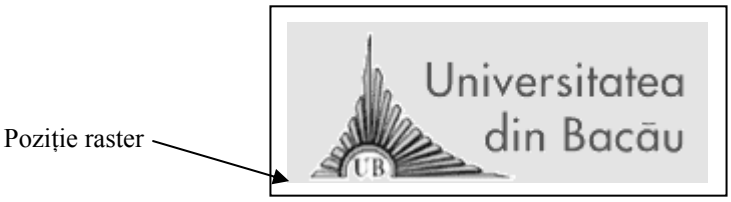


Figura 3.31 Poziția raster

Redarea Bitmap-urilor

glBitmap (width, height, xorig, zorig, xmove, zmove, bitmap) ;

- funcția redă bitmap-ul în culoarea curentă;
- după redarea primitivei, poziția raster curentă este actualizată.

Bitmap-urile sunt utilizate ca o mască pentru a determina care pixeli să fie actualizați. Un bitmap este specificat ca un tablou împachetat de biți într-un tablou de tip byte. Pentru fiecare valoare de 1 dintr-un bitmap, este generat un fragment în culoarea curentă și apoi este procesat de operațiile pe fragment.

Bitmap-urile pot avea propria lor origine, care asigură o poziție relativă față de poziția raster curentă. Suplimentar, după ce se redă bitmap-ul, poziția raster este actualizată în mod automat prin offset-ul furnizat în (xmove, ymove). Dacă originea bitmap-ului este (0,0) atunci colțul stânga jos al bitmap-ului va fi plasat în poziția raster curentă. Dacă originea bitmap-ului este diferită de (0,0) atunci poziția raster curentă va fi utilizată pentru plasarea originii bitmap-ului. Dacă poziția raster curentă este în exteriorul viewport-ului, chiar dacă pentru această poziție s-ar putea reprezenta parțial bitmap-ul, totuşi el nu este redat deloc. Deci, din acest punct de vedere, primitivele raster nu se comportă la fel cu primitivele vectoriale, care sunt decupate față de viewport.

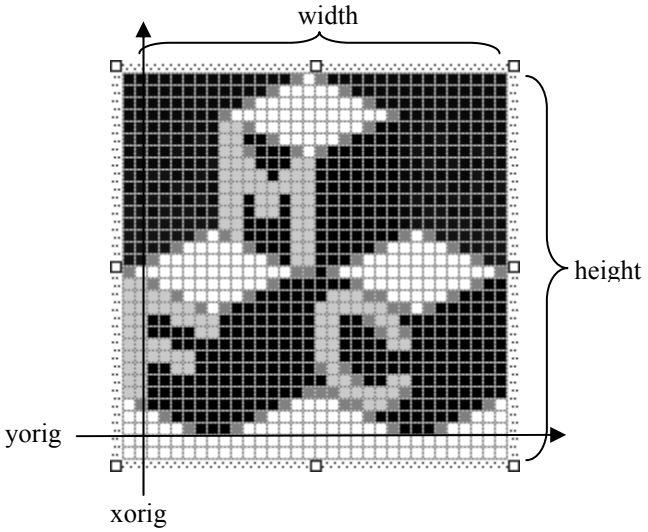


Figura 3.32

Coordonatele originii bitmap-ului se stabilesc relativ la colțul stânga-jos al bitmap-ului. Pentru figura 3.32 coordonatele originii sunt (xorig=2, yorig=2). Dimensiunea bitmap-ului este height=32, width=32.

Bitmap-urile sunt utile în mod particular pentru redarea font-urilor de tip bitmap, despre care se va discuta în continuare. în acest caz facilitatea de actualizare a poziției raster curente este utilizată din plin.

Exemplul 1

Exemplul următor redă bitmap-ul din figura 3.33, succesiv în fereastra aplicației.

Dimensiunea ferestrei inițial este de 200X200 pixeli. Pentru viewport-ul aplicației, de 10X10 unități, va rezulta că fiecare unitate are 20 de pixeli. Bitmap-ul fiind de 32X32 pixeli, vom stabili prin program o repetare a bitmap-ului din 2 în 2 unități, adică un bitmap la 40 de pixeli. Distanța dintre două bitmap-uri succesive va fi de 8 pixeli. În figura 3.34 se poate vedea cum va arăta fereastra aplicației.

Figura 3.33

Mai trebuie subliniat felul în care se specifică un bitmap. Bitul 7 din primul octet al tabloului care descrie bitmap-ul corespunde pixelului stânga jos al bitmap-ului. Deci descrierea bitmap-ului începe cu primul rând din partea de jos a bitmap-ului.

Figura 3.34

În continuare, se dă codul aplicației.

```
/*utilizarea bitmap-urilor în OpenGL*/
#include "glos.h"
#include <GL/gl.h>
```

```
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>

void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
void CALLBACK display(void)
{    int i, j;
   GLubyte model[] = {
   0x03, 0xc0, 0, 0,
   0x0f, 0xf0, 0, 0,
   0x1e, 0x78, 0, 0,
   0x39, 0x9c, 0, 0,
   0x77, 0xee, 0, 0,
   0x6f, 0xf6, 0, 0,
   0xff, 0xff, 0, 0,
   0xff, 0xff, 0, 0,
   0xff, 0xff, 0, 0,
   0xff, 0xff, 0, 0,
   0x73, 0xce, 0, 0,
   0x73, 0xce, 0, 0,
   0x3f, 0xfc, 0, 0,
   0x1f, 0xf8, 0, 0,
   0x0f, 0xf0, 0, 0,
   0x03, 0xc0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0,
         0, 0, 0, 0
         };
         glClearColor(1.0, 1.0, 1.0, 1.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
                  for(i=-5; i<5; i+=2)
         for(j=-5; j<5; j+=2)
```

```
            {
            glRasterPos2i(i,j);
            glBitmap(32, 32, 0.0, 0.0, 0.0, 0.0, model);
            }
    glFlush();
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
       5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
    else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
       5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 200, 200);
    auxInitWindow ("O fereastră cu bitmap-uri");
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

3.4.2 Reprezentarea fonturilor prin bitmap-uri

În OpenGL, pentru redarea caracterelor, se pot utiliza fonturi vectoriale sau fonturi bitmap. Fonturile vectoriale sunt construite din segmente de dreaptă şi se pretează pentru dispozitivele vectoriale (plotere spre exemplu). Fonturile bitmap construiesc fiecare caracter ca un bitmap. Pentru a se facilita utilizarea fonturilor, în OpenGL se utilizează listele de display. Fiecare listă de display conține doar un apel glBitmap(), pentru redarea caracterului corespunzător. Listelor de display le sunt asociate în mod unic, identificatori (numere) care pentru caracterele fontului pot fi alese ca valori succesive. În felul acesta pentru redarea unui text trebuie apelate listele de display corespunzătoare caracterelor din textul respectiv. Pentru construirea bitmap-urilor corespunzătoare fiecărui caracter dintr-un font există două posibilități:

Se declară tabloul de tip unsigned byte (tipul GLubyte în OpenGL) care conține caracterele respective.

Se construiesc bitmap-urile corespunzătoare fiecărui caracter pe baza fontului curent din sistemul de ferestre cu care se lucrează. Sistemele de ferestre au pentru aceasta funcţii proprii. Windows-ul, spre exemplu, dispune de funcţia wglUseFontBitmaps(). Şi în acest caz, se utilizează pentru manevrarea fontului tot listele de display. Fiecare caracter este memorat într-o listă de display care este parte a unui set creat la procesarea fontului.

În continuare, se vor exemplifica ambele modalităţi.

Construirea unui caracter şi afişarea sa

Vom începe cu exemplificarea modului în care primitivele bitmap sunt utilizate în OpenGL pentru construirea şi afişarea unui caracter. în acest caz nu apar modificări faţă de afişarea obişnuită a unui bitmap. Se construieşte tabloul care conţine caracterul. Se ţine seama de faptul că lăţimea bitmap-ului trebuie să fie un număr par de octeţi. Pentru caracterul "E" afişat de exemplul nostru (figura 3.35), utilizăm un bitmap de 10X12. Avem nevoie de un tablou de octeţi, câte doi octeţi pentru fiecare linie a bitmap-ului. Din acest tablou, pentru construirea bitmap-ului vor fi utilizaţi doar primii 10 biţi din fiecare rând. Originea bitmap-ului se va considera colţul stânga jos al bitmap-ului. Ce mai trebuie remarcat, este faptul că funcţia glRasterPos2i () utilizează coordonate logice iar offset-ul pentru actualizarea poziţiei curente, din funcţia glBitmap() este dat în pixeli. în cazul acestui exemplu, fereastra are 200X200 pixeli (unităţi fizice). Funcţia myReshape () stabileşte câte 10 unităţi logice pe fiecare axă, în intervalul [-5, 5]. Deci fiecărei unităţi logice îi corespund 20 unităţi fizice (pixeli). Pentru a deplasa cu o unitate fiecare caracter, în funcţia glBitmap, actualizarea poziţiei curente s-a făcut cu 20 pixeli pe axa x şi cu 0 pixeli pe axa y.

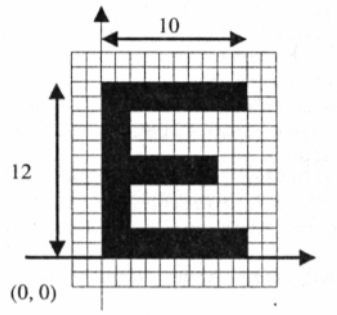


Figura 3.35

Exemplu:

/* Programul este un exemplu de utilizare a primitivei bitmap * in OpenGL, pentru desenarea fonturilor. */

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
// tabloul care conține litera E, 2 octeți X 12 rânduri
GLubyte litera_E[24] = {
   0xff, 0xc0, 0xff, 0xc0, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
   0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
   0xff, 0xc0, 0xff, 0xc0};

void CALLBACK display(void)
{
   glPixelStorei (GL_UNPACK_ALIGNMENT, 1);
   glClearColor (1.0, 1.0, 1.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f (0.0, 0.0, 0.0);
   glRasterPos2i (-2.0, 0.0);
//poziția curentă pentru primul bitmap
// dată în unități logice (sunt 10 unități pe axa x în intervalul(-5, 5))
   glBitmap (10, 12, 0.0, 0.0, 20.0, 0.0, litera_E);
// deplasarea poziției curente este dată în pixeli - 20 de pixeli pe axa x
   glBitmap (10, 12, 0.0, 0.0, 20.0, 0.0, litera_E);
   glBitmap (10, 12, 0.0, 0.0, 20.0, 0.0, litera_E);
   glBitmap (10, 12, 0.0, 0.0, 20.0, 0.0, litera_E);
   glFlush();
}
// proiecție ortogonală
//se mapează 10 unități pe fiecare axă, o unitate are 20 pixeli

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
 if (!h) return;
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
   glOrtho (-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
      5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
   else
   glOrtho (-5.0*(GLfloat)w/(GLfloat)h,
      5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
   glMatrixMode(GL_MODELVIEW);
}
int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (0, 0, 200, 200);
   auxInitWindow ("Bitmap_E");
   auxReshapeFunc (myReshape);
```

```
    auxMainLoop(display);
    return(0);
}
```

Figura 3.36

În figura 3.36 este dată fereastra afişată de această aplicație.

Funcții pentru manevrarea listelor de display utilizate pentru fonturile bitmap.

Pentru utilizarea fonturilor bitmap în OpenGL, anumite funcții pentru listele de display au o relevanță deosebită. Trebuie subliniat că aceste funcții se utilizează la fel şi pentru fonturile vectoriale. Acestea sunt:

- void glNewList(GLuint list, GLenum mode);
- void glEndList(void);

  Funcțiile glNewListO şi glEndListO creează lista de display cu identificatorul list. Parametrul mode stabileşte modul de compilare al listei de display.
- GLuint glGenLists(GLsizei range);

  Funcția creează un set de liste de display, cu identificatori succesivi şi care sunt goale (nu conțin nici o comandă). Funcția returnează identificatorul primei liste de display n, iar celelalte liste vor avea identificatorii n+J, n+2,..., n+range-1. Parametrul range este numărul listelor de display create.
- void glListBase(GLuint base);

  Funcția glListBase() setează baza listelor de display, pentru funcția glCallLists.
- void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);

  Funcția execută o listă de liste de display. Parametrul n este numărul listelor de display care vor fi executate. Parametrul type este tipul elementelor din lists. Parametrul lists este adresa unui tablou cu numele ofseturilor în listele de display.
- void glCallList(GLuint list);

  Funcția execută lista de display al cărui identificator (întreg) este dat ca parametru.
- void glDeleteLists(GLuint list, GLsizei range);

  Funcția şterge un grup de liste de display ("range" liste de display) având identificatori succesivi, începând cu identificatorul list.

Un font, aşa cum probabil se ştie, este un şir de caractere, fiecare caracter având asociat un număr de identificare, de obicei acesta fiind codul ASCII al caracterului. Fiecare caracter are şi o anumită formă grafică, aceasta fiind conținută în bitmap-ul asociat caracterului respectiv. Spre exemplu, caracterul 'A' are codul ASCII 65 (41H), caracterul 'B' are codul ASCII 66 (42H), caracterul 'C are codul ASCII 67 (43H), ş.a.m.d. La modul cel mai simplu, listele de display asociate cu afişarea caracterelor pot avea ca identificatori chiar codurile ASCII ale caracterelor. în acest caz, pentru afişarea şirului "ABC" se vor executa listele de display având identificatorii 65, 66, 67. Pentru aceasta se poate apela funcția glCallLists(). Parametrul n va conține lungimea şirului de caractere, 3 în cazul nostru, iar tabloul lists va conține codurile ASCII ale caracterelor.

- glCallLists(3, GL_UNSIGNED_BYTE, "ABC");
  Având acum în vedere faptul că există mai multe tipuri de fonturi, având corpuri de caractere diferite, utilizarea codurilor ASCII nu mai pare a fi o soluție convenabilă. Se poate însă proceda în felul următor: pentru fiecare font se adaugă un offset, față de care se stabilesc identificatorii fiecărei liste de display. Deoarece numărul maxim al caracterelor dintr-un font nu depăşeşte 255, un offset de 1000, spre exemplu, este o soluție convenabilă. In felul acesta un font va utiliza liste de display cu identificatorii cuprinşi între 1000 şi 1255, următorul font va utiliza liste cu identificatorii cuprinşi între 2000 şi 2255, ş.a.m.d. Pentru stabilirea offset-ului se poate utiliza comanda glListBase().
  Deoarece este necesar ca listele de display corespunzătoare caracterelor dintr-un font să aibă identificatori succesivi se va utiliza pentru aceasta comanda glGenLists(). Spre exemplu, dacă se doreşte un font cu toate cele 255 de caractere apelul va fi:
- glGenLists(255);
  Funcția va stabili identificatori unici pentru fiecare listă de display. Dacă funcția nu găseşte un bloc de 255 de identificatori succesivi, va returna 0. Pentru a şterge o parte din listele de display se poate utiliza funcția glDeleteLists().

Cum se utilizează aceste funcții pentru fonturile bitmap, se va vedea în cele două exemple care vor urma.

Construirea internă a unui font

Înainte de toate, trebuie spus că exemplul următor se bazează pe exemplul din fişierul font.c, care face parte din exemplele pentru utilizarea bibliotecii OpenGL, existente în nucleul de instalare pentru Visual C 6.0, preluate din OpenGL Programming Guide. În acest exemplu, tabloul rasters[] conține toate cele 95 de caractere ASCII tipăribile (coduri ASCII cuprinse între 32 şi 127), inclusiv spațiul. Fiecare caracter este construit în tabloul rasters, aşa cum s-a construit şi caracterul E din exemplul anterior. Listele de display sunt construite în funcția makeRasterFont(). Fiecare listă de display are asociat ca identificator codul ASCII al caracterului la care se adaugă un offset obținut de funcția glGenLists() (apelată în funcția makeRasterFont()), şi conține ca şi comandă funcția glBitmap(). După afişarea fiecărui caracter, poziția curentă este actualizată prin adăugarea unui offset pe axa x, de 10 pixeli. În figura 3.7 se poate vedea fereastra aplicației.

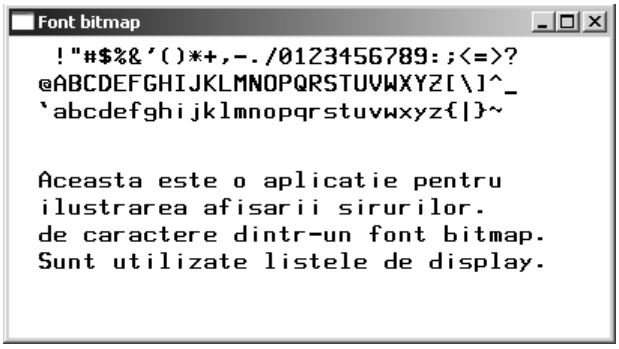


Figura 3.37

Exemplu:

```
/*
Aplicația afișează texte în OpenGL, utilizând un font bitmap.
Pentru construirea caracterelor fontului utilizează glBitmap() şi alte funcții pentru primitive raster. Aplicația utilizează de asemenea listele de display. */

#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void makeRasterFont(void);
void printString(char *s);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
GLubyte rasters[][13] = {
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
//caracterul spațiu
{0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18},
//caracterul !
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36, 0x36, 0x36, 0x36},
{0x00, 0x00, 0x00, 0x66, 0x66, 0xff, 0x66, 0x66, 0xff, 0x66, 0x66, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x7e, 0xff, 0x1b, 0x1f, 0x7e, 0xf8, 0xd8, 0xff, 0x7e, 0x18},
{0x00, 0x00, 0x0e, 0x1b, 0xdb, 0x6e, 0x30, 0x18, 0x0c, 0x76, 0xdb, 0xd8, 0x70},
```

```
        {0x00, 0x00, 0x7f, 0xc6, 0xcf, 0xd8, 0x70, 0x70, 0xd8, 0xcc, 0xcc, 0x6c,
0x38},
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x1c, 0x0c,
0x0e},
        {0x00, 0x00, 0x0c, 0x18, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x18,
0x0c},
        {0x00, 0x00, 0x30, 0x18, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x18,
0x30},
        {0x00, 0x00, 0x00, 0x00, 0x99, 0x5a, 0x3c, 0xff, 0x3c, 0x5a, 0x99, 0x00,
0x00},
        {0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0xff, 0xff, 0x18, 0x18, 0x18, 0x00,
0x00},
        {0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x60, 0x60, 0x30, 0x30, 0x18, 0x18, 0x0c, 0x0c, 0x06, 0x06, 0x03,
0x03},
        {0x00, 0x00, 0x3c, 0x66, 0xc3, 0xe3, 0xf3, 0xdb, 0xcf, 0xc7, 0xc3, 0x66,
0x3c},
        {0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x78, 0x38,
0x18},
        {0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0xe7,
0x7e},
        {0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0x07, 0x03, 0x03, 0xe7,
0x7e},
        {0x00, 0x00, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0xff, 0xcc, 0x6c, 0x3c, 0x1c, 0x0c},
        {0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
        {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
        {0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x03, 0x03,
0xff},
        {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7,
0x7e},
        {0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x03, 0x7f, 0xe7, 0xc3, 0xc3, 0xe7,
0x7e},
        {0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x1c, 0x1c, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x06, 0x0c, 0x18, 0x30, 0x60, 0xc0, 0x60, 0x30, 0x18, 0x0c,
0x06},
        {0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00},
        {0x00, 0x00, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x06, 0x0c, 0x18, 0x30,
0x60},
```

```
        {0x00, 0x00, 0x18, 0x00, 0x00, 0x18, 0x18, 0x0c, 0x06, 0x03, 0xc3, 0xc3,
0x7e},
        {0x00, 0x00, 0x3f, 0x60, 0xcf, 0xdb, 0xd3, 0xdd, 0xc3, 0x7e, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x18},
        {0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
        {0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7,
0x7e},
        {0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xfc},
        {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
        {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xff},
        {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
        {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
        {0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x7e},
        {0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
0x06},
        {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc3},
        {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
        {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff, 0xff, 0xe7, 0xc3},
        {0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe3},
        {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xe7,
0x7e},
        {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
        {0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x66,
0x3c},
        {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
        {0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7,
0x7e},
        {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0xff},
        {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3},
        {0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3},
        {0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
        {0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66,
0xc3},
        {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66,
0xc3},
        {0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x03, 0xff},
        {0x00, 0x00, 0x3c, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x3c},
        {0x00, 0x03, 0x03, 0x06, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0x30, 0x30, 0x60,
0x60},
        {0x00, 0x00, 0x3c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
0x3c},
```

```
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc3, 0x66, 0x3c,
0x18},
        {0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x38, 0x30,
0x70},
        {0x00, 0x00, 0x7f, 0xc3, 0xc3, 0x7f, 0x03, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
        {0x00, 0x00, 0x7e, 0xc3, 0xc0, 0xc0, 0xc0, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x03, 0x03, 0x03, 0x03,
0x03},
        {0x00, 0x00, 0x7f, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30, 0x33,
0x1e},
        {0x7e, 0xc3, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0},
        {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x00, 0x00, 0x18,
0x00},
        {0x38, 0x6c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x00, 0x00, 0x0c,
0x00},
        {0x00, 0x00, 0xc6, 0xcc, 0xf8, 0xf0, 0xd8, 0xcc, 0xc6, 0xc0, 0xc0, 0xc0, 0xc0},
        {0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x78},
        {0x00, 0x00, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xfe, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xfc, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x7c, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x7c, 0x00, 0x00, 0x00,
0x00},
        {0xc0, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0x00, 0x00, 0x00, 0x00},
        {0x03, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe0, 0xfe, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xfe, 0x03, 0x03, 0x7e, 0xc0, 0xc0, 0x7f, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x1c, 0x36, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30,
0x00},
        {0x00, 0x00, 0x7e, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0x00, 0x00, 0x00,
0x00},
```

```
        {0x00, 0x00, 0xc3, 0xe7, 0xff, 0xdb, 0xc3, 0xc3, 0xc3, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xc3, 0x66, 0x3c, 0x18, 0x3c, 0x66, 0xc3, 0x00, 0x00, 0x00,
0x00},
        {0xc0, 0x60, 0x60, 0x30, 0x18, 0x3c, 0x66, 0x66, 0xc3, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0xff, 0x60, 0x30, 0x18, 0x0c, 0x06, 0xff, 0x00, 0x00, 0x00,
0x00},
        {0x00, 0x00, 0x0f, 0x18, 0x18, 0x18, 0x38, 0xf0, 0x38, 0x18, 0x18, 0x18,
0x0f},
        {0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x18},
        {0x00, 0x00, 0xf0, 0x18, 0x18, 0x18, 0x1c, 0x0f, 0x1c, 0x18, 0x18, 0x18,
0xf0},
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x8f, 0xf1, 0x60, 0x00, 0x00,
0x00}
        };

        GLuint fontOffset;
        void makeRasterFont(void)
        {
           GLuint i;
           glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
           fontOffset = glGenLists (128);
           for (i = 32; i < 127; i++) {
                        glNewList(i+fontOffset, GL_COMPILE);
                                glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, rasters[i-32]);
                        glEndList();
           }
        }
        void printString(char *s)
        {
           glPushAttrib (GL_LIST_BIT);
                        glListBase(fontOffset);
                        glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
           glPopAttrib ();
        }

        void CALLBACK display(void)
        {
           int i, j;
           char teststring[33];
                glClearColor(1.0, 1.0, 1.0, 0.0);
           glClear(GL_COLOR_BUFFER_BIT);
           glColor3f(0.0, 0.0, 0.0);
                glShadeModel (GL_FLAT);
           makeRasterFont();
```

```
   for (i = 32; i < 127; i += 32)
         {
                  glRasterPos2i(20, 200 - 18*i/32);
                  for (j = 0; j < 32; j++)
                           teststring[j] = (char) (i+j);
                  teststring[32] = 0;
                  printString(teststring);
   }
   glRasterPos2i(20, 100);
   printString("Aceasta este o aplicate pentru");
   glRasterPos2i(20, 82);
   printString("ilustrarea afişării şirurilor.");
         glRasterPos2i(20, 64);
   printString("de caractere dintr-un font bitmap.");
         glRasterPos2i(20, 46);
   printString("Sunt utilizate listele de display.");
   glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho (0.0, w, 0.0, h, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (0, 0, 400, 200);
   auxInitWindow ("Font bitmap");
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
   return(0);
}
```

Crearea textelor în OpenGL cu fonturi Windows

Aşa după cum s-a arătat, aplicațiile Windows beneficiază de funcții suplimentare (aparținând sistemului) pentru utilizarea fonturilor sistemului. Funcția wglUseFontBitmaps() creează un set de liste de display pentru a fi utilizate într-un context de redare OpenGL. Fontul obținut se bazează pe fontul curent selectat în contextul de dispozitiv.

BOOL wglUseFontBitmaps (HDC hdc,

```
//contextul de dispozitiv al cărui font se va utiliza
DWORD first,   // caracterul corespunzător primei liste de display
DWORD count,  // numărul caracterelor ce vor fi plasate în listele de display
DWORD 1istBase // specifică identificatorul listei de display a
// primului caracter din font
) ;
```

Exemplu:

În exemplul următor se arată cum se pot afişa caractere în OpenGL utilizând un font bitmap construit pe baza fontului curent din contextul de dispozitiv curent. Pentru aceasta s-a utilizat funcția wglUseFontBitmaps(), care construieşte câte o listă de display pentru fiecare caracter al fontului. Fiecare listă de display nu conține altceva decât o funcție glBitmap care construieşte bitmap-ul corespunzător caracterului respectiv. Spre deosebire de aplicația anterioară, funcția makeRasterFont este înlocuită de funcția CreazaFontBitmap(), cu acelaşi rol. Funcția printstring() este înlocuită de funcția AfiseazaSirBitmap(), cu acelaşi rol. Rezultatul rulării programului este afişat în figura 3-38. Codul aplicației este dat în continuare.

Figura 3.38

```
/* Aplicația arată felul în care sunt redate fonturile
*        în aplicații OpenGL care utilizează ferestrele Windows */

#include <windows.h>
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
void CALLBACK myReshape(GLsizei w, GLsizei h);
void SetDCPixelFormat(HDC hdc);
```

```
        GLuint CreazaFontBitmap(HGLRC hdc, HFONT font, char *typface, int height,
int weight, DWORD italic);
        void StergeFontBitmap(GLuint font);
        void AfiseazaSirBitmap(GLuint font, char *s);
        int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        PSTR szCmdLine, int iCmdShow)
            {
            static char szAppName[] = "Fonturi" ;
            HWND        hwnd ;
            MSG         msg ;
            WNDCLASSEX  wndclass ;
            wndclass.cbSize        = sizeof (wndclass) ;
            wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
            wndclass.lpfnWndProc   = WndProc ;
            wndclass.cbClsExtra    = 0 ;
            wndclass.cbWndExtra    = 0 ;
            wndclass.hInstance     = hInstance ;
            wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
            wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
            wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
            wndclass.lpszMenuName  = NULL ;
            wndclass.lpszClassName = szAppName ;
            wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
            RegisterClassEx (&wndclass) ;
            hwnd = CreateWindow (szAppName, "Fonturi bitmap",

WS_OVERLAPPEDWINDOW|WS_CLIPCHILDREN|WS_CLIPSIBLINGS,
                        50, 50,
                        300, 300,
                        NULL, NULL, hInstance, NULL) ;
            ShowWindow (hwnd, iCmdShow) ;
            UpdateWindow (hwnd) ;
            while (GetMessage (&msg, NULL, 0, 0))
                {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
                }
            return msg.wParam ;
            }
        LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM
wParam, LPARAM lParam)
            {
            static HDC  hDC ;
                  static HGLRC hRC;
                  static GLuint base;
                  HFONT font;
                  switch (iMsg)
```

```
        {
         case WM_SIZE:
                    myReshape(LOWORD(lParam), HIWORD(lParam));
                    break;
                    case WM_CREATE:
                    hDC=GetDC(hwnd);
                    SetDCPixelFormat(hDC);
                    hRC=wglCreateContext(hDC);
                    wglMakeCurrent(hDC,hRC);
                    break;
                    case WM_PAINT:
                                        {
                    glClearColor(1.0, 1.0, 1.0, 1.0);
                    glClear(GL_COLOR_BUFFER_BIT);
                    glColor3f(1.0,0.0,0.0);
                    base=CreazaFontBitmap(hDC, font, "Ariel", 0, 0, FALSE);
                    glRasterPos2i(-4, 0);
                    AfiseazaSirBitmap(base, "123456789!%&'()*+,-");
                    glRasterPos2i(-4,-1);
                    AfiseazaSirBitmap(base, "OpenGl");
                    glRasterPos2i(-4,-2);
AfiseazaSirBitmap(base, "Prelucrare grafica- Tehnologia Informatiei");
                                        glFlush();
                                        ValidateRect(hwnd, NULL);
                                        }
                                        break;
                                case WM_DESTROY:
            StergeFontBitmap(base);
                                        DeleteObject(font);
                                        wglMakeCurrent(hDC, NULL);
                                        wglDeleteContext(hRC);
                                        PostQuitMessage (0) ;
             return 0 ;
        }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
 if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
       5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
    else
    glOrtho (-5.0*(GLfloat)w/(GLfloat)h,
```

```
            5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
          glMatrixMode(GL_MODELVIEW);
        }
        void SetDCPixelFormat(HDC hdc)
        {
                int nPixelFormat;
                static PIXELFORMATDESCRIPTOR pfd={
                        sizeof(PIXELFORMATDESCRIPTOR),
                                1,
                                PFD_DRAW_TO_WINDOW|
                                PFD_SUPPORT_OPENGL,
                                PFD_TYPE_RGBA,
                                24,
                                0,0,0,0,0,0,
                                0,0,
                                0,0,0,0,
                                32,
                                0,
                                0,
                                PFD_MAIN_PLANE,
                                0,
                                0,0,0};
                nPixelFormat=ChoosePixelFormat(hdc, &pfd);
                SetPixelFormat(hdc, nPixelFormat, &pfd);
        }
        GLuint CreazaFontBitmap(HDC hdc, HFONT font, char *typface, int height, int
weight, DWORD italic)
        {
                GLuint base;
        if((base=glGenLists(255))==0)
                return(0);
        if(stricmp(typface, "symbol")==0)
          font=CreateFont(height, 0, 0, 0, weight, italic, FALSE,
        FALSE,SYMBOL_CHARSET,                                   OUT_TT_PRECIS,
CLIP_DEFAULT_PRECIS, DRAFT_QUALITY,
          DEFAULT_PITCH, typface);
        else
        font=CreateFont(height, 0, 0, 0, weight, italic, FALSE, FALSE,
        ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
        DRAFT_QUALITY,
        DEFAULT_PITCH, typface);
        SelectObject(hdc, font);
        wglUseFontBitmaps(hdc, 0, 255, base);
        return(base);
        }
        void StergeFontBitmap(GLuint base)
        {
```

```
if (base==0)
        return;
glDeleteLists(base, 255);
}
void AfiseazaSirBitmap(GLuint base, char *s)
{
        if(base==0)
                return;
        if(s==NULL)
                return;
        glPushAttrib(GL_LIST_BIT);
          glListBase(base);
          glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
        glPopAttrib();
}
```

3.4.3 Redarea pixmap-urilor

Imaginile care folosesc mai mult de două culori sunt numite pixmap-uri (prescurtarea de la "pixel map" din limba engleză). Pixmap-urile sunt utilizate în principal, în OpenGL, ca imagini de fundal sau ca texturi. Un pixmap este un grup de valori destinat memoriei video. Un pixmap (dreptunghi de pixeli) este similar cu un bitmap, cu excepția faptului că, de obicei, valorile reprezintă culori, deşi este alocat spațiu şi pentru alte tipuri de date cum ar fi valori ale adâncimii. în OpenGL, pixmap-urile sunt, în general, fie imagini în index de culoare (8 biți/pixel) fie imagini RGB (24 biți/pixel).

Funcția de redare a pixmap-urilor

Pentru redarea pixmap-urilor, OpenGL dispune de o singură funcție, şi anume glDrawPixels(). Spre deosebire de funcția pentru desenarea bitmap-urilor glBitmap(), această funcție nu permite specificarea unei originii a pixmap-ului şi nici a unui offset pentru poziția raster curentă.

Utilizând glDrawPixels valorile memorate ca un bloc de date în memoria gazdă sunt transmise spre memoria video.

```
void glDrawPixels (GLsizei  width,    //dimensiunea dreptunghiului de pixeli
                   GLsizei height,          //care se va înscrie în memoria video
                   GLenum format,       // formatul datelor pentru un pixel
                   GLenum type,// tipul datelor pentru pixeli
                   const GLvoid *pixels) ;   //un pointer spre tabloul de pixeli
```

Funcția glDrawPixels redă pixelii având colțul stânga jos al imaginii în poziția raster curentă. Un bloc de pixeli din memoria gazdă CPU este transmis spre OpenGL cu un format şi un tip de date specificat. Pentru fiecare pixel din imagine, se generează un

fragment utilizând culoarea restabilită din imagine, şi în continuare este procesat. Odată obținute, valorile rezultate produc un dreptunghi de fragmente. Localizarea acestui dreptunghi este controlată de poziția raster curentă, care este tratată ca un punct (incluzând culoarea şi coordonatele texturii), cu excepția faptului că este setată cu o comandă separată (glRasterPos) care nu apare între glBegin() şi glEnd(). Dimensiunea dreptunghiului este determinată de lățimea şi înălțimea specificată precum şi de setarea parametrilor de zoom ai dreptunghiului de pixeli (setați cu glPixelZoom).

Există numeroase formate şi tipuri de date pentru specificarea depozitării în memorie a pixmap-urilor. Cele mai bune performanțe se obțin prin utilizarea formatului şi a tipului care se potriveşte hardware-ului.

OpenGL permite formate diferite pentru imagini incluzând:

- Imagini RGB sau RGBA conținând un triplet RGB pentru fiecare pixel. Parametrul format este GL_RGB sau GL_RGBA, şi specifică valorile exacte pentru roşu, verde şi albastru pentru fiecare pixel din imagine.
- Imagini color. Parametrul format este GL_COLOR_INDEX şi arată că fiecare valoare din pixmap este un index în paleta de culori curentă a Windows-ului.
- Imagini intensitate care conțin doar intensitatea pentru fiecare pixel. Aceste imagini sunt convertite intern în imagini RGB în tonuri de gri. Parametrul format este GL_LUMINANCE sau GL_LUMINANCE_ALPHA. In acest format, fiecare valoare este mapată la o valoare de intensitate, valoarea maximă corespunzând albului iar valoarea minimă corespunzând negrului, iar valorile intermediare diferitelor tonuri de gri.
- Imagini adâncime (format este GL_DEPTH_COMPONENT) care conțin valoarea adâncimii (coordonata z) pentru fiecare pixel, valoare înscrisă şi în buffer-ul de adâncime, corespunzător buffer-ului de cadru care conține culoarea corespunzătoare fiecărui pixel. Acesta este util în încărcarea buffer-ului de adâncime cu valori şi apoi redarea imaginilor color corespunzătoare cu activarea testului de adâncime.
- Imagini stencil (format este GL_STENCIL_INDEX) care copiază măşti şablon în buffer-ul stencil. Aceasta permite o modalitate uşoară pentru a încărca o mască per pixel complicată.

Parametrul type descrie datele pentru tabloul pixels. Acestea ar putea fi valori ca: GL_FLOAT, GL_INT, GL_BYTE (valori cu semn între -128 şi 127), GL_BITMAP (două valori 0 şi 1) GL_UNSIGNED_BYTE (valori fără semn între 0 şi 255) sau pixeli cu toate componentele de culoare împachetate într-un tip ca GL_UNSIGNED_SHORT_5_6_5.

Parametrul pixels indică adresa de memorie a datelor.

Citirea pixelilor

Aşa cum se pot transmite pixeli spre framebuffer (buffer-ul de cadru), se pot şi citi valorile pixelilor înapoi din framebuffer în memoria gazdă pentru realizarea memorării sau procesării imaginilor. O aplicație evidentă a acestei funcții este salvarea imaginilor create în fişiere. O altă aplicație ar putea fi realizarea unor efecte speciale cu maparea texturilor. Funcția care realizează citirea pixelilor de pe ecran este glReadPixels().

void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei

height, GLenum format, GLenum type, GLvoid *pixels);

Funcția citeşte din framebuffer un dreptunghi de pixeli al cărui colț stânga-jos are poziția specificată prin (x, y). Dimensiunea dreptunghiului este dată de parametrii width şi height. Pixelii se convertesc în mod automat din formatul framebuffer în formatul şi tipul cerut şi sunt plasați în memoria internă la adresa pixels. Pixelii citiți din framebuffer sunt procesați prin modurile de transfer şi depozitare.

Copierea pixelilor în buffer-ul de cadru

Suplimentar, pixelii pot fi copiați dintr-o locație în alta a buffer-ului de cadru utilizând funcția glCopyPixels(). De o asemenea funcție este nevoie, spre exemplu, atunci când sunt mărite anumite porțiuni din imagine. Pixelii sunt procesați de modurile de transfer şi depozitare înainte de a fi returnați în buffer-ul de cadru.

void glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei
height, GLenum type);

Parametrii (x, y) reprezintă poziția colțului stânga jos a dreptunghiului de pixeli care se copiază. Dimensiunile acestui dreptunghi sunt date de parametrii width şi height. Datele sunt copiate într-o nouă poziție din buffer-ul de cadru, care este poziția raster curentă. Parametrul type este fie GL_COLOR, GL_STENCIL sau GLJDEPTH şi specifică ce valori se copiază. Spre exemplu dacă type este GL_COLOR se vor copia valori RGB sau indecşi de culoare.

Scalarea pixmap-urilor

In afara ajustării culorilor unei imagini, folosind funcțiile de mapare a culorilor, un alt atribut care poate fi modificat este dimensiunea unei imagini. Pentru aceasta se utilizează funcția glPixelZoom(). în mod normal fiecare pixel al imaginii este redat pe un pixel al dispozitivului de ieşire. Funcția acceptă doi parametrii în virgulă mobilă, care specifică factorii de scalare ai imaginii pe axele x şi y.

void glPixelZoom(Glfloat zoomx, Glfloat zoomy);

Pentru valori egale ale factorilor de scalare imaginea este scalată păstrând proporțiile pe cele două axe. Valori supraunitare ale factorilor de scalare conduc la o mărire a imaginii iar valori subunitare conduc la o micşorare a imaginii. Dacă se utilizează valori negative se va realiza în plus şi o oglindire a imaginii după axa respectivă în jurul poziției raster curente.

Exemplu 1:

```
glPixelZoomd(1.0, 1.0);       //nu scalează imaginea
glPixelZoom( -l .0,   1.0);   //oglindeşte imaginea pe orizontală
glPixelZoom (1.0,   -2.0);    //oglindeşte imaginea pe verticală, şi dublează
//dimensiunea pe verticală
glPixelZoom(0 .33,   0.33);
//redă imaginea la 1/3 din dimensiune
```

Exemplu 2:

În exemplul următor o imagine de 16 X 16 pixeli va fi copiată în colțul stânga jos al ferestrei şi scalată la 32 x 32 pixeli.

```
int pozx,   pozy;
glPixelZoom(2.0, 2.0);
glRasterPos2i(0, 0);
glCopyPixels(pozx-8, pozy-8, 16, 16, GL_COLOR);
```

Remaparea culorilor

Atunci când o imagine este transferată din memorie în buffer-ul de cadru sau invers din buffer-ul de cadru în memorie, OpenGL poate realiza unele operații asupra ei. Spre exemplu, domeniile componentelor de culoare pot fi alterate. în mod obişnuit, componenta roşu este cuprinsă între 0.0 şi 1.0, dar este posibil să doriți memorarea ei în alte domenii, sau este posibil ca datele utilizate de la diferite sisteme grafice să utilizeze alte domenii de valori. Se pot crea mapări pentru a asigura conversii arbitrare ale indecşilor de culoare sau ale componentelor de culoare în timpul transferării pixelilor. Conversiile asemănătoare celor asigurate în timpul transferării pixelilor la şi de la buffer-ul de cadru sunt numite moduri de transfer a pixelilor (pixel-transfer modes). Aceste conversii sunt controlate de funcțiile glPixelTransfer#() şi glPixelMap#().

Alte moduri de conversie ce pot fi controlate includ buffer-ul de cadru din care se citesc pixelii, şi orice mărire ce se face asupra pixelilor la scrierea lor în buffer-ul de cadru.

Buffer-ele de culoare, adâncime şi stencil deşi au multe similitudini, nu se comportă identic şi câteva moduri au cazuri speciale pentru buffer-e speciale.

Dacă se utilizează formatul GL_COLOR_INDEX, se pot remapa culorile din pixmap sau bitmap utilizând funcțiile glPixelMap() sau glPixelTransfer(). Funcția glPixelTransfer() permite specificarea scalarii şi a offset-urilor pentru valori RGB sau color index. Spre exemplu codul următor arată modul de creştere a luminozității unei imagini cu 10%.

```
glPixelTransferf(GL_RED_SCALE, 1.1);
glPixelTransferf(GL_GREEN_SCALE, 1.1);
glPixelTransferf(GL_BLUE_SCALE, 1.1);
```

În mod similar, pentru a deplasa indecşii de culoare ai unui bitmap la intrări definite special pentru bitmap-ul respectiv, se foloseşte apelul următor:

```
glPixelTransferi(GL_INDEX_OFFSET, bitmap_entry);
```

Exemplu:

Exemplul următor reia aplicația de la redarea bitmap-urilor. Aceeaşi imagine este redată acum ca pixmap, utilizând funcția glDrawPixels(). în continuare sunt date doar funcțiile display() şi main(), care au suferit modificări față de aplicația anterioară. Cele două culori ale bitmap-ului sunt remapate la diferiți indecşi. Pentru valori diferite ale offset-ului, cele două culori cu care sunt redate bitmap-ul se vor modifica.

```
void CALLBACK display(void) {    int i, j;
//urmează tabloul cu bitmap-ul 32X32 biți
void CALLBACK display(void)
{    int i, j;

        GLubyte model[] = {
```

```
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0x03, 0xc0, 0,
                0, 0x0f, 0xf0, 0,
                0, 0x1e, 0x78, 0,
                0, 0x39, 0x9c, 0,
                0, 0x77, 0xee, 0,
                 0, 0x6f, 0xf6, 0,
                 0, 0xff, 0xff, 0,
                0, 0xff, 0xff, 0,
                0, 0xff, 0xff, 0,
                0, 0xff, 0xff, 0,
                0, 0x73, 0xce, 0,
                0, 0x73, 0xce, 0,
                0, 0x3f, 0xfc, 0,
                0, 0x1f, 0xf8, 0,
                0, 0x0f, 0xf0, 0,
                0, 0x03, 0xc0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0,
                0, 0, 0, 0
                };
        glClearIndex(10.0);
           glClear(GL_COLOR_BUFFER_BIT);
                glPixelTransferi(GL_UNPACK_ALIGNMENT, 10);
                glPixelTransferi(GL_INDEX_OFFSET, 16);
                for(i=-5; i<5; i+=2)
                        for(j=-5; j<5; j+=2)
                        {
                        glRasterPos2i(j,i);
                        glDrawPixels(32,    32,    GL_COLOR_INDEX,    GL_BITMAP,
model);
                        }
           glFlush();
        }
        int main(int argc, char** argv)
```

```
{
   auxInitDisplayMode (AUX_SINGLE | AUX_INDEX);
   auxInitPosition (0, 0, 200, 200);
   auxInitWindow ("O fereastrã cu bitmap-uri");
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
   return(0);
}
```

Observații:

Pentru valorile din aplicație ale culorilor, fundalul este gri foarte deschis, bitmap-ul este redat în combinația de culori galben şi albastru (figura 3.39). Trebuie remarcat că funcția auxInitDisplayMode() s-a modificat pentru a lucra cu modelul de culoare index (AUX_INDEX). De asemenea pentru stabilirea culorii background-ului s-a folosit funcția glClearIndex(). Bitmap-ul s-a redat cu funcția glDrawPixels() dar utilizând tipul de date GL_BITMAP, şi formatul GL_COLOR_INDEX.

Figura 3.39

Tabele de mapare a culorilor

Uneori este necesar să se aplice corecții ale culorilor care sunt mai complicate decât simpla scalare liniară sau offset-ul. O aplicație este corecția gamma, în care intensitatea fiecărei valori de culoare este ajustată pentru a compensa iregularitățile de pe monitor sau imprimantă. Funcția glPixelMap() permite realizarea acestui lucru.

void glPixelMap (GL_enum map, GL_int mapsize, const TYPE *values) /

Funcția realizează o mapare al cărui nume simbolic este indicat de map. Valoarea simbolică map poate lua 10 valori:

GL_PIXEL_MAP_I_TO_I - se realizează o mapare a indecşilor de culoare la indecşi de culoare;

GL_PIXEL__MAP_S_TO_S - se realizează o mapare a indecşilor şablon (stencil) la indecşi stencil;

GL_PIXEL_MAP_I_TO_R - se realizează o mapare a indecşilor de culoare la componente roşu;

GL_PIXEL_MAP_I_TO_G - se realizează o mapare a indecşilor de culoare la componente verde;

GL_PIXEL_MAP_I_TO_B - se realizează o mapare a indecşilor de culoare la componente albastru;

GL_PIXEL_MAP_I_TO_A - se realizează o mapare a indecşilor de culoare la componente alfa;

GL_PIXEL_MAP_R_TO_R - se realizează o mapare a componentelor roşu la componente roşu;

GL_PIXEL_MAP_G_TO_G - se realizează o mapare a componentelor verde la componente verde;

GL_PIXEL_MAP_B__TO_B - se realizează o mapare a componentelor albastru la componente albastru;

GL_PIXEL_MAP_A_TO_A - se realizează o mapare a componentelor alfa la componente alfa;

Parametrul mapsize arată dimensiunea mapării care se va realiza. Parametrul values este un pointer spre tabloul care conţine valorile pentru mapare.

Exemplu:

In exemplul următor se arată felul cum se poate realiza o corecţie gamma.

```
GLfloatl 1ut[256];
GLf loat gamma_value;
int i ;
gamma_value=l. 7;    //Pentru monitoare video NTSC
//se încarcă tabela de căutare cu valori corespunzătoare
// monitorului cu care se lucrează
for(i=0;   i<256;   i++)
lut[i]=pow (i/255.0, 1.0/gamma_value); //se activează maparea
glPixelTransferi(GL_MAPCOLOR,   GL_ TRUE);
//se realizează maparea
glPixelMap(GL_PIXEL_MAP_R_TO_R/ 256, lut);
glPixelMap(GL_PIXEL_MAP_G_TO_G/ 256, lut);
glPixelMap(GL_PIXEL_MAP_B_TO_B/ 256, lut);
```

Moduri de memorare sau de transfer a pixelilor

În această secţiune se arată detalii ale modurilor de memorare şi de transferare.

O imagine depozitată în memorie conţine pentru fiecare pixel între unul şi patru elemente. Aceste elemente pot fi indexul de culoare sau intensitatea luminoasă dar pot fi şi componentele roşu, verde, albastru şi alfa. Aranjamentele posibile pentru datele corespunzătoare pixelilor, sau formatele, determină numărul elementelor memorate pentru fiecare pixel şi ordinea lor.

Unele elemente sunt valori întregi iar altele sunt valori în virgulă mobilă cuprinse între 0 şi 1. Numărul exact de biţi utilizat pentru reprezentarea comp0nentelor diferă de la un hardware la altul. De aceea uneori este o risipă să se memoreze fiecare componentă ca un număr în virgulă mobilă pe 32 de biţi, mai ales că imaginile pot, uşor, conţine un milion de pixeli.

Elementele pot fi memorate ca diferite tipuri de date, de la octeţi la întrebi sau numere în virgulă mobilă pe 32 de biţi. OpenGL defineşte explicit conversia fiecărei componente, în fiecare format, pentru fiecare din tipurile posibile de date. Trebuie reţinut că se pierde din rezoluţie atunci când se încearcă memorarea pe un număr mai mic de biţi.

Datele corespunzătoare unei imagini sunt memorate în memoria procesorului ca tablouri cu două sau trei dimensiuni. Deseori, este nevoie să se afişeze sau sa se memoreze o subimagine care corespunde unui sudreptunghi dintr-un tablou. Suplimentar, este nevoie să se ia în considerare diferite maşini care au diferite convenţii de ordonare a octeţilor. În sfârşit, unele maşini au astfel hardware-ul încât este mai eficient să se mute datele înspre şi dinspre buffer-ul de cadru dacă datele sunt aliniate pe doi octeţi, patru octeţi sau opt octeţi în memoria procesorului. În asemenea cazuri poate fi necesar să se controleze alinierea octeţilor. Toate aceste lucruri, descrise în acest subcapitol, sunt controlate de modurile de stocare al pixelilor. Aceste moduri se specifică utilizând comanda glPixelStore#(). De obicei trebuie făcute câteva apeluri succesive ale acestei comenzi pentru a seta valorile câtorva parametrii.

void glPixelStore{if}(Glenum pname, TYPE param) ;

Funcţia afectează modurile de depozitare, care intervin în operaţiile realizate de funcţiile: glDrawPixels, glReadPixels, glBitmap, glPolygonStippla, glTexImage şi glGetTexImage.

Parametrul pname poate lua 12 valori. Dintre acestea, 6 valori intervin în felul în care datele sunt scrise în memorie şi deci afectează doar comenzile glReadPixels() şi glTexImage().

GL_PACK_SWAP_BYTES
GL_PACK_LSB_FIRST
GL_PACK_ROW„LENGTH
GL_PACK_SKIP_ROWS
GL_PACK_SKIP_PIXELS
GL_PACK_ALIGNMENT

Celelalte 6 tipuri de valori ale parametrului pname intervin în felul în care sunt citiţi pixelii din memorie şi afectează funcţiile glDrawPixels, glBitmap, glPolygonStipple, glTexImage.

GL_UNPACK_SWAP_BYTES
GL_UNPACK_LSB_FIRST
GL_UNPACK_ROW_LENGTH
GL_UNPACK_SKIP_ROWS
GL_UNPACK_SKIP_PIXELS
GL_UNPACK_ALIGNMENT

Valorile #SWAP_BYTES controlează ordinea octeţilor în memorie care poate sau nu să fie inversată după cum parametrul este true sau false. Acest parametru poate fi ignorat dacă nu se lucrează cu imagini create pe procesoare diferite.

Valoarea #LSB_FIRST (low significant bit) este utilă atunci când se lucrează cu pixmap-uri de un bit sau cu bitmap-uri. Dacă parametrul este false, biţii sunt preluaţi din octet începând cu bitul cel mai semnificativ.

Atunci când dorim să desenăm sau să citim un subdreptunghi dintr-un dreptunghi al unui pixmap memorat în memorie, se utilizează celelalte valori enumerate. Dacă dreptunghiul din memorie este mai mare decât subdreptunghiul care se va desena, trebuie să se specifice lungimea dreptunghiului mai mare cu #ROW_LENGTH. Dacă parametrul #ROW_LENGTH este 0 (implicit), atunci se sub înţelege că lungimea este valoarea parametrului width specificat în funcţiile

glReadPixels(), glDrawPixels(), glCopyPixels(). Trebuie să se specifice de asemenea numărul rândurilor şi al pixelilor care se sar înainte de a se începe copierea datelor din subdreptunghi. Aceste numere sunt setate utilizând parametrii

Exemplu:

Spre exemplu, pentru a afişa o imagine de 200 x 200 pixeli, aflată în centrul unei imagini de 500 x 300 se va utiliza următorul cod:

```
glPixelStorei(GL_UNPACK_ROW_LENGTH, 500);
 glPixelStorei(GL_UNPACK_SKIP_PIXELS, (500-200)/2);
glPixelStorei(GL_UNPACK_SKIP_ROWS, (300-200)12);
glDrawPixels(200, 200, GL_RGB, GL_UNSIGNED_BYTE, model);
```

3.5 Utilizarea atributelor de redare în OpenGL

Un obiect poate fi redat cu diferite atribute:

a) reprezentare wireframe

b) reprezentare wireframe cu poligoanele având atribut de umplere;

c) reprezentare cu iluminare folosind un model de iluminare constantă pe poligoane;

d) reprezentare cu iluminare Phong;

e) reprezentare cu texturare;

f) reprezentare cu texturare.

OpenGL poate reda obiectele de la modul wireframe până la texturarea acestora sau la utilizarea unor modele complexe de iluminare cum ar fi modelul Gouraud. Aşa cum s-a mai arătat în capitolul introductiv, OpenGL se comportă ca o maşina de stare. Toate atributele de redare sunt încapsulate în starea OpenGL:

- Stiluri de redare
- Umbrire
- Iluminare
- Maparea texturii

De fiecare dată când OpenGL procesează un vârf, utilizează datele memorate în tabelele sale interne de stare pentru a determina cum se va transforma vârful, cum se va lumina, textura, etc.

3.5.1 Controlarea stării OpenGL

Aspectul obiectelor desenate de aplicațiile OpenGL este controlat de starea curentă. Fluxul general al oricărei redări OpenGL este de a se seta starea curentă, apoi de a se transmite primitiva de redat, şi de a se repeta aceşti paşi pentru fiecare primitivă.

pentru fiecare (primitivă de reprezentat) {

- se actualizează starea OpenGL
- se redă primitiva }

În general, metoda cea mai obişnuită de a controla starea OpenGL este de a seta atributele vârfurilor, care includ culoarea, normalele pentru iluminare, coordonatele de texturare.

glColor#() / glIndex()
glNormal#()
glTexCoord#()

3.5.2 Setarea stării curente

Setarea stării OpenGL de obicei include modificarea atributelor de redare, cum ar fi încărcarea unei texturi sau setarea grosimii unei linii. Alte atribute necesită însă activarea lor. Aceasta se face prin utilizarea funcției glEnable(), şi transmiterea numelui stării respective, cum ar fi GL_LIGHTO (pentru activarea sursei de lumină 0) sau GL_POLYGON_STRIP (pentru activarea şabloanelor de umplere ale poligoanelor).

Exemple:

Setarea stării

```
glPointSize(size) ;      //stabileşte grosimea liniei
glLineStipple (repeat,   pattern) ; //stabileşte stilul liniei
glShadeModel (GL_SMOOTH) ;  //umbrire Gouraud
```

Activarea atributelor

```
glEnable (GL_LIGHTING) ;        //se activează reprezentarea cu iluminare
glDisable (GL_TEXTURE_2D) ; //se dezactivează reprezentarea cu texturare
                            //2D
```

3.5.3 Interogarea stării şi stiva de parametrii

Aşa cum s-a mai spus, OpenGL este o maşină de stare. Ea poate fi pusă în diferite stări (sau moduri) care au efect până când sunt schimbate. Fiecare variabilă de stare sau mod are o valoare implicită, şi în orice punct al programului se poate interoga sistemul cu privire la fiecare din valorile curente ale variabilelor. În mod obişnuit, se va folosi una din următoarele patru comenzi pentru a afla aceste valori: glGetBooleanv(), glGetDoublev(), glGetFloatv(), glGetIntegerv(). Care dintre aceste comenzi se va alege depinde de tipul datelor care se doresc a fi obținute. Unele variabile de stare au comenzi de interogarea stării mult mai specifice (cum ar fi glGetLight(), glGetError(), glGetPolygonStipple()). Există de asemenea o stivă pentru valorile parametrilor care pot fi salvați sau restaurați din stivă. Comenzile pentru salvarea respectiv restaurarea din stiva de atribute de stare sunt: glPushAttrib() şi glPopAttrib(). Comenzile get şi stivele de parametrii fac posibilă implementarea diferitelor biblioteci, fiecare fără a interfera cu altă utilizare OpenGL.

4 INDRUMAR DE LABORATOR

Programare Grafică – OpenGL

4.1 Lucrare de laborator 1.

4.1.1 Introducere în OpenGL

OpenGL (Open Graphic Library) nu este cum s-ar putea crede un limbaj de programare ci un standard de programare al aplicațiilor 3D. A fost primul standard care s-a impus pe piață fiind inclus deja la primele versiuni de Windows95 şi WindowsNT4.0. Practic odată cu biblioteca de funcții Win32 a apărut şi suportul pentru OpenGL (separat desigur…).

OpenGL este independent de mediul de programare, fiind definite aceleaşi tipuri de date şi aceleaşi funcții indiferent dacă se programează în Visual C, Visual Basic, Delphi, CBuilder ş.a. Totuşi se poate observa o oarecare înrudire între OpenGL şi C pe măsură ce se avansează în programare şi se capătă o oarecare experiență şi familiaritate cu OpenGL. Nu se pot nega nici oarecare asemănări cu Delphi, dar mai puține ca număr şi mai subtile.

Astfel chiar dacă veți învăța OpenGL utilizând Vizual C, veți putea trece relativ rapid şi fără dificultați la alt mediu de programare mai familiar, dar care are biblioteci OpenGL. Saltul mai mare se va face dacă se va aborda alt standard de programare 3D, Direct 3D spre exemplu, dar având concepte comune se va însuşi mult mai uşor. Ca fapt divers în versiunile Visual C, s-a renunțat la un moment dat la actualizare bibliotecii OpenGL, aceasta rămânând la versiune 1.1, în schimb s-a introdus suport pentru Direct 3D. Alte medii de programare continuă încă să includă şi să actualizeze bibliotecile OpenGL, şi aceasta cu atât mai mult cu cât marea parte a distribuțiilor de Linux se bazează pe acest standard. În Linux s-a impus deja GLUT (OpenGL Utility Toolkit) şi mai nou freeGLUT.

Deci dacă doriți implementare în medii freeware de aplicații 3D aceasta este soluția.

Linkuri utile:

- http://freeglut.sourceforge.net/
- www.**opengl**.org

4.1.2 Crearea unei aplicații OpenGL

Pentru a crea o aplicație OpenGL vom folosi în principal 4 biblioteci :

- gl.h – bibliotecă exclusiv după standardul OpenGL;

- glu.h – bibliotecă auxiliară pentru integrarea OpenGL în mediul de programare şi nu numai;
- glaux.h – bibliotecă auxiliară pentru crearea şi testarea rapidă de aplicaţii OpenGL;
- glos.h – microbibliotecă pentru corecţia unui mic bug din bibliotecile OpenGL din Visual C.

Alături de acestea mai avem nevoie de 3 librării:

- glu32.lib
- glaux.lib
- opengl32.lib

Ce se găsesc în directorul LIB al Visual C.

În plus sistemul de operare trebuie să includă neapărat în directorul system32 – opengl32.dll.

Vom utiliza pentru aceste laboratoare Visual C++ 6.0. Programele putând funcţiona şi în Visual C++ 5.0 , ca de altfel şi în Visual Studio Net, şi ultima versiune de Visual Studio 2008.

Iniţial vom lucra în modul consolă.

Vom proceda astfel:

1. Rulăm Visual C++;
2. Din File alegem New;
3. Trecem pe primul tab Files;
4. Alegem şi selectăm C++ source file;
5. Denumim fişierul xxxx;
6. Scriem programul.
7. Apăsam F7 pentru a compila programul.
8. Apăsăm Yes pentru a crea un spaţiu de lucru pentru program.
9. Următorul meniu Yes ,salvăm modificările;
10. Din meniul Project – alegem Settings ;
11. Alegem tabul Link din fereastra Project Settings ce s-a deschis;
12. La project options scriem librăriile folosite de OpenGL – glu32.lib, glaux.lib, opengl32.lib cu spaţii între ele şi fără virgulă şi la sfârşit apăsăm OK.
13. Rulăm programul cu F5.

Exemplu de program:

```
#include <glos.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include <conio.h>
void main()
{
        //initializare modului de afisare –
        //un singur buffer – afisare direct pe ecran –
    //culori după standardul RGBA
                auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
```

```
//poziția si dimensiunile ferestrei de afisare
        auxInitPosition(100,100,250,250);
//initializam fereastra precizand titlul acesteia
  auxInitWindow("My first OpenGL Program");

// aici se scrie codul OpenGL pentru afişare
//stergem fereastra folosind culoarea albastru
glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

glFlush();
// Nu inchide! Asteapta apasarea unei taste
        getch();

}
```

Observați că avem o secvență de inițializare a mediului OpenGL:

```
        auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
        auxInitPosition(100,100,250,250);
  auxInitWindow("My first OpenGL Program");
```

Şi o secvență de sfârşit care poate lipsi. Între aceste putând scrie instrucțiuni OpenGL de
afişare.

Avem instrucțiunea glClearColor(…) care stabileşte culoare de ştergere a ferestrei sau a buferului având prototipul:

```
void glClearColor(GLclampf rosu, GLclampf verde,
        GLclampf albastru,GLclampf alpha);
```

GLclampf este un tip de date numeric asemănător cu float din C.

Culoare se specifică prin componenții ei rosu, verde şi albastru, şi o componentă de transparență alpha care dacă nu este folosită este setată pe 1, adică complet opac.

1.0f este intensitatea maximă a componentei 0.0f cea minimă. Observați că după fiecare constantă numerică de tip float se pune litera f, pentru a se evita ambiguitățile şi în special mesajele de avertizare (warning) de la compilare.

Instrucțiunea glClear (…) stabileşte buferul care va fi şters prin culoare specificată în glClearColor, ea are prototipul:

```
void glClear(GLbitfield mask);
```

unde mask poate lua valorile:

- GL_COLOR_BUFFER_BIT;
- GL_DEPTH_BUFFER_BIT;

- GL_ACCUM_BUFFER_BIT;
- GL_STENCIL_BUFFER_BIT.

Pentru primele aplicații o vom folosi doar cu masca GL_COLOR_BUFFER_BIT.

Pentru a desena un dreptunghi folosim funcția glRect cu următoarele prototipuri:

```
void glRectd(GLdouble x1, GLdouble y1, GLdouble x2, GLdouble y2);
void glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
void glRecti(GLint x1, GLint y1, GLint x2, GLint y2);
void glRects(GLshort x1, GLshort y1, GLshort x2, GLshort y2)
```

Se observă că OpenGL lucrează atât cu date specificate prin tipuri întregi cât şi prin date cu virgulă. Rezultatul va fi desenarea unui dreptunghi ce are colțul din stânga sus descris de x1,y1 şi cel din dreapta jos descris de x2,y2.

În OpenGL sistemul de coordonate nu mai este ca cel definit în Windows sau BGI, axa Oy este îndreptată în sus şi nu în jos ca în cazul acesta. Axa Oz iese din ecran, Ox este de la stânga spre dreapta.

Exercițiu 1. Modificați programul anterior pentru a desena câte patru dreptunghiuri, unul verde, unul roşu, unul albastru şi unul galben pe un fond negru. Pentru a modifica culoare dreptunghiului se foloseşte funcția glColor3f() care are prototipul:

```
void glColor3f(GLfloat red, GLfloat green, GLfloat blue);
```

4.2 Lucrare de laborator 2.

4.2.1 Utilizarea funcţiilor bibliotecii glaux

Biblioteca glaux este o bibliotecă auxiliară, care deşi compatibilă cu standardul OpenGL nu este prevăzută de acesta. Ea fost implementată în special pentru Visual C++, celelalte medii de obicei având doar bibliotecile glu şi gl, sub acest nume sau altul. Biblioteca glaux oferă funcţii de integrare a aplicaţiei 3D în mediul de programare. Aplicaţiile vor rula iniţial o aplicaţie de tip consolă, aceasta la rândul ei deschizând o altă fereastră în care va fi afişată grafica OpenGL. Pentru început se va dovedi destul de utilă, codul OpenGL putând fi testat rapid, dar dacă se va dori o aplicaţie mai serioasă se va renunţa la glaux şi se vor folosi opţiunile de integrare a graficii OpenGL într-o aplicaţie gen Win32 sau MFC (aceasta prin funcţiile bibliotecii glu).

Funcţiile disponibile în biblioteca glaux sunt prefixate întotdeauna de cuvântul „aux”.

Aceste funcţii pot fi împărţite în trei categorii:

- funcţii de iniţializare – care definesc bufferele folosite, poziţia ferestrei de afişare, generează fereastra;
- funcţii de interfaţă cu utilizatorul – pentru mouse şi tastatura;
- funcţii pentru desenarea de primitive 3D - se pot desena primitive ca model din sârme (wire frame), sau cu suprafaţă continuă (solids).

4.2.2 Funcţii de iniţializare:

Funcţia auxInitDisplayMode realizează iniţializarea modului grafic folosit de OpenGL ea având următorul prototip:

void auxInitDisplayMode(GLbitfield mask);

unde mask este o mască cu diferite opţiuni de configurare – aceste opţiuni sunt:

- AUX_SINGLE imaginea se va afişa prin intermediul unui singur buffer direct legat de ecran;
- AUX_DOUBLE imaginea va fi desenată mai întâi într-un buffer auxiliar, fiind afişată apoi pe ecran prin intermediul bufferului principal ;
- AUX_RGBA – culoare va fi specificată prin cele trei componente de culoare roşu, verde, albastru şi o componentă de transparenţă alfa;
- AUX_INDEX – culoare va fi specificată prin index şi nu direct;
- AUX_DEPTH – specifică că bufferul de adâncime folosit va fi pe 32 de biţi; AUX_DEPTH16 – specifică că bufferul de adâncime folosit va fi pe 16 de biţi;
- AUX_STENCIL – specifică că va fi folosit un buffer şablon;
- AUX_ACCUM – specifică că va fi folosit un buffer de acumulare;
- AUX_ALPHA – specifică că va fi folosit un buffer pentru parametrul de transparenţă alfa ;
- AUX_FIXED_332_PAL – specifică o paletă de culoare 3-3-2 pentru fereastră ;

În mod obişnuit vom folosi această funcție astfel :

auxInitDisplayMode(AUX_DOUBLE | AUX_RGBA) ;

,pentru a lucra cu două buffere, evitând astfel efectul de pâlpâire la animații sau compoziții complexe ;

sau mai simplu :

auxInitDisplayMode(AUX_SINGLE | AUX_RGBA) ;

, grafica va fi afişată direct pe ecran, relativ nepretențios pentru început ;

Pentru a specifica poziția ferestrei de afişare se va folosi funcția auxInitPosition cu
următorul prototip :

void auxInitPosition(GLint x, GLint y, GLsizei lungime, GLsizei inaltime);

În care sunt specificate coordonatele colțului stânga – sus şi dimensiunile ferestrei.

Exemplu:

auxInitPosition(10,10,200,200);

Pentru a genera efectiv fereastră, inclusiv pentru a preciza titlul acesteia se va folosi funcția auxInitWindow cu următorul prototip:

void auxInitWindow(GLBYTE *titlul ferestrei);

Exemplu:

auxInitWindow(“ Primul program în OpenGL”);

4.2.3 Funcții de interfață şi de ciclare

Până acum pentru a putea vedea grafica desenată în fereastră, evitând închidere imediată a ferestrei am folosit funcția getch() din biblioteca conio.h. Acelaşi lucru se poate realiza însă prin folosirea unor funcții de ciclare. Acestea au prototipul :

void auxIdleFunc(AUXIDLEPROC NumeleFunctiei);
void auxMainLoop(AUXMAINPROC NumeleFunctiei);

Funcția auxIdleFunc este apelată atunci când programul a terminat de desenat o scenă şi aşteaptă alte comenzi, o putem folosi fără probleme pentru afişarea aceleaşi scene sau pentru generarea de animații.

auxMainLoop este utilizată exclusive pentru ciclări, odată terminat ciclul, acest ciclu este reapelat, totul terminându-se la închiderea ferestrei.

Variabila NumeleFunctiei este o numele unei funcții definită astfel:

```
void CALLBACK NumeleFunctiei(void)
        {
        //codul OpenGL prin care desenam ceva în mod repetat

        }
```

În program putem apela doar odată ori una ori alta din funcțiile de ciclare.

NumeleFunctiei o putem denumi fie "Desenare", fie "RandareScena", cum anume ni se pare mai sugestiv.

Exercițiu 1. Creați un program în care un dreptunghi să se mişte de la stânga la dreapta în fereastră. Când a ajuns la marginea din dreapta să se oprească. Odată realizat programul încercați ca odată ajuns la margine să se întoarcă înapoi, repetând indefinit această mişcare.

4.2.4 Funcțiile de lucrul cu mouse

Avem o funcție de asociere a unei funcții unui eveniment de mouse aceasta are prototipul :

void auxMouseFunc(int button, int mode, AUXMOUSEPROC func);

, unde button este definit prin următoarele constante:

- AUX_LEFTBUTTON – se referă la butonul stâng al mouse-ului;
- AUX_MIDDLEBUTTON- se referă la butonul din mijloc al mouse-ului;
- AUX_RIGHTBUTTON – se referă la butonul drept al mouse-ului;

, mode specifică evenimentul în care este implicat butonul precizat anterior :

- AUX_MOUSEDOWN – butonul a fost apăsat şi este ținut apăsat;
- AUX_MOUSEUP – butonul a fost eliberat;

În ceea ce priveşte func aceasta este numele unei funcții şi are următorul mod de declarare :

```
void CALLBACK MouseFunc(AUX_EVENTREC *event)
{
// operatii legate de mouse, desenari…etc.
}
```

Unde event este o variabilă de tip structură prin care se transmit informații despre starea mouse-ului.

```
typedef struct _AUX_EVENTREC {
        GLint event;
        GLint data[4];
    }
```

Event specifică evenimentul – definit prin constantele – AUX_MOUSEUP / AUX_MOUSEDOWN ;

Data este apelat cu următoarele secvențe :

- data[AUX_MOUSEX] – coordonata x a pointerului de mouse;
- data[AUX_MOUSEY] - coordonata y a pointerului de mouse;
- data[MOUSE_STATUS] – starea butoanelor de mouse;

Exemplu:

```
void CALLBACK MouseFunc(AUX_EVENTREC *event)
{
int mousex,mousey;
mousex = data[AUX_MOUSEX] ;
mousey = data[AUX_MOUSEY] ;

}

….
void main(void)
{
…
auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN, MouseFunc);
…
}
```

Exercițiu 2. Creați un program pe baza secvenței de mai sus, care la apăsarea primului buton al mouse-ului să deseneze un dreptunghi pe ecran cu centrul exact la poziția pointerului de mouse. La fiecare apăsare a butonului, la diverse poziții dreptunghiul va fi redesenat.

4.2.5 Funcțiile de lucru cu tastatura

Pentru a se lucra cu tastatura se foloseşte funcția cu următorul prototip:

```
void auxKeyFunc(GLint key, void(*function(void));
```

,la care key este codul tastei cărei i se asociază funcția function.

O listă de coduri de taste este următoarea:

- AUX_ESCAPE –pentru tasta Escape;
- AUX_SPACE – pentru bara de spațiu;
- AUX_RETURN – pentru tasta Enter;
- AUX_LEFT The – pentru tasta spre stânga;
- AUX_RIGHT – pentru tasta spre dreapta;
- AUX_UP The – pentru tasta în sus;
- AUX_DOWN – pentru tasta în jos;
- AUX_A … AUX_Z pentru tastele cărora le corespund litere;
- AUX_0 … AUX_9 pentru tastele cărora le corespund cifre;

Funcția asociată tastei are codul general valabil:

```
void CALLBACK NumeFunctie(void)
{
…
}
```

Exercițiu 3. Scrieți un program pentru a deplasa cu tastele săgeți un dreptunchi în fereastră.

Funcțiile grafice propriu-zise :

Pentru a desena obiecte grafice 3D predefinite se folosesc funcțiile ce au următoarele prototipuri :

```
void auxSolidBox(GLdouble lungime, GLdouble inaltime, GLdouble adancime
);
void auxSolidCone(GLdouble raza, GLdouble inaltime);
void auxSolidCube(GLdouble lungimeLatura);
void auxSolidCylinder(GLdouble raza, GLdouble inaltime);
void auxSolidDodecahedron(GLdouble raza);
void auxSolidIcosahedron(GLdouble raza);
void auxSolidOctahedron(GLdouble raza);
void auxSolidSphere(GLdouble raza);
void auxSolidTeapot(GLdouble dimensiune);
void auxSolidTetrahedron(GLdouble raza);
void auxSolidTorus(GLdouble RazaInterioara, GLdouble RazaExterioara);
```

Observați că toate aceste funcții au un anumit mod de contrucție al numelui :

aux + ModDeDesenare + TipulCorpuluiGeometric

ModDeDesenare poate fi Solid- adică cu suprafață continuă sau Wire – cu suprafață reprezentată prin linii interconectate.

Pentru tipul geometric avem următoarele cazuri :

- Box - un paralelipiped dreptunghic – definit prin lungime, înălțime, adâncime ;
- Cone – un cod – definit prin raza bazei şi înălțimea ;

- Cube – un cub definit prin mărimea laturii ;
- Cylinder – un cilindru definit prin rază şi înălţime;
- Dodecahedron – un dodecaedru definit prin rază sferei cel circumscrie;
- Icosahedron – un icosaedru definit prin rază sferei cel circumscrie;
- Octahedron – un octaedru definit prin rază sferei cel circumscrie;
- Sphere – o sferă definită prin rază;
- Teapot – un ceanic definit de o mărime specifică după care se scalează;
- Tetrahedron – un tetraedru definit prin rază sferei cel circumscrie;
- Torus – un tor (un corp de forma unei gogoşi) definit prin rază zonei umplute şi raza exterioară.

Se observă că la aceste corpuri nu li s-a precizat poziţie. Ele vor desenate la poziţia actuală a centrului sistemului de coordinate. Pentru a modifica poziţia se foloseşte funcţia

glTranslatef cu prototipul :

void glTranslatef(GLfloat x, GLfloat y, GLfloat z);

Unde x, y, z semnifică deplasări pe ox, oy, oz ale centrului de referinţă.

Exerciţiu 4. Desenaţi pe ecran în modul Wire toate corpurile 3D precizate mai sus, încercaţi să le afişaţi pe toate în fereastră pentru a acoperi toată fereastra folosiţi translaţii cu ajutorul funcţiei glTranslate(…). Sunt nevoie doar de translaţii de ox şi oy.

“Duble – buffering”

Pentru a înlătura pâlpâirea ecranului la animaţii veţi folosi următoarele secvenţe de cod:

```
auxInitDisplayMode(AUX_DOUBLE | AUX_RGBA) ;
```

,pentru iniţializare, iar pentru a afişa bufferul secundar se va apela funcţia următoare, aceasta după ce s-a desenat tot ce se dorea :

```
auxSwapBuffers();
```

4.3 Lucrare de laborator 3.

4.3.1 Aplicații OpenGL în Win32

În laboratorul anterior s-a spus că pentru aplicații serioase ale OpenGL se va lucra cu aplicații Win32 sau MFC. În continuare vom prezenta paşii ce trebuie urmați pentru a crea o astfel de aplicație. Este relativ simplu de a crea o aplicație Win32 necompletată, mai greu însă este crearea codului de bază ce va inițializa grafic OpenGL şi mai ales fereastra de aplicație. Pentru astfel de tipuri de aplicații nu vom scrie pentru fiecare aplicație nouă codul de la zero, ar fi contraproductiv, vom crea în schimb o aplicație şablon în care să avem definite elementele generale pe care le dorim a le folosi în orice aplicație viitoare, urmând să copiem codul acestei aplicații ori de câte ori dorim să realizăm o aplicație OpenGL în stilul Win32.

4.3.2 Crearea unei aplicații Win32 necompletate

Vom folosi pentru aceasta meniurile de creare de proiect specifice Visual C.
Vom parcurge următorii paşi:

1. Din meniul File apăsăm New;
2. Alegem din tabul Projects – Win32 Application;
3. Denumim proiectul în caseta de editare etichetată Project Name;
4. Apăsăm OK.
5. În fereastra următoare alegem - An empty project- şi apăsăm Finish;
6. Mai apare o fereastră de confirmare în care apăsăm OK – Am creat proiectul;
7. Din fereastra de vizualizare din partea stângă – selectăm modul FileView – ce specifică că se doreşte a se vizualiza fişiere;
8. Din meniul File apasăm New;
9. Din fereastră selectăm tabul –files- şi de aici alegem -C++ source file-;
10. Denumim fişierul şi apăsăm OK, va apărea o fereastră goală în care vom scrie codul programului.

Acesta va conține în primul rând includerea bibliotecilor:

```
#include <windows.h> //pentru lucrul cu ferestre şi funcții Win32
#include "glos.h"
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include <stdio.h> //facultativa
#include <string.h>//pentru lucrul cu sirurile de caractere
#include <math.h>// pentru lucrul cu funcții matematice
```

Apoi o definire a constantelor şi variabilelor globale.

```
const char titlulferestrei[]="OpenGL Afisarea texturarii unui cilindru";
```

```
static HGLRC hRC;  // dispozitivul de randare - obligatoriu
static HDC hDC;    // dispozitivul de context – suprafața ferestrei adică - obligatoriu
```

Aceste sunt recomandabile a fi definite sau chiar obligatorii.
Corpuri de funcții utilizate.
Funcția principala de desenare a graficii OpenGL:

```
void CALLBACK deseneaza(void)
{
// comenzi – functii opengl
}
```

Funcția de setare modului grafic, esențială pentru crearea unei ferestre compatibile cu OpenGL:

```
void SetDCPixelFormat(HDC hDC)
    {
    int nPixelFormat;

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),  // dimensiunea structurii
        1,                      // Versiunea structurii
      PFD_DRAW_TO_WINDOW |          // desenează în ferestre
        PFD_SUPPORT_OPENGL |          // suportă grafica OpenGL
        PFD_DOUBLEBUFFER,           // se suporta duble - buffering
        PFD_TYPE_RGBA,              // culoare respecta standardul RGBA
        24,                     // culoare pe 24 de biti
        0,0,0,0,0,0,                // nefolositi
        0,0,                    // nefolositi
        0,0,0,0,0,                  // nefolositi
        32,                     // Buffer de adâncime pe 32 de biți
        0,                      // nefolosit
        0,                      // nefolosit
        PFD_MAIN_PLANE,             // deseneaza in planul principal
        0,                      // nefolosit
        0,0,0 };                // nefolositi

    // Alege un mod cât mai apropiat de cel solicitat in structura
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);
    // Stabileste modul grafic asociat contextului de dispozitiv hDC
    SetPixelFormat(hDC, nPixelFormat, &pfd);
    }
```

Funcția de răspuns la redimensionarea ferestrei:

```
void CALLBACK resizefunc(GLsizei w,GLsizei h)
{
        if (w==0)        w=1;
        if (h==0)       h=1;
   glViewport(0,0,w,h);
        glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
        GLsizei w2,h2;
        if (w>h){
                w2=nRange*w/h;
                h2=nRange;
        }else
        {
                w2=nRange;
                h2=nRange*h/w;
        }
        glOrtho(-w2,w2,-h2,h2,-nRange,nRange);
        glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();

}
```

Funcția repetată prin intermediul unui timer:

Exemplu:

```
void CALLBACK Repeta(void)
{
        if (roteste_on)
        {
        a=(a+4)%360;
        b=(b+4)%360;
   deseneaza();
        }
}
```

Şi cel mai important funcția de gestionare a mesajelor, practic nucleul unei aplicații Win32:

Exemplu:

```
LRESULT CALLBACK WndProc(     HWND    hWnd,
                        UINT    message,
                        WPARAM  wParam,
                        LPARAM  lParam)
      {
```

```
  switch (message)
          {
          // Apelata la crearea ferestrei
          case WM_CREATE:
                // extragem contextul de dispozitiv
                hDC = GetDC(hWnd);

                // selectam modul grafic compatibil OpenGL asociat ferestrei
                SetDCPixelFormat(hDC);

                //creem contextul de randare al graficii OpenGL
                hRC = wglCreateContext(hDC);
               // asociam contextului de dispozitiv curent contextul de randare
              //facandu-l pe acesta activ
                wglMakeCurrent(hDC, hRC);

                // creem un timer ciclat la un interval de o milisecunda
                SetTimer(hWnd,10000,1,NULL);
                break;

  // La inchiderea ferestrei
  case WM_DESTROY:
        // dezactivam timerul
     KillTimer(hWnd,101);

         //deselectam contextul de randare din contextul de dispozitiv
        wglMakeCurrent(hDC,NULL);
         // stergem contextul de randare
        wglDeleteContext(hRC);
        // va iesi din aplicatie odata ce fereastra de aplicatia a disparut
        PostQuitMessage(0);
        break;

  // Apelat la redimensionarea ferestrei
  case WM_SIZE:
    //apelam functia de redimensionare a ferestrei
        resizefunc(LOWORD(lParam), HIWORD(lParam));
        break;

 // apelat prin intermediul timerului o dată la fiecare milisecunda
  case WM_TIMER:
        {
//functie care se repeta o data la o milisecunda
        Repeta();
      }
        break;
```

```
      // Apelat la desenarea sau redesenarea ferestrei
      case WM_PAINT:
            {
            // desenam grafica OpenGL propriu-zisa
            deseneaza();

           // afisam buferul secundar in fereastra pe ecran
   SwapBuffers(hDC);
           //  validam suprafata de desenare
ValidateRect(hWnd,NULL);
           }
           break;
          //apelata la apasarea unei taste
                case WM_KEYDOWN:
                      {
                      switch (keydata)
                      {
         case 75://executa ramura la apasarea tastei spre stanga
                                  ScadeUnghiA();
                                  break;
        case 77:// executa ramura la apasarea tastei spre dreapta
                                  CresteUnghiA();
                                  break;
         case 72:// executa ramura la apasarea tastei in sus
                                  ScadeUnghiB();
                                  break;
         case 80:// executa ramura la apasarea tastei in jos
                                  CresteUnghiB();
                                  break;
case 57:// executa ramura la apasarea barei de spatiu
                                  AltCorp();
                                  break;
                           }
                          }
                          break
// apelata la mişcare cursorului mouselui
                case WM_MOUSEMOVE:
//Roteste Corpul pe 2 axe cat timp butonul stang este tinut apasat
                          {
          int mx,my;
                           mx=LOWORD(lParam);
                           my=HIWORD(lParam);
          if (wParam==MK_LBUTTON)
                           {
                                  a=a_+(mmy-my)*2;
                                  b=b_+(mmx-mx)*2;
                                  deseneaza();
```

```
                                }
                                }break;
        //apelata la apăsarea butonului stang al mouse-lui
                        case WM_LBUTTONDOWN://Activeaza rotatia prin mouse
                                        //fixand punctul de referinta
                                {
                int mx,my;
                                 mx=LOWORD(lParam);
                                 my=HIWORD(lParam);
                                 a_=a;
                                 b_=b;
                                 mmx=mx;
                                 mmy=my;
                                }
                                break;
        //apelata la apasarea butonului drept al mouse-ului
                        case WM_RBUTTONDOWN://Schimba corpul prin clic dreapta
mouse
                                {
                                        AltCorp();
                                }
                                break;
        //apelata la apasarea butonului din mijloc al mouselui
                        case WM_MBUTTONDOWN:
                                         //Roteste automat si in mod continuu corpul
                                        //activarea dezactivarea facandu-se prin butonul
                                        //de mijloc al mouse-ului
                                {
                                        roteste_on=!roteste_on;
                                }
                                break;


            default:   // daca nu s-a apelat nici o alta ramura
                // apelam functia de procesare implicita a mesajelor
              return (DefWindowProc(hWnd, message, wParam, lParam));

            }
        return (0L);
        }
```

În locul funcției main() din C, C++ vom avea următoare secvență, care rămâne aproape indentică pentru orice aplicație Win32 vom crea de acum încolo:

```
int APIENTRY WinMain(  HINSTANCE       hInstance,
```

```
                HINSTANCE       hPrevInstance,
                LPSTR           lpCmdLine,
                int             nCmdShow)
        {
        MSG           msg;           // mesaj window
        WNDCLASS      wc;            // clasa fereastră.
        HWND          hWnd;          // pointer - indice la o fereastră
        // initializam parametrii ferestrei:
        wc.style             = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc       = (WNDPROC) WndProc;
        wc.cbClsExtra        = 0;
        wc.cbWndExtra        = 0;
        wc.hInstance         = hInstance;
        wc.hIcon             = NULL;
        wc.hCursor           = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground     = NULL;
        wc.lpszMenuName      = NULL;
        wc.lpszClassName     = windowtitle;

        // Inregistram clasa fereastra
        if(RegisterClass(&wc) == 0)
              return FALSE;

        // Creem fereastra aplicatiei
                   hWnd = CreateWindow(
                windowtitle,
                windowtitle,
                // OpenGL requires WS_CLIPCHILDREN and
                WS_CLIPSIBLINGS |
     WS_OVERLAPPEDWINDOW        |        WS_CLIPCHILDREN        |
WS_CLIPSIBLINGS,
                100, 100,
                width0, height0,
                NULL,
                NULL,
                hInstance,
                 NULL);
        // daca fereastra nu s-a creat functia  WinMain va returna FALSE
        if(hWnd == NULL)
              return FALSE;

        // Afisam fereastra aplicatiei
        ShowWindow(hWnd,SW_SHOW);
        UpdateWindow(hWnd);

        // Procesam mesajele spre aplicatie prin intermediul unei bucle
        while( GetMessage(&msg, NULL, 0, 0))
```

```
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        }
 return msg.wParam;// inchidem ciclul returnand un parametru al mesajului
}
```

Exercițiu. Pe baza unei aplicații șablon disponibile pe calculatorul pe care lucrați. Construiți un program care la ținerea apăsată a butonului stâng al mouselui să deplaseze un dreptunghi care să fie afișat tot timpul cu centrul sub pointerul mouseului.

4.4 Lucrare de laborator 4.

4.4.1 Primitive grafice OpenGL

În momentul de față din laboratoarele precedente stăpânim inițializarea şi crearea unui mediu pentru OpenGL atât folosind biblioteca glaux cât şi Win32. Cu toate aceste nu prea se ştie să se deseneze prea mult în OpenGL cu excepția câtorva elemente de bază. Pentru a desena orice corp în OpenGL indiferent de cât de complex este, este nevoie de utilizarea unui set de primitive. Acestea sunt elemente grafice de bază.

4.4.2 Desenarea de puncte

Pentru a se desena un punct se foloseşte următoarea secvență:

```
glBegin(GL_POINTS);
// va desena un punct la coordonatele (50,50,50)
      glVertex3f(50.0f, 50.0f, 50.0f);
glEnd();
```

Se observă folosirea unei funcții de început de desenare glBegin – la care specificăm ce anume se va desena. O funcție de finalizare a secvenței de desenare glEnd(). Întotdeauna aceste funcții vor lucra în pereche, iar între ele nu vor fi permise decât anumite operații de desenare. Una dintre acestea este specificare unui vârf ce se face cu funcția glVertex3f care are următorul prototip:

```
void glVertex3f(GLfloat x,  GLfloat y,  GLfloat z);
```

Unde x,y,z sunt coordonatele punctului sau vârfului. Această funcție are o multitudine de alte abordări, aceasta fiind cea mai folosită.
Altă funcție este specificare culorii glColorf3f având prototipul:

```
void glColor3f(GLfloat rosu, GLfloat verde,GLfloat albastru);
```

Aceasta precede un vârf, sau vârfuri cărora le specifică culoarea putând tot la fel de bine sta şi în afara perechii glBegin …glEnd.

4.4.3 Desenarea de linii

Pentru a desena o linie folosim secventa:

```
glBegin(GL_LINES);
      glVertex3f(0.0f, 0.0f, 0.0f);//primul capat al liniei
      glVertex3f(50.0f, 50.0f, 50.0f);//al doilea capat al liniei
glEnd();
```

Se observă că şi de această dată s-a specificat în glBegin tipul de primitivă dorită. Făcând această practic îi precizăm secvenței cum va interpreta vârfurile ce se află în interiorul perechii glBegin … glEnd. Oricare două perechi de vârfuri luate în ordine vor specifica o nouă linie.
Exemplu :

```
glBegin(GL_LINES);
      glVertex3f(0.0f, 0.0f, 0.0f);//primul capat al liniei 1
      glVertex3f(50.0f, 50.0f, 50.0f);//al doilea capat al liniei 1
         glVertex3f(0.0f, 20.0f, 0.0f);//primul capat al liniei 2
      glVertex3f(50.0f, 40.0f, 50.0f);//al doilea capat al liniei 2
         glVertex3f(20.0f, 0.0f, 0.0f);//primul capat al liniei 3
      glVertex3f(10.0f, 30.0f, 50.0f);//al doilea capat al liniei 3
glEnd();
```

Dacă este un vârf necuplat acesta va fi ignorat.

Pentru desenarea unor linii la care ultimul capăt este început pentru o nouă linie se foloseşte constanta GL_LINE_LOOP specificată în glBegin(…).
Pentru a desena un mănunchi de linii în care un punct să fie originea unui grup de linii se foloseşte constanta GL_LINE_STRIP.
La GL_LINE_LOOP primul vertex este începutul înlănțuirii de linii şi ultimul sfârşitul.
La GL_LINE_STRIP primul vertex(vârf) este originea mănunchiului de linii şi celelalte vertexuri capete de linie.

Exercitiu 1. Desenați din linii un hexagon centrat pe ecran.
Exercițiu 2. Desenați folosind GL_LINE_STRIP un mănunchi de linii care să semene cu razele de lumină emise de o stea.(De acum încolo vom folosi abordare desenați când ne vom referi la crearea unui program care să deseneze un anume lucru, sau ori de câte ori ne referim la construcția unei scene 3D.)

4.4.4 Desenarea unui triunghi

Pentru a desena un triunghi avem la dispoziție trei constante:

GL_TRIANGLES – desenează triungiuri independente – grupuri de trei vârfuri luate în ordine constituie un triunghi;
GL_TRIANGLE_STRIP – desenează triunghiuri interconectate – ultimele două vârfuri al oricărui triughi sunt vârfuri de generare ale următorului;
GL_TRIANGLE_FAN – desenează triunghiuri interconectate sub forma unui evantai – la care primul vârf al primului triunghi este comun tuturor celorlalte triunghiuri ;

Deşi sunt disponibile o varietate de alte poligoane pe care le poate genera OpenGL, din punct de vedere hardware triunghiurile sunt procesate cel mai rapid. De aceea la creare unui obiect din elemente este indicat ca acestea să fie triunghiuri, sau o altă abordare ar fi descompunerea poligoanelor complexe în triunghiuri.

Exercițiu 3. Desenați o piramidă folosind evantaiul de triunghiuri, fără de desena şi baza. Fiecare față desenata va avea altă culoare.

4.4.5 Desenarea unui patrulater

Pentru a desena un patrulater se folosesc constantele :

GL_QUADS – desenează un patrulater independent;
GL_QUAD_STRIP – desenează nişte patrulatere interconectate, ultimele două vârfuri fiind primele două ale următorului patrulater;

4.4.6 Desenarea unui poligon

Pentru a desena un poligon se foloseşte constanta GL_POLYGON, fiecare vârf va fi câte un colț al poligonului ultimul vârf fiind automat unit cu primul pentru crearea unui contur închis. Aceasta este valabilă şi pentru patrulatere.

4.4.7 Funcții grafice auxiliare

În acest moment au fost expuse toate primitivele grafice de bază şi cu toate acestea mai râmân de precizat elemente tipice pentru primitive : mărimea punctului, grosimea liniei, model de linie, model de poligon, renunțarea la desenarea unei fețe,desenarea umplută sau doar conturată a unei primitive închise. Aceste lucruri şi altele vor fi expuse în continuare.

4.4.8 Modificare mărimii punctului

Spre deosebire de puncte afişate pe un ecran punctele din OpenGL au dimensiuni. Punctele au o forma de pătrat când sunt mărite şi nu de cerc cum s-ar putea crede. Pentru a modifica mărimea unui punct se foloseşte secvența următoare :

```
GLfloat sizes[2];       // pastreaza limitele intre care
                          //variaza marimea punctului
GLfloat step;          // pastrează pasul de incrementare al marimii

// se citesc valorile stocate implicit
glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
```

De obicei limitele mărimii variază între 0.5 şi 10.0, iar pasul de incrementare al mărimii este de 0.125.

Pentru a seta mărimea punctului se foloseşte această apelare :

glPointSize(MarimeaPunctului);

4.4.9 Modificare parametrilor liniei

Pentru a afla limitele între care variază grosimea liniei, precum şi pasul de incrementare al grosimii se foloseşte o secvență asemănătoare ca la punct:

```
GLfloat sizes[2];//  intervalul de valori
GLfloat step;   //pasul
// Get supported line width range and step size
glGetFloatv(GL_LINE_WIDTH_RANGE,sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY,&step);
```

Pentru a seta grosimea liniei vom apela funcția:

glLineWidth(GrosimeLinie);

, înaintea desenării liniilor ce vor avea grosimea precizată.

Pentru a modifica modelul liniei – spre exemplu pentru a trasa o linie segmentată sau compusă din puncte, sau alt model. Folosim secvența :

```
GLushort model = 0x5555;
GLint factor = 4;

glEnable(GL_LINE_STIPPLE);
…
glLineStipple(factor,model);

…
```

La care model semnifică o secvență de 16 biți fiecare bit semnificând absența sau prezență unui segment din linie, segment de lungime specificată de –factor-.

4.4.10 Modificare parametrilor figurilor închise

Pentru a specifica faptul că tranzițiile de culoare de pe suprafața poligonului se fac liniar sau fin se va folosi funcția cu prototipul :

void glShadeModel(GLenum mod);

Unde mod poate fi una dintre constantele:

GL_FLAT – aspect mai necizelat al tranziției de culoare;
GL_SMOOTH – aspect lin, fin al tranziției de culoare;

Pentru a specifica că poligoanele (prin acest termen se face referire la toate figurile închise fie ele triunghiuri, patrulater…) vor fi desenate astfel încât cele din spate să fie acoperite de cele din față se va folosi următoarele secvențe :

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

, pentru ştergerea ecranului. Se observă că se şterge şi bufferul de adâncime în care sunt păstrate pozițiile punctelor desenate în raport cu axa oz.

```
glEnable(GL_DEPTH_TEST);
```

, pentru a activa opțiunea de folosire a buferului de adâncime.

Uneori pentru a creşte viteza de procesare dorim să nu mai fie desenate suprafețele interioare ale corpurilor. În acest sens se va folosi o funcție care să specifice care poligoane vor constitui fața poligonului şi care spatele. Esențial în acest proces este ordinea în care sunt definite vârfurile unui polygon. Ea poate fi în sensul acelor de ceas sau invers.

Secvența care specifică că spatele poligoanelor nu va fi desenat este:

```
glEnable(GL_CULL_FACE);
…
glFrontFace(GL_CW);// fata poligonului este cea în care vârfurile
                   // sunt definite în sensul acelor
…//definim poligoane conform acestei reguli
glFrontFace(GL_CCW); // fata poligonului este cea în care vârfurile
                   // sunt definite contrar sensului acelor
 …//definim poligoane conform acestei reguli
```

Pentru a specifica modul în care este desenat poligonul vom folosi secvențele:

```
glPolygonMode(GL_BACK,GL_LINE);// spatele este desenat numai conturat
…
glPolygonMode(GL_FRONT,GL_LINE);// fața este desenat numai conturat
…
glPolygonMode(GL_FRONT,GL_FILL);// fața este desenata umpluta
```

Primul parametru poate lua valorile :
GL_FRONT – se refera doar la fața poligonului
GL_BACK – se refera la spatele poligonului
GL_FRONT_AND_BACK – se refera la ambele suprafețe ale poligonului

Al doilea parametru poate lua valorile :
GL_POINT - se desenează numai vârfurile;
GL_LINE - se desenează numai conturul;
GL_FILL – se desenează suprafața umplută;

Pentru a specifica că se va folosi un efect de umplere vom folosi secvențele de inițializare :

```
glEnable(GL_POLYGON_STIPPLE);

glPolygonStipple(model);
```

Unde model este o matrice 32x32 de biți. Fiecare bit reprezentând un punct din model. Acest model este repetat pe verticala şi pe orizontala suprafeței poligonale la care se aplică acoperind-o în întregime.

```
GLubyte model[] = { 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0xc0,
                    0x00, 0x00, 0x01, 0xf0,
                    0x00, 0x00, 0x07, 0xf0,
                    0x0f, 0x00, 0x1f, 0xe0,
                    0x1f, 0x80, 0x1f, 0xc0,
                    0x0f, 0xc0, 0x3f, 0x80,
                    0x07, 0xe0, 0x7e, 0x00,
                    0x03, 0xf0, 0xff, 0x80,
                    0x03, 0xf5, 0xff, 0xe0,
                    0x07, 0xfd, 0xff, 0xf8,
                    0x1f, 0xfc, 0xff, 0xe8,
                    0xff, 0xe3, 0xbf, 0x70,
                    0xde, 0x80, 0xb7, 0x00,
                    0x71, 0x10, 0x4a, 0x80,
                    0x03, 0x10, 0x4e, 0x40,
                    0x02, 0x88, 0x8c, 0x20,
                    0x05, 0x05, 0x04, 0x40,
                    0x02, 0x82, 0x14, 0x40,
                    0x02, 0x40, 0x10, 0x80,
                    0x02, 0x64, 0x1a, 0x80,
                    0x00, 0x92, 0x29, 0x00,
```

```
0x00, 0xb0, 0x48, 0x00,
0x00, 0xc8, 0x90, 0x00,
0x00, 0x85, 0x10, 0x00,
0x00, 0x03, 0x00, 0x00,
0x00, 0x00, 0x10, 0x00};
```

Un exemplu de definire de model.

Pentru construcția unui poligon se aplică o serie de reguli. Încălcarea acestor reguli crea probleme pe plăcile video mai vechi dar pentru cele noi aceste restricții nu mai sunt valabile. În orice caz este bine să fie respectate pentru o mai bună portabilitate a codului OpenGL. Astfel :
Laturile poligonului nu trebuie să se intersecteze.
Poligonul trebuie să fie convex.
Vârfurile poligonului trebuie să se afle în acelaşi plan.

Exercițiu 4. Creați un poligon hexagonal care să fie umplut cu un model, care să reprezinte o față zâmbitoare (smiley).

4.5 Lucrare de laborator 5.

4.5.1 Transformări geometrice în OpenGL

În OpenGL transformările geometrice pe care le suportă un set de puncte, respectiv un corp 3D format din acestea, sunt mediate de o matrice de transformări cu patru coloane şi patru linii. Înmulțind vectorul de patru elemente ce specifică punctul cu o astfel de matrice se obține un nou vector care reprezintă punctul ce a suferit transformările.

Am spus patru elemente şi nu trei pentru că primele trei sunt coordonatele x, y, z ale punctului şi a patra valoare este un coeficient de scalare w. Prin înmulțirea unei matrice corespunzătoare de transformări cu un punct se pot face rotații, translații, scalări. Acestea sunt transformări clasice realizate prin funcțiile OpenGL, dar lucrând direct cu matricea se pot realiza transformări personalizate – ca de exemplu crearea de umbre. Nu există un acces direct la matrice dar în schimb există unul mediat, matricea poate fi încărcată, descărcată, încărcată într-o stivă, descărcată dintr-o stivă, inițializată cu matricea unitate, şi în plus o mediere de gradul doi în care modificam matricea prin înmulțiri cu matrice ce specifică translații, rotații, etc. Orice transformare de coordonate în OpenGL se realizează prin înmulțirea matricei de lucru cu o matrice ce specifică o transformare.

4.5.2 Încărcarea şi descărcarea din stivă a unei matrice

Matricea de lucru poate fi salvată într-o stivă, ca mai apoi când se doreşte revenirea la starea inițială să fie descărcată din stivă în matricea de lucru curentă. Aceasta este utilă pentru a păstra o stare a sistemului la un moment pentru ca mai apoi să se revină la ea.

De exemplu am făcut nişte transformări şi acestea vor fi comune unui grup de obiecte ce urmează, dar pentru aceste obiecte se mai doreşte încă o transformare pentru orientare de exemplu, înainte de a face rotația vom salva starea matricei ca după afişarea obiectului să se revină la starea inițială.

Pentru a salva matricea de transformare prin încărcarea ei în stivă se foloseşte funcția:

```
glPushMatrix( );
```

, care nu are parametri.

Iar pentru a descărca matricea de transformare salvată din stivă se foloseşte o funcție asemănătoare:

```
glPopMatrix( );
```

De obicei acestea vor lucra împreună. O apelare glPushMatrix fiind urmată mai jos în program de glPopMatrix.

Pentru a încărca matricea unitate se foloseşte funcţia:

glLoadIdentity( );

,care anulează orice transformare anterioară.

Mai există o funcţie care specifică la care tip de matrice se referă:

void glMatrixMode(GLenum mode);

unde mode poate fi definit de una din constantele:

GL_MODELVIEW – se referă la matricea ce afectează modul de vizualizare a obiectului ;
GL_PROJECTION – se referă la matricea de proiecţie;
GL_TEXTURE - se referă la matricea prin care se realizează texturarea :

Exemplu :

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();

glTranslatef(0.0f, 0.0f, -300.0f);
glColor3f(1, 1, 0);
auxSolidSphere(30.0f);

glRotatef(a, 0.0f, 1.0f, 0.0f);

glColor3f(0,0,1.0);
glTranslatef(105.0f,0.0f,0.0f);
auxSolidSphere(15.0f);

glColor3f(0.7,0.6,0.8);
glRotatef(b,0.0f, 1.0f, 0.0f);
glTranslatef(30.0f, 0.0f, 0.0f);
auxSolidSphere(6.0f);

glPopMatrix();//

glFlush();
```

4.5.3 Transformări elementare de coordonate

OpenGL pune la dispoziție trei tipuri de transformări prin intermediul următoarelor funcții:

glRotatef – pentru rotații;
glTranslatef – pentru translații;
glScalef – pentru scalări;

glRotatef are următorul prototip:

void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

,unde angle reprezintă valoarea unghiului de rotație exprimat în grade, unghi ce semnifică o rotație invers acelor de ceas ;
x, y, z reprezintă unul din punctele ce definesc axa de rotație, celălalt fiind poziția curentă a centrului sistemului de referință ;

glTranslatef are următorul prototip :

void glTranslatef(GLfloat x, GLfloat y, GLfloat z);

,unde x, y, z sunt valori ale tranlației pe cele trei axe de coordonate.

glScalef are următorul prototip :

void glScalef(GLfloat x, GLfloat y, GLfloat z);

,unde x, y, z sunt valori ale factorilor de contracție respectiv dilatare a corpului pe cele trei axe de coordonate .

Trebuie reținut faptul că orice transformare este aditivă şi se perpetuează dacă nu se folosesc funcțiile glPushMatrix, glPopMatrix sau glLoadIdentity.

Pentru funcțiile de rotații pentru a nu apărea confuzii este bine să se facă rotații numai câte pe o axa odată astfel :

Pentru a roti pe axa Ox cu un unghi de 45 de grade vom avea secvența :

glRotatef (45.0f, 1.0f, 0.0f, 0.0f) ;

,pentru oy avem ceva asemănător :

glRotatef(45.0f, 0.0f, 1.0f, 0.0f).

Exercițiu 1. Creați un program care folosind glaux, să afişeze un cub (auxWireCube) şi apoi apăsând tastele săgeți pentru stânga – dreapta să se facă o rotație pe ox, sus-jos pe oy, si a – z pe oz.

Exercițiu 2. Creați un program pentru simularea sistemului Soare, Lună, Pământ. Acest sistem se va afla în rotație, Luna se va învârti de două ori mai repede în jurul Pământului decât Pământul în jurul Soarelui. Soarele va fi o sferă galbenă, Luna o sferă cenuşie iar Pământul o sferă albastră.

Exercițiu 3. Creați un atom cu electroni – nucleul va fi compus din 2 sfere albastre şi două sfere roşii. Vor fi 4 electroni ce se vor roti fiecare pe o orbită proprie, fiecare cu raze diferite. Electronii vor fi sfere galbene.

Exercițiu 4. Creați o compoziție formată dintr-un cub, iar în fiecare colț al său va fi desenată o sferă.

4.5.4 Configurarea mediului vizual

Dacă veți crea programe în Win32, de OpenGL, veți avea nevoie să cunoaşteți o serie de funcții de configurare a mediului vizual. Mai întâi se vor expune o serie de secvențe de cod în care sunt implicate astfel de funcții:

```
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

gluPerspective(60.0f,(GLfloat)w/(GLfloat)h, 1.0, 400.0);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

În această secvență avem următoarele funcții:

glViewport stabileşte parametrii ferestrei software de desenare(Viewport) specificați prin poziția colțului stânga - sus , şi lungimea w, şi înălțimea h a ferestrei.

glMatrixMode(GL_PROJECTION) stabileşte că se va lucra cu matricea de proiecție;

glLoadIdentity() stabileşte că matricea de proiecție va fi resetată la starea inițială;

gluPerspective face modificări asupra matricei de proiecție – stabilind unghiul percepției – 60.0 de grade, aspectul proiecției adică raportul dintre înălțimea şi lungimea imaginii, cel mai apropiat şi cel mai îndepărtat punct pe oz pe baza cărora se stabileşte planurile de decupare a scenei, astfel că ce este înafara acestor limite nu este afişat ;

glMatrixMode(GL_MODELVIEW) – se stabileşte că se va lucra cu matricea modelelor 3D ;

glLoadIdentity() – se setează mediul 3D la starea inițială;

De obicei o astfel de secvență este apelată la fiecare redimensionare a ferestrei, sau înainte de prima afişare a ferestrei aplicației.

O altă funcție de configurare ar fi următoarea:

void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

, unde eyex, eyey, eyez – este poziția observatorului sau mai bine spus a ochiului acestuia;
, centerx, centerzy, center z este centrul scenei spre care priveşte.
, upx, upy, upz – specfică direcția privirii observatorului.

Exemplu:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(locX, locY, locZ, dirX, dirY, dirZ, 0.0f, 1.0f, 0.0f);
```

În acest exemplu matricea de vizualizare este resetată, şi apoi setată în raport cu ochiul observatorului.

Folosind această funcție putem crea un program prin care să explorăm o anumită scenă.

Exercițiu 5. Modificând exemplul 4, scrieți o secvență de program în care la apăsarea tastelor sus – jos ochiul observatorului să se deplaseze pe oz, iar stânga – dreapta pe ox. Veți folosi desigur funcția gluLookAt prezentată mai sus. Mai puteți modifica programul astfel încât la apăsarea tastelor a – z să se modifice upx, s – x upy, d - c upz, modificând astfel şi orientarea privirii observatorului.

4.6 Lucrare de laborator 6.

4.6.1 Iluminare, lumini şi materiale.

Până acum s-a lucrat în medii în care nu s-a pus problema iluminării scenelor, obiectele solide, triunghiurile erau toate uniform colorate, nu conta distanța, orientarea acestora. Corpurile care aveau suprafață opacă, sferele spre exemplu erau simple pete de culoare mai mult discuri, decât corpuri cu volum. Scopul acestui laborator este chiar rezolvarea acestei probleme. Se vor prezenta atât moduri de iluminare, poziționări de spoturi, efecte de iluminare cât şi un aspect complementar materialul din care este făcut corpul. La material vor conta proprietățile reflexive ale acestuia adică modul cum acesta răspunde la lumină.

4.6.2 Configurarea iluminării

Iluminarea naturală are trei componente principale :

componenta ambientală – în care lumina nu pare a avea o sursă anume ea emanând din toate punctele din spațiu şi din nici unul în mod particular, obiectele sunt luminate uniform;

componenta difuză – la care lumina are o sursă anume, dar nu apar suprafețe lucioase;

componenta de „luciu" – la care lumina este direcțională şi mai apare şi un luciu al suprafețelor;

Pentru a configura aceste trei componente se foloseşte funcția cu următorul prototip:

```
void glLightfv( GLenum sursa,  GLenum mod,  const GLfloat *parametri);
```

,in care sursa este o constanta de forma GLLightx. Unde x poate lua valori de la 0 la o valoare maximă precizată de constanta GL_MAX_LIGHTS.

, mod este tipul componentei de iluminare aceasta poate fi :

GL_AMBIENT - pentru specificare componentei ambientale;
GL_DIFFUSE - pentru specificare componentei difuze;
GL_SPECULAR – pentru specificare componentei de luciu;
GL_POSITION – pentru specificare poziției sursei de lumină.

ar mai şi alte moduri dar pentru moment acestea sunt suficiente,

, parametri este un vector de patru componente care pentru modurile de iluminare reprezintă elementele RGBA – adică intensitatea culorii roşii, a celei verzi şi a celei albastre plus intensitatea componentei alfa toata date prin valori de la 0.0 la 1.0.

Pentru poziţii parametri specifică exact coordonatele x, y şi z ale sursei plus o componentă de scalare care trebuie setată întotdeauna la 1.0.

Exemplu :

```
        GLfloat ambient[] = { 0.8f, 0.8f, 0.8f, 1.0f };
 GLfloat diffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f };
        GLfloat specular[] = { 0.5f, 0.5f, 0.5f, 1.0f };
        GLfloat position[] = { 0.0f, 200.0f, 200.0f, 0.0f };

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, diffuse);
        glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Se observă modul cum sunt declaraţi vectorii. Se mai observă folosirea unor funcţii de activare a iluminării – pentru iluminarea generala a scenei glEnable(GL_LIGHTING) - iar pentru activarea exclusivă a unei surse glEnable(GL_LIGHT0).

În cazul iluminării ambientale nu se pune problema orientării suprafeţei faţă de sursa de lumină. Pentru iluminării direcţionale în schimb orientarea suprafeţei devine esenţială. În matematică pentru a defini orientare unei suprafeţe se specifică vectorul normal pe acea suprafaţă, practic o perpendiculară care iese din suprafaţă. Când se construieşte o suprafaţă înainte de specifica vârfurile pentru aceea suprafaţă (glVertex…)

se va specifica în interiorul secvenţei glBegin …. glEnd normala la acea suprafaţă folosind funcţia cu prototipul :

```
void glNormal3f(GLfloat nx,  GLfloat ny,  GLfloat nz);
```

, unde nx, ny, nz sunt coordonatele vârfului normalei.

Deşi OpenGL oferă o modalitate de precizare a normalei unei suprafeţe nu oferă şi o funcţie care dând o serie de vârfuri să calculeze normala pentru suprafaţa definite de ele. Din aceasta cauza va trebui definită o funcţie de calcul a normalei. Un exemplu de astfel de funcţie este :

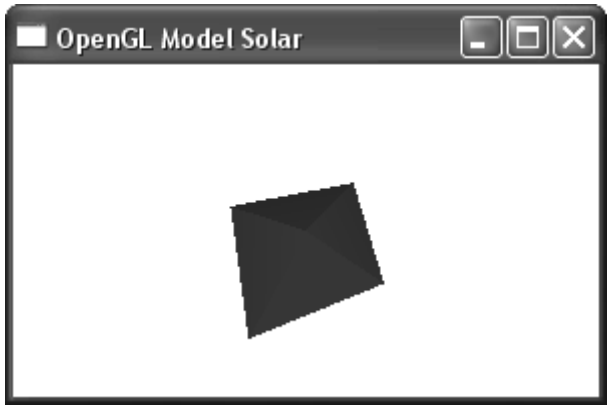
```
void ReduceToUnit(float vector[3])
	{
	float lungime;

	// se calculează lungimea vectorului
	lungime = (float)sqrt((vector[0]*vector[0]) +
			(vector[1]*vector[1]) +
			(vector[2]*vector[2]));

	// evitarea impartirii la zero
	if(lungime == 0.0f)
		lungime = 1.0f;

	// se impart toate componentele vectorului la lungime
	// normalizandu-le
	vector[0] /= lungime;
	vector[1] /= lungime;
	vector[2] /= lungime;
	}
void calcNormal(float vv0[3],float vv1[3],float vv2[3], float out[3])
	{
	float v1[3],v2[3];
	static const int x = 0;
	static const int y = 1;
	static const int z = 2;

	// determină  doi vectori pe baza a trei puncte
	v1[x] = vv0[x] - vv1[x];
	v1[y] = vv0[y] - vv1[y];
	v1[z] = vv0[z] - vv1[z];

	v2[x] = vv1[x] - vv2[x];
	v2[y] = vv1[y] - vv2[y];
	v2[z] = vv1[z] - vv2[z];

	// se face produsul vectorial al celor doi vectori
	// obținându-se varful normalei
	out[x] = v1[y]*v2[z] - v1[z]*v2[y];
	out[y] = v1[z]*v2[x] - v1[x]*v2[z];
	out[z] = v1[x]*v2[y] - v1[y]*v2[x];
	//se normalizeaza vectorul
	ReduceToUnit(out);
	}
```

Funcția care calculează normala este calcNormal, se mai foloseşte şi o funcție de normalizare a vectorilor – ReduceToUnit. vv0, vv1, vv3 constituie vârfurile, iar out vârful vectorului normal.

Exemplu de folosire:

```
glFrontFace(GL_CW);
glColor3f(1,0,0);
calcNormal(triunghi[2],triunghi[1],triunghi[0],mynormal);
glBegin(GL_TRIANGLES);
glNormal3fv(mynormal);
glVertex3fv(triunghi[0]);
glVertex3fv(triunghi[1]);
glVertex3fv(triunghi[2]);
calcNormal(triunghi[3],triunghi[2],triunghi[0],mynormal);
glNormal3fv(mynormal);
glVertex3fv(triunghi[0]);
glVertex3fv(triunghi[2]);
glVertex3fv(triunghi[3]);
calcNormal(triunghi[4],triunghi[3],triunghi[0],mynormal);
glNormal3fv(mynormal);
glVertex3fv(triunghi[0]);
glVertex3fv(triunghi[3]);
glVertex3fv(triunghi[4]);
calcNormal(triunghi[5],triunghi[4],triunghi[0],mynormal);
glNormal3fv(mynormal);
glVertex3fv(triunghi[0]);
glVertex3fv(triunghi[4]);
glVertex3fv(triunghi[5]);
glEnd();
glColor3f(1,0,0);
glFrontFace(GL_CCW);
calcNormal(triunghi[2],triunghi[3],triunghi[4],mynormal);
glBegin(GL_QUADS);
glNormal3fv(mynormal);
glVertex3fv(triunghi[1]);
glVertex3fv(triunghi[2]);
glVertex3fv(triunghi[3]);
glVertex3fv(triunghi[4]);
glEnd();
```

Secvența de mai sus desenează o piramida rosie, cu tot cu bază.



Se observă folosirea funcției glVertex3fv aceasta este o variantă a funcției glVertex3f, numai că în acest caz parametrii x, y, z sunt preluați dintr-un vector cu trei elemente. Acest mod de folosire pentru, acelaşi tip de acțiune, de variante de funcții este o caracteristică OpenGL. Un exemplu ar fi :

glVertex2d, glVertex2f, glVertex2i, glVertex2s, glVertex3d,
glVertex3f, glVertex3i, glVertex3s, glVertex4d, glVertex4f,
glVertex4i, glVertex4s, glVertex2dv, glVertex2fv, glVertex2iv,
glVertex2sv, glVertex3dv, glVertex3fv, glVertex3iv, glVertex3sv,
glVertex4dv, glVertex4fv, glVertex4iv, glVertex4sv

Aceasta pentru a vă face o idee de flexibilitate OpenGL, exemplele putând continua.

Crearea automată de normale

Pentru corpuri 3D predefinite în OpenGL există posibilitatea generării automate a normalelor, astfel:

```
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
```

,tot ce trebuie făcut este să se insereze în program înainte de desenarea corpurilor respective a acestor două linii de cod.

Pentru ce a fost nevoie de normale? Pentru ca reflecțiile venite de la corp şi tonurile de culoare să pară naturale, altfel acestea ar fi fost arbitrare fără vreo legătură cu iluminarea în sine.

4.6.3 Configurarea materialului

Odată ce s-au stabilit luminile următorul pas este configurarea materialului. Materialul precizează modul în care lumina va fi reflectată, difuzată sau emanată de către obiect. În acelaşi timp specificăm şi culoare suprafeței prin precizarea materialului. Pentru a specifica materialul se poate folosi următoarea secvență:

```
glMaterialfv (GL_FRONT, GL_AMBIENT, mat_amb);
glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_dif);
glMaterialfv (GL_FRONT, GL_SPECULAR, mat_spec);
 glMaterialf  (GL_FRONT, GL_SHININESS, 0.7f*128.0f);
```

Observăm folosirea funcției glMaterialfv aceasta are prototipul:

```
void glMaterialfv(GLenum fata,  GLenum proprietate,
  const GLfloat *parametri );
```

, fata poate lua următoarele valori:
GL_FRONT – se referă la față ;
GL_BACK – se referă la spate;
GL_FRONT_AND_BACK – se referă la față şi spate;
, proprietate stabileşte care aspect al materialului va fi configurat:
GL_AMBIENT – reflexia luminii ambientale (4 valori);
GL_DIFFUSE – reflexia luminii difuze(4 valori);
GL_SPECULAR – reflexia luciului suprafeței(4 valori);
GL_EMISSION – emisia de lumină de către suprafață(4 valori);
GL_SHININESS – gradul de reflexie lucioasă a suprafeței (cu cât este mai mare cu atât suprafața lucioasă este mai extinsă) (o valoare);
GL_AMBIENT_AND_DIFFUSE - reflexia luminii ambientale şi difuze(4 valori);
GL_COLOR_INDEXES – se referă la indecşii de culoare – se pot configura în acelaşi toate cele trei proprietăți reflexive : ambientală, difuză şi lucioasă. (3valori)
, parametrii vor fi patru valori ale componentelor culorii RGBA.

glMaterialf este folosit în special pentru a specifica gradul de lucire al suprafeței. Intervalul de valori pentru acest parametru este între 1 şi 128. 128 este cel mai înalt grad de lucire.

Exercițiu1. Realizați un program cu Soarele, Luna şi Pământul, În care soarele să fie o sursă de lumină şi în acelaşi timp o sferă galbenă. Luna va fi o sferă cenuşie şi Pamântul o sferă albastră. Acestea se vor roti.

Exercițiu 2. Construiți o moleculă de metan (CH_4). Atomul de carbon va fi o sferă cenuşie. Iar atomii de hidrogen nişte sfere albastru deschis de 3 ori mai mici ca atomul de carbon, plasate pe suprafața acestuia astfel încât cam 40% din ele să fie înglobate în atomul de carbon. Atomii de hidrogen vor fi dispuşi aproximativ în vârfurile unui tetraedru ce ar avea centrul de greutate în centrul atomului de carbon. Molecula se va putea roti cu tastele săgeți după cele două axe.

4.7 Lucrare de laborator 7.

4.7.1 Generarea de umbre prin utilizarea matricelor.

Până acum am realizat efecte de tonuri de culoare dar efectiv nu am generat nici o umbră. Adică având un plan, un obiect şi o sursă de lumină pe acest plan să apară clar definită umbra obiectului. OpenGL nu prevede o funcție sau o configurație în care să genereze astfel de umbre. Totuşi printr-o manevrare dibace a matricelor de transformare şi a modurilor de afişare se poate genera o umbră dar nu pentru un întreg ansamblu ci doar pentru fiecare obiect luat în parte. Se foloseşte următorul principiu scena se desenează odată normal, şi a doua oară se desenează corpul a cărui umbră se doreşte , de data aceasta înainte de desena corpul îi schimbăm culoare se fie un gri – cenuşiu cum sunt în general umbrele şi îi producem o transformare geometrică. Astfel corpul va fi turtit până va fi extrem de subțire, turtirea se va face ținând seama de poziția sursei de lumină şi poziția şi orientarea planului pe care se proiectează umbra. Acest corp turtit va fi poziționat deasupra planului dar foarte aproape de el exact în poziția unde normal ar trebui să cadă umbra.

Pentru a genera matricea de transformare vom folosi următoare funcție:

```
void MakeShadowMatrix(GLfloat points[3][3], GLfloat lightPos[4],
      GLfloat destMat[4][4])
      {

     GLfloat planeCoeff[4];
      GLfloat dot;

      // se calculează coeficienții ecuației planului
// pe baza a trei puncte distincte din plan
      calcNormal(points,planeCoeff);

      //se determina al patrulea coeficient
            planeCoeff[3] = - (
            (planeCoeff[0]*points[2][0]) +
            (planeCoeff[1]*points[2][1]) +
            (planeCoeff[2]*points[2][2]));

      // facem produsul dintre coordonatele sursei si coeficientii planului
      dot = planeCoeff[0] * lightPos[0] +
                  planeCoeff[1] * lightPos[1] +
```

```
                    planeCoeff[2] * lightPos[2] +
                    planeCoeff[3] * lightPos[3];

    // facem proiectiile pe plan
    // calculand prima coloana a matricei de umbrire
        destMat[0][0] = dot - lightPos[0] * planeCoeff[0];
        destMat[1][0] = 0.0f - lightPos[0] * planeCoeff[1];
        destMat[2][0] = 0.0f - lightPos[0] * planeCoeff[2];
        destMat[3][0] = 0.0f - lightPos[0] * planeCoeff[3];

        // a doua coloana
        destMat[0][1] = 0.0f - lightPos[1] * planeCoeff[0];
        destMat[1][1] = dot - lightPos[1] * planeCoeff[1];
        destMat[2][1] = 0.0f - lightPos[1] * planeCoeff[2];
        destMat[3][1] = 0.0f - lightPos[1] * planeCoeff[3];

        // a treia coloana
        destMat[0][2] = 0.0f - lightPos[2] * planeCoeff[0];
        destMat[1][2] = 0.0f - lightPos[2] * planeCoeff[1];
        destMat[2][2] = dot - lightPos[2] * planeCoeff[2];
        destMat[3][2] = 0.0f - lightPos[2] * planeCoeff[3];

        // a patra coloana
        destMat[0][3] = 0.0f - lightPos[3] * planeCoeff[0];
        destMat[1][3] = 0.0f - lightPos[3] * planeCoeff[1];
        destMat[2][3] = 0.0f - lightPos[3] * planeCoeff[2];
        destMat[3][3] = dot - lightPos[3] * planeCoeff[3];

        }
```

Unde points sunt trei puncte ce definesc planul;

- lightPos – sunt coordonatele sursei de lumina;
- destMat – este matricea de umbrire;

Orice obiect asupra căruia se aplică matricea de umbrire este turtit şi poziționat în planul de proiecție al umbrei.

Exemplu de utilizarea a matricei de umbrire:

```
void CALLBACK deseneaza(void)
{
    GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
    GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
    GLfloat  specularLight[]={1.0f,1.0f,1.0f,1.0f};
    GLfloat  lightPos[] = { 0, 50, -100, 1 };
    GLfloat  specref[]={1.0f,1.0f,1.0f,1.0f};
```

```
GLfloat mynormal[3];
        GLfloat plan[4][3]={    //parametrii planului pe care proiectata
umbra


                                {-500,-50,500},
                                {500,-50,500},
                                {500,-50,-500},
                                {-500,-50,-500}
                                                };
    GLfloat plan_[4][3]={   //parametrii efectivi ai planului unde este
  desenata
                        //umbra
                                {-500,-49.9,500},
                                {500,-49.9,500},
                                {500,-49.9,-500},
                                {-500,-49.9,-500}};

                GLfloat shadowMat[4][4];
                GLfloat pi=3.14159f;
                GLint i,j,lx=100,ly=100;
  // sursa de lumina se poate roti deasupra în jurul obiectului

                lightPos[0]=sin(arot[1]*pi*2/360)*40;
                                        lightPos[1]=50;
                lightPos[2]=cos(arot[1]*pi*2/360)*40-150;

     // generam matricea de umbrire
                MakeShadowMatrix(plan_,lightPos,shadowMat);

        glEnable(GL_DEPTH_TEST);   // inlaturam fetele invizibile
        glFrontFace(GL_CCW);        // fata poligonului va fi cea definita invers
acelor de ceas

        // activam iluminarea
        glEnable(GL_LIGHTING);

        // configuram lumina GL_LIGHT0
        glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
         glLightfv(GL_LIGHT0,GL_SPECULAR,specularLight);
        glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

       // activam lumina GL_LIGHT0
        glEnable(GL_LIGHT0);
```

```
            // activam colorarea indirecta a materialului
            glEnable(GL_COLOR_MATERIAL);

            // specificam ce parametri vor fi preluati indirect de material prin glColorf
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);

           //stergem ecranul
                    glClear(GL_COLOR_BUFFER_BIT                                    |
GL_DEPTH_BUFFER_BIT);
        //specificam materialul
                    glMaterialfv(GL_FRONT,GL_SPECULAR,specref);
                    glMateriali(GL_FRONT,GL_SHININESS,128);

        // matricea de lucru este cea de vizualizare a modelului
            glMatrixMode(GL_MODELVIEW);

            glPushAttrib(GL_LIGHTING_BIT);//Salvam luminile in stiva
                    glDisable(GL_LIGHTING);//Dezactivam luminile

                    glPushMatrix();

                            glColor3f(1,1,0);
                            glTranslatef(lightPos[0],lightPos[1],lightPos[2]);
                            auxSolidSphere(4.0);//desenam sursa de lumina

                    glPopMatrix();

                   glPushMatrix();

                    glMultMatrixf((GLfloat *)shadowMat);//incarcam matricea de
umbra                  // umbra va fi de un verde închis
                            glColor3f(0,0.3,0);
                            // facem aceleasi transformari ca si cum am lucra cu
obiectul
                    glTranslatef(0,0,-150);
                            glRotatef(a,0,1,0);
                            glRotatef(b,1,0,0);
                            auxSolidCube(20.0);//desenam umbra corpului

               glPopMatrix();

            glPopAttrib();//Incarcam luminile din stiva

            glPushMatrix();
                    //corpul este rosu
                     glColor3f(1,0,0);
                   // dupa cum vedeti aceleasi transformari
```

```
            glTranslatef(0,0,-150);
            glRotatef(a,0,1,0);
            glRotatef(b,1,0,0);
            auxSolidCube(20.0);// desenam corpul
        glPopMatrix();

            glPushMatrix(); // desenam planul pe care se proiecteaza umbra
                glColor3f(0,1.0,0);
                glBegin(GL_QUADS);
                    glVertex3fv(plan[0]);
                    glVertex3fv(plan[1]);
                    glVertex3fv(plan[2]);
                    glVertex3fv(plan[3]);
                glEnd();
            glPopMatrix();


    glFlush();
}
```

Exercițiu 2. Reutilizând secvențele de cod precedente – creați un ceainic căruia să-i afișați umbra, lumina se va roti deasupra corpului la apăsare tastei bara de spațiu. Corpul
se va putea roti prin apăsarea tastelor săgeți.

Observați această linie de cod :

```
glMultMatrixf((GLfloat *)shadowMat);//incarcam matricea de umbra
```

Prin aceasta înmulțim matricea de lucru cu matricea de umbrire, rezultatul fiind depus în matricea de lucru.

O altă linie de cod interesantă este următoarea:

```
glPushAttrib(GL_LIGHTING_BIT);
```

Această linie specifică că se va salva setările pentru lumini în stivă pentru a fi apoi recuperate. Se pot salva cu aceeaşi funcție şi parametru corespunzător şi alți parametri de stare ai OpenGL.

Exemplu:

- GL_ACCUM_BUFFER_BIT – culoarea de ştergere a bufferului;
- GL_COLOR_BUFFER_BIT –biți de țin de setările de culoare ale OpenGL
- GL_CURRENT_BIT – biți de stare curentă a OpenGL
- GL_DEPTH_BUFFER_BIT - biți ce țin de configurarea bufferului de adâncime

- GL_ENABLE_BIT - biți ce țin de activarea sau dezactivare unei opțiuni a mediului OpenGL;
- GL_EVAL_BIT - biți de evaluatori OpenGL şi activarea normalelor generate automat;
- GL_FOG_BIT - biți ce țin de parametrii de ceață;
- GL_HINT_BIT – biți ce țin calitatea trasării unei linii,a unui punct, a unui polygon, a cețiii;
- GL_LIGHTING_BIT - biți de țin de parametrii de iluminare;
- GL_LINE_BIT - biți de țin de configurarea liniei;
- GL_LIST_BIT - biți refiritori la listele de afişare ;
- GL_PIXEL_MODE_BIT – biți referitori la modul de desenare al unui punct din imagine;
- GL_POINT_BIT - biți referitori la desenare unui punct individual;
- GL_POLYGON_BIT – biți referitori la desenarea poligoanelor;
- GL_POLYGON_STIPPLE_BIT - biți referitori la modelul de umplere a poligoanelor;
- GL_SCISSOR_BIT - biți referitor la decupari;
- GL_STENCIL_BUFFER_BIT - biți referitori la bufferul şablon;
- GL_TEXTURE_BIT - biți referitori la textură;
- GL_TRANSFORM_BIT - biți referitori la limitele de decupare a scenei;
- GL_VIEWPORT_BIT - biți referitori la limitele pe Oz ale volumului de decupare.

Exercițiu 3. Creați un program cu două obiecte un con(auxSolidCone) şi un tor (auxSolidTorus) la care să le generați umbrele. La apăsare tastei enter va fi selectat conul, la o nouă apăsare torul şi tot aşa. Obiectul selectat va putea fi rotit prin tastele săgeți.

Exercițiu 4. Creați un program cu două surse de iluminare şi un obiect – un tetraedru.

Generați ambele umbre produse de cele două surse de lumina, pentru acelaşi obiect.

Imagine a ferestrei programului exemplu :

1

4.8 Lucrare de laborator 8.

4.8.1 Utilizarea listelor

În momentul de față cunoaşteți îndeajuns de multe elemente de OpenGL pentru a construi element cu element orice structură vă imaginați şi orice scenă. Cu ceea ce ştiți până acum nu puteți reutiliza codul OpenGL decât prin simpla lui copiere, însoțită de transformări geometrice, în zona de program în care aveți nevoie. Anticipând această nevoie OpenGL a introdus listele de desenare prin care codul de program scris exclusiv în OpenGL este definit odată în cadrul unei liste, şi apoi prin apeluri la acea listă codul poate fi refolosit. Listele oferă nu numai o metodă mai comodă de gestionare a codului dar la apelările listei desenarea se face mai rapid. Aceasta deoarece prin listă se păstrează o structură 3D specificată de acel cod, structură gata de desenare, în timp ce în secvența propriu-zisă de cod trebuie efectuate de fiecare dată calcule. Astfel orice operație ce lucrează cu mediul OpenGL, sau indirect modifică un parametru sau o structură OpenGL este luată în calcul. Însă dacă secvența respectivă este definită prin anumiți parametri independenți de mediul OpenGL, modificarea acestor parametri şi apelarea mai apoi a listei nu modifică de loc aspectul structurii desenate, doar asupra secvenței de cod originale, şi nu asupra listei au efect aceşti parametri. Totuşi este permisă orice transformare geometrică definită prin matricele OpenGL. Ce încărcăm în listele OpenGL? De obicei structuri geometrice, mai puțin materiale, lumini şi culori, pentru că uneori pe aceste dorim să le schimbăm la reapelări. Pe viitor vom lucru cu liste şi la păstrarea unor texturi încărcate.

4.8.2 Crearea unei liste.

Pentru a crea o lista folosim funcțiile glNewList…glEndList care au prototipurile:

```
void glNewList( GLuint numarlista,  GLenum mod);
void glEndList(  void);
```

, unde numarlista este un număr prin care vom identifica lista creată;
mod este modul cum va fi procesată liste putând lua valorile:
GL_COMPILE – doar compilează lista fără a o executa, adică nu va afişa nimic pe ecran din ceea ce este prevăzut în listă;
GL_COMPILE_AND_EXECUTE – compilează şi execută lista pe loc;

Exemplu de definire a unei liste:

```
glNewList (1, GL_COMPILE_AND_EXECUTE);
…
// instructiuni OpenGL
…
glEndList( ); // delimiteaza finalul zonei de definire a listei
```

Atunci când nu dorim să gestionăm manual numerele de listă, sau dorim să evităm redefinirea unei liste din neatenție sau uitare, vom folosi funcțiile ce au următoarele prototipuri:

```
GLuint glGenLists(GLsizei interval);
GLboolean glIsList(GLuint numarlista);
```

, glGenLists generează primul număr de listă, a unei serii continue de numere de listă, mărimea acestei serii fiind definite de interval. Acestea sunt doar numere disponibile, alocarea făcându-se pentru fiecare în parte mai pe urmă.

,glIsList – returnează TRUE dacă numărul de listă specificat a fost folosit la crearea unei liste, altfel FALSE.

Pentru a chema în vederea execuției o listă alocată şi compilată folosim funcția cu prototipul :

```
void glCallList(GLuint numarlista);
```

De reținut că în cazul apelării unei liste parametrii modificați de secvența compilată în listă rămân modificați şi după revenirea din apel. De aceea dacă doriți evitarea acestui lucru este recomandabil folosire funcțiilor de salvare prin stivă – glPushMatrix, glPopMatrix, glPushAttrib, glPopAttrib.

Când se doreşte apelarea mai multor liste la acelaşi apel se foloseşte funcția cu prototipul:

```
void glCallLists(GLsizei n,  GLenum type,  const GLvoid *lists);
```

,unde n este numărul de liste;

type este tipul valorilor din listă, aceste valori fiind convertite mai apoi la GLint ;

lists este un vector de tipul specificat în type ce conține numerele de lista.

Pentru type se pot folosi următoarele constante specificatoare de tip, acestea fiind selectate pe criteriul folosirii efective :

GL_INT – pentru tipul GLint ;

GL_UNSIGNED_INT – pentru GLuint ;

GL_FLOAT – pentru GLfloat ; ş.a.

Odată ce nu se mai foloseşte o listă este recomandabil ca aceasta să fie dealocată eliberând memoria asociată mediului OpenGL, memorie care cel mai adesea este mapată chiar în memoria plăcii grafice.

Pentru aceasta se va folosi funcția cu protipul :

void glDeleteLists(GLuint list, GLsizei range);

Aceasta va şterge un număr specificat de range de liste alocate în mod continuu, începând cu lista specificată de parametrul list.

Exemplu :

glDeleteLists(10,4);

Va şterge listele 10,11,12,13.

O altă funcție importantă este definirea bazei listei adică numerele de listă vor începe de la valoare specificată, la apelarea listelor numerele de listă folosite vor fi cele stabilite minus valoare bazei listelor.

Exemplu:

```
int lista[10];//un vector de numere de lista
   int i;

   for(i  = 0; i < 10;  i++) //alocam un numar continuu de liste
         lista[i] = i;

   // Construim 30 de liste alocand numere de listă consecutive
    // Definim prima lista
   glNewList(1,GL_COMPILE);
         …
         …
   glEndList();

   // ... a doua lista
   glNewList(2,GL_COMPILE);
         …
         …
   glEndList();

   // si asa mai departe
   // ... ultima lista
   glNewList(29,GL_COMPILE);
         …
```

```
        …
glEndList();


// executam primele 10 liste – de la 0 la 9
glCallLists(10,GL_INT,lista);

// setam baza la 10
glListBase(10);
// executam următoarele 10 liste   - de la 10 la 19
glCallLists(10,GL_INT,lista);

// setam baza la 20
glListBase(20);
// executam ultimele 10 liste – de la 20 la 29
glCallLists(10,GL_INT,lista);
```

O lista de desenare poate conține la rându-i alte liste de desenare pe care le apelează:

Exemplu :

```
glNewList(10,GL_COMPILE);

        // instructiuni OpenGL
        glCallList(1);
        glCallList(2);
        …
        glCallList(5);
            …
        glCallList(6);

glEndList();
```

Totuşi există o serie de comenzi OpenGL care nu sunt salvate prin punerea lor într-o listă, ceea ce înseamnă cu nu vor fi executate la chemarea listei, acestea sunt :

glIsList
glGenLists
glDeleteLists
glFeedbackBuffer
glSelectBuffer
glRenderMode
glReadPixels

glPixelStore
glFlush
glFinish
glIsEnabled
glGet

Exercițiu 1. Creați patru tipuri de obiecte din primitive OpenGL – un corp prismatic de forma lui L, un corp prismatic nerotunjit de forma lui U, unul de forma lui T şi altul de forma lui Z. Aceste corpuri vor avea specificate materiale diferite (folosiți glMaterialfv), scena va fi iluminată. Aceste obiecte vor fi păstrate în liste, care la creare vor fi doar compilate. Să se afişeze pe 4 rânduri, în 7 coloane într-o fereastră, corpuri de tipul respectiv, pe fiecare rând de acelaşi tip, iar pe coloană corpurile vor avea orientări diferite la alegere. Pe aceeaşi coloană corpurile vor avea aceeaşi orientare.

4.9 Lucrare de laborator 9.

4.9.1 Lucrul cu imagini rastru în OpenGL

Ca orice standard grafic OpenGL trebuia să includă şi funcţii de desenare de imagini de tip rastru. Ce înseamnă rastru? Înseamnă că imaginea este compusă din puncte, şi nu din entităţi grafice scalabile ca în cazul imaginilor vectoriale (wmf), ale fonturilor truetype etc. Un reprezentant important al acestei clase este BMP , un altul este Iconul (iconiţa).

Fişierele Bitmap utilizate de OpenGL trebuie să fie necomprimate având culoarea pe 24 de biţi. Oricum chiar dacă se dispune de astfel de fişiere OpenGL nu are funcţii care să le citească pentru că nu oferă suport de lucrul de fişiere. Citirea fişierului, alocarea memoriei în care acesta va fi citit este treaba mediului de programare în care este integrat OpenGL. Totuşi odată ce matricea de pixeli – mai corect spus de culori de pixel – este încărcată în memorie, într-un vector, aceasta poate fi folosită de către OpenGL.

Pentru a afişa un bitmap păstrat într-un vector se poate folosi funcţia cu prototipul:

```
void glBitmap(GLsizei width,  GLsizei height,
                    GLfloat xorig,  GLfloat yorig,
                     GLfloat xmove,  GLfloat ymove,
                            const GLubyte *bitmap
);
```

, unde width este lungimea imaginii afişate,
height este înălţimea imaginii;
xorig, yorig este poziţia zonei de afişare în cadrul imaginii;
xmove, ymove valori ale actualizării poziţiei pointerului de rastru (acesta specifică poziţia pentru orice grafică de tip rastru în mediul OpenGL)
vector la imaginea de tip rastru – specificat ca o colecţie de octeţi.

Pentru a specifica poziţia curentă a unui obiect rastru se foloseşte funcţia glRasterPos, care are o multitudine de variante, recomandabil ar fi folosirea funcţiei cu următorul prototip:

void glRasterPos3f(GLfloat x, GLfloat y, GLfloat z);

, unde x, y, z sunt coordonatele pointerului de rastru.

Alte funcţii de grafică rastru sunt :
glDrawPixels – pentru afişarea de imagine de tip rastru;
glCopyPixels – pentru copierea unei imagini din buffer pe ecran;
glReadBuffer – specifică bufferul din care se citeşte imaginea;
glReadPixels – citeşte într-un vector pixeli de pe ecran;
glPixelMap – specifică un mod de interpretarea a culorii pixelilor manipulaţi;
glPixelStore – specifică cum sunt citiţi sau scrişi pixelii din memorie;
glPixelZoom – specifică cum sunt scalaţi pixelii;
Funcţia glDrawPixels este o altă abordare a afişării imaginilor de tip rastru ea are prototipul:

```
void glDrawPixels(GLsizei width,  GLsizei height,
                  GLenum format,  GLenum type,
                  const GLvoid *pixels
);
```

,unde width este lungimea imaginii,
height este înălţimea imaginii păstrate în vector;
format este formatul pixelilor păstraţi în vector;
type este tipul elementelor vectorului;
pixels este un vector în care este păstrată imagine.

În ceea ce priveşte formatul sunt disponibile următoarele opţiuni :

GL_COLOR_INDEX – culoare pixelor este definită pe bază de indici şi paletă;
GL_STENCIL_INDEX - se referă la indici din bufferul şablon;
GL_DEPTH_COMPONENT - se referă la indici din bufferul de adâncime ;
GL_RGBA - se referă la pixeli a căror culoare este după standardul RGBA;
GL_RED - se referă la intensităţile componentei roşii a culorii pixelilor, fiecare valoare din vector fiind o intensitate;
GL_GREEN - acelaşi lucru dar pentru verde;
GL_BLUE - acelaşi lucru dar pentru albastru;
GL_ALPHA - pentru alpha;
GL_RGB – defineşte un pixel compus din trei componente roşu , verde şi albastru, practic o imagine având culoarea pe 24 de biţi;
GL_LUMINANCE – se specifică culoare pixelului pe baza luminozităţii;
GL_LUMINANCE_ALPHA – specifică atât luminozitatea cât şi alpha ;
GL_BGR_EXT – specifică pixeli a căror culoare este precizată de componenta albastră, verde, roşie efectiv în această ordine;
GL_BGRA_EXT specifică pixeli a căror culoare este precizată de componenta albastră, verde, roşie, alpha efectiv în această ordine.

Tipul poate fi precizat prin următoarele constante:

GL_UNSIGNED_BYTE – întreg pe opt biți fără semn – la care corespund tipul OpengGL - GLubyte;

GL_BYTE – întreg pe opt biți cu semn – la care corespund tipul OpengGL - GLbyte;

GL_BITMAP – câte un bit – stocați pe câte un întreg pe 8 biți;

GL_UNSIGNED_SHORT – întreg pe 16 biți fără semn – GLushort;

GL_SHORT –întreg pe 16 biți cu semn – Glshort;

GL_UNSIGNED_INT – întreg pe 32 de biți fără semn – Gluint;

GL_INT - întreg pe 32 de biți cu semn – Glint ;

GL_FLOAT – tip real în simplă precizie - glFloat;

Exemplu de folosire:

```
glDrawPixels(BitmapInfo->bmiHeader.biWidth,
        BitmapInfo->bmiHeader.biHeight,
        GL_RGB, GL_UNSIGNED_BYTE, BitmapBits);
```

, cu acest apel se afişează o imagine bitmap citită în memorie, imagine pe 24 de biți. BitmapBits este un vector spre pixelii imaginii, se observă că primi doi parametri specifică lungimea şi înălțimea imaginii.

Funcția glPixelStore este deosebit de importantă când se lucrează cu funcțiile de afişare prin această specificându-se cum vor fi interpretate valorile din vector. Ea are variantele:

```
void glPixelStoref( GLenum pname, GLfloat param);

void glPixelStorei( GLenum pname, GLint param);
```

Unde pname specifică tipul modificării vizate, aceasta prin folosirea unei constante specificator. Aceste constante sunt:

GL_PACK_SWAP_BYTES - inversează ordinea octeților componenți ai pixelului (dacă param este TRUE);

GL_PACK_LSB_FIRST – aranjează biții dintr-un byte astfel încât astfel încât cel mai puțin semnificativ să fie procesat primul;

GL_PACK_ROW_LENGTH – specifică numărul de pixeli dintr-un rând – folosind param;

GL_PACK_SKIP_PIXELS – nu mai afişează un număr de pixeli specificat în lparam, afişare începând de la pixelii care urmează ;

GL_PACK_SKIP_ROWS – nu mai afişează primele param linii ;

GL_PACK_ALIGNMENT – specifică modul de aranjare a datelor în vector, de obicei se foloseşte valoare 4 – grupare de 16 biți adică, se mai pot folosi valorile 1,2, 8 ;

GL_UNPACK_SWAP_BYTES – schimbă ordine octeților la despachetare bitmapului;
GL_UNPACK_ROW_LENGTH – la fel ca la împachetare;
GL_UNPACK_SKIP_PIXELS – la fel ca la împachetare;
GL_UNPACK_SKIP_ROWS – la fel ca la împachetare;
GL_UNPACK_ALIGNMENT – la fel ca la împachetare;

Exemplu :

```
glPixelStorei(GL_PACK_ALIGNMENT, 4);
glPixelStorei(GL_PACK_ROW_LENGTH, 0);
glPixelStorei(GL_PACK_SKIP_ROWS, 0);
glPixelStorei(GL_PACK_SKIP_PIXELS, 0);
glReadPixels(0, 0, viewport[2], viewport[3], GL_RGB, GL_UNSIGNED_BYTE,
        bits);
```

sau…

```
glPixelTransferi(GL_UNPACK_ALIGNMENT, 4);
glPixelTransferi(GL_INDEX_OFFSET, 1);
```

Folosind glPixelZoom putem scala o imagine, această funcție are prototipul:

void glPixelZoom(GLfloat xfactor, GLfloat yfactor);

, unde xfactor şi yfactor sunt indicii de scalare pe orizontală şi pe verticală e imaginii, unde valori supraunitare pozitive mărim imaginea, pentru valori subunitare pozitive micşorăm imaginea iar pentru valori negative inversăm în oglindă imaginea.

Pentru a citi pixeli dintr-un buffer al OpenGL într-un vector folosim funcția cu prototipul:

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GLenum format, GLenum type, GLvoid *pixels
);
```

, unde x, y sunt poziția colțului din stânga – sus de unde va începe citirea
, width, height – sunt lungimea şi înălțimea zonei citite
, format – specifică atât bufferul din care se face citirea cât şi componenta dorită,
, type – este tipul elementelor vectorului;
, pixels – este vectorul în care va postată imaginea.

Specificatori de format :

GL_COLOR_INDEX - indice de culoare – când se utilizează paleta;
GL_STENCIL_INDEX – indice din bufferul şablon;
GL_DEPTH_COMPONENT – indice din bufferul de adâncime;
GL_RED – componenta roşie a culorii din bufferul de culoare (de adâncime) ;
GL_GREEN - componenta verde a culorii din bufferul de culoare (de adâncime) ;
GL_BLUE - componenta albastră a culorii din bufferul de culoare (de adâncime) ;
GL_ALPHA - componenta alfa a culorii din bufferul de culoare (de adâncime) ;
GL_RGB – culoare efectiva ;
GL_RGBA – culoare plus indicele alfa;
GL_BGR_EXT – culoare dar în format BGR;
GL_BGRA_EXT – culoare plus alfa în format BGRA;
GL_LUMINANCE – luminozitatea;
GL_LUMINANCE_ALPHA – luminozitatea plus alfa;

De exemplu pentru a salva o anumită regiune dintr-un ecran într-un vector, formatul va ales GL_RGB, vectorul respectiv va putea fi scris ca imagine într-un fişier bitmap.
Tipul type are aceleaşi caracteristici ca şi la glDrawPixels.

glReadbuffer cu prototipul următor permite stabilirea unei surse de pixeli color pentru o serie de funcţii:

void glReadBuffer(GLenum mode);

, mode poate lua valorile –
GL_FRONT_LEFT- buffer de afişare principal stâng – pentru display stereoscopic;
GL_FRONT_RIGHT- buffer de afişare principal drept – pentru display stereoscopic;
GL_BACK_LEFT- buffer de afişare secundar stâng – pentru display stereoscopic;
GL_BACK_RIGHT- buffer de afişare secundar drept – pentru display stereoscopic;
GL_FRONT – buffer de afişare principal;
GL_BACK- buffer de afişare secundar – când se realizează double – buffering;
GL_LEFT- buffer stâng;
GL_RIGHT- buffer drept;
GL_AUXi – buffer auxiliary;

In mod obişnuit se va folosi apelarea:

```
glReadBuffer(GL_FRONT);
```

Funcția glCopyPixels cu următorul prototip permite copierea de pixeli între bufferele OpenGL:

```
void glCopyPixels(  GLint x,  GLint y,
                    GLsizei width,  GLsizei height,
                    GLenum type
);
```

, unde x, y, width, height specifică regiunea ce se copie şi în care se copie,

, type specifică bufferul în care se copie, bufferul din care se copie fiind specificat prin glReadBuffer.

Type poate lua valorile:

GL_COLOR – destinația este bufferul de afişare curent;

GL_DEPTH – destinația este bufferul de adâncime;

GL_STENCIL – destinația este bufferul şablon.

Exercițiu 1. Alocați un vector de tip GLubyte (cu funcția malloc) în care să păstrați o imagine pe 24 de biți. Imaginea va conține steagul României – culorile roşu, galben şi albastru. Lungimea imaginii va fi de două ori înălțimea. Afişați steagul. Acelaşi lucru pentru steagul Franței şi Germaniei. Afişați la final imaginea fiecărui steag în acelaşi ecran. Creați o funcție care să selecteze steagurile. Schimbarea steagului selectat se va face apăsând bara de spațiu. Steagurile au aceeaşi mărime, sunt desenate pe acelaşi rând,

Steagul selectat va fi adus mai în față decât celelalte neselectate.

4.9.2 Încărcarea unei imagini bitmap din fişier

În continuare se va prezenta o funcție de încărcare într-o structură din memorie a unei imagini bitmap pe 24 de biți:

```
struct bmptype{
        GLubyte *bits;
        BITMAPFILEHEADER bmpheader;
        BITMAPINFOHEADER bmpinfo;
                        };
bmptype imaginebmp;

int bmpload(char *bmpfilename,bmptype *bmp)
{
FILE *fp;
int infosize,bitsize;
```

```
                        if ((fp = fopen(bmpfilename, "rb")) == NULL)
                                return (NULL);
                        fread(&bmp->bmpheader,
sizeof(BITMAPFILEHEADER), 1, fp);
                        if (bmp->bmpheader.bfType != 'MB')
                        {

                                fclose(fp);
                                return (NULL);
                        } else
                        {
        infosize = bmp->bmpheader.bfOffBits - sizeof(BITMAPFILEHEADER);
                        fread (&bmp->bmpinfo, 1, infosize, fp);
                        if ((bitsize = bmp->bmpinfo.biSizeImage) == 0)
                         bitsize = bmp->bmpinfo.biWidth *
                        (bmp->bmpinfo.biBitCount + 7) / 8 *
                                abs(bmp->bmpinfo.biHeight);
                        bmp->bits = (GLubyte *)malloc(bitsize);
                           fread(bmp->bits, 1, bitsize, fp);
                                fclose(fp);
        }
        }
```

Se observă că se alocă în funcţia de citirea a fişierului şi memorie pentru vectorul ce conţine imaginea. Bmpfilename este numele fişierului de tip bitmap ce se deschide.

Exemplu de utilizare:
Pentru încarcare:

```
bmpload(”romania.bmp”,imaginebmp);
```

Pentru afişare:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
   glPixelZoom(1, 1);
   glRasterPos3f(-15,-11,-20);
         glDrawPixels(imaginebmp.bmpinfo.biWidth,
                  imaginebmp.bmpinfo.biHeight,
            GL_RGB, GL_UNSIGNED_BYTE imaginebmp.,bits);
```

Exercițiu 2. Folosind funcția de încărcare de bitmapuri, încărcați steagurile oficiale ale României, Marii Britanii, Franței, Germaniei, Grecie, UE şi creați un program de selectarea ca cel anterior.

4.10 Lucrare de laborator 10.

4.10.1 Aplicarea texturilor

Până acum s-au creat obiecte, s-au iluminat, s-au creat efecte de reflexie. Totuşi obiectele create deşi au formă şi culoare încă nu seamănă prea bine cu cele din realitate. Acest lucru se poate rezolva aplicându-le o textură. Dacă se dispune de forma reală şi de o textură potrivită, combinate cu un efect de iluminare şi o modalitate de creare realistică a umbrelor , imaginea obiectului simulat față de cel real va fi greu sau chiar imposibil de deosebit cu ochiul liber. Ceea ce oferă OpenGL este o imagine care se apropie în proporție de 80 – 90 % de imaginea obiectului real. În jocurile actuale aceasta valoare poate fi împinsă până la 95%, iar în editoarele specializate, profesionale de grafică 3D aceasta poate fi împinsă până la 98-99% şi chiar 99.7% diferențele fiind practic insesizabile. Ce înseamnă o textură? O textură este o imagine bidimensională (sau unidimensională) care acoperă o anumită suprafață a unui corp, în special cea vizibilă, mulându-se de obicei după forma corpului. Texturarea este procesul de asociere dintre o anumită suprafață şi o imagine. În OpenGL pentru texturare se pot folosi imagini uni- şi bidimensionale.

4.10.2 Texturarea unidimensională

Texturarea unidimensională sau 1D foloseşte drept textură un singur rând de pixeli. Este o texturare nepretențioasă rapidă. Ea poate fi utilizată pentru texturarea unui curcubeu sau a inelelor unei planete (Saturn de exemplu).

Pentru a specifica o texturare 1D se foloseşte funcția cu prototipul:

```
void glTexImage1D(  GLenum target,  GLint level,  GLint internalformat,
  GLsizei width,  GLint border,  GLenum format,  GLenum type,
  const GLvoid *pixels
);
```

, unde target specifică tipul texturii totdeauna egal cu GL_TEXTURE_1D,
level specifică nivelul de detaliu al texturii , 0 este nivelul de bază,
internalformat, poate fi o valoare 1,2,3 sau 4 sau una din constantele :

GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12,

GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12 sau GL_RGBA16.

width – lungime liniei de culoare, sau a dimensiuni texturii ;
border – grosimea marginii – poate fi 0 sau 1 ;
format – este formatul pixelilor texturii – poate fi :
GL_COLOR_INDEX
GL_RED
GL_GREEN
GL_BLUE
GL_ALPHA
GL_RGBA
GL_BGR_EXT
GL_BGRA_EXT
GL_LUMINANCE
GL_LUMINANCE_ALPHA
constante descrise în laboratoarele precedente.
type – este tipul elementelor vectorului în care este definită textura;
pixels – este chiar vectorul în care se află textura;

Exemplu :

```
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage1D(GL_TEXTURE_1D, 0, 3, 8, 0, GL_RGB, GL_UNSIGNED_BYTE,
        textura);
```

Se observă folosirea funcției de configurare a modului de texturare. În exemplu se specifică o tranziție liniară pentru zonă texturată mai mare, respectiv mai mică decât rezoluția texturii.

Pentru a activa texturarea 1D se foloseşte următoarea linie de cod :

```
glEnable(GL_TEXTURE_1D);
```

, aceasta se plasează în program înaintea oricărei afişări de obiecte texturate 1D.

Pentru a specifica poziția în cadrul texturii pe baza căreia se va textura o regiune se foloseşte funcția cu prototipul :

```
void glTexCoord1f(  GLfloat poz);
```

, poz este o valoare cuprinsă între 0.0 şi 1.0 respectiv începutul texturii şi sfârşitul ei.

4.10.3 Texturarea 2D

Pentru a textura un obiect cu o textură bidimensională (o imagine) se poate folosi funcția cu prototipul :

```
void glTexImage2D(  GLenum target,  GLint level,
                    GLint internalformat,
                    GLsizei width,  GLsizei height,
                    GLint border,  GLenum format,  GLenum type,
  const GLvoid *pixels
);
```

,unde target va fi întotdeauna egal cu GL_TEXTURE_2D,
level – nivelul de detaliu al texturării – 0 nivel de bază imaginea originală, n – nivel redus al detaliului față de imaginea de bază ;
internalformat – acelaşi ca la texturarea 1D ;
width, height – lungimea, înălțimea texturii ;
border – grosimea marginii poate fi 0 sau 1;
format – acelaşi ca la texturarea 1D;
type - acelaşi ca la texturarea 1D;
pixels – textura bidimensională sub forma unui vector ;

Exemplu :

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
glPixelStorei(GL_UNPACK_SKIP_ROWS, 0);
glPixelStorei(GL_UNPACK_SKIP_PIXELS, 0);
glTexImage2D(GL_TEXTURE_2D, 0, 3, info->bmiHeader.biWidth,
        info->bmiHeader.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
        rgb);
```

Pentru a specifica coordonatele texturii vom folosi o varianta a funcției glTexCoord :

```
void glTexCoord2f(  GLfloat s,  GLfloat t);
```

Coordonatele unei texturi sunt date prin folosirea indicilor s,t,r,q.
Pentru texturi 2D sunt utili doar s,t. acestea specificând coordinate în cadrul texturii. Ei variază între 0.0 şi 1.0, valori prin care parcurge întreaga suprafață a texturii.

Pentru texturi se pot seta anumiți parametri de texturare prin intermediul funcției cu prototipul:

```
void glTexParameteri(GLenum target, GLenum pname, GLint param);
```

,unde target poate lua valorile GL_TEXTURE_1D sau GL_TEXTURE_2D, în funcție de textura la care ne referim;

pname poate lua una din valorile :

GL_TEXTURE_MIN_FILTER – specifică comportarea texturării când suprafața texturată are o întindere în pixeli mai mică decât cea a texturii;

GL_TEXTURE_MAG_FILTER - specifică comportarea texturării când suprafața texturată are o întindere în pixeli mai mare decât cea a texturii ;

GL_TEXTURE_WRAP_S – specifică modul de texturare după s când zona texturată este mai mare decât textura folosită ;

GL_TEXTURE_WRAP_T - specifică modul de texturare după s când zona texturată este mai mare decât textura folosită ;

param – specifică o valoare pentru proprietatea specificată în pname:

pentru GL_TEXTURE_MIN_FILTER / GL_TEXTURE_MAG_FILTER poate fi una valorile –

GL_NEAREST – returnează pixelul cel mai aproape de centru;

GL_LINEAR - aproximează liniar textura;

GL_NEAREST_MIPMAP_NEAREST – pixelul se obține din texturi mipmap –

GL_LINEAR_MIPMAP_NEAREST pixelul se obține din texturi mipmap –

GL_NEAREST_MIPMAP_LINEAR pixelul se obține din texturi mipmap –

GL_LINEAR_MIPMAP_LINEAR pixelul se obține din texturi mipmap –

pentru GL_TEXTURE_WRAP_S / GL_TEXTURE_WRAP_T se pot alege valorile:

GL_CLAMP – dacă s-a depăşit textura nu se mai desenează nimic;

GL_REPEAT– dacă s-a depăşit textura se reia aceeaşi textură prin repetare;

O altă funcție de configurare ar fi glTexEnv cu prototipul :

void glTexEnvi(GLenum target, GLenum pname, GLint param);

La care:

target este scopul funcției – aceste trebuie să fie GL_TEXTURE_ENV(Env – vine de la Environment ce se traduce prin mediu, adică mediul OpenGL)

pname – trebuie să fie setat or pe GL_TEXTURE_ENV_MODE or pe GL_TEXTURE_ENV_COLOR

param – poate fi una din constantele :

GL_MODULATE — realizează un produs între textură şi bufferul de afişare;

GL_DECAL – textura este afişata direct pe ecran;

GL_BLEND - textura este combinată cu o culoare înainte de a fi afişată pe ecran;

Pentru GL_TEXTURE_ENV_COLOR – se va specifica culoarea de combinare ;

În mod obişnuit se foloseşte apelarea :

glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

Un alt aspect este generarea automată a coordonatelor texturilor. Pentru a activa această obțiune se folosesc secvențele:

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Pentru a configura generarea :

```
static GLint s_vector[4] = { 2, 0, 0, 0 };
static GLint t_vector[4] = { 0, 0, 2, 0 };

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeniv(GL_S, GL_OBJECT_PLANE, s_vector);

glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeniv(GL_T, GL_OBJECT_PLANE, t_vector);
```

Un s_vector şi t_vector sunt folosiți ca parametri într-o expresie care generează valoare efectivă a coordonatei texturii pe suprafața texturată astfel :

```
g= v[0]*x+v[1]*y+v[2]*z+v[3]*w;
```

,unde g este coordonata de texturare generată, x,y,z,w coordinate ale punctului de pe suprafață, v este vectorul specificat în exemplu fie prin s_vector fie prin t_vector.

Exercițiu 1. Texturați o piramidă cu o textură care să creeze efectul de trepte. Mai întâi calculați texturile şi afişați apoi încercați o generare automată a coordonatelor de texturare.

Exercițiu 2. Creați un cilindru pe care să-l texturați.

4.10.4 Utilizarea texturilor „mipmap”

Texturile mipmap se folosesc pentru a creşte viteză de procesare, pentru suprafețe îndepărtate se folosesc texturi cu rezoluție inferioară, în timp ce pentru a aceleaşi suprafețe apropiate se folosesc texturi cu cea mai bună rezoluție disponibile. OpenGL generează plecând de la o imagine de bază texturi cu rezoluții înjumătățite, pe câte nivele se doreşte.

Pentru a specifica folosirea texturilor mipmap respectiv generarea acestora se poate folosi o secvență asemănătoare cu următoarea:

```
// pentru texturi 1D
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
        GL_NEAREST_MIPMAP_LINEAR);
gluBuild1DMipmaps(GL_TEXTURE_1D, 3, 8, 0, GL_RGB,
GL_UNSIGNED_BYTE,
         roygbiv_image);

// pentru texturi 2D
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_NEAREST_MIPMAP_NEAREST);
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, info->bmiHeader.biWidth,
         info->bmiHeader.biHeight, 0, GL_RGB,
         GL_UNSIGNED_BYTE, rgb);
```

Aceste funcții au prototipurile:

```
int gluBuild1DMipmaps(  GLenum target,  GLint components,
                        GLint width,  GLenum format,GLenum type,
                        const void *data
);

int gluBuild2DMipmaps(  GLenum target,  GLint components,
                        GLint width,  GLint height,
                        GLenum format,  GLenum type,
```

const void *data

);

Se observă că sunt aceeaşi parametrii de la glTexImage.

Texturile care vor fi generate vor trebui încărcate cu glTexImage specificând prin level nivelul mipmap utilizat.

Este recomandat ca la încărcarea texturilor să se folosească liste de afişare care reduc mult timpul de procesare.

Exercițiu 3. Desenați un dreptunghi care va fi texturat cu o imagine de înaltă rezoluție. Imaginea va fi centrată pe ecran, la apăsare tastei sus imaginea se va apropia, la apăsare tastei jos imaginea se va depărta. Utilizați mipmap şi listele de afişare pentru a optimiza programul.

4.11 Lucrare de laborator 11.

4.11.1 Cuadrice

Alternativa la obiectele grafice predefinite în glaux sunt o serie de obiecte definite în glu, obiecte care sunt grupate sub denumirea de cuadrice. Acestea sunt:

- gluSphere – desenează o sferă;
- gluCylinder – desenează un cilindru;
- gluDisk – desenează un disc;
- gluPartialDisk – desenează un fragment de disc;

Spre deosebire de sfera desenată prin glaux sfera generată prin cuadrice poate fi configurată, calitatea obiectului, a formei, poate fi îmbunătățită, desigur cu timpi de procesare suplimentari.

Pentru a putea afişa o cuadrică mai întâi trebuie creată, iar la sfârşit după ce s-a încheiat lucrul cu ea trebuie dealocată.

Astfel avem funcţiile cu următoarele prototipuri :

pentru crearea de cuadrice:

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

pentru dealocarea de cuadrice:

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

, exemplu:

```
myquadric=gluNewQuadric();
…
gluDeleteQuadric(myquadric);
```

, pentru a configura o cuadrica vom folosi secvența:

```
gluQuadricDrawStyle(myquadric,GLU_FILL);//1
gluQuadricNormals(myquadric,GLU_SMOOTH); //2
gluQuadricOrientation(myquadric,GLU_OUTSIDE); //3
gluQuadricTexture(myquadric,GL_TRUE); //4
```

Linia 1 specifică că modelul va fi acoperit în întregime.
Linia 2 specifică că normalele vor fi generate pentru un aspect lin, fin al suprafeței.
Linia 3 specifică orientarea normalelor cuadricei spre exterior;
Linie 4 specifică că cuadrica va fi texturată.

Aceste funcții au următoarele prototipuri:

```
void gluQuadricDrawStyle( GLUquadricObj *qobj, GLenum drawStyle);
void gluQuadricNormals( GLUquadricObj *qobj, GLenum normals);
void gluQuadricOrientation( GLUquadricObj *quadObject,
                            GLenum orientation);
void gluQuadricTexture( GLUquadricObj *quadObject,
                        GLboolean textureCoords);
```

unde drawStyle specifică stilul de desenat al cuadricei:
GLU_FILL – cuadrica va fi desenata prin poligoane umplute;
GLU_LINE – cuadrica va fi desenata din linii;
GLU_SILHUETTE – cuadrica este desenata prin linii, cu câteva excepții, în care unele linii nu se mai trasează;
GLU_POINT – cuadrica este desenată numai prin vârfurile poligoanelor componente;
pentru normals sunt disponibile două opțiuni:
GLU_NONE – nu se generează normale ;
GLU_FLAT – pentru fiecare fațetă a cuadricei se generează câte o normală ;
GLU_SMOOTH – este generată câte o normală pentru fiecare vârf al cuadricei ;
pentru orientation avem :
GLU_INSIDE – normalele sunt orientate spre interior ;
GLU_OUTSIDE – normalele sunt orientate spre exterior ;
pentru texturare – există – doar două opțiuni GL_TRUE sau GL_FALSE.

Funcțiile care desenează efectiv cuadricele au prototipurile:

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

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

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

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

Se observă că slices este caracteristic pentru toate cuadricele acesta specifică numărul de bucăți în care va fi descompus corpul geometric, cu cât acesta este mai mare cu atât aspectul corpului va fi mai precis.

Al doilea element de descompunere este stacks sau loops, care descompune pe cealaltă axă analizabilă corpul geometric.

Pentru a defini parametrii efectivi geometrici pentru sferă s-a dat doar raza – Radius, pentru cilindru raza bazei baseRadius, raza capătului superior topRadius, şi înălțimea. Pentru disc s-a dat doar raza internă – innerRadius şi cea externă – outerRadius. Pentru discul parțial s-a dat şi unghiul de început al secțiunii – startAngle şi valoarea unghiului de acoperit – sweepAngle.

Exercițiu 1. Creați patru cuadrice câte una de acelaşi tip, şi cam de aceleaşi dimensiuni şi afişați-le pe toate în aceeaşi fereastră.

Exercițiu 2.Pentru programul precedent la apăsare barei de spațiu, toate cuadricele vor fi afişate pe rând în cele patru stiluri de desenare.

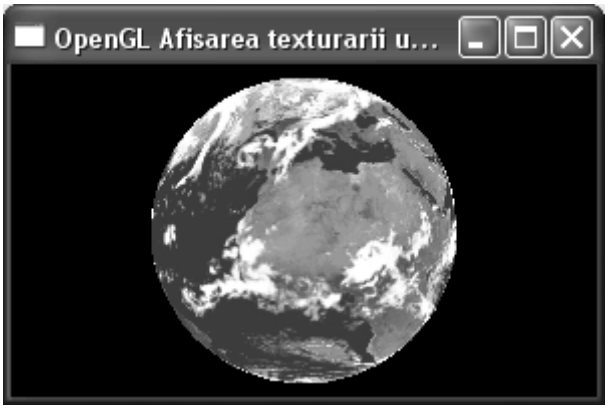
Exercițiu 3.Pentru cuadrica de tip sferă, setați iluminarea,o lumină, şi materialul sferei. Alegeți stilul de desenare plin. Prin tastele direcționale, creşteți sau scădeți cu o unitate numărul de slices sau de stacks, şi afişați schimbarea.

Exercițiu 4. Pentru o sferă încercați să o texturați cu bitmapul numit moon.bmp. Afişați. Acum încercați să creați o secvență de program astfel încât prin tastele direcționale să rotiți sfera. Veți folosi texturarea şi încărcarea de bitmapuri din laboratoarele anterioare.

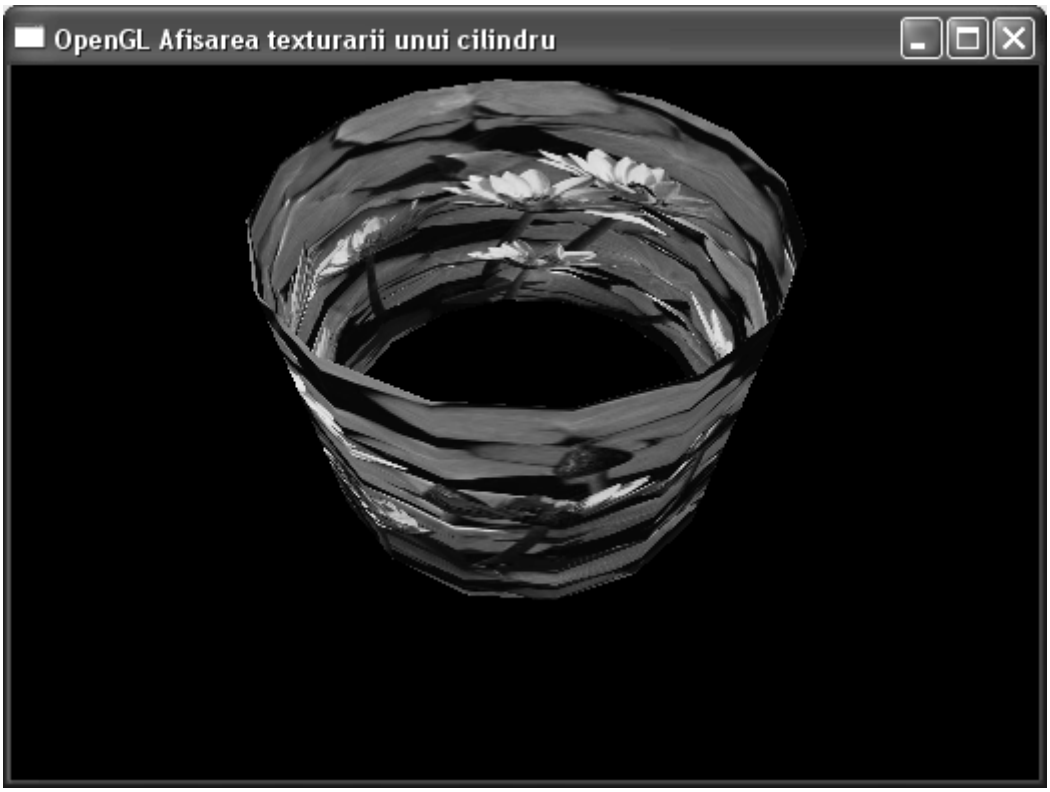
Exercițiu 5. Creați modelul Soare, Luna, Pământ, fiecare fiind o cuadrica. Soarele va fi o sferă galbenă lucioasă. Pentru Lună si Pamant se folosesc texturile moon.bmp şi earth.bmp. La rularea programului sferele să se afle în mişcare. Luna va avea o rază jumătate din raza Pământului. Iar Soarele o rază de trei ori câte cea Pământului. Pe baza funcției glLookAt şi a tastelor direcționale deplasați-vă prin sistemul solar creat.

Exemplu de texturare a Lunii:

Exemplu de texturare a Terrei:



Exemplu de texturare a unui cilindru:



4.12 Lucrare de laborator 12.

4.12.1 Lucrul cu bufferele în OpenGL

Operaţiile de afişare în OpenGL se fac prin intermediul bufferelor. Acestea sunt zone de memorie care acoperă (mapează) suprafaţa ferestrei de lucru, fiecărui pixel din fereastra de lucru corespunzându-i un pixel din buffer. Există mai multe tipuri de buffere. Există un buffer de afişare numite buffer de culoare,un buffer de culoare secundar pentru creşterea calităţii animaţiei,un buffer de adâncime, un buffer şablon (stencil), un buffer de acumulare.

4.12.2 Bufferul de culoare

La bufferul de culoare fiecare element din buffer constituie culoarea unui pixel, fie ea un indice de culoare, fie un set de componente de culoare RGBA. În OpenGL pentru a realiza minimum de afişare grafică avem nevoie doar de un buffer de culoare – numit şi buffer principal, buffer frontal sau buffer vizibil. Ori de câte ori se face o desenare vizibilă, aceasta se face în acest buffer, dacă desenăm în alt buffer trebuie să transferăm informaţia din bufferele respective în acest buffer pentru a o face vizibilă. Pentru a preciza că desenare se face direct în acest buffer în glaux folosim secvenţa:

```
auxInitDisplay(AUX_SINGLE | AUX_RGBA) ;
```

La animaţii sau la desenarea de scene complexe observăm că se produc pâlpâiri, aceasta deoarece în timpul desenării se reactualizează în permanenţă suprafaţa ecranului prin citirea acestui buffer, astfel chiar dacă suntem în mijlocul unei desenări, se va afişa pe ecran conţinutul bufferului. Pentru a evita acest lucru vom folosi două buffere cel vizibil, discutat anterior, şi unul invizibil un buffer secundar. Odată ce s-a specificat că se lucrează cu două buffere toate desenele ce se vor face din acel moment vor fi desenate în bufferul invizibil. Astfel vom desena tot ce dorim în bufferul secundar iar după ce operaţiile de desenare s-au terminat vom transfera datele din acest buffer în bufferul principal ce este vizibil. Acest transfer este îndeajuns de rapid pentru a se evita pâlpâirea.

În glaux pentru a specifica că se va lucra cu două buffere se foloseşte secvenţa:

```
auxInitDisplay(AUX_DOUBLE | AUX_RGBA) ;
```

, iar în Win32, acest mod de lucru este implicit, dar cu toate acestea trebuie specificat prin folosirea constantei PFD_DOUBLEBUFFER.

Pentru a transfera datele din bufferul invizibil în cel vizibil se foloseşte o funcţie care în glaux este :

```
auxSwapBuffers( ) ;
```

iar în Win32 :

SwapBuffers(hDC) ;

, unde hDC este dispozitivul de context ataşat ferestrei aplicației.

Totuşi dacă se doreşte schimbarea setării implicite prin care desenarea se face în bufferul secundar. Se poate folosi funcția cu prototipul :

void glDrawBuffer(GLenum mode);

,unde mode specifică bufferul în care OpenGL va desena acesta fiind specificat prin următoarele constante:

GL_NONE – nu se va desena în nici un buffer;
GL_FRONT_LEFT – specifică bufferul principal stâng – pentru aplicații stereoscopice;
GL_FRONT_RIGHT - specifică bufferul principal drept – pentru aplicații stereoscopice;
GL_BACK_LEFT - specifică bufferul secundar stâng – pentru aplicații stereoscopice;
GL_BACK_RIGHT specifică bufferul secundar drept – pentru aplicații stereoscopice;
GL_FRONT – specifică bufferul principal;
GL_BACK – specifică bufferul secundar;
GL_LEFT - specifică bufferul stâng atât cel principal cât şi cel secundar, pentru aplicații stereoscopice;
GL_RIGHT – specifică bufferul drept atât cel principal cât şi cel secundar, pentru aplicații stereoscopice;
GL_FRONT_AND_BACK - specifică atât bufferul principal cât şi cel secundar, cât şi bufferele stâng şi drept ale acestora;
GL_AUXi – specifică un buffer auxiliar i, i fiind o valoare între 0 şi GL_AUX_BUFFERS;

În mod implicit pentru lucrul cu un singur buffer mode este setat la GL_FRONT, iar pentru lucrul cu două buffere mode este setat la GL_BACK.

Exercițiu 1. Creați o aplicație în care animați un corp folosind glaux. Setați mai întâi utilizare unui mod de lucru cu un singur buffer, apoi cu două buffere. Observați schimbările în calitatea animației.

4.12.3 Bufferul de adâncime

Atunci când se creează o scenă complexă, sau un obiect compus din mai multe poligoane, se doreşte ca poligoanele din spate să nu apăra desenate peste cele din față. Sau la intersecții de suprafețe să se deseneze partea din suprafață vizibilă şi nu cea invizibilă. Acest lucru s-a rezolvat prin ataşarea la bufferul de culoare a unui buffer de adâncime. În bufferul de adâncime se păstrează poziția pe oz, a punctului desenat în bufferul de culoare cu o anumită precizie. Astfel dacă se va desena o nou obiect, „adâncimea" punctelor noului obiect va fi comparată cu cea din bufferul de adâncime, dacă va fi mai mică aceste puncte(condiție ce se analizează pentru fiecare punct în parte) vor fi desenate în bufferul de culoare şi se va actualiza valorile de adâncime din bufferul de adâncime, altfel nici unul dintre aceste buffere nu va fi modificat. Pentru a activa folosirea bufferelor de adâncime se va utiliza secvența:

glEnable(GL_DEPTH_TEST);

Pentru a stabili condiția de alterare a bufferelor, se va folosi funcția cu următorul prototip:

void glDepthFunc(GLenum func);

, unde func este o condiție, care implicit este GL_LESS adică dacă adâncime sau coordonată pe Oz a pixelului este mai mică decât a celui anterior desenat (la poziția respectivă de pe ecran, buffer) atunci acesta va fi desenat – pot fi precizate şi alte condiții prin următoarele constante:

GL_NEVER - afişare nu este permisă;
GL_LESS – afişarea este permisă dacă Z nou < Z vechi;
GL_EQUAL – afişarea este permisă dacă Z nou = Z vechi.
GL_LEQUAL – afişarea este permisă dacă Z nou <= Z vechi.
GL_GREATER – afişarea este permisă dacă Z nou > Z vechi.
GL_NOTEQUAL – afişarea este permisă dacă Z nou diferă de Z vechi.
GL_GEQUAL – afişarea este permisă dacă Z nou >= Z vechi.
GL_ALWAYS – modificare este permisă oricând.

Pe lângă aceste funcții mai sunt disponibile şi altele pentru configurarea lucrului cu buffere de adâncime în OpenGL acestea au următoarele prototipuri:

void glDepthRange(GLclampd znear, GLclampd zfar);

, care stabileşte limitele între care se încadrează factorii de adâncime în mod implicit acestea fiind 0.0 pentru znear şi 1.0 pentru zfar;

void glClearDepth(GLclampd depth);

, care stabileşte valoare de ştergere a bufferului de adâncime, implicit aceasta fiind egală 1.0.

Atunci când se şterge bufferul de culoare trebuie să se şteargă şi cel de a adâncime altfel, la o nouă scenă vor fi luate în considerare adâncimile punctelor precedente, aceasta se face cu secvența:

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

Crearea efectului de secționare folosind bufferul de adâncime

Putem folosi următoarea secvență:

```
glDrawBuffer(GL_NONE);

glBegin(GL_POLYGON);
  glVertex3f(-100.0, 100.0, cutting_plane);
  glVertex3f(100.0, 100.0, cutting_plane);
  glVertex3f(100.0, -100.0, cutting_plane);
  glVertex3f(-100.0, -100.0, cutting_plane);
glEnd();

glDrawBuffer(GL_BACK);
```

, ia funcționează în felul următor – înainte de secvență desenăm corpul, apoi desenăm un plan de secționare într-un buffer de culoare inexistent, fiind afectat doar planul de adâncime, tot ce este în spatele planului de secționare nu v-a mai fi afişat, planul efectiv va fi invizibil. Se observă că după trasarea acestui plan, se revine la modul normal de afişare.

Această secvență înlătură prin secționare doar partea din spate.

Exercițiu 2. Creați un program care să secționeze un corp afişând doar partea din spate, acest corp va fi un ceainic (auxSolidTeapot), veți folosi secvența precedentă, plus funcția
glDepthFunc ( ).

Precizia de calculare a adâncimii pentru acest buffer poate fi de 16biți sau de 32 de biți, valoare implicită este de 32 de biți. Cu cât această precizie este mai mare scena va putea fi mai complexă.

4.12.4 Folosirea bufferului şablon

Bufferul şablon (stencil) permite păstrarea anumitor zone din fereastră. Astfel dacă se proiectează un simulator de zbor, sau de curse, bordul avionului, sau cel al maşinii rămâne neschimbat comparativ cu mediul exterior, pentru aceasta se poate folosi bufferul şablon. Bufferul şablon are o abordare asemănătoare cu a bufferului de adâncime, având cu toate aceste a serie de funcţii specifice.

Pentru a activa bufferul şablon vom folosi secvența:

glEnable(GL_STENCIL_TEST);

În Win32 la specificatorul de format pentru pixel trebuie setat cStencilBits la 1(adică fiecărui pixel i se va asocia doar un singur bit în şablon)

Pentru se specifica în glaux folosirea bufferului de tip şablon se foloseşte secvența:

auxInitDisplay(AUX_RGB | AUX_DOUBLE | AUX_DEPTH | AUX_STENCIL);

În plus se mai folosesc funcțiile:

void glClearStencil(GLint s);

, care stabileşte valoare prin care se va şterge bufferul şablon;

void glStencilFunc(GLenum func, GLint ref, GLuint mask)

, stabileşte care puncte vor fi înlocuite şi care păstrate,
- func este acelaşi cal glDepthFunc
- ref este o valoare de refeință a bufferului şablon;
- mask este valoarea prin care se specifică o condiție suplimentară pentru păstrarea unui pixel;

void glStencilMask(GLuint mask)

, setează condiția de modificare a unui pixel, de obicei se foloseşte valoarea 1, pentru mască;

void glStencilOp(GLenum fail, GLenum zfail, GLzpass)

, stabileşte testele pentru trecerea prin şablon:
fail – specifică ce se întâmplă dacă pixelul nu a trecut de testul şablon – aceasta este definită prin constantele :
GL_KEEP – se păstrează valoarea din şablon;
GL_ZERO – se setează cu 0 valoarea din şablon;
GL_REPLACE – se înlocuieşte valoarea din şablon;

GL_INCR – se incrementează valoarea din şablon;
GL_DECR – se decrementează valoarea din şablon;
GL_INVERT – neagă pe biți valoare din şablon;
zfail – stabileşte acțiunea ce se întreprinde dacă testul şablon este trecut dar nu s-a trecut de testul de adâncime (acțiunea este descrisă de aceleaşi constante de mai înainte) ;
zpass – stabileşte acțiunea ce se întreprinde dacă s-a trecut de testul şablon şi de testul de adâncime ;

Exemplu :

```
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

sau

```
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

În plus mai este secvența de ştergere completă a bufferului şablon:

```
glClear(GL_STENCIL_BUFFER_BIT);
```

Exercițiu 3. Creați un program în care să animați un corp solid. Corpul va fi văzut printr-o deschidere circulară poziționată în mijlocul ecranului. Veți folosi bufferele şablon.

4.12.5 Folosirea bufferului de acumulare

Bufferul de acumulare poate fi folosit la cumularea, adunare de imagini de afişare, peste alte imagini, putând fi folosit pentru a simula efecte de inerție a vederii la care marginile corpului în mişcare sunt vizualizate începețoşat, suprapus. Sau mai poate fi folosit pentru o trecere mai lină de la muchiile ce delimitează un corp la mediul ambiant (anti – aliasing).

Pentru a activa folosirea bufferului de acumulare se foloseşte:
pentru Win32 se setează cAcumBits la o valoare diferită de 0,
pentru glaux se foloseşte secvența:

```
auxInitDisplayMode(AUX_RGB | AUX_DOUBLE | AUX_DEPTH | AUX_ACCUM);
```

, se observă folosirea constantei AUX_ACCUM.

Pentru a specifica modul cum se face acumularea în acest buffer se foloseşte funcția cu prototipul :

void glAccum(GLenum op, GLfloat value);

,unde op specifică acest mod prin constantele:

- GL_ACCUM – valoare componentelor culorii (0.0 – 1.0) este citită din bufferul sursă specificat prin glReadBuffer, înmulțită cu value şi adunată la valoarea din bufferul de acumulare;
- GL_LOAD - valoare componentelor culorii (0.0 – 1.0) este citită din bufferul sursă specificat prin glReadBuffer, înmulțită cu value şi scrisă direct în bufferul de acumulare înlocuind valoare precedentă;
- GL_ADD - adună valoare specificată de value la fiecare componentă a culorii (RGBA) specificată în bufferul de acumulare;
- GL_MULT - înmulțeşte fiecare componentă a culorii din bufferul de acumulare cu value şi produsul obținut este pus în bufferul de acumulare;
- GL_RETURN – transferă datele din bufferul de acumulare în bufferul de afişare, mai întâi înmulțind fiecare componenetă cu value şi normalizând-o pentru intervalul 0.0-1.0.
  , value este o valoare între 0.0 şi 1.0 prin care configurează acumularea;

Exemplu:

```
// se desenează scena ce va fi încetosata
draw_frame(0);
// transferăm in bufferul de acumulare 50% din intensitatea pixelilor
//din scena
glAccum(GL_LOAD, 0.5);

// redesenam de zece ori scena in la momente de timp successive
// si acumulam in bufferul de acumulare 5% din intensitatea pixelor din
// aceste secvente cadru
for (i = 1; i <= 10; i ++)
{
 draw_frame(-i);
 glAccum(GL_ACCUM, 0.05);
};

// afisam continutul bufferului de acumulare
glAccum(GL_RETURN, 1.0);
```

Exercițiu 4. Creați un program pe baza secvenței prezentate în care un corp în rotație sau alt gen de mişcare să prezinte astfel de încețoşări ca urmare a inerției ochiului.

4.13 Lucrare de laborator 13

4.13.1 Efecte speciale în OpenGL

OpenGL punea la dispoziția programatorului o serie de efecte speciale predefinite, din care două ceața (fog) şi îmbinarea(blend) sunt cele mai utilizate. Deşi au fost concepute vizând în principal scopul specificat prin denumire în anumite configurații se pot crea şi alte genuri de efecte surprinzătoare cu ajutorul acestora două şi lucrul cu bufferul de acumulare sau componenta alfa a culorii.

4.13.2 Crearea efectului de ceață

Efectul de ceață în OpenGL se manifesta ca o diminuare a culorii obiectelor scenei, până la dispariția completă a acestora. Culorile obiectelor sunt alterate prin culoarea cetii făcându-se o trecere graduală de la culoarea corpului la cea a cetii. Trecerea poate fi mai lină sau mai abruptă în funcție de parametrii configurabili ai cetii. Totuşi ceața în OpenGL, nu este mai mult decât acest lucru. Închipuirea că prin ceață s-ar dispune de nişte mase de aburi sub forma unor nori sau pâcle care s-ar putea dispune convenabil în fața sau printre obiecte, nu este situația de fapt a cetii standard din OpenGL. Totuşi efectul prezentat anterior se poate obține cu mai mult efort prin folosirea unor combinații de efecte. Ceața se poate configura pentru fiecare obiect în parte sau pentru un grup de obiect.

Pentru a activa ceața se foloseşte secvența:

```
glEnable(GL_FOG);
```

, iar pentru a o dezactiva:

```
glDisable(GL_FOG);
```

Pentru a configura ceața se poate folosi secvența :

```
glFogf(GL_FOG_MODE, GL_LINEAR);
glFogfv(GL_FOG_COLOR, fog_color);
```

Funcția glFogf specifică modul de estompare al luminii ce traversează ceața, ea are următorul prototip:

```
void glFogf(  GLenum pname,  GLfloat param);
```

, unde pname specfică parametrul configurat:

GL_FOG_MODE - specifică funcția de estompare(atenuare) folosită, pentru aceasta param poate lua valorile:

GL_LINEAR – funcție liniara;

GL_EXP – funcție exponențială;

GL_EXP2 – funcție exponențială pătratică;

GL_FOG_DENSITY – specifică densitatea ceții – implicit param este 1.0, acesta trebuie să fie pozitiv – folosit pentru atenuarea exponențială;

GL_FOG_START - pentru a specifica poziția pe oz de început al ceții,care implicit este 0.0, folosit pentru atenuare liniară;

GL_FOG_END – pentru a specifica poziția pe oz de depărtare a zonei de ceață , implicit este 1.0, folosit pentru atenuare liniară ;

GL_FOG_INDEX - pentru a specifica indicele de ceață, implicit este 0.0;

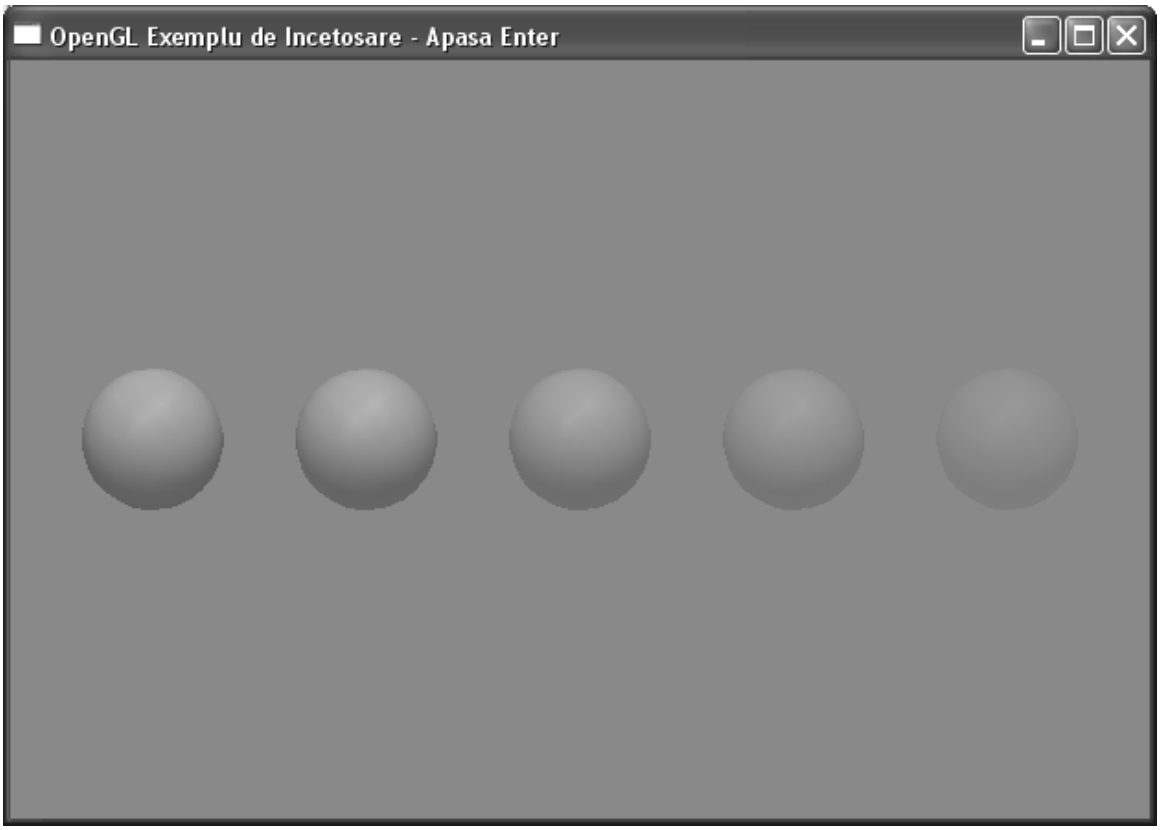
GL_FOG_COLOR - pentru a specifica culoarea ceții - unde param este un vector de patru elemente – este folosit în combinație cu varianta glFogfv sau glFogiv a funcției.

O altă secvență folosită este :

```
glHint (GL_FOG_HINT, GL_NICEST);
```

,care specifică calitatea maximă a efectului de ceață în dauna vitezei de procesare, altă variantă ar fi fost folosire constantei GL_FASTEST – care pune accentul de viteză.

Exemplu de folosire a ceții:



În care ceața s-a configurat cu secvența:

```
glEnable(GL_FOG);
glFogf(GL_FOG_MODE,GL_LINEAR);
glFogfv(GL_FOG_COLOR,fog_color);
glFogf(GL_FOG_DENSITY, FogDensity);
glFogf (GL_FOG_START, -maxim(30,400.0*(1-FogDensity)));
glFogf (GL_FOG_END, maxim(30,400.0*(1-FogDensity)));
```

Exercițiu 1.Creați un program în care să puteți selecta cele trei tipuri de extompări specifice ceții – liniară, exponențială, exponențială pătratică prin apăsare unei taste. Veți lucra cu glaux, desenând un singur corp. Adăugați programului şi opțiunea de a apropia – depărta corpul.

4.13.3 Crearea efectului de îmbinare

Cu ajutorul efectului de îmbinare(blend) se pot crea corpuri transparente, de diferite culori, chiar cu anumite efecte de iluminare.

Pentru a activa opțiunea de îmbinare se poate folosi secvența:

```
glEnable(GL_BLEND);
```

Pentru a configura îmbinarea se foloseşte funcția cu prototipul:

```
void glBlendFunc(  GLenum sfactor,    GLenum dfactor   );
```

, unde sfactor specifică sursa de culoare şi poate lua valorile:

- GL_ZERO – culoare sursă va fi negru (0,0,0,0)
- -GL_ONE – culoare sursă va fi alb(1,1,1,1);
- GL_SRC_COLOR – culoare sursă propriu-zisă;
- GL_DST_COLOR – culoare sursă este înmulțită prin culoare destinație;
- GL_ONE_MINUS_DST_COLOR – culoarea sursă este înmulțită cu 1 minus culoarea destinației;
- GL_SRC_ALPHA culoarea sursei este înmulțită cu alfa sursei;
- GL_ONE_MINUS_SRC_ALPHA culoarea sursei este înmulțită cu 1 minus alfa sursei;
- GL_DST_ALPHA- culoarea sursei este înmulțită cu alfa destinației
- GL_ONE_MINUS_DST_ALPHA – culoarea sursei este înmulțită cu 1 minus alfa destinației;
- GL_SRC_ALPHA_SATURATE – culoarea sursei este înmulțită cu minimul dintre alfa sursei şi 1 minus alfa destinației;

,iar dfactor specifică culoarea destinației luând valori descrise de constante asemănătoare:

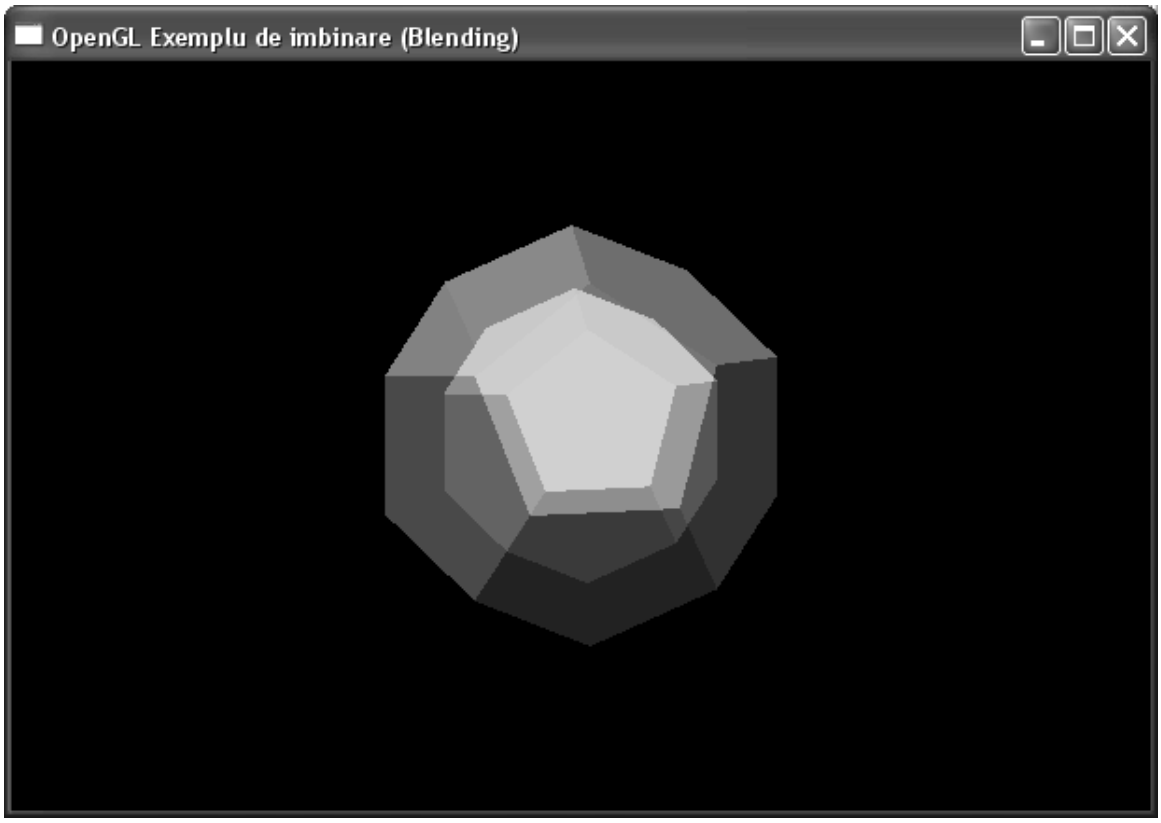
- GL_ZERO
- GL_ONE
- GL_SRC_COLOR
- GL_ONE_MINUS_SRC_COLOR
- GL_SRC_ALPHA
- GL_ONE_MINUS_SRC_ALPHA
- GL_DST_ALPHA
- GL_ONE_MINUS_DST_ALPHA.

dar de data acesta referitoare la destinație.

Exemplu:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

 glPushMatrix();
   glTranslatef(0.0, 0.0, -15.0);
   glRotatef(-rotation, 0.0, 1.0, 0.0);
  // desenam ceainicul netransparent
   glDisable(GL_BLEND);
   glColor3f(1.0, 1.0, 0.0);
   auxSolidTeapot(1.0);
 glPopMatrix();

 glPushMatrix();
   glTranslatef(0.0, 0.0, -10.0);
   glRotatef(rotation, 0.0, 1.0, 0.0);
  // desenam ceainicul transparent
   glEnable(GL_BLEND);
   // se observa ca alfa este setat la 0.25
   glColor4f(1.0, 1.0, 1.0, 0.25);
   auxSolidTeapot(1.0);
 glPopMatrix();
```

Exemplu de program de îmbinare:



Exercițiu 2. Creați un program în care în interiorul unui cub transparent să fie poziționat un cub netransparent, rotit puțin față de cel transparent.

Exercițiu 3. Modificând programul precedent – creați două cuburi transparente unul conținându-l pe celălalt şi al doilea pe un al treilea netransparent. Aceste cuburi vor fi centrate şi se vor roti în direcții diferite.

4.14 Lucrare de laborator 14.

4.14.1 Curbe Bezier şi Nurbs

Până acum cu primitivele de care dispunem putem desena curbe, suprafețe curbe doar prin nişte structuri complicate de cod. Însă mediul OpenGL pune la dispoziție nişte structuri ce permit generarea unor asemenea curbe doar prin specificare unor „centre de curbare" şi a capetelor curbelor,respectiv a marginilor suprafețelor. O abordare este cea prin evaluatori prin care construim curbe şi suprafețe şi gestionăm manual parametrii generați, o alta este folosind structurile NURBS, în care totul este prelucrat automat.

4.14.2 Curbele şi suprafețele Bezier

Teoria matematică şi formulele ce stau în spatele generării acestor curbe sunt destul de complexe pentru a fi expusă aici. În acest sens vom aborda acest subiect dintr-o perspectivă practică – adică se doreşte crearea unei structuri se va explica exact secvențele de cod folosite.

Exemplu:

```
   glMap1f(GL_MAP1_VERTEX_3,      // tipul curbei desenate
0.0f,              // limita inferioară a intervalului
100.0f,             // limita superioară a intervalului
3,              // distanta dintre puncte în vectori
nNumPoints,                 // numărul de puncte de control
&ctrlPoints[0][0]);          // vectorul punctelor de control

// activăm evaluatorul
glEnable(GL_MAP1_VERTEX_3);

// genereaz curba cu o precizie de o sută de diviziuni
glBegin(GL_LINE_STRIP);
      for(i = 0; i <= 100; i++)
            {
 // generează un vârf pe baza evaluatorului specificat
            glEvalCoord1f((GLfloat) i);
            }
glEnd();
```

Numărul de diviziuni poate di crescut în funcție de precizia grafică dorită.

Funcția glMap1f configurează un evaluator de curbe – unidimensional ea are următorul prototip:

```
void glMap1f(  GLenum target,  GLfloat u1,  GLfloat u2,
               GLint stride,  GLint order,  const GLfloat *points
);
```

, unde target – specifică rezultatul dorit a fi obținut – fie că se referă la nişte vârfuri, fie la normale, fie la culori, fie la coordonate de texturare, el este definit prin constantele:

GL_MAP1_VERTEX_3 – se referă la nişte vârfuri definite prin x, y, z (glVertef3f);

GL_MAP1_VERTEX_4 – se referă la nişte vârfuri definite prin x, y, z, w(glVertef4f) ;

GL_MAP1_INDEX - se referă la colori definite prin index;

GL_MAP1_COLOR_4 – se referă la culori RGBA(glColor4f);

GL_MAP1_NORMAL – se referă la o normală definită prin coordonatele vârfului x, y, z (glNormal3f);

GL_MAP1_TEXTURE_COORD_1 – se referă la coordonatele unei texturări cu texturi unidimensionale (glTexCoord1f);

GL_MAP1_TEXTURE_COORD_2 – se referă la coordonatele unei texturări cu texturi bidimensionale(glTexCoor21f) ;

GL_MAP1_TEXTURE_COORD_3 – se referă la coodonatele unei texturări cu texturi – specificate prin s, t, r(glTexCoord3f);

GL_MAP1_TEXTURE_COORD_4 – se referă la coordonatele unei texturări cu texturi – specificate prin s, t, r, q(glTexCoord4f);

, u1,u2 – valoare inferioară, respectiv superioară a mapării liniare a punctelor de pe curbă ;

, stride – numărul de componente ale unui punct definit în vectorul points ;

, order – numărul de puncte de control ;

, points – vectorul punctelor de control ;

Apoi urmează funcția prin care activăm generarea automată e elementelor grafice prin evaluatori. –

```
glEnable(tip_element_grafic);
```

, unde tipul este acelaşi ca şi target.

Urmează generarea efectivă a punctelor de către funcția cu prototipul:

```
void glEvalCoord1f(  GLfloat u );
```

, unde u este o valoare din intervalul u1, u2, cu semnificație specificată anterior, la prototip, şi-n secvența de program.

Exercițiu 1. Creați un program care să genereze o curbă alcătuită din o sută de diviziuni, această curbă este definită de 4 puncte de control. Cu ajutorul mouselui selectați câte un punct de control la care să-i schimbați poziția, apoi să-l deselectați.

4.14.3 Evaluatori pentru suprafețe.

Sunt practic identici cu cei de la curbe, diferența fiind că mai apare a coordonată.

Exemplu:

```
glMap2f(GL_MAP2_VERTEX_3,      // tipul evaluatorului
      0.0f,    // limita inferioara a coordonatei u
      10.0f,  // limita superioara a coordonatei u
      3,       // numarul de componente al unui element al vectorului
      3,      // numărul de puncte de control pe direcția u
      0.0f,  // limita inferioara a coordonatei v
      10.0f, // limita superioara a coordonatei v
      9,   // distanta dintre două elemente consecutive pe v
      3,  // numărul de puncte de control pe direcția v
      &ctrlPoints[0][0][0]); // vectorul cu puncte

      // activam evaluatorul
      glEnable(GL_MAP2_VERTEX_3);

      // generează punctele suprafeței
      glMapGrid2f(10,0.0f,10.0f,10,0.0f,10.0f);

      // afişează suprafața sub forma unei rețele de linii
      glEvalMesh2(GL_LINE,0,10,0,10);
```

Funcția glMap2f are prototipul:

```
void glMap2d(  GLenum target,  GLdouble u1,  GLdouble u2,
               GLint ustride,  GLint uorder,
                 GLdouble v1,  GLdouble v2,
                 GLint vstride,  GLint vorder,
                 const GLdouble *points
);
```

, pentru target se vor folosi practic aceleaşi constante cu diferența că în loc de MAP1 vom avea MAP2;
, restul sunt similare cu cele de la glMap1d;

Funcția glMapGrid2f care generează punctele suprafeței are prototipul:

```
void glMapGrid2f(  GLint un,  GLfloat u1,  GLfloat u2,
                   GLint vn,  GLfloat v1,  GLfloat v2
);
```

, unde un ,vn sunt numărul de diviziuni pe u şi pe v;
, u1,u2 – sunt limitele intervalului de generare pe u(adică de la ce coordonată se începe generarea punctelor pe u);
,v1, v2 - sunt limitele intervalului de generare pe v;

Funcția glEvalMesh2 generează efectiv suprafața. Are următorul prototip :

```
void glEvalMesh2(  GLenum mode,
                   GLint i1,  GLint i2,
                   GLint j1,  GLint j2
);
```

, unde mode este definit prin constantele:
GL_POINT – se desenează doar punctele;
GL_LINE – se va desena o suprafață din linii;
GL_FILL – se va desena o suprafață continuă;
, i1,i2 – intervalul de puncte definite în grid ce se vor afişa pe u ;
,j1,j2 – intervalul de puncte definite în grid ce se vor afişa pe v ;

Exercițiu 2. Generați o suprafață plină folosind o matrice de 3 x 3 puncte de control. Numărul de diviziuni va fi de 30 x 30. Se va selecta prin apăsarea barei de spațiu în mod ciclic modul de desenare al suprafeței din cele trei moduri existente.

4.14.4 Curbele şi suprafețele de tip NURBS

Acest tip de curbe este similar tipului Bezier, cu unele diferențe de calcul efectiv al curbelor şi de implementare OpenGL. Denumirea NURBS provine de la inițialele denumirii complete a acestui gen de curbe Non-Uniform Rational B-Spline.
Curbele NURBS sunt definite în biblioteca glu şi sunt apelate doar după ce au fost create. Asfel pentru a crea o structură NURBS se foloseşte secvența :

```
GLUnurbsObj *pNurb = NULL; //declaram structura
…
…
pNurb = gluNewNurbsRenderer();// creem structura
```

, la fel când nu se mai foloseşte se dealocă structura prin secvența :

```
if(pNurb) // daca este alocata o dealocam
```

gluDeleteNurbsRenderer(pNurb);

Pentru a configura o structură NURBS folosim funcția cu prototipul:

```
void gluNurbsProperty(GLUnurbsObj *nobj,
                      GLenum property,  GLfloat value
);
```

,unde nobj este structura NURBS,
, property este proprietatea ce se configurează, ea este definită prin constantele:
GLU_SAMPLING_TOLERANCE - specifică lungimea maximă în pixeli pentru eşantionarea curbei – implicit este 50 , folosită împreună cu GLU_PATH_LENGTH ;

GLU_DISPLAY_MODE – specifică modul cum va fi desenată suprafața sau curba - GLU_FILL – trasare completă, GLU_OUTLINE_POLYGON - trasare din linii, GLU_OUTLINE_PATCH - trasarea acelor linii specificate de utilizator;
GLU_CULLING – generează eliminarea suprafețelor din afara zonelor stabilite de afişare daca este setat pe GL_TRUE, implicit este setat cu GL_FALSE;
GLU_PARAMETRIC_TOLERANCE – specifică distanța maximă în pixeli, când este folosită abordarea parametrică, implicit este 0.5 ;
GLU_SAMPLING_METHOD – stabileşte metoda de eşantionare: GLU_PATH_LENGTH este setarea implicită,alta ar fi GLU_PARAMETRIC_ERROR, definită de GLU_PARAMETRIC_TOLERANCE, şi GLU_DOMAIN_DISTANCE definită prin numărul de puncte pe u şi pe v ;
GLU_U_STEP – specifică numărul de puncte pe direcția u, pe unitate, implicit este 100.0 ;
GLU_V_STEP specifică numărul de puncte pe direcția v, pe unitate, implicit este 100.0 ;
GLU_AUTO_LOAD_MATRIX - apelează automat prin internet serverul OpenGL pentru a genera matricele, de proiecție, afişare.

Exemplu :

```
gluNurbsProperty(pNurb, GLU_SAMPLING_TOLERANCE, 25.0f);

gluNurbsProperty(pNurb, GLU_DISPLAY_MODE, (GLfloat)GLU_FILL);
```

Pentru a genera o curbă NURBS se foloseşte funcția cu prototipul:

```
void gluNurbsCurve(
  GLUnurbsObj *nobj,
  GLint nknots,  GLfloat *knot,
  GLint stride,  GLfloat *ctlarray,
  GLint order,  GLenum type
);
```

,un nobj este strutura NURBS,
,nknots – este numărul de noduri,
,knot – este un vector de noduri,
,stride – este distanța dintre punctele de control din vector,
,ctlarray – este vectorul de generare al curbei NURBS,
,order – este ordinul curbei nurbs implicit 4,
,type – este tipul evaluatorului folosit definit mai sus – este setat la GL_MAP1_VERTEX3, cel mai frecvent.

Această funcție se încadrează între gluBeginCurve(nobj) … gluEndCurve(nobj).

Pentru a genera o suprafață NURBS se foloseşte o secvență de genul :

```
gluBeginSurface(pNurb);

// crează suprafața
gluNurbsSurface(pNurb,      // structura NURBS
  8, Knots,           // numărul de noduri şi vectorul de noduri pe s
  8, Knots,          // numărul de noduri şi vectorul de noduri pe t
  4 * 3,             // Distanta dintre punctele de control pe s
  3,               // Distanta dintre punctele de control pe t
  &ctrlPoints[0][0][0],    // vector cu punctele de control
  4, 4,                // ordinal curbei pe s şi pe t
  GL_MAP2_VERTEX_3);       // tipul evaluatorului folosit


gluEndSurface(pNurb);
```

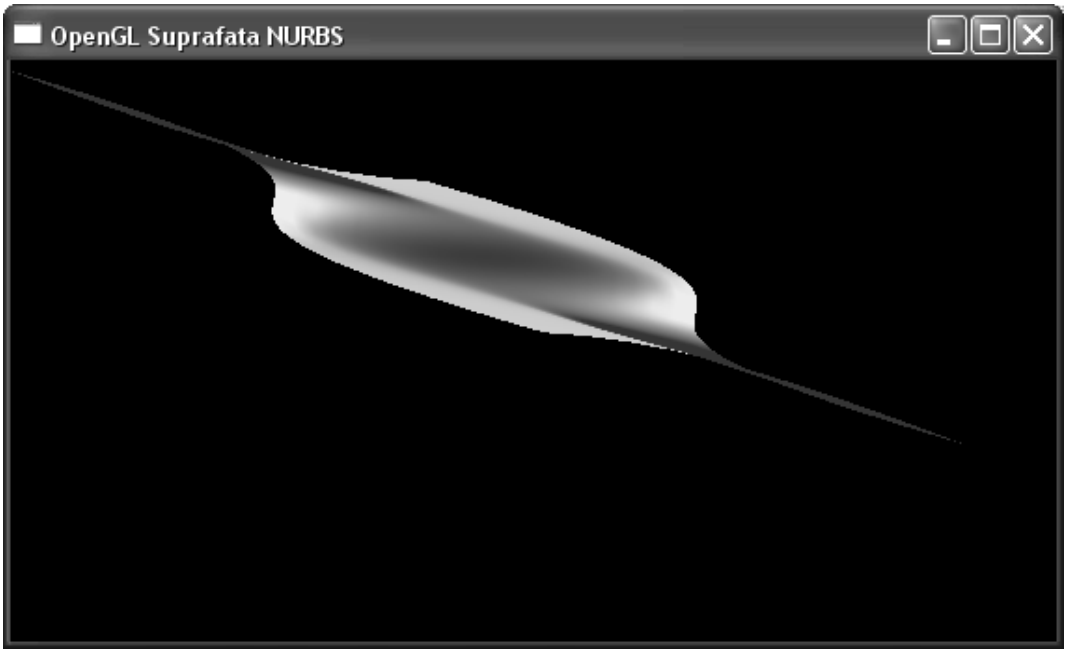
Această funcție are prototipul:

```
void gluNurbsSurface(  GLUnurbsObj *nobj,
          GLint sknot_count,  GLfloat *sknot,
          GLint tknot_count,  GLfloat *tknot,
          GLint s_stride,  GLint t_stride,
          GLfloat *ctlarray,
          GLint sorder,  GLint torder,
          GLenum type
);
```

Exercițiu 3. Realizați aceleaşi programe de la exercițiu 1 şi 2 dar folosind curbe NURBS.

Exercițiu 4. Realizați o suprafață NURBS ondulată texturată cu o imagine încărcată din fişier.

Exemple de aplicație NURBS :



5 BIBLIOGRAFIE

1. Dave Astle, Kevin Hawkins Beginning OpenGL Game Programming , Premier Press, Boston, 2004;
2. Paul Martz,OpenGL(R) Distilled (OpenGL), Addison Wesley Professional, 2006;
3. Richard S. Wright Jr., Michael R. Sweet ,OpenGL Super Bible, Waite Group Press. ,2007;
4. Rodica Baciu , Programarea Aplicațiilor Grafice 3D cu OpenGL, Editura Albastră, Cluj-Napoca 2005;
5. The Red Book of OpenGL
6. http://www.opengl.org
7. http://www.sgi.com/software/opengl
8. http://www.cs.utah.edu/~narobins/opengl.html
9. www.mesa3d.org/