EGC - Laborator 1

Dezvoltarea pornind direct de la sursã a unei aplicaţii grafice simple.
Primitive grafice OpenGL.

Informaţia din fişier este structuratã în 3 pãrţi. Prima parte se referã la tema propriu-zisã a laboratorului. Ea trebuie parcursã şi cerinţele temei trebuie implementate independent de cãtre fiecare student.

A doua parte cuprinde explicaţii asupra sursei, pentru cei doritori sã înţeleagã (sã aibã o idee) asupra API-ului (Application Programming Interface) OpenGL, precum şi asupra structurii (conceptuale) interne a motorului (serverului) OpenGL.

Partea i

1. Prezentarea unei aplicaţii simple OpenGL de desenare a primitivelor grafice
2. Sursa aplicaţiei (realizata în versiuni OpenGL > 3.3 – pipeline-ul funcţiilor programabile, cu shadere)
3. Indicaţii privind modul de construire a unui proiect Visual C++ pentru o aplicaţie OpenGL.
4. Cerinţele temei de laborator (ce trebuie sã implementeze studenţii)

Partea ii

5. Prezentarea/explicarea apelurilor OpenGL şi freeglut folosite în sursa de la punctul (2)
6. Prezentarea unor concepte legate de pipeline-ul grafic (adicã structura motorului (serverului) care executã comenzile OpenGL)

Partea iii

7. Prezentarea implementãrii aceleiaşi aplicaţii (de la punctul (1)) în versiunea OpenGL 1.1 (pipeline-ul funcţiilor fixe)

Noţiunile din partea a ii-a vor fi prezentate la curs, dar abia dupã ce se desfãşoarã prima şedinţã de laborator de aceea nu este obligatoriu (ci numai recomandat) sã fi citite. Lectura pãrţii a iii-a este complet opţionalã; ea conduce numai la o mai bunã înţelegere a modului cum a evoluat API-ul OpenGL.

Observaţii:

- **Textele de culoare roşie conţin informaţie care trebuie cititã obligatoriu şi (eventual) gândit asupra ei.**
- **Figurile nu sunt numerotate în ordine dar sunt referite corect în textul laboratorului.**

Partea I

1. Prezentarea aplicaţiei

Definiţie: Primitivele grafice sunt elemente de bazã constitutive, ale geometriei unui model sau ale unui desen pe calculator. De obicei, obiectele modelate sunt corpuri 3D şî prin urmare primitivele vor fi tot 3D. Este posibil de asemeni sã folosim numai primitive 2D (adicã obiectele modelate sau pãrţile constituente ale unui desen sã fie toate plasate într-un singur plan).
Diferite platforme[1] grafice (cum ar fi de exemplu OpenGL) suportã/implementeazã diferite seturi de primitive grafice (evident în cazul diferitelor platforme grafice, aceste seturi au o intersecţie nevidã). De exemplu, ultimele versiuni de OpenGL au ca primitive: GL_POINTS (colecţii de puncte), GL_LINES (segmente de dreaptã/vectori izolaţi), GL_LINE_STRIP (linii poligonale închise sau deschise), GL_TRIANGLES (triunghiuri izolate care pot fi reprezentate cu interiorul umplut sau numai prin laturile/vârfurile frontierei lor), GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN (diferite combinaţii de triunghiuri adiacente. Aceste ultime variante de primitive de tip triunghi sunt folosite pentru a reduce numãrul date (vârfuri) care se transmit vãtre serverul OpenGL, pentru a fi vizualizate triunghiurile – i.e. vârfurile care delimitează o laturã de adiacenţã a douã triunghiuri nu se transmit de douã ori), GL_PATCHES (se mai poate folosi şi GL_QUAD dar tot la triunghiuri se ajunge). Primitivele grafice fundamentale OpenGL (la care se reduc în cele din urmã toate desenele/modelele) sunt: GL_POINTS, GL_LINES şi GL_TRIANGLES.

Aplicaţia a cãrei sursã e prezentatã în laborator realizează afişarea a douã triunghiuri cu interiorul umplut (primitive grafice 2D). Pentru a simplifica (evita) la acest laborator abordarea unor aspecte legate de vizualizare, se va respecta urmãtoarea regulã: Toate primitivele desenate vor avea vârfurile cu coordonatele cuprinse într-un pãtrat unitate centrat în origine şi care corespunde domeniului plan cartezian $[-1,1]^2$ Acesta este un dreptunghi, având laturile paralele cu axele de coordonate şi se numeşte în argoul graficii pe calculator "AABB – Axis Aligned Bounding Box" iar în matematicã "(hiper)paralelipiped izotetic" (dreptunghiul este 2D, paralelipipedul este 3D iar un hiperparalelipiped este nD cu n>3).
Dacã vom folosi numai coordonate cuprinse între -1 şi 1, sistemul OpenGL va şti sã afişeze (într-un dreptunghi AABB de pe suprafaţa ecranului, specificat de utilizator/programator) figurile ale cãror vârfuri au astfel de coordonate, folosind pentru aceasta un singur apel numit glViewport() care va fi descris ulterior în cuprinsul acestui laborator. De fapt, deoarece informaţia graficã (cele douã triunghiuri) este afişatã în întreaga fereastrã de pe ecran care este alocatã aplicaţiei, nici mãcar nu este necesarã apelarea rutinei glViewport() ci numai crearea ferestrei (în S.O. Windows), folosind

---

Prin platformã se înţelege ansamblul alcãtuit din maşina hardware şi sistemul de operare (inclusiv stratul de software format din bibliotecile, dll-urile) pe care rulează o aplicaţie OpenGL.

apelul glutCreateWindow(). Aceasta corespunde specificării zonei de afişare curente (viewport) pe întreaga fereastră.
Observaţie: (Cei ce doresc amănunte suplimentare să citească în secţiunea (6) a acestei lucrări, despre pipeline-ul grafic, volumul canonic de vizualizare şi etapa de mapare pe ecran a pipeline-ului conceptual geometric. În acest laborator, nu vom folosi nici o transformare de modelare, vizualizare, proiecţie).

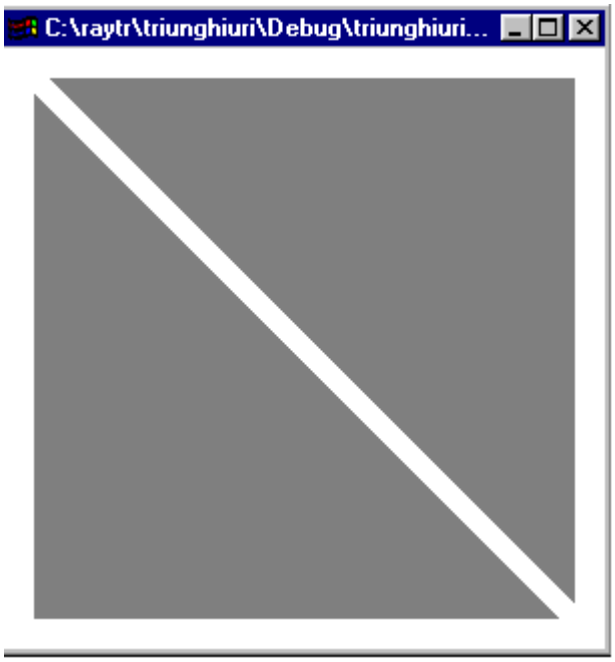

Figura 1 – Rezultatul afişat de aplicaţia noastră

Retur punct start fisier

2. Sursa aplicaţiei realizată într-o versiune OpenGL mai mare ca 3.3 (pipeline-ul funcţiilor programabile)

Iată sursa completă a aplicaţiei OpenGL care vizualizează două triunghiuri 2D T1, T2, cu vârfurile de coordonate (-0.9, -0.9), (0.85, -0.9), (-0.9, 0.85) respectiv (0.9, -0.85), (0.9, 0.9), (-0.85, 0.9).
Acestă aplicaţie apelează rutine OpenGL precum şi câteva rutine ale bibliotecii freeglut care ajută pe programator să gestioneze ferestrele sistemului de operare (Windows/Linux) pe care este implementată aplicaţia. Structura de principiu (abstractizată) a maşinii grafice care este comandată de aplicaţia OpenGL este descrisă (mai ales) în figura 1.6 (din secţiunea (6) a prezentei lucrări) împreună cu paragraful de text care o urmează.

Fişierul - sursat2d.cpp

```
using namespace std;

#include <iostream>
#include "vgl.h"

#define BUFFER_OFFSET(offset) ((void *) (offset))
```

```
enum VAO_IDs {Triangles, NumVAOs};
enum Buffer_IDs {ArrayBuffer, NumBuffers};
enum Attrib_IDs {vPosition = 0};

GLuint VAOs[NumVAOs];
GLuint Buffers[NumBuffers];

const GLuint NumVertices = 6;

GLuint LoadShadersSimplu(void);
// definitia functiei LoadShadersSimplu() este data dupa main()

void init(void)
//*************
{

   glGenVertexArrays(NumVAOs, VAOs);
   glBindVertexArray(VAOs[Triangles]);
   GLfloat vertices[NumVertices][2] = {
   // primul triunghi 3 varfuri  * 2 coordonate pe varf
      { -0.9f, -0.9f },
      { 0.85f, -0.9f },
      { -0.9f, 0.85f },
   // al doilea triunghi
      { 0.9f,  -0.85f},
      { 0.9f,  0.9f  },
      { -0.85f,0.9f  }
   };
   // rezerva id (nume) pentru un obiect buffer (OB)
   glGenBuffers(NumBuffers, Buffers);
   // creeaza un OB de tipul indicat
   glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
   glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),vertices,
                GL_STATIC_DRAW);

   // construieste un program shader cu doua shadere simple
   GLuint program = LoadShadersSimplu();
   glUseProgram(program);

   glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0,
                         BUFFER_OFFSET(0));
   glEnableVertexAttribArray(vPosition);
}

void display(void)
//****************
{   glClear(GL_COLOR_BUFFER_BIT);
    glBindVertexArray(VAOs[Triangles]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);

    glFlush();
}


void main(int argc, char** argv)
//*****************************
{     glutInitContextVersion(4, 3);
      glutInitContextProfile(GLUT_CORE_PROFILE);
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_RGBA);
```

```
	glutInitWindowPosition(100,100);
	glutInitWindowSize(512, 512);
	glutCreateWindow(argv[0]);

	if(glewInit() != GLEW_OK) {
		cerr << " Nu se poate initializa glew ... exit" << endl;
		exit(EXIT_FAILURE);
	}

	cout << "GLEW initiaizat : "<< glewGetString(GLEW_VERSION) <<
endl;

	init();

	glutDisplayFunc(display);

	glutMainLoop();
}

GLuint LoadShadersSimplu(void)
//***************************
// cel mai simplu mod de a crea un program shader care contine
// numai cele doua shadere obligatorii (vertex si fragment)
// folosite numai in aplicatia triunghiuri2Dsimplu
// Explicatii mai detaliate despre rutinele folosite in implementare
// la laborator sau in bibliografie. In sursa sunt numai
// comentarii scurte
{
   int compile_result=0, link_result=0;
   char info_log_message[80];
// triunghiuri2Dsimplu.cpp foloseste numai
// shaderele obligatorii intr-o aplicatie
// ("vertex shader" si "fragment shader") aici sursele shaderelor
// sunt date ca o secventa de siruri de caractere,
// fiecare sir de caractere al secventei fiind terminat cu '\0'
// (acesta e formatul implicit de stocare a sursei unui shader)

// functia "LoadShadersSimplu() intoarce numarul de identificare al
// program shaderului, care va fi folosit in aplicatia OpenGL

// sursa vertex shader-ului
   const char* vShader = {
       "#version 330 core\n"
       "layout (location = 0) in vec4 vPosition;\n"
       "void main()\n"
       "{ gl_Position = vPosition;\n"
       "}\n"
      };


// sursa fragment shader-ului
    const char* fShader = {
       "#version 330 core\n"
       "out vec4 fColor;\n"
       "void main()\n"
       "{ fColor = vec4(0.0, 0.0, 1.0, 1.0);\n"
       "}"
      };

// Constructia unui program shader se deruleaza in mai multe etape
```

```
// Etapa 1 -- creaza si compileaza toate obiectele shader folosite
// de aplicatie

// mai intai vertex shaderul
// se creaza obiectul vertex shader identificat prin id_vertex_shader
    GLuint id_vertex_shader = glCreateShader(GL_VERTEX_SHADER);
//  se asociaza textul sursa din vShader, obiectului shader
//  creat anterior
//  parametrul NULL indica formatul (implicit) folosit
//  in amplasarea liniilor sursa in tabloul vShader
    glShaderSource(id_vertex_shader, 0, &vShader, NULL);
//  compileaza sursa asociata obiectului shader identificat de
//  id_vertex_shader
    glCompileShader(id_vertex_shader);
//  se verifica daca s-a incheiat cu succes compilarea.
//  parametrul compile_result este boolean si ia valoarea GL_TRUE
//  in caz de succes al compilarii
    glGetShaderiv(id_vertex_shader, GL_COMPILE_STATUS,
                  &compile_result);
    if(compile_result == GL_FALSE) {
      // in caz de esec al compilarii se obtine mesajul de eroare
      // in sirul din ultimul parametru actual
       glGetShaderInfoLog(id_vertex_shader, 80, NULL,
                          info_log_message);
       cout<<"EROARE COMP. vertex shader"<<info_log_message<<  endl;
       return 0;
    }
// creaza si compileaza fragment shader-ul
    GLuint id_fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(id_fragment_shader, 0, &fShader, NULL);
    glCompileShader(id_fragment_shader);
    glGetShaderiv(id_fragment_shader, GL_COMPILE_STATUS,
                  &compile_result);
    if(compile_result == GL_FALSE) {
        glGetShaderInfoLog(id_fragment_shader, 80, NULL,
                           info_log_message);
        cout<<"ERR. COMP. fragment shader "<<info_log_message<<endl;
        return 0;
     }

// Etapa 2 -- link ale tuturor "programelor obiect" obtinute
// in Etapa 1 si constructia programului shader
// care va fi folosit in aplicatia OpenGL

// creaza un frame pentru un program shader
// si ii aloca un numar de identificare
   GLuint id_program_shader = glCreateProgram();
// se specifica care programe (obiect) rezultate prin
// compilare vor fi
// cuprinse in programul shader.
// (Prin atasarea acestor "programe obiect" la "programul shader")
    glAttachShader(id_program_shader, id_vertex_shader);
    glAttachShader(id_program_shader, id_fragment_shader);
//  se linkediteaza toate programele obiect atasate
//  programului shader
    glLinkProgram(id_program_shader);
//  se verifica daca link-ul s-a incheiat cu eroare
//  si in caz afirmativ
//  se obtine si apoi se afiseaza mesajul de eroare la link
    glGetProgramiv(id_program_shader,  GL_LINK_STATUS, &link_result);
    if(link_result == GL_FALSE) {
```

```
        // parametrul NULL inseamna ca nu se intoarce
        // lungimea mesajului
        // de eroare, acesta e un sir terminat cu '\0'
        glGetProgramInfoLog(id_program_shader, 80, NULL,
                            info_log_message);
        cout <<"ERR. LINK program shader "<<info_log_message<<endl;
        return 0;
    }
  //  daca link-ul s-a incheiat cu succes
  // se marcheaza pentru stergere
  //  rezultatele compilarilor. Acestea vor fi efectiv sterse
  //  in momentul stergerii (nefolosirii) programului shader rezultat
  //  in urma link-ului
    glDeleteShader(id_vertex_shader);
    glDeleteShader(id_fragment_shader);

    return id_program_shader;
}
```

Sursa conţinutã în fişierul “sursat2d.cpp” referã un header « vgl.h » care conţine includefile-urile legate de OpenGL pe care le referã textul sursã al al aplicaţiei. De exemplu, continutul acestui fisier ar putea fi :

Fişierul - vgl.h

```
#include "..\dependente\glew.h"
#include "..\dependente\glcorearb.h"
#include "..\dependente\freeglut.h"
//#include <GL\glew.h>
//#include <GL\glcorearb.h>
//#include <GL\freeglut.h>
// daca dorim sa lucram cu acest tip de directive #include
// o solutie comoda ar fi sa folosim comanda
// Project-->{project name}properties-->configuration properties
//-->C/C++-->general-->additional include directories
// care adauga ../dependente\GL In loc sa folosim catalogul
// include file-urilor sistem <VCpath>\include\GL
```

Retur la index inițial (punct start fișier)

3. Indicaţii privind modul de construire a unui proiect Visual C++ pentru o aplicaţie OpenGL.

Etapele construirii unui proiect/soluţie Visual C++

Toate indicaţiile date mai jos, sunt valabile pentru Visual studio 2012 (v110) i.e. exact versiunea referitã de Bjarne Stroustrup (creatorul C++) în (ultima versiune disponibilã a) cartea lui *The C++ programming language*. Dacã aveţi versiuni superioare e posibil sã se schimbe cãile din arborele de comenzi al meniului prin care se tratează anumite faze ale construirii proiectului. Subproblema (faza/etapa) care trebuie rezolvatã este conceptual aceeaşi şi rãmâne în sarcina dumneavoastrã sã identificaţi comenzile concrete pe care sã le daţi pentru a rezolva fiecare subproblemã.

O aplicaţie (sau mai multe) Visual C++ este caracterizatã printr-o soluţie şi un proiect.

O soluţie poate conţine un singur proiect sau mai multe. Când se doreşte lucrul cu o soluţie Visual C++, care a fost creatã în prealabil, cel mai comod procedeu este sã dai click (în Windows Explorer de exemplu) pe fişierul ".sln" care corespunde soluţiei. Când se doreşte lucrul asupra unui proiect se selectează dintr-un meniu respectivul proiect (activ) din mulţimea de proiecte care corespund unei anumite soluţii ar trebui urmatã calea de meniu:
View→Solution explorer→Startup project şi aici se completează o formă

În cazul nostru, soluţia va conţine un singur proiect.

După compilarea modulelor C++, urmează faza legării în care se primesc la intrare mai multe module obiect şi produce la ieşire un modul încărcabil. În funcţie de sistemul de operare pe care este implementatã platforma OpenGL, pot fi folosite diferite tipuri de legare. În cazul sistemului de operare Windows cu care lucrăm la acest laborator se foloseşte un aşa numita « legare dinamicã la momentul execuţiei ». În acest caz, referirile externe la module ale sistemului (i.e. de exemplu o aplicaţie conţine apeluri (call) ale unor funcţii care nu sunt definite în sursa aplicaţiei, astfel de funcţii externe sunt definite în module numite şi module ţintã/target/externe). În sistem există două feluri de biblioteci (de rutine/ module obiect). Biblioteci .lib utilizate în faza de legare a aplicaţiei noastre, care în locul apelurilor simple (CALL f()) ale unei funcţii externe conţin nişte mici programme numite *stub*. Biblioteci de legare dinamicã (.dll) ale cãror module sunt încărcate în memorie (dacã nu existau deja acolo pentru cã au fost eventual referite de o altã aplicaţie care rulează în sistem).abia la momentul execuţiei aplicaţiei, prin execuţia stubului corespunzător încorporat în programul obiect după ce a fost cãutat în biblioteca .lib. Mai amănunţit despre tipurile/variantele de legare veţi discuta la cursul de S.O.

Iatã principalele biblioteci referite de o aplicaţie OpenGL:

- (1) opengl32.lib, opengl32.dll - conţin printre altele (cod de) rutine ale cãror nume începe cu gl… (glClear() de exemplu).
- (2) freeglut.lib, freeglut.dll (OpenGL Utility Toolkit) - conţin în principal rutine al cãror nume începe cu glut… Ele realizează gestiunea sistemului de ferestreşi evenimente necesare aplicaţiei nosatre,
- (3) glew32/glew32s (OpenGL Extension Wrangler - vezi apelul glewinit() din sursa aplicaţiei “t2d”) –. Această ultimă rutinã realizează încărcarea corectã a

pointerilor către rutinele care suportă o anumită extensie OpenGL (versiune). Această bibliotecă are o variantă (glew32.lib) în care rutinele sunt încărcate dinamic, şi o variantă glew32s.lib în care funcţiile referite vor rămâne rezidente în memorie tot timpul (i.e. vor putea fi eventual evacuate numai împreună cu întregul proces) de îndată ce au fost încărcate). O astfel de bibliotecă se numeşte statică.

Observaţie: (Horea Cărămizaru) La laboratoarele următoare, framework-ul va folosi, începând din acest an o altă bibliotecă în locul bibliotecii freeglut. Această nouă bibliotecă, numită glfw, pune la dispoziţia programatorului un API simplu pentru gestunea ferestrelor, contextelor şi evenimentelor (ca şi freeglut).
Un exemplu de avantaj al folosirii glfw (în locul lui freeglut) este că în locul unui singur apel: glutMainLoop() (specific freeglut) care încapsulează întreaga buclă ((a)afişare cadru; (b)verificare recepţie eveniment; (c)tratare eveniment), biblioteca g;fw permite codificarea explicită sub forma unei bucle while() (în limbjul C) a acţiunilor (a), (b), (c) specificate mai sus. Prin urmare depanarea buclei principale a unui program dezvoltat folosind glfw se poate face mai uşor, pas cu pas (spre deosebire de cazul freeglut unde nu se pot pune breakpoint-uri în corpul funcţiei glutMainLoop()). În paragrafele de la începutul secţiunii (5) a prezentului document, în care se prezintă apelul glutMainLoop(), este pusă şi o sursă C cu apeluri glfw cu o funcţionalitate echivalentă cu a lui glutMainLoop().
Dăm şi aici site-urile de interes pentru această bibliotecă:
download general : http://www.glfw.org/
documentatie http://www.glfw.org/documentation.html
Pentru ca, la compilare să se cunoască exact numărul şi tipul fiecărui parametru al unei funcţii (care a fost referită în sursa aplicaţiei) dintr-o bibliotecă, vor trebui incluse în sursele aplicaţiei, fişierele header: “opengl.h”, “freeglut.h”,”glew.h” care terbuie puse în cataloage binecunoscute sistemului Visual C++. (Asta deoarece compilarea se face într-o singură parcurgere a fişierului sursă)

În proiect vor trebui adăugate, în afară de surse şî bibliotecile .lib mai sus menţionate, precum şi (prin consecinţă) fişierele .dll[2] corespunzătoare. Fişierele header corespunzătoare se găsesc de obicei în catalogul <VCpath>/include. (Prin notaţia <VCpath> am desemnat (rădăcina căii) calea de cataloage în care se află sistemul Visual C. Pe PC-ul meu această cale este:
C : \Program files (x86)/Microsoft Visual Sturio 11.0/VC
)

Dacă o sursă C va conţine linia:
#include <GL/nume.h>

fişierul nume.h va fi căutatde compilator, în catalogul <VCpath>\GL.

Fişierele ale căror nume sunt scrise în (1), vin (sunt livrate) sigur cu sistemul de operare Windows (şi se află în cataloagele S.O.-ului (în PC-ul meu este vorba de catalogul:

[2] dynamic link library – biblioteci cu legare dinamica, se vor studia la sisteme de opeare

C:\Windows\System32) sau cu sistemul Visual C++ (Visual Studio). Aceste cataloage ar putea în principiu diferi de la o instalare (versiune) la alta a sistemului de operare de aceea, pentru a asigura o independenţã mai mare de platformã a surselor aplicaţiei (portabilitate mai mare la nivel de sursã) am creat un catalog numit "dependente" pe care l-am pus pe acelaşi nivel al ierarhiei de cataloage cu catalogul "t2dproj" şi aici am pus tot ce referã proiectul (tehnica este învãţatã de la L.P. (care numea acest catalog "resurse" şi o folosea nu chiar în acest sens, dar nu e rafinatã de loc; n-am gândit asupra selectării/structurării informaţiei în catalogul "dependente").

Catalogul "..\dependente" va fi adãugat în listele care conţin numele cataloagelor în care compilerul/linkerul VisualC++ îşi cautã bibliotecile (pentru a rezolva referinţele nerezolvate la diverşi simboli) sau include file-urile. Acest lucru va fi fãcut dupã construirea proiectului şi înainte de primul compile/build.

- La versiunile mai noi de Visual C++ se selectează din meniul de comenzi : Project→t2dprojproperties→VC++ directories şi se modificã acolo diferitele intrări (lib path/ include path)
- La versiunile mai vechi de Visual C++ calea în arborele de comenzi (meniu) este : Tools→options→projects and solutions→VC++ directories (şi se modificã acolo diferite nume de cataloage)

Vom asocia şi soluţiei şi proiectului câte un catalog separat. Decizia pe care am luat-o eu a fost sã numesc catalogul soluţiei "t2dsol", catalogul proiectului" t2dproj" şi sã formez o relaţie de filiaţie (cale/ierarhie în arborele de cataloage) t2d →t2dsol→t2dproj

Catalogul t2dsol va conţine printre altele nişte fişiere (cu extensii de forma ".sln", ".sdf", catalogul "t2dproj" şi un catalog "dependenţe".

Catalogul "t2dproj" va conţine fişiere cu extensia ".vcxproj" şi toate sursele care intrã în componenţa proiectului (în cazul nostru fişierele "sursat2d.cpp" (programul sursã prezentat în textul laboratorului) şi "vgl.h".

Exerciţiu : desenaţi pe o hârtie (sub)arborele de cataloage având rãdãcina în catalogul t2d şi descris de propoziţiile precedente. În dreptul fiecãrui nod al subarborelui (care corespunde unui catalog) scrieţi numele fişierelor care vor fi incluse în acel catalog.

Această grupare pe fişiere/cataloage este fixatã de mine (care am o experienţã limitatã în folosirea mediului de programare Visual C++ cu versiuni mari, >11) m-am mai inspirat şi din sursele proiectelor (laboratoarelor) de anul trecut care au fost construite de asisenţi (L.P.), Fiecare din voi îşi poate structura altfel gruparea pe cataloage a soluţiilor/proiectelor/surselor. În laboratoarele 2, 3, … se va lucra cu o soluţie şi un proiect gata fãcute (vezi framework-ul Ivãnicã) si va trebui, pentru a rezolva cerinţele temei de laborator sã modificaţi sursa aplicaţiei şi sã compilaţi/linkeditaţi proiectul

Etapele construirii unui proiect care produce o astfel de structurare/grupare (în cataloage/fişiere) a (surselor ş.a.) aplicaţiei sunt:

1. *Se creazã un catalog rãdãcinã absolutã a aplicaţiei fie el "<path>\t2d"*

Se urmează calea de comenzi meniu : file→new→project→empty project
(în principiu s-ar putea alege/crea dintr-o mare varietate de tipuri de proiecte, noi alegem în acest caz empty project). Se completează în formă câmpurile :
Project name : t2dproj
Solution name : t2dsol
Location : <path>\t2d
Se setează tag-urile “create directory for solution” respectiv “add to source control”

2. *Se completează cu informaţie (fişiere) catalogul "dependente" (de exemplu fişiere precum glew32.lib sau freeglut.h)*

3. *Se adaugă la proiectul creat, sursele care îl compun* (ca să pot să le văd/modific/depanez cu uşurinţă) cu:

Project→add existing item (se consideră că înainte am copiat sursele (.cpp, .h) în catalogul t2dproj)

4. *Se modifică listele de cataloage în care Visual C++ caută fişierele include şi bibliotecile (libraries) necesare unei aplicaţii OpenGL.* (Se presupune că toate dll-urile sunt la locul lor, în cataloagele sistem)

Adăugarea în lista de cataloage în care se caută include file-uri am făcut-o cu:
Project→t2dproperties→configuration properties→VC++ Directories→Include Directories
unde am dat numele catalogului adăugat "..\dependente" (cu tagul inherit setat). Aici am pus headere ca freeglut.h etc.

Adăugarea în lista de cataloage în care se caută biblioteci de simboli am făcut-o cu
Project→t2dproperties→configuration properties→VC++ Directories→Library Directories
unde am dat numele aceluiaşi catalog (în care am pus glew32.lib etc.) (flagul inherit setat)

5. Se adaugă *la lista (numelor) bibliotecilor în care se caută simbolii nerezolvaţi*, (numele) *bibliotecile care conţin aceşti simboli* (care sunt indicaţi la faza de Build a aplicaţiei, prin mesaje de eroare de tipul « Error LNK 2019 … »). Trebuie avut în vedere şi pentru ce variantă Windows: Win32 sau Win64 se compilează, linkează proiectul. (Eu aş recomanda să folosiţi (încă) Win32 pentru a genera aplicaţii mai portabile). (În versiunile timpurii de Visual C++ platforma de lucru era Win32, în versiunile mai noi se poate comuta între platformele Win32/Win64 folosind calea de meniu pentru comutarea Win32/64 cu o comandă de genul : Project→t2dproperties→configuration properties→platform)

Această adăugare (a numelor de bibliotecă) se face folosind calea de meniu :
Project→t2dproperties→configuration properties→linker→input→additional dependencies
şi se introduc numele bibliotecilor. Eu am folosit această metodă pentru biblioteca glew32.lib al cărei nume l-am adăugat la şirul celorlalte nume de biblioteci. (Am setat flagul inherit)
Am pus (copiat) această bibliotecă (glew32.lib) în catalogul "dependente" şi am adăugat acest catalog la lista de cataloage în care Visual C++ caută bibliotecile.

Urmează bucla cu care sunteţi obişnuiţi de la gcc/Linux: compile/build/debug/make release version. Comenzile corespunzătoare le găsiţi în meniul Build.

Observaţie: La depanare, în cazul compilării/linkeditării, dacă nu sunteţi în stare să vă daţi seama singuri care este cauza erorii (uneori mi-a fost şi mie foarte greu) puteţi folosi următorul procedeu:

(a) Selectaţi toată linia care conţine mesajul de eroare (din fereastra build a mediului de programare Visual C++) şi
(b) copiaţi-o (copy-paste) în linia de căutare de pe google.com; veţi obţine indicaţii asupra cauzei posibile a erorii. Se poate folosi şi un site: *stack overflow* care îi asistă pe programatori în identificarea modului de rezolvare a unei erori.

retur punct start fişier

4. Cerinţele temei de laborator (ce trebuie să implementeze studenţii)

- Să se construiască, pornind de la sursa prezentată în textul laboratorului (2), un proiect/soluţie/aplicaţie care prin rulare să producă un rezultat asemănător cu cel din figura 1
- Să se modifice programul (sursa) a.î. în loc de triunghiuri cu interiorul colorat să se afişeze numai frontiera lor (câte 3 segmente (muchii/laturi) pentru fiecare triunghi). Se poate folosi apelul glPolygonMode() sau să se folosească primitive de tip GL_LINES în loc de GL_TRIANGLES cu aceleaşi coordonate ale vârfurilor.
- Să se traseze primitive grafice cu diferite atribute. De exemplu în loc să trasăm cele 3 laturi ale unui triunghi cu linie continuă, să le trasăm cu linie punctată/întreruptă/sau de diferite grosimi/culori. Ce atribute de afişare există în cazul primitivelor de tip triunghi sau punct? Pentru început ne vom limita la cazul primitivelor care au toate punctele care le compun de aceeaşi culoare. (Dacă nu ştiţi de unde să începeţi, porniţi de la căutarea apelurilor numite glLineStipple(), glPointSize(), glEnable(). şi exploraţi.)
- Încercaţi să repetaţi rezolvarea cerinţelor (buleţilor) de mai sus şi pentru programul OpenGL cu funcţii fixe.

Dacã, în cursul şedinţei de laborator aţi terminat rezolvarea cerinţei roşii (primul bullet) a temei în mai puţin de o orã şi jumãtate, puteţi continua fie cu rezolvarea buletului (buletilor) urmãtori sau sa începeţi sã citiţi partea a doua. În curs de o sãpãmânã puteţi sã faceţi (singuri) (în funcţie de timpul disponibil) câţi buleţi puteţi din aceste teme dar primul bulet este obligatoriu. Dacã nu reuşiţi sa-l terminaţi în timpul pe care vi l-aţi alocat în sãptãmâna de lucru (max 2 ore) veniţi în laboratorul urmãtor la şcoalã cu proiectul în stadiul în care aţi ajuns, pentru ca asistentul sã-şi dea seama în ce stadiu sunteţi cu el.
Oricum, în urmãtoarele laboratoare veţi lucra cu un proiect (de fapt framework) creat în prealabil de Gabi Ivãnicã şi veţi avea în principiu numai de completat/modificat sursele care vã vor fi livrate în componenţa acestui proiect.
Retur punct start

Partea ii

5. Prezentarea/explicarea apelurilor OpenGL şi freeglut folosite în sursa de la punctul (2)

Rutina main()

creeazã o fereastrã în sistemul de operare Windows folosind apeluri din biblioteca freeglut, intrã în bucla (afişare, testare recepţie eveniment şi tratare) apelând *glutMainLoop()*.

Notã : Iatã sursa (încãrcatã de pe net) referitã în secţiunea (3) a prezentului document, în observaţia (Horea Cãrãmizaru) care introduce biblioteca glfw.

```
#include <GLFW/glfw3.h>
int main(void)
{
    GLFWwindow* window;
    /* Initialize the library */
    if (!glfwInit())  // un fel de glutInit()
        return -1;
    /* Create a windowed mode window and its OpenGL context */
    // ceva mai mult decât glutCreateWindow()
    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }
    /* Make the window's context current */
    glfwMakeContextCurrent(window);
    /* Loop until the user closes the window */
```

```
    // Aceasta este codificarea C-glfw care ţine locul
    // lui glutMainLoop()
    while (!glfwWindowShouldClose(window))
    {
        /* Render here */
        glClear(GL_COLOR_BUFFER_BIT);

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

        /* Poll for and process events */
        // termenul poll vine de la polling = interogare
        glfwPollEvents();
    }
    glfwTerminate();
    return 0;
}
```

glutInit() iniţializează toate elementele necesare funcţionarii apelurilor glut. Parametrii argc şi argv care sunt primiţi din linia de comandă, pot specifica diferite moduri de funcţionare a glut. De obicei executabilul care foloseşte glut se lansează fără parametri în linia de comandă. În cazul nostru, se foloseşte un singur parametru în linia de comandă (argv[0]) care reprezintă titlul ferestrei (care va fi afişat la crearea ferestrei) create cu *glutCreateWindow()*.

glutInitDisplayMode() configurează tipul ferestrei pe care dorim să o afişam în aplicaţie. Frame-bufferul asociat ferestrei poate conţine mai multe (≤4) color buffere (fiecare color buffer (buffer de culoare) asociază fiecărui pixel al ferestrei, o informaţie de culoare (informaţie specificată prin apelul rutinei *glutInitDisplayMode()*). Prin acest ultim apel (i.e. *glutInitDispayMode()*) se poate specifica prezenţa unui z-buffer (GL_DEPTH), stencil buffer (buffer de marcaj), sau buffere auxiliare care intră în componenţa frame-bufferului asociat ferestrei. În cazul nostru parametrul de apel al rutinei *glutInitDisplayMode()* este GL_RGBA. Acesta specifică asocierea la fiecare pixel al interiorului ferestrei, a 4 octeţi care reprezintă respectiv intensitatea componentelor roşie, verde şi albastră a culorii pixelului precum şi factorul de amestec alfa (opacitate, A=alpha blending factor). Se foloseşte dublu color buffer.

glutInitWindowSize() specifică width-ul şi height-ul (lăţimea şi înălţimea exprimate în numere de pixeli) ferestrei care se va crea.
Există şi un apel *glutInitWindowPosition()* care în programul nostru nu e folosit şi care specifică poziţia unde va fi plasat colţul din stânga jos al ferestrei nou create, poziţie specificată în sistemul de coordonate asociat ecranului.

glutInitContextVersion() specifică versiunea de OpenGL folosită. Parametrii de apel indică versiunea 4.3 Acest apel este specific freeglut. Dacă lipseşste se va utiliza versiunea OpenGL 1.1.

glutInitContextProfile() specifică profilul “core”. În general un context specifică o structură de date internă OpenGL care stochează starea maşinii OpenGL şi operaţiile efectuate. În sursa shaderelor folosite în program (vezi tablourile vShader[], fShader[] din definiţia funcţiei LoadShaderssimplu()), prima linie este: “#version 330 core”.

Aceasta indicã ce versiune a limbajului GLSL se foloseşte. Numãrul 330 indicã sã se folosească versiunea GLSL asociatã cu versiunea OpenGL 3.3 specificatã cu *glutInitContextVersion()*. Tokenul "core" din sursa shader-ului (vezi şirurile vShader şi fShader) indicã profilul specificat în sursa aplicaţiei, prin apelul *glutInitContextProfile()*. Schema de denumire a versiunilor pe care am folosit-o în program este valabilã numai pentru versiuni ≥3.3. Fiecare shader trebuie sã înceapã cu o linie "#version" altfel se foloseşte implicit versiunea 1 (110 în convenţia de numire cea mai recentã).

Funcţia init() alocã şi iniţializează datele folosite în programul OpenGL.

Rutina *LoadShadersSimplu()* este cel mai simplu (şi nu neaparat elegant/general) mod de a obţine programul shader (care se executã pe GPU – graphic processing unit) asociat aplicaţiei t2d. E explicatã în comentariile din sursã.

Apeluri OpenGL (specifice versiunilor ≥3.3) folosite în programul prezentat:

*void glGenVertexArrays(GLuint n, GLuint *tvert);*

glGenVertexArrays() întoarce în tabloul tvert, n numere de identificare (nefolosite pânã la momentul apelului), ale unor obiecte (un numãr de identificare mai este numit "nume" sau "id" al unui obiect) de tip "vertex_array" (VAO). Tabloul de vertecşi nu e alocat ci numai se rezervã un nume pentru el. Mai sunt şi alte rutine ale cãror nume încep cu glGen… care au o funcţionalitate analoagã cu *glGenVertexArray()*.

void glBindVertexArray(GLuint array);

Dacã parametrul actual e diferit de 0 şi a fost întors de un apel *glGenVertexArray()*, se creează un nou VAO (vertex array object) care va avea numele indicat de valoarea parametrului actual. Obiectul VAO este creeat numai dacã nu a fost creat anterior apelului *glBindVertexArray()*. Dacã VAO era deja creat, el va deveni obiectul VAO curent. În OpenGL pot exista mai multe tipuri de obiecte precum VAO şi existã pentru fiecare apeluri *glBind*()* cu funcţionalitate analoagã. În programul nostru, funcţia *glBindVertexArray()* este apelatã de douã ori: prima datã în corpul rutinei *init()* şi a doua oarã în corpul rutinei *display()*.

Datele dintr-un VAO (care sunt asociate în general unei colecţii de vertecşi) sunt stocate într-unul sau mai multe obiecte buffer (OB).

*void glGenBuffers(GLsizei n, GLuint *buffers);*

alocã nume (id-uri) pentru n obiecte buffer (OB).

void glBindBuffer(GLenum tipOB, GLuint idbuffer);

dacã parametrul “idbuffer” este diferit de 0 şi *glBindBuffer()* este apelatã pentru prima oarã cu valoarea lui idbuffer, se va crea un nou OB cu id-ul asociat valorii actuale a parametrului “idbuffer”. Dacã idbuffer corespunde unui OB deja creat, acel OB va deveni activ. Parametrul tipOB are în programul nostru valoarea de apel GL_ARRAY_BUFFER dar pot exista multe alte tipuri de obiecte buffer (OB) folosite în programele OpenGL.

*void glBufferData(GLenum tipbuffer, GLsizeiptr size, const GLvoid *data, GLenum modutilizare);*

alocã size octeţi în memoria serverului OpenGL (pe GPU) pentru a stoca diferite informaţii (date sau indici). Primul parametru cu care este apelatã aceastã funcţie în programul nostru, are valoarea GL_ARRAY_BUFFER şi aratã cã bufferul va conţine atribute asociate vertecşilor dintr-o colecţie de vertecşi.

Parametrul data este un pointer cãtre o zonã de memorie din programul client (aplicaţia OpenGL) care va iniţializa obiectul buffer. Dacã data este NULL se face numai rezervare în server, fãrã iniţializare.

*void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalised, GLsizei stride, const GLvoid *pointer);*

asociazã variabilele dintr-un vertex shader cu datele care sunt stocate într-un OB. Se conectează astfel aplicaţia la un shader. În cazul nostru variabila “in” vPosition declaratã în vertex shader. Funcţia specificã în parametrul index, unde poate fi accesatã valoarea acestei variabile. În aplicaţie parametrul actual corespunzător are valoarea unei constante (cu valoarea 0) a tipului enumerare Attrib_IDs. În vertex shader valoarea ID-ului asociat variabilei vPosition se gãseşte în specificatorul “layout” din declaraţia variabilei vPosition a vertex shader-ului. Parametrul pointer este deplasamentul în bytes (octeţi) faţã de începutul OB al zonei unde este stocatã prima valoare a variabilei vPosition.(la noi este setat la valoarea 0 printr-un macro), size reprezintã numãrul de componente ale OB care trebuie actualizate pentru fiecare vertex (în cazul nostru variabila vPosition are 2 componente de tip GLfloat (i.e. coordonatele unui punct 2D din planul xOy) stocate în OB), (GLfloat reprezintã tipul fiecãrei componente din tabloul vertices stocat în OB), stride este deplasamentul în bytes între douã valori consecutive ale lui vPosition stocate în OB. La noi stride are valoarea 0 ceea ce înseamnã cã stocarea e compactã.

Un shader OpenGL reprezintã în general datele pe care le primeşte prin valori în virgulã mobilã (float). Aplicaţia poate furniza date cãtre OpenGL ca şi colecţii de valori de tip int (sau derivate), float (sau derivate). Se poate alege ca OpenGL sã converteascã automat valorile care nu sunt de tip reprezentat în virgulã mobilã în valori normalizate de tip float sau double. Acest lucru e specificat de parametrul normalised cu valoarea GL_TRUE. De exemplu, dacã tipul de date al unei valori trimise de aplicaţie este GLshort (reprezentare în virgulã fixã cu domeniul de valori [-32768, 32767]) OpenGL va mapa valorile din acest interval în valori reale din domeniul [-1.0, 1.0] dacã parametrul normalised are valoarea GL_TRUE. Domeniile de valori ale tipurilor întregi

fãrã semn sunt mapate normalizat în intervalul [0.0,1.0]. În programul nostru, nu se face mapare normalizatã.

void glEnableVertexAttribArray(Gluint index);

aratã cã tabloul de vertecşi asociat cu variabila specificatã prin parametrul index este activat (existã şi un apel disable corespunzãtor).

Funcţia display() afişeazã scena.

Se folosesc apelurile *glClear()* care indicã bufferele asociate frame-buffer-ului ferestrei curente (bufferul de culoare, z-bufferul, bufferul de marcaj – stencil -) care se curãţã. La noi se va umple (iniţializa) color bufferul cu negru (culoarea de iniţializare implicitã). Dacã vrem sa iniţializãm cu altã culoare apelãm în corpul funcţiei *init()* funcţia *glClearColor()*.

void glDrawArrays(GLenum mod, GLint first, Glsizei n);

Afişeazã primitivele geometrice specificate în parametrul mod, utilizând elementele din tabloul care iniţializeazã OB-ul curent (în cazul nostru tabloul vertices[]) începând de la elementul de indice first şi pânã la elementul de indice "first + n – 1". Valori posibile pentru parametrul mod (care e folosit în faza de "primitive assembly" (PA) a pipeline-ului grafic OpenGL) i.e. altfel spus primitivele grafice suportate de OpenGL sunt: GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_PATCHES (se mai poate folosi şi GL_QUAD dar tot la triunghiuri se ajunge)

void glFlush(void);

Comandã începerea execuţiei comenzilor de redare OpenGL şi garanteazã cã execuţia acestor comenzi se va termina în timp finit. Pentru a forţa şi trimiterea comenzilor care aşteaptã, cãtre serverul OpenGL se foloseşte *glFinish()*, care se terminã numai dupã terminarea tuturor comenzilor OpenGL în curs de execuţie. Se recomandã sã folosim *glFinish()* când punem la punct programul şi sã-l înlocuim cu *glFlush()* cand am obţinut versiunea finalã.
Este o practicã utilã de a da nume id-urilor obiectelor (VAO, VBO, IBO) folosite într-o aplicaîie, utilizând în acest scop tipuri enumerare (cum sunt în programul nostru VAO_IDs, Buffer_IDs sau Attrib_IDs) care ar putea avea mai multe constante, ultima constantã din tipul enumerare are valoarea egalã cu numãrul constantelor care o preced. (Dupã cum se ştie, în mod implicit, primei constante a unui tip enumerare îi corespunde valoarea 0.)

Consideraţii referitoare la aplicaţia OpenGL (pipeline-ul funcţiilor programabile)

Informaţia transmisã între aplicaţie şi shadere (sau între shadere) este conţinutã în variabilele declarate “in” sau “out” în sursa shaderelor. (existã şi alţi calificatori). Funcţia *glVertexAttribPointer()* specificã locul în fluxul de date FIFO (trimise de aplicaţie) în care este plasatã variabila “in” vPosition folositã în vertex shader. Direcţia transferului de date “in” sau “out” corespunde succesiunii etapelor de pipeline specifice OpenGL (vezi figura 1.6).
Această aplicaţie este un exemplu al unei categorii de aplicaţii distincte care pot fi scrise în OpenGL. Este vorba de aplicaţiile care bypasseaza (ocolesc) primele 3 etape funcţionale ale pipeline-ului conceptual geometric (model&view transform, lighting, projection) (vezi curs 1). Nu apare în sursa nici unui shader o matrice ca MVP sau matricele componente. Dacã se ocolesc cele 3 etape menţionate mai sus, aplicaţia OpenGL va reda numai conţinutul volumului canonic de vizualizare care la OpenGL este cubul de colţuri diametral opuse ((-1., -1., -1.), (1., 1., 1.)). Vertecşii folosiţi în aplicaţie (sursat2d.cpp) au ca informaţie asociatã numai informaţia de poziţie (câte două coordonate ale fiecãrui punct asociat fiecãrui vertex) care este în această aplicaţie o informaţie 2D (vezi conţinutul tabloului vertices[][]). Coordonatele sunt cuprinse în domeniul [-1, 1] x [-1, 1] adicã sunt coordonatele unor puncte situate în planul xOy şi interioare volumului de vizualizare canonic;.deci ambele triunghiuri 2D vor fi afişate.
Aceste coordonate sunt transformate în coordonate în spaţiul omogen 4D, de cãtre vertex shader. Variabila vPosition care ocupã spaţiu 2*GL_FLOAT (după cum se vede din parametrii 2 şi 3 ai apelului *glVertexAttribPointer()*), va fi convertitã de la vec2 la vec4 în definirea variabilei “in” din vertex shader care e de tip vec4 (vezi instrucţiunea "layout (location = 0) in vec4 vPosition;\n" din sursa vertex shaderului) această conversie adaugã (0, 1) la cele două coordonate preluate din sursa aplicaţiei (vezi linia “glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));” din sursa aplicaţiei).
Se transformã astfel un punct reprezentat în spaţiul Euclidian 2D, în acelaşi punct în spaţiul omogen 4D. Reprezentarea informaţiei poziţionale (punct) asociate unui vertex în spaţiul omogen este folositã intern de OpenGL pentru etapele pipeline-ului.
Coordonatele poziţiei (x, y) (se decarteazã coordonata z) ale vertexului curent, vor fi mapate pe ecran în etapa funcţionalã a a pipeline-ului conceptual geometric, pe care am numit-o plasare pe ecran (la cursul 1).
Un astfel de shader se numeşte “pass through shader” sau “copy-paste shader” pentru cã nu face nici un calcul pentru obţinerea lui gl_Position (variabilã internã a OpenGL folositã în shadere).
Ambele triunghiuri vor avea interiorul umplut cu aceeaşi culoare. Culoarea folositã pentru fiecare triunghi (RGBA - albastru opac) este specificatã în variabila “out” a fragment shader-ului (variabila fColor) care are valoare constantã pentru toate fragmentele generate.

Retur index iniţial (punct start fişier)

6. Pipeline-ul grafic conceptual

Procesele de redare (rendering) graficã se caracterizează printr-un numãr mare de dependenţe între operaţiile aplicaţiei (i.e. rezultatele fiecãrei operaţii depind de rezultatele operaţiei efectuate anterior). Metodologia cea mai raspânditã de a paraleliza (pe cât posibil) aplicaţiile din această categorie este folosirea pipeline-ului. Această metodã poate fi implementatã numai dacã un proces poate fi împãrţit în mai multe etape distincte. De exemplu, un procesor clasic von Neumann poate fi considerat ca având funcţionalitatea formatã din trei etape ("procesoare funcţionale"): prima aduce (fetch) instrucţiunea din memorie într-un registru al procesorului, a doua decodificã instrucţiunea şi a treia executã instrucţiunea decodificatã.

În general, dându-se o anumitã problemã spre a fi paralelizatã, ea poate fi divizatã într-un numãr (Nmax) finit de subprobleme ce corespund celor mai "mici" procese posibile (vezi conceptul de etapã propiu-zisã a pipeline-ului grafic, prezentat mai jos). O soluţie optimã din p.d.v. al timpului de execuţie ar putea sa fie compusã din cel mult Nmax subprobleme (etape în cazul unui pipeline), deoarece pot apare constrângeri suplimentare în rezolvarea problemei care sã necesite "sincronizarea" taskurilor care se executã paralel sau concurent. Se pot produce "gâtuiri" în desfãşurarea algoritmului paralel.

Pentru o funcţionare optimã a pipeline-ului, toate etapele trebuie sã se desfãşoare în acelaşi timp. În cazul unor viteze diferite (de realizare a fiecãrei etape) la ieşirea pipeline-ului se produce un articol la fiecare $\max(t_{P1}, t_{P2}, \ldots, t_{Pn})$ unitãţi de timp, unde prin t_{Pi} am notat timpul necesitat pentru prelucrarea unui articol în etapa a i-a a pipeline-ului.

O aplicaţie graficã de redare a unei scene poate fi consideratã (la nivelul unei diagrame bloc) o variantã a paradigmei de programare concurentã "producãtor-consumator" binecunoscutã celor ce au urmat cursul de Sisteme de Operare (sisteme concurente). Funcţionarea unui sistem grafic standard este descrisã printr-o abstractizare numitã *pipeline* grafic (numit uneori *bandã graficã*). În termenii unui sistem de operare, un *pipe* (o parte de comunicare a pipeline-ului) este un mecanism de comunicare şi sincronizare între procese (de calcul) numite şi etape ale pipeline-ului. Comunicarea se face FIFO iar sincronizarea înseamnã cã procesul receptor se blocheazã dacã pipe-ul este vid iar procesul producãtor se blocheazã dacã pipe-ul este plin.

Termenul de pipeline grafic se utilizează datoritã faptului cã transformarea datelor de la un model geometric/matematic la pixeli pe ecran este realizatã în mai multe etape (procese de calcul care comunicã date între ele). Calculele sunt realizate în secvenţe, iar rezultatele unei etape sunt transmise prin pipe-uri cãtre etapa urmãtoare a.î. fiecare etapã sã poatã începe imediat prelucrarea elementelor primite la intrare

Vom prezenta o structurã conceptualã (simplificatã) de pipeline grafic (urmând **[Moll 02]**), ce stã la baza sistemelor grafice, pentru a pune în evidenţã principalele etape ale transformãrii datelor grafice (de modelare) în pixeli. Această structurã conceptualã este

implementatã diferit – de la un sistem la altul - în sistemele grafice actuale. De exemplu, în variantele actuale ale platformelor OpenGL, o parte a etapelor conceptuale este structuratã diferit (în etape mai amãnunţite), unele din aceste etape sunt implementate în programul de aplicaţie iar altele direct în GPU (graphic processing unit – unitatea de prelucrare graficã a calculatorului).

O prima împarţire a pipeline-ului grafic este fãcutã în trei etape conceptuale: *aplicaţia*, *geometria* şi *rasterizarea*. Fiecare din aceste etape poate fi la rândul ei un pipeline (ceea ce înseamnã cã e divizatã în subetape). Trebuie sã distingem între *etapele conceptuale*, *etapele funcţionale* şi *etapele propriu-zise* de pipeline. O etapã funcţionalã are o anumitã sarcinã de îndeplinit, dar nu specificã modul concret în care sarcina este executatã în pipeline. O etapã propriu-zisã de pipeline este o etapã elementarã care poate fi executatã simultan cu celelalte etape propriu-zise de pipeline.

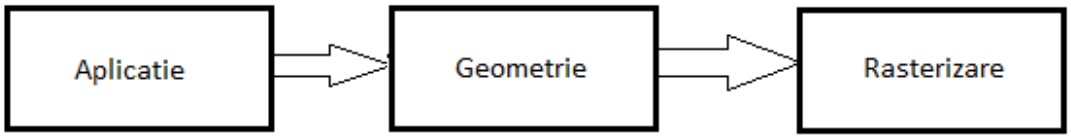

Figura 1.1 - Etapele conceptuale ale pipeline-ului grafic

Etapa conceptualã a aplicaţiei este (dupã cum îi aratã şi numele) întotdeauna implementatã în software-ul de aplicaţie. Aceastã etapã poate conţine algoritmi de detectare a coliziunilor între obiecte, algoritmi de accelerare sau cei de animaţie (vezi **[Hug 14]**). Urmãtoarea etapã conceptualã este cea a geometriei; ea foloseşte transformãri geometrice 3D/2D, proiecţii, calcule de iluminare, Aceastã etapã conceptualã calculeazã ce trebuie desenat, cum ar trebui desenat şi unde. O ultima etapã a pipeline-ului conceptual este etapa rasterizãrii i.e. desenarea pixelilor unei imagini, utilizând datele generate în etapele conceptuale precedente.

Etapa conceptualã a geometriei este împãrţitã în mai multe etape funcţionale dupã cum se aratã în Figura 1.2

Figura 1.2 – Etapele funcţionale ale pipeline-ului conceptual geometric

Depinzând de implementare, aceste etape funcţionale pot sau nu pot să fie echivalente cu etapele propriu-zise ale pipeline-ului. În unele cazuri mai multe etape funcţionale consecutive formează o singură etapă propriu-zisă de pipeline, în alte cazuri o etapă funcţională poate fi divizată în mai multe etape propriu-zise de pipeline.

Prima etapa funcţională ce compune etapa conceptuală a geometriei (transformarea de modelare/vizualizare)

Pe calea pe care o parcurge către coordonatele ecran, un model este transformat în mai multe spaţii sau sisteme de coordonate. La început un model este descris în propriul său *spaţiu model* (numit uneori *spaţiu local*) i.e. nu e transformat deloc. Fiecare model are asociată o transformare de modelare care îl poziţionează şi îl orientează într-un spaţiu global de coordonate (numit uneori *world coordinate system* sau *spaţiu utilizator* sau *sistemul de coordonate global*) asociat întregii scene care trebuie afişată. Este posibil ca mai multe transformări de modelare să fie asociate cu acelaşi model. Aceasta permite ca mai multe instanţe ale aceluiaşi model să aibă diferite locaţii/orientări/mărimi în aceeaşi scenă. Aceasta se face fără replicarea geometriei modelului. Transformarea de modelare transformă vârfurile (poziţiile vertecşilor) şi normalele (la vârf) ale unui model.

Spaţiul utilizator este unic şi după ce toate modelele au fost transformate cu respectivele transformări de modelare, toate modelele vor fi localizate în acelaşi spaţiu global (utilizator).

Scena este privită folosind o cameră de luat vederi. Aceasta este caracterizată printr-o locaţie în spaţiul utilizator şi o direcţie înspre care priveşte. Pentru a facilita proiecţia şi

decuparea, camera şi toate modelele sunt transformate folosind *transformarea de vizualizare (3D)* (*view transform*). Scopul transformării de vizualizare este (să fixeze ideile) să plaseze camera în originea sistemului de coordonate, să o facă sa privească de-a lungul axei Oz negative, cu axa Oy pointând în sus şi axa Ox pointând spre dreapta. Spaţiul de coordonate astfel specificat se numeşte *spaţiul coordonatelor de vizualizare*. Folosirea acestui spaţiu de coordonate uşurează calculele ulterioare. Vor fi redate numai modelele pe care camera de luat vederi (observatorul) le vede, i.e. modelele care sunt situate în interiorul unui *volum de vizualizare* asociat transformării de vizualizare. Transformările de modelare şi vizualizare sunt implementate folosind matrice 4x4 în coordonate omogene (vezi secţiunile 4.2.1 şi 4.3).

A doua etapă funcţională a etapei conceptuale a geometriei (iluminarea)

Pentru acele modele care sunt afectate de surse luminoase, se utilizează o ecuaţie a iluminării pentru a calcula culoarea în fiecare vârf (vertex) al modelului (vezi secţiunile 2.1 si 8.4). Culoarea în fiecare vârf al suprafeţei modelului este determinată utilizând locaţiile surselor luminoase şi proprietăţile lor, poziţiile vârfurilor şi normalele la vârf şi proprietăţile de material asociate fiecarui vârf (vezi ca exemplu ecuaţia iluminării (8-4)). Luând ca exemplu un obiect modelat prinr-o plasă de triunghiuri (vezi secţiunea 2.2) culorile vârfurilor unui triunghi sunt interpolate în fiecare din punctele interioare ale triunghiului la redarea pe ecran a respectivului triunghi. Această tehnică de interpolare este numită *shading Gouraud*[3]. (Termenul shading poate fi tradus prin "colorare realistă", în funcţie de gradul de întunecare/luminozitate al unei suprafeţe).

A treia etapă funcţională a etapei conceptuale a geometriei (proiecţia)

Înainte de a elimina coordonata z (a aduce "totul" într-un plan) este necesar să trecem din spaţiul de vizualizare (view space) în *spaţiul proiecţie* (projection space). După calculele de iluminare, sistemele grafice aplică o transformare proiectivă (vezi secţiunea 5.4.2) care transformă *volumul de vizualizare*, într-un cub unitate având două colţuri diametral opuse de coordonate (-1, -1, -1) respectiv (1, 1, 1). Cubul unitate se numeşte *volum canonic de vizualizare*. Aceasta transformare (view space → projection space) este implementată folosind altă matrice 4x4 (matricea de proiecţie).

Există două metode de proiecţie (şi două tipuri de matrice de proiecţie corespunzătoare) folosite în OpenGL şi în majoritatea platformelor grafice 3D: *proiecţia ortografică* (caz particular de *proiecţie paralelă*) şi *proiecţia perspectivă*. Volumul de vizualizare folosit de proiecţia ortografică este un paralelipiped dreptunghic; proiecţia paralelă transformă volumul de vizualizare în cubul unitate.

În cazul proiecţiei perspectivă, volumul de vizualizare, numit *frustum,* este un trunchi de piramidă dreaptă cu baza dreptunghiulară şi vârful piramidei în poziţia camerei de luat vederi. Trunchiul de piramidă este transformat în cubul unitate prin transformarea de proiecţie perspectivă. Ambele transformări (proiecţia ortografică/perspectivă) sunt implementate prin matrice 4x4 iar, după aplicarea acestor transformări, modelul se zice reprezentat în *sistemul coordonatelor normalizate de dispozitiv*. În funcţie de metoda de proiecţie folosită, poate fi necesar un pas suplimentar (omogenizarea coordonatelor vertecşilor) deoarece, în general, o transformare proiectivă nu transformă puncte din E^3 tot în puncte din E^3, (care au a patra coordonată din spaţiul omogen egală cu 1)

A patra etapă funcţională a etapei conceptuale a geometriei (decuparea-clipping)

În această etapă, numai primitivele (elemente de bază constitutive ale geometriei modelului cum ar fi liniile, triunghiurile, punctele) care sunt în întregime sau parţial în interiorul volumului de vizualizare, vor fi transmise mai departe spre etapa de rasterizare. Primitivele care sunt localizate în afara volumului de vizualizare nu vor fi deloc afişate. Primitivele care sunt parţial în afara volumului de vizualizare necesită decuparea faţă de frontiera volumului de vizualizare (vezi capitolul 6). Datorită aplicării prealabile (într-o etapa anterioară etapei de decupare) a transformării de proiecţie, primitivele vor fi decupate faţă de cubul unitate (volumul canonic de vizualizare).

A cincea etapă funcţională a etapei conceptuale a geometriei (maparea pe suprafaţa ecranului – screen mapping)

Coordonatele sunt încă tridimensionale ((x, y, z)) la intrarea în aceasta etapă. Coordonatele (x,y) ale punctelor fiecărei primitive sunt transformate pentru a forma coordonatele ecran. Coordonatele ecran, împreună cu coordonatele z vor fi folosite în etapa de rasterizare.

Etapa conceptualã a rasterizãrii

Această etapă primeşte la intrare (de la procesele anterioare ale benzii grafice) coordonatele (proiectate ale) vertecşilor, culorile lor şi coordonatele texturã si asociazã culori corecte pixelilor imaginii (de sintezã) finale. Spre deosebire de etapa geometriei care conţine operaţii per-poligon, faza rasterizării conţine operaţii per-pixel. Informaţia asociată fiecărui pixel este stocată în *bufferul de culoare* (o componentă a frame-bufferului), care este un tablou dreptunghiular care conţine pentru fiecare pixel al imaginii, codificarea culorii respectivului pixel. De obicei se foloseşte codificarea RGB a culorii fiecărui pixel (se reţin componentele (intensităţile) roşie, verde respectiv albastră ale culorii fiecărui pixel – vezi secţiunea 8.2). Pot exista două buffere de culoare asociate unei aceaşi imagini redate, ele se folosesc pentru afişarea imaginilor stereoscopice de sintezã.

Pentru a permite observatorului uman să nu vadă primitivele pe măsură ce sunt rasterizate şi trimise spre afişare pe ecran, se utilizează dubla bufferare. Redarea (modificarea biţilor codurilor de culoare a pixelilor) unei scene are loc într-un back-buffer (sau chiar două în cazul imaginilor stereoscopice) (numit şi buffer de fundal). Odată ce întreaga scenă a fost redată, conţinutul back-bufferului este swapat (interschimbat într-un singur ciclu de refresh al display-ului) cu conţinutul front-bufferului (buffer de prim-plan) care este bufferul afişat pe ecran.

Etapa rasterizării este folosită şi pentru asigurarea vizibilităţii (eliminarea suprafeţelor ascunse ale obiectelor). Asta înseamnă că după redarea întregii scene, bufferul culorilor va trebui sa conţină numai culorile primitivelor scenei care sunt vizibile când sunt observate din camera de luat vederi. În cazul majorităţii dispozitivelor hardware de afişare, vizibilitatea este implementată cu ajutorul *algoritmului z-buffer* (numit şi *depth-buffer* sau *buffer de adâncime*) (Catmull[4], 1974). Un z-buffer are aceleaşi dimensiuni ca şi bufferul de culoare şi, pentru fiecare pixel, stochează valoarea z (sau uneori distanţa) de la camera de luat vederi până la primitiva cea mai apropiată de cameră (primitiva redată, la momentul curent, în acel pixel). Când o primitivă (notată în continuare **P**) este redată într-un anumit pixel, va fi calculată valoarea z pe acea primitivă în acel pixel (notata $\mathbf{z_P}$) şi această valoare va fi comparată cu conţinutul z-bufferului în acel pixel (notat $\mathbf{z_D}$). Daca $\mathbf{z_P}<\mathbf{z_D}$ atunci primitiva **P** este mai apropiată de observator şi culoarea curentă a pixelului va fi actualizată la culoarea primitivei **P**.

O altă tehnică folosită în etapa rasterizării pentru a creşte nivelul de realism al unei imagini este aplicarea unei texturi (*texturarea)* pe suprafaţa unui obiect. Texturarea unui obiect poate fi asimilată intuitiv cu "lipirea" unei imagini pe suprafaţa obiectului.

Etapa conceptualã a rasterizării poate fi la rândul ei descompusă în mai multe etape funcţionale (precum etapa conceptualã a geometriei) dupa cum se aratã în figura 1.4.

[4] Edwin Catmull (nascut in 1945) specialist celebru, cu multe contribuţii teoretice şi practice de bazã în domeniul graficii pe calculator, a studiat fizicã şi calculatoare la Universitatea Utah (S.U.A.), actualmente director la studiourile Pixar este considerat autorul algoritmului z-buffer alãturi de germanul Wolfgang Strasser

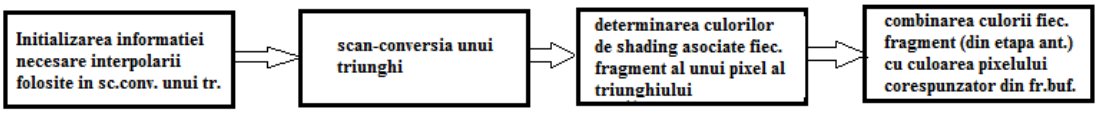

Figura 1.4 – Etapelele funcţionale ce corespund (etapei conceptuale a) rasterizării

- *Iniţializarea informaţiei* – realizează calculul tuturor datelor constante care sunt folosite în etapa următoare a pipeline-ului funcţional de rasterizare. De exemplu în cazul folosirii interpolării liniare, ciclul de generare a pixelilor din faza de scan-conversie (baleiere) poate exploata coerenţa imaginii şi folosi calcule incrementale (adunări cu incrementi constanţi, calculaţi prin hardware FF în etapa iniţializării), pentru generarea datelor de shading.
- *Baleierea (scan-conversia) primitivelor* (triunghiurilor) – în aceasta etapă se prelucrează, pentru fiecare pixel, toate eşantioanele interioare pixelului acoperite de primitiva curentă şi, pentru fiecare astfel de eşantion, se generează câte un fragment ce corespunde porţiunii din pixel care este acoperită de primitiva curentă. Se generează (prin interpolare) pentru fiecare fragment, date asociate fragmentului cum ar fi adâncimea sau coordonatele textură. Operaţiile acestei etape funcţionale sunt implementate de obicei direct prin hardware (FF).
- *Determinarea culorilor de shading* corespunde generării, pentru fiecare pixel care acoperă (parţial sau total) interiorul unei primitive (de exemplu triunghi), a culorilor fragmentelor asociate respectivului pixel (i.e. a fragmentelor ce rezultă din acoperirea pixelului de către primitivele redate – vezi figura 1.7). Această etapă funcţională este implementată în platformele grafice actuale în shaderele programabile care rulează pe GPU (mai precis în fragment shader). (vezi sectiunea 1.3)
- *Combinarea (amestecul) culorilor* fragmentelor asociate unui pixel este etapa în care se determină culoarea asociată întregului pixel (i.e. codul de culoare stocat în color buffer). În etapa de combinare, se folosesc adâncimile fiecărui fragment, i.e. a primitivei asociate fragmentului, (vezi algoritmul z-buffer), şi se fac eventuale teste suplimentare cum ar fi testul stencil (de marcaj). Această etapă mai este numită şi rezolvarea pixelului.

1.3 Evoluţia unei platforme grafice 3D (OpenGL)

OpenGL este platforma grafică pe care se desfaşoară laboratorul la acest curs.

Hardware-ul grafic şi API-urile IM au evoluat din 1990 până acum. Noi caracteristici ale hardware-ului au cerut îmbunătăţiri ale API-ului pentru a accesa aceste noi caracteristici. Identificarea de către utilizatori a limitărilor provocate de restricţiile ce apăreau în apelurile interfeţei IM a determinat apariţia unei reacţii din partea

utilizatorilor, care a condus la modificări in hardware-ul grafic efectuate de proiectanţii acestuia. Evoluţia bazatã pe aceste douã elemente prezentate anterior, a platformelor IM, a parcurs urmatoarele etape principale:

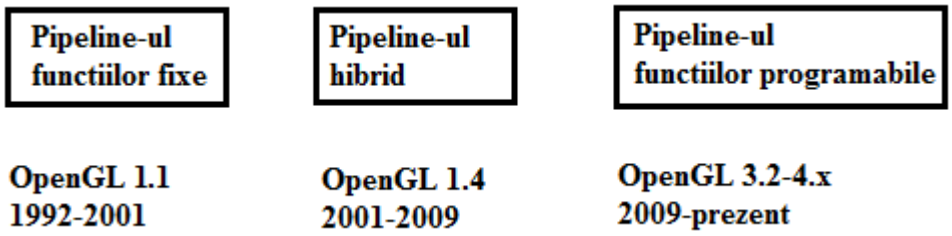

Figura 1.5 – Etapele evoluţiei OpenGL

- *Pipelineul funcţiilor fixe*. Modulele pipeline-ului erau configurabile folosind parametri ai funcţiilor API (Application Program Interface) OpenGL. Algoritmii acestor module erau implementaţi în hardware şi nu puteau fi inlocuiţi sau configuraţi (vezi un exemplu în secţiunea A1.2). Platformele IM-FF (fixed functions) furnizau accesul programatorului la caracteristicile unor funcţii (fixe) şi, dacã apãreau noi caracteristici ale hardware-ului, API-urile IM erau extinse pentru a accesa aceste noi caracteristici.
- *Extensibilitate prin shadere* (*pipelineul hibrid*). Shaderele au fost introduse în 1984 într-un articol al lui Rob Cook (de la Pixar Renderman software). Popularitatea lor a crescut începând cu anul 2000 deoarece programatorii cereau posibilitãţi de control mai mari asupra procesului de redare graficã şi acces mai fin la facilitãţile oferite de GPU. Shaderele rulează pe GPU şi descriu prelucrarea datelor în pipelineul grafic.Termenul *shader* se referã la orice modul programabil care poate fi instalat dinamic în pipelineul grafic. La început programarea shaderelor se fãcea în limbaj de asamblare; începând din 2003-2004 shaderele au fost dezvoltate în limbaje similare limbajului C de exemplu GLSL (limbaj specializat pentru scrierea shaderelor, proiectat de “OpenGL Architecture Review Board” – ARB şi introdus de OpenGL 2.0). Stratul software IM a tratat iniţial shaderele ca pe o adãugire la pipeline-ul funcţiilor fixe şi, câţiva ani, caracteristicile fixe şi programabile au coexistat în aplicaţii. Acestea utilizau de la caz la caz pipelineul. FF sau instalau shadere suplimentare daca era necesar. Despre shadere se vorbeşte şi în secţiunea 8.3 precum şi în anexa 2.
- *Pipelineul funcţiilor programabile*. Folosirea funcţiilor fixe a fost cerutã din ce în ce mai puţin pe mãsurã ce a evoluat cererea pentru aplicaţiile de generare de desene animate şi jocuri video cu imagini de calitate. Astfel a apărut o adevaratã cursã între programatorii de aplicaţie şi proiectanţii de GPU-uri pentru furnizarea de efecte vizuale cât mai spectaculoase. Începând din 2010 (OpenGL 3.2) funcţiile fixe OpenGL au fost mutate în “OpenGL Compatibility Profile” (vezi sursa din anexa 1.1) care nu a mai fost considerat ca fãcând parte din partea principalã a API-ului OpenGL. API-ul OpenGL are mai puţine puncte de intrare grafice. GPU-urile devin

unitãţi de prelucrare cu un grad ridicat de paralelism şi s-a ajuns ca ray-tracingul sã se facã în timp real (vezi exemple în **[Pet 15]**).

1.4 Structura pipeline-ului grafic la un sistem cu funcţii programabile (OpenGL 4.3)

OpenGL este implementat ca un sistem cu *arhitecturã client-server*, unde aplicaţia este consideratã client şi implementarea OpenGL furnizatã de producãtorul unui hardware grafic (cum ar fi NVIDIA) este serverul. În unele implementãri OpenGL (cea de pe X-windows) clientul şi serverul se executã pe maşini diferite, conectate prin reţea.

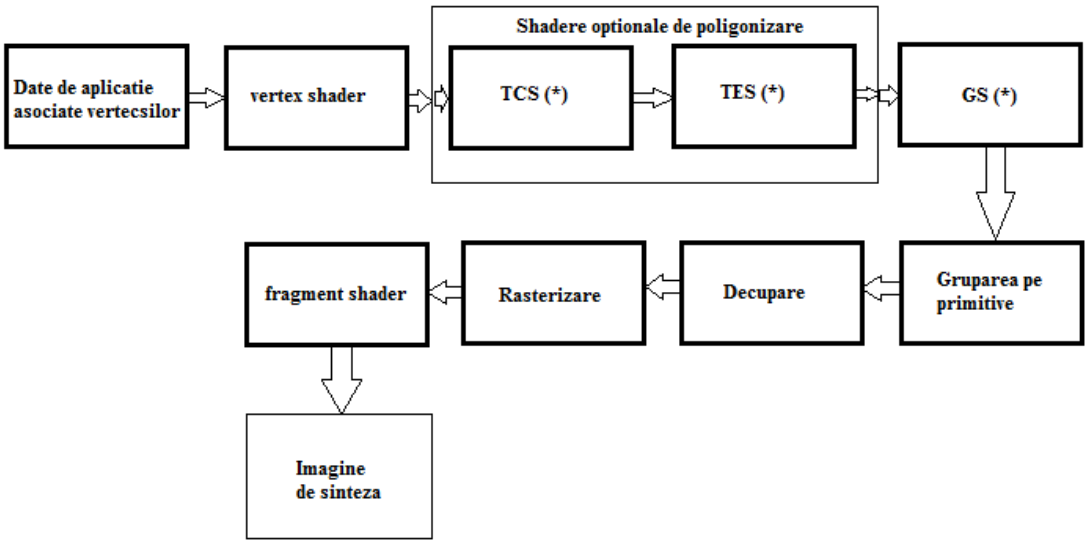

Figura 1.6 – Etapele principale ale pipeline-ului cu funcţii programabile, caracteristic platformelor OpenGL actuale

Fiecare porţiune a pipeline-ului grafic poate cuprinde mai multe procese de calcul care toate sunt considerate îndeplinite în mod secvenţial. Implementarea exactã a acestor task-uri poate varia, în funcţie de arhitectura maşinii de calcul şi a GPU-ului. Programatorii aplicaţiilor grafice au aceastã abstractizare în minte (caracterul secvenţial) când scriu aplicaţiile. Majoritatea API-urilor care sunt utilizate pentru a controla aplicaţiile grafice, furnizeazã acest mod de gândire programatorului de

aplicaţii grafice. În practicã ordinea de execuţie a task-urilor poate fi modificatã dar unui sistem grafic i se cere sã producã rezultate ca şi cum datele ar fi prelucrate în ordinea descrisã în specificaţia sistemului. Deci pipelineul grafic este o abstracţie, un mod de a gândi despre calculele care sunt fãcute, fãrã sã se ia în considerare implementarea concretã. Pipeline-ul grafic ne face sã ne creem o reprezentare asupra rezultatelor prelucrãrii grafice.

Iatã explicarea pe scurt a etapelor pipeline-ului grafic prezentate în figura 1.6.

Pregãtirea trimiterii datelor la server. OpenGL cere ca datele sã fie stocate în obiecte buffer (OB) (i.e. zone de memorie gestionate de serverul OpenGL), popularea acestor buffere cu date se poate face cu apelul *glBufferData()*. (vezi exemplul din anexa 1.1)

Trimiterea datelor. Dupa iniţializarea bufferelor, se poate cere redarea primitivelor grafice, prin apelul comenzilor de desenare (de exemplu *glDrawBuffers()*) OpenGL. Desenarea OpenGL înseamna transferul datelor asociate vertecşilor (locaţii de puncte, vectori normalã, coordonate texturã etc.) cãtre serverul OpenGL.

Vertex shader (etapã obligatorie): Fiecare vertex trimis serverului OpenGL de o comandã de desenare va fi prelucrat de cãtre un vertex shader. Pot fi "vertex shadere" simple (pass-through shadders) care copiazã datele cãtre alte shadere sau vertex shadere complicate care determinã poziţia vertexului pe ecran aplicând transformari grafice, determinã culoarea vertexului utilizând calcule de iluminare.

Dupã prelucrarea datelor grafice de cãtre un vertex shader, urmeazã (opţional) prelucrarea acestor date de *shadere de poligonizare* (care sunt în numãr de douã şi sunt numite pe englezã *tesselation shadders*) dacã acestea sunt invocate. Un shader de poligonizare are la intrare geometria unor petice de suprafaţã (vezi secţiunea 3.2) şi le transformã în primitive geometrice suportate de sistemul OpenGL (de exemplu triunghiuri).

În pipeline-ul prezentat în figura 1.6, sunt evidenţiate douã tipuri de shadere de poligonizare: *Tesselation control shadder* (numit si *Hull shadder*, notat *TCS*) este un shader opţional, el ia primitivele calculate în urma unei prime etape de *asamblare a primitivelor* (*primitive assembly – PA*) şi calculeazã factorii de poligonizare care sunt utilizaţi de cãtre un *program de poligonizare* (*tesselator*, notat T) care creazã noi vertecşi şi topologie. O etapã PA grupeazã vertecşii asociaţi unei primitive geometrice, pentru a pregãti informaţia necesarã etapelor ulterioare ale pipeline-ului grafic: decuparea şi rasterizarea.

Aceşti noi vertecşi sunt transformaţi de *Tesselation evaluation shadder* (un shader opţional, numit şi *Domain shadder*, notat (in figura 1.6 *TES*) iar vertecşii transformaţi şi topologia creatã de poligonizator sunt asamblaţi (asociaţi/grupaţi pe primitive grafice) într-o a doua etapã de asamblare a primitivelor (PA).

Datele de ieşire din ultima faza PA constituie intrarea în *geometry shadder* (un shader opţional) care prelucrează fiecare primitivă şi poate chiar să modifice topologia primitivei.

Clipping – în principiu, vertecşii din afara volumului canonic de vizualizare sunt eliminaţi.

Rasterizare – primitivele actualizate de etapa de decupare (clipping) sunt trimise modulului de rasterizare care genereaza fragmentele ce compun fiecare primitiva.

Fragment shaderul (shader obligatoriu) este o etapă finală, în care există control programabil asupra culorii fiecărei locaţii de pe ecran. Fragment shaderul determină culoarea fiecărui fragment şi eventual adâncimea lui. Fragment shaderul poate mapa texturi pentru a îmbunătăţi culorile furnizate de vertex shader.

Vertex shaderul, (eventual TCS, TES sau geometry shaderul daca acestea există) determină unde este situată pe ecran o primitivă grafică, fragment shaderul determină ce culoare are fiecare fragment al primitivei.

Acest model de structurare a shaderelor din pipeline-ul grafic OpenGL este cunoscut sub numele de *modelul cu 5 shadere* (un model cu 5 etape programabile în pipeline).

Mai exista şi un alt tip de shader (al 6-lea) numit *compute shader*, care nu face parte din pipeline-ul grafic dar poate fi folosit pentru a efectua calcule diverse pe GPU (vezi explicaţiile referitoare la noţiunile SIMD, GPGPU din secţiunea 8.3).

O etapă finală a pipeline-ului grafic OpenGL este numită "*operatii cu fragmente* (*per fragment operations*)". În fragment shader, se face şi *testul de adâncime* (z-buffer) (vezi capitolul 10) (şi eventual un alt test numit *stencil test – testul de marcaj*). Dacă un fragment trece toate testele activate, ar putea fi scris direct în frame buffer actualizând culoarea pixelului corespunzator (vezi concept*ul de acoperire* (*coverage*) explicat mai jos si (mai detaliat) in capitolul 11) şi eventual adâncimea pixelului. Etapa "per fragment operations" face eventuale operaţii (de combinare) între culori de pixeli dacă amestecul culorilor (*color blending*) este activat.

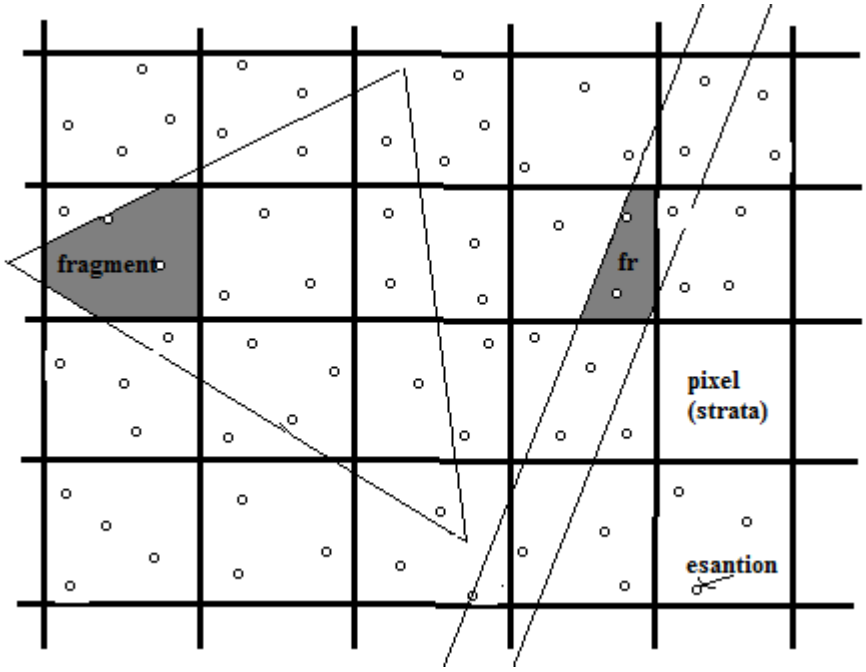

Figura 1.7 - Ilustrarea terminologiei legate de rasterizare

Explicarea unor termeni folosiţi anterior: Un pixel este o regiune dreptunghiulară a imaginii asociate frame-bufferului (vezi si notiunea de strata prezentată în secţiunea 7.2.2). O primitivă este forma geometrică ce trebuie afişată. De exemplu, primitivele grafice suportate de ultimele versiuni OpenGL sunt puncte, linii (şi lanţuri de linii) şi triunghiuri (şi combinaţii de triunghiuri). Un fragment este porţiunea unei primitive care este inclusă într-un pixel. Un eşantion (sample) este un punct într-un pixel. Acoperirea unui pixel de un fragment (coverage) şi shadingul (determinarea culorii unui fragment) sunt calculate într-unul sau mai multe eşantioane (posibil nu în aceleaşi eşantioane). Culoarea unui pixel e determinată printr-o operaţie de rezolvare a pixelului care filtrează eşantioane apropiate (de exemplu mediază valorile de culoare din eşantioanele din interiorul unui pixel).

Retur index iniţial (punct start fişier)

Partea III

7. Prezentarea implementării aceleiaşi aplicaţii (de la punctul (1)) în versiunea OpenGL 1.1 (pipeline-ul funcţiilor fixe)

Această secţiune se poate citi la sfârşit, (împreună cu secţiunea 1.3 a cursului inclus în textul laboratorului) când vă găsiţi timp, până la sfârşitul semestrului (eventual în timpul vacanţei de iarnă dacă aveţi chef).

```
#include "vgl.h"

void Init(void)
{
  // culoarea de fond alb
  glClearColor(1.0, 1.0, 1.0, 0.0);
  // triunghiurile vor fi afisate cu interiorul umplut cu aceeasi culoare
  // flat shading
  // shadingul nu mai e programabil in shadere
  // ci este specificat prin apeluri inghetate
  glShadeModel(GL_FLAT);
  // matricea transformarii de proiectie corespunde VCO
  // i.e. volumul canonic ortografic
  // in aceasta aplicatie apelul glOrtho() poate fi ignorat
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  // glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  // matricea transformarii de modelare-vizualizare
  // lasa vertecsii neschimbati
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

void DesenTriunghiuri(void)
{ glClear(GL_COLOR_BUFFER_BIT);
  // culoarea triunghiurilor este o nuanta de gri (pentru editura)
  // in fragment shaderul din A1.1 triunghiurile au culoarea albastra
  glColor3f(0.5f, 0.5f, 0.5f);
  // lipsesc apelurile de comunicare aplicatie GPU
  // lipsesc shaderele
  // se specifica direct vertecsii care sunt trimisi
  // in pipeline-ul functiilor fixe
  glBegin(GL_TRIANGLES);
    // primul triunghi
    glVertex2f(-0.9, -0.9);
    glVertex2f(0.85, -0.9);
    glVertex2f(-0.9, 0.85);
    // al doilea triunghi
    glVertex2f(0.9, -0.85);
    glVertex2f(0.9, 0.9);
    glVertex2f(-0.85, 0.9);
  glEnd();
  glFlush();
}


void main(int argc, char *argv[])
```

```
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA);
  glutInitWindowSize(300, 300);
  glutInitWindowPosition(100, 100);
  /* numele ferestrei corespunde numelui executabilului */
  glutCreateWindow(argv[0]);
  Init();
  glutDisplayFunc(DesenTriunghiuri);
  glutMainLoop();
}
```

Un apel tipic OpenGL 1.1 este definirea modului de shading utilizat. În pipeline-ul funcţiilor fixe este posibil să se utilizeze un număr limitat de metode de shading. Ele sunt specificate prin apelul:

```
void glShadeModel(GLenum m);
```

În programul din anexa 1.2 se foloseşte shadingul uniform/plat (parametrul m are valoarea GL_FLAT) asta înseamnă că toate punctele interioare unui triunghi vor fi colorate cu culoarea primului vârf al triunghiului (în cazul nostru toate vârfurile triunghiului au aceeaşi culoare, un gri opac). Acest tip de shading este implementat în pipeline-ul funcţiilor programabile în fragment shader-ul programului din anexa 1.1.
Dacă parametrul m ia valoarea GL_SMOOTH, metoda de shading folosită (în OpenGL 1.1) este shadingul Gouraud (shading cu interpolarea culorii la vărfurile triunghiului). Secvenţa prin care se asociază culori diferite la fiecare din vârfurile unui triunghi (în OpenGL 1.1) ar fi:

```
  glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 1.0f);
    glVertex2f(-0.9, -0.9);
    glColor3f(0.0f, 1.0f, 1.0f);
    glVertex2f(0.85, -0.9);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex2f(-0.9, 0.85);
  glEnd();
```

Folosind culorile la vârf specificate în secvenţa de mai sus, aspectul triunghiului (GL_SMOOTH) va fi asemănător (dar nu identic) cu gamutul unui monitor, pe diagrama de cromaticitate CIE.

Culorile asociate vârfurilor se calculează în aplicaţiile OpenGL 1.1 prin secvenţe de program de aplicaţie C, asemănătoare celei din corpul rutinei *shade()* (vezi 7.2.4). În pipeline-ul funcţiilor programabile, calculul culorii la vârf este implementat în vertex shader (vezi A2.1) printr-o secvenţă GLSL iar culoarea interpolată este primită automat (pentru fiecare fragment) de către fragment shader. Folosirea shaderelor permite scrierea unei varietăţi de metode de shading folosite în programele OpenGL actuale, nu numai două metode GL_FLAT şi GL_SMOOTH ca în aplicaţiile OpenGL1.1.

Matricele transformãrilor folosite în pipeline-ul grafic OpenGL cele a cãror folosire am ocolit-o când am implementat aplicaţia OpenGL cu funcţii programabile (shadere) după cum am menţionat în secţiunea iniţialã (numitã “prezentarea aplicaţiei”) a laboratorului, corespund în versiunea cu funcţii fixe a implementãrii OpenGL apelurilor *glMatrixMode()*, şi *glOrtho()*; aceastã matrice este utilizatã direct în pipeline-ul grafic FF.

Retur index iniţial (punct start fişier)

Referinţe

[Hug 14] John Hughes, Andries van Dam, Morgan McGuire, David Sklar, James Foley, Steven Feiner, Kurt Akeley, *Computer Graphics Principles and Practice*, 3-rd edition, Addison Wesley, 2014, copyright Pearson Education.

[Moll 02] Tomas Akenine Moller, Eric Haines, *Real time rendering*, 2-nd edition, A.K.Peters, 2002.

[Pet 15] Lucian Petrescu, *Redarea de scene masive 3D în timp real*, Tezã de doctorat susţinutã la Universitatea Politehnica Bucureşti, 25 septembrie 2015.