10/27/2013

FACULTATEA DE AUTOMATICA SI CALCULATOARE

ELEMENTE DE GRAFICA PE CALCULATOR

Laborator 4

OpenGL – introducere

OpenGL

OpenGL este un standard/API pe care il putem folosi pentru a desena. Este aproape echivalent cu Direct3D cu care de-a lungul timpul a avut o relatie de influenta reciproca. O scurta istorie o puteti gasi pe pagina https://en.wikipedia.org/wiki/OpenGL. Versiunea curenta a acestui standard este 4.4. Pentru cursul de EGC vom folosi standardul 3.0/3.3, care este in acelasi timp si versiunea actuala pentru varianta pentru mobile a OpenGL, numita OpenGL ES https://en.wikipedia.org/wiki/OpenGL_ES .

Freeglut

Ca sa putem lucra cu OpenGL ne trebuie un **Context.** Acesta poate fi obtinut manual prin API dependent de sistemul de operare. Mai mult ne dorim sa avem o **Fereastra** ca sistemul de operare sa ne deseneze comenzile in ceva afisabil. In al treilea rand ne dorim sa putem da comenzi programului nostru, adica sa avem un sistem de **Input.** Putem rezolva toate acest probleme prin utilizarea unei librarii toolkit ca Freeglut (alternative: glow, glfw, sdl, fltk, etc). Cum freeglut are nevoie de pointeri de functie de tip C, nu este compatibil direct cu designul de tip OOP, deci in laborator vom folosi o interfata prin care vom lucra cu freeglut. In codul efectiv de laborator NU exista comenzi freeglut. FREEGLUT NU FACE PARTE DIN OPENGL. Aceasta interfata include functii pentru a notifica de apasarea unei taste, apasarea pe un buton al mouseului, redimensionarea ferestrei, etc. In codul de laborator este folosit un namespace lab::glut in care este pusa toata functionalitatea folosita de freeglut.

Pentru a initializa glut avem nevoie sa specificam exact tuplul (Context, Fereastra, Framebuffer), unde:

- Context e contextul de OpenGL(ex: 3.3 compatibility)
- Fereastra descrie tipul de fereastra pentru aplicatia noastra (nume, pozitie, dimensiuni)
- Framebuffer reprezinta bufferele initiale cu care va fi initializata fereastra (culoare pe 32 de biti, etc).

In laborator avem 2 fisiere: lab_glut.hpp si lab_glut_suport.hpp, in primul este implementata functionalitatea glut iar in al doilea sunt implementate structurile de date folosite la initializarea glut si o interfata din care trebuie sa mostenim clasa laborator pentru a primi notificari asupra evenimentelor (redimensionare ferestra, e momentul sa afisam, mouse Click, tasta apasata,etc). Implementarea din laborator ofera un mediu in care se deseneaza cadre la rand de cand este freeglut initializat pana cand este oprit. Cateva functii utile oferite de aceasta interfata:

- notifiyBeginFrame, apelate inainte de a fi apelata functia de afisare, in aceasta functie ne putem updata partea ce nu tine de desenare a programului (pozitii, orientari,stari,etc)
- notifyDisplay, apelata atunci cand este momentul sa desenam ceva
- notifyEndFrame, apelata dupa ce procesul de afisare pentru cadrul curent s-a terminat.
- notifyReshape, apelata atunci cand am redimensionat fereastra programului.

GLEW

Cum OpenGL are o arhitectura bazata pe extensii fiecare versiune a adus cu sine noi functionalitati ce au fost implementate ca extensii. Ar fi extrem de ineficient sa cerem pointerii pentru aceste functii manual prin API-ul nativ al sistemului de operare (si ar trebui sa scriem asta pentru toate platformele cu care am dori sa lucram), de aceea vom folosi GLEW care este o librarie pentru incarcare usoara a tutoror extensiilor. GLEW nu face parte din OpenGL dar ne face viata mai usoara. Glew este initializat in 3 linii, DUPA initializarea contextului OpenGL, deci dupa initializarea freeglut.

GLM

In grafica, matematica este folosita peste tot, de la simple matrici pentru rotatii pana la integrale infinit dimensionale pentru algoritmii folositi in industria filmului, de aceea ne dorim sa avem un suport de matematica robust, bine documentat si nu il ultimul rand cat mai apropiat de formatul OpenGL. In loc sa scriem noi o librarie de matematica vom folosi libraria GLM. GLM ne ofera rotatii, translatii, vectori de dimensiune 2/3/4, matrici si multe alte functionalitati avansate (e.g. modele de zgomot). Vom folosi doar cele mai simple functionalitati in laboratoarele de la aceasta materie.

Cateva exemple de folosire GLM:

glm::mat4 matrice_identitate = glm::mat4(1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1) creeaza o matrice identitate.

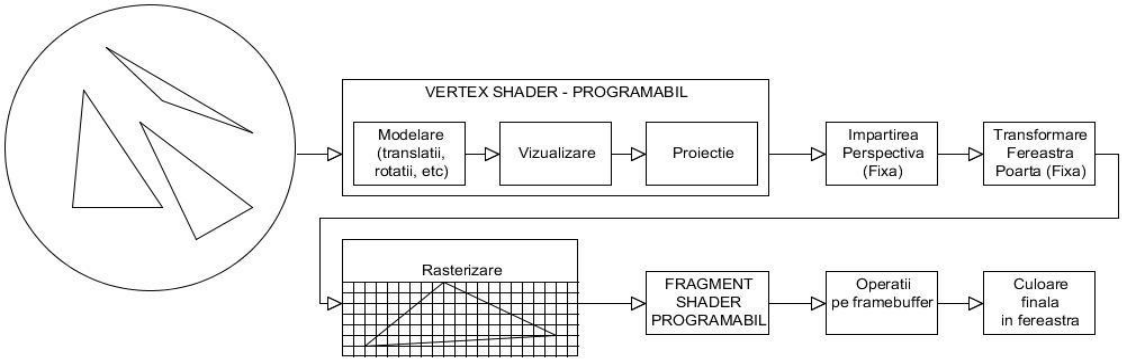
glm::mat4 matrice_translatata = glm::translate(matrice_identitate, glm::vec3(10,5,2)) creeaza matricea matrice_translatata prin translatia continutului matrice_identitate cu vectorul(10,52).

glm::vec3 pozitie, creeaza un vector tridimensional.

Un manual se gaseste aici: http://glm.g-truc.net/0.9.3/glm-0.9.3.pdf iar site-ul librariei este acesta: http://glm.g-truc.net/0.9.4/index.html . GLM foloseste matrici coloana.

Banda Grafica

Dupa cum deja stiti de la curs afisarea pe ecran este reprezentata printr-un lung lant de transformari. Anumite din aceste transformari sunt combinate in etape programabile numite shadere. Scriind shadere puteti procesa FIECARE vertex si FIECARE fragment si calitatea vizuala va este marginita doar de limita imaginatiei (si un pic de shader model☺)



Banda Grafica este un lant de transformari:

1. incepem cu primitivele in coordonate obiect (de modelare, ex: obiect facut de un artist in Blender e in coordonate de modelare)

2. in etapa programabila VERTEX SHADER se fac transformarile de modelare (iau modelul artistului si il pun in scena cu o pozitie, rotatie, etc) ce ne duc in coordonate lume.

3. din coordonate lume, tot in VERTEX SHADER prin transformarea de vizualizare ajungem in coordonate de vizualizare (modelul din scena cu coordonate relativ la camera).

4. din coordonate de vizualizare, tot in VERTEX SHADER, ajungem in coordonate de proiectie prin transformarea de proiectie (ca la curs!)

5. din coordonate de proiectie ajungem in coordonate normalizate de device prin impartirea perspectiva (etapa neprogramabila, o face banda grafica)

6. folosind aceste coordonate primitivele sunt rasterizate in fragmente (bucatele de primitive de dimensiune egala cu a unui pixel). Aceasta e o etapa fixa, o face banda grafica.

7. pentru fiecare fragment e executat FRAGMENT SHADER, care e programabil. Fragment shader-ul are coordonate bidimensionale (tehnic include si adancime ca propietate).

8. cu culorile rezultate din fragment shader putem face diferite operatii (combinare, compresie, etc), numite operatii pe framebuffer. Functionalitate fixa.

9. dupa operatiile pe framebuffer se face transformata fereastra poarta (de exemplu noi desenam doar intr-un colt al ecranului o imagine si apoi o mapam pe toata fereastra). Functionalitate fixa.

10. la sfarsit avem culoarea pentru fiecare pixel din fereastra, o punem pe ecran. Functionalitate fixa.

Ce e un shader?

Un shader e un program executat de GPU, e scris in limbajul GLSL(GL Shading Language).

Un VERTEX SHADER e un program care se executa pentru fiecare vertex trimis catre banda grafica. Este neaparata nevoie ca un vertex shader sa scrie ceva in gl_Position, pentru ca valoarea scrisa reprezinta coordonata post-proiectie a vertexului procesat care e folosita apoi de banda grafica. Un vertex shader are tot timpul o functie numita main. Un exemplu de vertex shader:

```
#version 330
layout(location = 0) in vec3 in_position;
uniform mat4 model_matrix, view_matrix, projection_matrix;
void main(){
gl_Position = projection_matrix*view_matrix*model_matrix*vec4(in_position,1);
}
```

Un FRAGMENT SHADER e un program ce este executat pentru fiecare fragment generat de rasterizator. Fragment shader are tot timpul o functie numita main. Un exemplu de fragment shader:

```
#version 330
layout(location = 0) out vec4 out_color;
void main(){
out_color = vec4(1,0,0,0);
}
```

Cum trimitem date la un shader?

La un shader putem trimite date de la CPU prin variabile uniforme. Se numesc uniforme pentru ca nu variaza pe durata executiei shader-ului. Ca sa putem trimite date la o variabila din shader trebuie sa gasim locatia variabilei in programul shader cu functia

glGetUniformLocation(program_shader, „cum_se_numeste_variabila_in_shader")

Apoi, dupa ce avem locatia (care reprezinta un offset/pointer) putem trimite la acest pointer informatie cu functii de tipul:

glUniformMatrix4fv(locatie, numar, transpusa?, pointer_la_matrice4x4)

glUniform4f(locatie, pointer_la_un_vector_de_4_floats)

glUniform3i(locatie, pointer_la_un_vector_de_3_intregi)

Pe masura ce vom progresa vom invata ca exista si alte metode.

Pentru usurinta puteti gasi API-ul de OpenGL aici: https://www.opengl.org/sdk/docs/man/

Si API-ul pentru GLSL aici: https://www.opengl.org/sdk/docs/manglsl/

Anexe

Matricea de vizualizare se poate calcula astfel:

```
vec3 forward = center – eye;
forward.normalize();
 up.normalize();
vec3 right = forward CROSS up;
vec3 newup = right CROSS forward;
m[0]=right.x; m[4]=right.y; m[8]=right.z;
m[1]=newup.x; m[5]=newup.y; m[9]=newup.z;
m[2]= -forward.x; m[6]= -forward.y; m[10]= -forward.z;
m[15]=1;
translate(-eye.x, -eye.y, -eye.z)
```

Si puteti folosi functia glm::LookAt(pozitie_ochi, unde_ma_uit, directie_sus_camera) pentru a obtine o matrice 4x4 de vizualizare.

Matricea de proiectie perspectiva se poate calcula astfel:

```
float f=1.0f/tan(field_of_view_y_IN_RADIENI/2);
m[0]=f/aspect;
m[5]=f;
m[10]=(zfar+znear)/(znear-zfar);
m[11]= -1;
m[14]=2*zfar*znear/(znear-zfar);
```

Si puteti folosi functia glm::perspective(field_of_view_y, aspect, z_near, z_far)

Viewportul se poate seta cu functia glViewport.

Demonstratii pentru ambele matrici puteti gasi la http://www.songho.ca/opengl/gl_transform.html iar formele matriceale le puteti gasi la https://www.opengl.org/sdk/docs/man2/xhtml/gluLookAt.xml si la https://www.opengl.org/sdk/docs/man2/xhtml/gluPerspective.xml