4/11/2013

FACULTATEA DE AUTOMATICA SI CALCULATOARE

ELEMENTE DE GRAFICA PE CALCULATOR

Laborator 5

OpenGL – de la date la shadere

Introducere

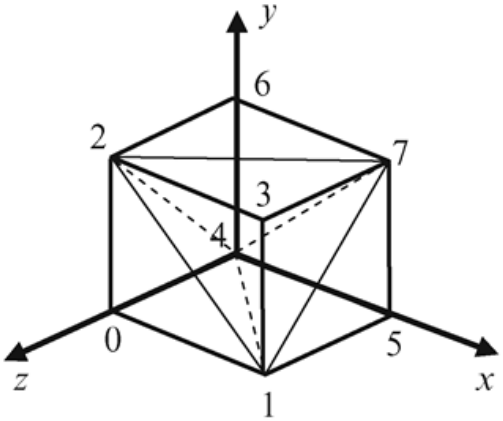
Daca am incerca sa reducem intregul API de OpenGL la mari concepte acestea ar fi:
- date
- shader
- stare

Shaderele au fost introduse in laboratorul precedent si reprezinta programele ce sunt executate pe GPU. Starea reprezinta un concept mai delicat, pentru ca OpenGL este de fapt un mare automat finit cu o multime de stari si posibilitati de a trece de la o stare la alta. De-a lungul laboratoarelor o parte din aceste stari vor fi folosite pentru a obtine efectele dorite, dar cum suntem la inceput atentia va fi focalizata pe date si shadere.

Datele contin informatiile ce ne definesc scena: obiecte tridimensionale, materialele obiectelor (plastic, sticla, etc) cat si pozitiile, orientarile si dimensiunile lor in scena. De exemplu pentru o scena cu un singur patrat avem urmatoarele date:
- varfurile patratului, deci 4 vectori tridimensionali ce ne definesc pozitia fiecarui varf in spatiu
- caracteristicile varfurilor. Daca singura caracteristica a unui varf in afara de pozitie ar fi culoarea am avea inca 4 vectori tridimensionali (culoare e in format RGB)
- topologia patratului, adica metoda prin care legam aceste varfuri
- vectorul de pozitie al patratului relativ la scena
- orientarea patratului (ex: o rotatie)

Topologie

Dintre toate aceste lucruri singurul mai complicat este topologia.



Lista Vertecsi		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

Lista Triunghiuri		
i	j	k
0	1	2
1	3	2
2	3	7
2	7	6
1	7	3
1	5	7
6	7	4
7	5	4
0	4	1
1	4	5
2	6	4
0	2	4

Dupa cum poate fi observat in imaginea de deasupra cubul este descris prin 8 vertecsi si o lista de indecsi. Aceasta lista de indecsi reprezinta fiecare din cele 12 triunghiuri ale cubului (2 pentru fiecare fata). Folosind vertecsi si indecsi putem descrie orice obiect tridimensional.

Daca dorim sa nu lucram cu indecsi putem folosi metode de definire neindexata, cum sunt strip-urile de triunghiuri https://en.wikipedia.org/wiki/Triangle_strip . Putem folosi si moduri in care topologia nu ar avea sens, de exemplu cand dorim sa desenam doar puncte.

O observatie importanta legata de topologie este ordinea varfurilor intr-o primitiva solida (nu linie, nu punct) cu mai mult de 2 varfuri. Aceasta ordine poate fi in sensul acelor de ceas sau in sens invers si desi nu o vom folosi la acest laborator este suficient de importanta pentru a fi mentionata.

Meshe

O mesha este un obiect tridimensional definit prin varfuri si indecsi. In laborator exista cod pentru a incarca obiecte din format .obj (https://en.wikipedia.org/wiki/Wavefront_.obj_file , in detaliu aici: http://paulbourke.net/dataformats/obj/). Acest format este popular in mediul aplicatiilor tridimensionale. Pentru a incarca o mesha puteti folosi functia:

Lab::loadObj(„fisier.obj”,**vao**, **vbo**, **ibo**, count);

VAO– Vertex Array Object, reprezinta un container de stare
VBO – Vertex Buffer Object, reprezinta un container de vertecsi
IBO – Index Buffer Object, reprezinta un container de indecsi (mai sunt numiti si elemente)
Count - reprezinta numarul de indecsi ai obiectului incarcat si este necesar pentru comenzile de desenare indexata.

Vertex Buffer Object

Un vertex buffer object reprezinta un container in care stocam TOATE datele ce tin de continutul vertecsilor. Pozitia in coordonate de modelare, normala (vom invata in laboratorul urmator la ce e folosita), culoarea, etc., toate sunt caracteristici ale unui vertex, care sunt numite in OpenGL **atribute.**

Un vertex buffer object este create cu comanda: `glGenBuffers(1, &vbo_id);`
Un vertex buffer object este distrus cu comanda: `glDeleteBuffers(1,&vbo_id);`

Pentru a putea pune date intr-un buffer trebuie intai sa il **MAPAM** la un punct de legatura. Pentru vertex buffer acest „binding point” se numeste GL_ARRAY_BUFFER, deci codul de stocare de date intr-un VBO este:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo_id);
glBufferData(GL_ARRAY_BUFFER,sizeof(Format)*nr_vertecsi,&vertecsi[0],GL_STATIC_DRAW);
```

Prima comanda leaga VBO-ul la punctul de legare GL_ARRAY_BUFFER iar a doua comanda citeste de la adresa primului vertex din array-ul de vertecsi si copiaza in VBO-ul mapat la GL_ARRAY_BUFFER sizeof(Format)*nr_vertecsi bytes de memorie. GL_STATIC_DRAW reprezinta un hint pentru driverul video in ceea ce priveste metoda de utilizare a bufferului. Format reprezinta structura unui vertex, de exemplu:

```
struct VertexFormat{
        glm::vec3 pozitie;
        glm::vec3 culoare;
        glm::vec3 viteza;
}
```

Index Buffer Object

Un index buffer object (numit si element buffer object) reprezinta un container in care stocam TOTI indecsii. Cum VBO si IBO sunt buffere, ele sunt extrem de similare in constructie, incarcare de date si destructie.

```
glGenBuffers(1, &ibo_id);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_id);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int)*nr_indecsi, &indecsi[0], GL_STATIC_DRAW);
```

La fel ca la VBO, creem un IBO si apoi il legam la un punct de legatura, doar ca de data aceasta punctul de legatura este GL_ELEMENT_ARRAY_BUFFER. Datele sunt trimise catre bufferul mapat la acest punct de legatura, in cazul indecsilor toti vor fi de dimensiunea unui singur intreg.

Vertex Array Object

Intr-un vertex array object putem stoca toata informatia legata de starea geometriei desenate. Putem folosi un numar mare de buffere pentru a stoca fiecare din diferitele atribute („separate buffers”). Putem stoca mai multe(sau toate) atribute intr-un singur buffer („interleaved” buffers). Datorita acestei complexitati nu putem ca de fiecare data inainte de comanda de desenare sa dam comenzile de binding pentru toate bufferele si atributele, de aceea se foloseste un vertex array object care tine minte toate aceste legaturi.

Un vertex array object este creat cu : `glGenVertexArrays(1 , &vao_id);`
`Este legat cu: glBindVertexArray(vao_id);`
`Si este distrus cu: glDeleteVertexArrays(1,&vao_id);`

Inainte de a crea VBO-urile si IBO-ul necesar pentru un obiect se leaga VAO-ul obiectului si acesta va tine minte automat toate legaturile. Inainte de comanda de desenare este suficient sa legam doar VAO-ul ca OpenGL sa stie toate legaturile create la constructia obiectului.

Desi am create toate aceste buffere cum specificam legatura intre date si shadere? – prin legare de atribute.

Legare de atribute

Legarea de atribute se poate face prin multe moduri distincte, ce au evoluat o data cu banda grafica. In lumea OpenGL-ului modern (3.3+) este recomandata prima metoda, numita si **metoda pe layout-uri**. In aceasta metoda se folosesc pipe-uri ce leaga un atribut din OpenGL de un nume de atribut in shader.

OpenGL:

```
glEnableVertexAttribArray(5);
glVertexAttribPointer(5,3,GL_FLOAT,GL_FALSE,sizeof(MyVertexFormat),(void*)0);
```

Prima comanda seteaza pipe-ul cu numarul 5 ca fiind folosibil.

Cu a doua comanda trimitem pe pipe-ul 5 (argument 1) cate 3(argument 2) floaturi (argument 3) pe care nu le normalizam(argument 4), adresa urmatorului atribut se afla peste sizeof(MyVertexFormat) bytes (argument 5, se numeste stride), iar toate datele le citesc din buffer-ul legat la GL_ARRAY_BUFFER, plecand cu offsetul initial 0 (argument 6).

Pe partea de shader folosim `layout(location = 5) in vec3 atributul_meu;`

Daca nu exista acces la OpenGL 3.3, in care e introduce metoda pe layouturi, se poate folosi metoda bazata pe **cautare de locatie**. Pe partea de OpenGL gasim pe ce pipe trimite OpenGL date atributului cu urmatoarea comanda:

```
Unsigned int pipe = glGetAttribLocation(gl_program_shader, "atributul_meu");
```

Iar apoi putem folosi comenzile introduse in paragraful ulterior, doar ca pe noul pipe.

```
glEnableVertexAttribArray(pipe);
glVertexAttribPointer(pipe,3,GL_FLOAT,GL_FALSE,sizeof(MyVertexFormat),(void*)0);

Pe partea de GLSL definem atributele cu: attribute vec3 atributul_meu;
```

Dezavantajul acestei metoda fata de prima este ca avem nevoie ca shaderul sa existe pentru a putea apela comenzile. Mai mult, cu aceasta metoda nu putem folosi mai multe shadere cu inputuri diferite in acelasi timp.

Daca nici aceasta metoda nu este disponibila datorita necesitatii de a lucra cu o versiune mai veche de OpenGL, atunci putem folosi metoda bazata pe **setare de locatie.** Pe partea de OpenGL putem folosi urmatoarea comanda pentru a lega la nivel de shader un pipe de un nume de atribut:

```
glBindAttribLocation(gl_program_shader, pipe , "in_position");
```

Ca mai apoi sa folosim comenzile introduse cu prima metoda:

```
glEnableVertexAttribArray(pipe);
glVertexAttribPointer(pipe,3,GL_FLOAT,GL_FALSE,sizeof(MyVertexFormat),(void*)0);

Pe partea de GLSL definem atributele cu: attribute vec3 atributul_meu;
Dupa acest proces trebui sa re-linkam shaderul!!!
```

Dezavantajul acestei metode este ca avem nevoie ca shaderul sa existe, nu putem folosi tehnica de legare cu mai multe shadere in acelasi timp si ca trebuie sa re-linkam shaderul dupa crearea legaturilor.

In cazul putin probabil in care nu dorim sa folosim nici un din metodele deja prezentate putem folosi **legarea implicita**, in care pe partea de OpenGL folosim doar :

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,sizeof(MyVertexFormat),(void*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,sizeof(MyVertexFormat),(void*)0);
...
```

Iar pe partea de GLSL nu avem decat:

```
in vec3 atributul_meu;
```

sau

```
attribute vec3 atributul_meu
```

Ordinea in care declaram atributele in shader este TEORETIC aceeasi cu ordinea de legatura de date, deci primul declarat e legat implicit la pipe-ul 0, al doilea la pipe-ul 1, samd. Dezavantajul acestei metode este ca nu avem garantia acestui comportament si putem ajunge usor la comportament nedefinit.