Laboratorul 02

Video Laborator 2: https://youtu.be/RtXuIQO8l0U.
Autor: Alex Gradinaru

OpenGL – Date

Dacă am încerca să reducem întregul API de OpenGL la mari concepte acestea ar fi:

- date
- stări
- shadere

Shaderele vor fi introduse pe parcursul cursului.
Stările reprezintă un concept mai larg, OpenGL fiind de fapt un mare automat finit cu o mulțime de stări și posibilități de a seta aceste stări. De-a lungul laboratoarelor o parte din aceste stări vor fi folosite pentru a obține efectele dorite.
Datele conțin informațiile ce definesc scena, precum:

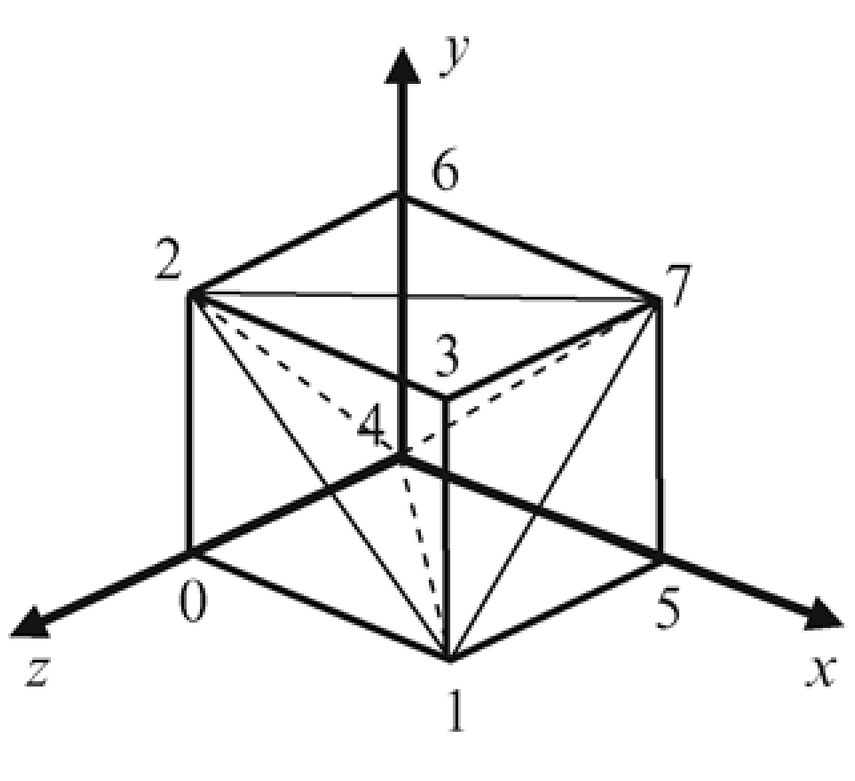
- obiecte tridimensionale
- proprietăți de material ale obiectelor (plastic, sticlă, etc)
- pozițiile, orientările și dimensiunile obiectelor în scenă
- orice alte informații necesare ce descriu proprietăți de obiecte sau de scenă

De exemplu pentru o scenă cu un singur pătrat avem următoarele date:

- vârfurile pătratului - 4 vectori tridimensionali ce definesc poziția fiecărui vârf în spațiu
- caracteristicile vârfurilor
  - dacă singura caracteristică a unui vârf în afară de poziție ar fi culoarea am avea încă 4 vectori tridimensionali (RGB)
- topologia pătratului, adică modul în care legăm aceste vârfuri

OpenGL este un API de grafică tridimensională, adică, toate obiectele care pot fi definite sunt raportate la un sistem de coordonate carteziene tridimensional. Cu toate acestea putem utiliza API-ul pentru a afișa obiecte bi-dimensionale chiar dacă acestea sunt definite prin coordonate (x,y,z) prin plasarea tuturor datelor într-un singur plan și utilizarea unei proiecții corespunzătoare.
În cadrul laboratorului vom utiliza coordonata Z = 0. Astfel orice punct tridimensional va deveni P(x,y,0)

Topologie



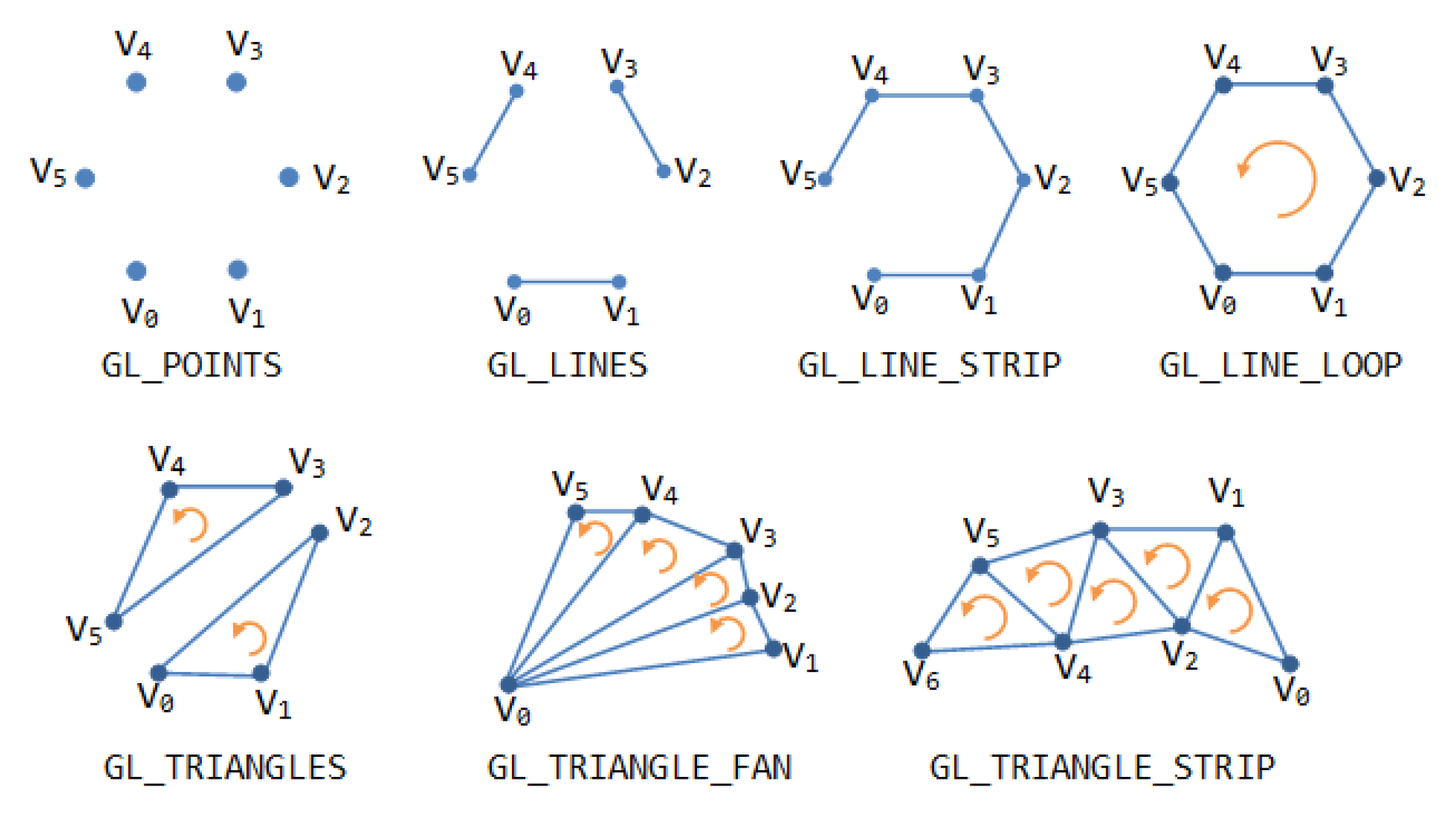
Lista Vertecsi		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

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

Primitiva de bază în OpenGL este triunghiul. Astfel, așa cum se poate observa și în imaginea de sus, pentru a desena un obiect acesta trebuie specificat prin triunghiuri.

Cubul descris mai sus este specificat prin lista celor 8 coordonate de vârfuri și o listă de 12 triunghiuri care descrie modul în care trebuie unite vârfurile specificate în lista precedentă pentru a forma fețele cubului. Folosind vârfuri și indici putem descrie în mod discret orice obiect tridimensional.

Mai jos regăsiți principalele primitive acceptate de standardul OpenGL 3.3+.



OpenGL Primitives

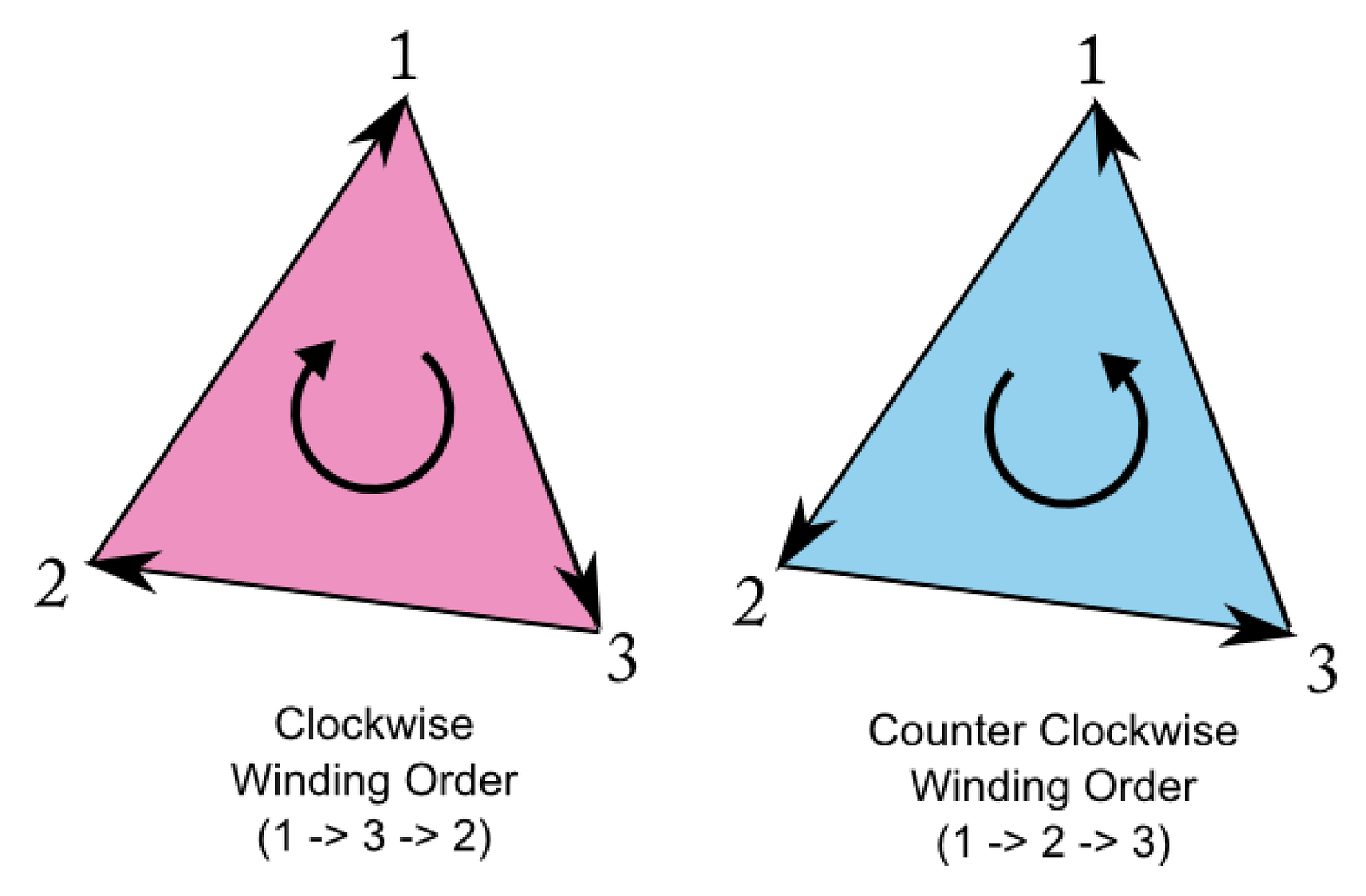
După cum se poate observa, există mai multe metode prin care geometria poate fi specificată:

- **GL_LINES** și **GL_TRIANGLES** sunt cele mai des utilizate primitive pentru definirea geometriei
- **GL_POINTS** este des utilizat pentru a crea sistemele de particule
- Celelalte modele reprezintă doar niște optimizari ale celor 3 primitive de bază, atât din perspectiva memoriei dar și a ușurinței în a specifica anumite topologii însă utilitatea lor este deseori limitată întrucât obiectele mai complexe nu pot fi specificate decât prin utilizarea primitivelor simple

În cadrul framework-ului puteți seta tipul de primitivă utilizat de către un obiect la randare prin intermediul funcției Mesh::SetDrawMode(GLenum primitive) unde `primitive` poate fi oricare dintre primitivele menționate în imaginea de mai sus.

Ordinea specificării vârfurilor

O observație importantă legată de topologie este ordinea vârfurilor într-o primitivă solidă (nu linie, nu punct) cu mai mult de 2 vârfuri. Această ordine poate fi în sensul acelor de ceas sau în sens invers.



Face Culling

API-ul OpenGL oferă posibilitatea de a testa orientarea aparentă pe ecran a fiecărui triunghi înainte ca acesta să fie redat și să îl ignore în funcție de starea de discard setată: **GL_FRONT** sau **GL_BACK**. Această funcționalitate poartă numele de **Face Culling** și este foarte importantă deoarece reduce costul de procesare total.

Modul cum este considerată o față ca fiind **GL_FRONT** sau **GL_BACK** poate fi schimbat folosind comanda glFrontFace (valoarea inițială pentru o față **GL_FRONT** este considerată ca având ordinea specificării vârfurilor în sens trigonometric / counter clockwise):

```
// mode can be GL_CW (clockwise) or GL_CCW (counterclockwise)
// the initial value is GL_CCW
void glFrontFace(GLenum mode);
```

Exemplu: pentru un **cub** maxim 3 fețe pot fi vizibile la un moment dat din cele 6 existente. În acest caz maxim 6 triunghiuri vor fi procesate pentru afișarea pe ecran în loc de 12.

În mod normal face-culling este dezactivat. Acesta poate fi activat folosind comanda glEnable:

```
glEnable(GL_CULL_FACE);
```

Pentru a dezactiva face-culling se folosește comanda glDisable:

```
glDisable(GL_CULL_FACE);
```

Pentru a specifica ce orientare a fețelor să fie ignorată se folosește comanda **glCullFace**

```
// GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK are accepted.
// The initial value is GL_BACK.
glCullFace(GL_BACK);
```

Meshe

Un „mesh" este un obiect tridimensional definit prin vârfuri și indici. În laborator aveți posibilitatea să încărcați meshe în aproape orice format posibil prin intermediul clasei Mesh.

Vertex Buffer Object (VBO)

Un vertex buffer object reprezintă un container în care stocăm date ce țin de conținutul vârfurilor precum:

- poziție
- normală
- culoare
- coordonate de texturare
- etc…

Un vertex buffer object se poate crea prin comanda OpenGL **glGenBuffers**:

```
        GLuint VBO_ID;                  // ID-ul (nume sau referinta) buffer-ului ce va fi cerut de la G
        glGenBuffers(1, &VBO_ID);       // se genereaza ID-ul (numele) bufferului
```

Așa cum se poate vedea și din explicația API-ului, funcția glGenBuffers primește numărul de buffere ce trebuie generate cât și locația din memorie unde vor fi salvate referințele (ID-urile) generate.
În exemplul de mai sus este generat doar 1 singur buffer iar ID-ul este salvat în variabila `VBO_ID`.

Pentru a distruge un VBO și astfel să eliberăm memoria de pe **GPU** se folosește comanda **glDeleteBuffers**:

```
glDeleteBuffers(1, &VBO_ID);
```

Pentru a putea pune date într-un buffer trebuie întâi să legăm acest buffer la un „target". Pentru un vertex buffer acest „binding point" se numește **GL_ARRAY_BUFFER** și se poate specifica prin comanda **glBindBuffer**:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO_ID);
```

În acest moment putem să facem upload de date din memoria **CPU** către **GPU** prin intermediul comenzii **glBufferData**:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices[0]) * vertices.size(), &vertices[0], GL_STATIC_DRAW);
```

- Comanda citește de la adresa specificată, în exemplul de sus fiind adresa primului vârf `&vertices[0]`, și copiază în memoria video dimensiunea specificată prin parametrul al 2-lea.
- **GL_STATIC_DRAW** reprezintă un hint pentru driver-ul video în ceea ce privește metoda de utilizare a bufferului. Acest simbol poate avea mai multe valori dar în cadrul laboratorului este de ajuns specificarea prezentată. Mai multe informații găsiți pe pagina de manual a funcției glBufferData

Pentru a înțelege mai bine API-ul OpenGL vă rocomandăm să citiți documentația indicată pentru fiecare comandă prezentată. Atunci când se prezintă o nouă comandă, dacă apăsați click pe numele acesteia veți fi redirecționați către pagina de manual a comenzii respective.
De asemenea, documentația oficială și completă a API-ului OpenGL poate fi gasită pe pagina **OpenGL 4 Reference Pages**

Index Buffer Object (IBO)

Un index buffer object (numit și element buffer object) reprezintă un container în care stocăm indicii vertecșilor. Cum **VBO** si **IBO** sunt buffere, ele sunt extrem de similare în construcție, încărcare de date și ștergere.

```
glGenBuffers(1, &IBO_ID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO_ID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices[0]) * indices.size(), &indices[0], GL_STATIC_DRAW);
```

La fel ca la VBO, creăm un IBO și apoi îl legăm la un punct de legatură, doar că de data aceasta punctul de legatură este **GL_ELEMENT_ARRAY_BUFFER**. Datele sunt trimise către bufferul mapat la acest punct de legatură. În cazul indicilor toți vor fi de dimensiunea unui singur întreg.

Vertex Array Object (VAO)

Într-un vertex array object putem stoca toată informația legată de starea geometriei desenate. Putem folosi un număr mare de buffere pentru a stoca fiecare din diferitele atribute („separate buffers"). Putem stoca mai multe (sau toate) atribute într-un singur buffer („interleaved" buffers). În mod normal înainte de fiecare comandă de desenare trebuie specificate toate comenzile de „binding" pentru buffere sau atribute ce descriu datele ce doresc a fi randate. Pentru a simplifica această operație se folosește un vertex array object care ține minte toate aceste legături.

Un vertex array object este creat folosind comanda **glGenVertexArrays**:

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
```

Este legat cu **glBindVertexArray**:

```
glBindVertexArray(VAO);
```

Înainte de a crea VBO-urile și IBO-ul necesar pentru un obiect se va lega VAO-ul obiectului și acesta va ține minte automat toate legăturile specificate ulterior.

După ce toate legăturile au fost specificate este recomandat să se dea comanda `glBindVertexArray(0)` pentru a dezactiva legătura către VAO-ul curent, deoarece altfel riscăm ca alte comenzi OpenGL ulterioare să fie legate la același VAO și astfel să introducem foarte ușor erori în program.

Înainte de comanda de desenare este suficient să legăm doar VAO-ul ca OpenGL să știe toate legatările create la construcția obiectului.

Laborator 2

Descriere laborator

În cadrul laboratorului vom învăța să folosim **VAO, VBO, IBO** și astfel să generăm și încărcăm geometrie simplă. Laboratorul pune la dispoziție structura VertexFormat ce va fi utilizată ca bază pentru a crea geometria.

```
struct VertexFormat
{
        // position of the vertex
        glm::vec3 position;

        // vertex normal
        glm::vec3 normal;

        // vertex texture coordinate
        glm::uvec2 text_coord;

        // vertex color
        glm::vec3 color;
};
```

Clasa `Mesh` pune la dispoziție posibilitatea de a încărca geometrie simplă folosind diverse metode:

```
// Initializes the mesh object using a VAO GPU buffer that contains the specified number of indices
bool InitFromBuffer(unsigned int VAO, unsigned short nrIndices);

// Initializes the mesh object and upload data to GPU using the provided data buffers
bool InitFromData(std::vector<VertexFormat> vertices,
                std::vector<unsigned short>& indices);

// Initializes the mesh object and upload data to GPU using the provided data buffers
bool InitFromData(std::vector<glm::vec3>& positions,
                std::vector<glm::vec3>& normals,
                std::vector<unsigned short>& indices);
```

Taste de control pentru cameră

- **W, A, S, D, Q, E** - deplasare față, stânga, spate, dreapta, jos, sus
- **MOUSE RIGHT + MOUSE MOVE** - rotație cameră

F3 - afișează/ascunde gridul din scenă
Space - desenează primitivele doar prin **puncte** sau **linii** (wireframe) sau **geometrie opacă**

Cerințe laborator

Toate cerințele ce țin de încărcare de geometrie trebuie rezolvate prin intermediul funcției `Laborator2::CreateMesh` dar puteți folosi metodele `Mesh::InitFromData()` pentru a verifica validitatea geometriei.

1. Descărcați framework-ul de laborator
2. Completați geometria și topologia unui cub: vectorii de vertecși și indecși din inițializare. `VertexFormat` este o structură pentru vertex cu 2 parametrii (poziție, culoare).
3. Completați funcția `Laborator2::CreateMesh` astfel încât să încărcați geometria pe GPU
   - creați un VAO
   - creați un VBO și adăugați date în el
   - creați un IBO și adăugați date în el
   - afișați noul obiect (RenderMesh[cube3]) astfel încât să nu se suprapună cu un alt obiect
4. Creați o nouă formă geometrică simplă, de exemplu un tetraedru și desenați-l în scenă
5. Atunci când se apasă tasta **F2** faceți toggle între modul de culling **GL_BACK** și **GL_FRONT**
   - nu uitați să activați și să dezactivați face culling folosind `glEnable()` / `glDisable()`
6. Creați un pătrat format din 2 triunghiuri astfel încât fiecare triunghi să fie vizibil doar dintr-o parte
   - în orice moment de timp nu trebuie să se vadă decât 1 triunghi

Search

Info curs
- Elemente de Grafică pe Calculator
- Infographie

Cataloage EGC
- TBA

Laboratoare
- Laboratorul 01
- Laboratorul 02
- Laboratorul 03
- Laboratorul 04
- Laboratorul 05
- Laboratorul 06
- Laboratorul 07
- Prezentare Tema 1
- Laboratorul 08
- Laboratorul 09
- Vacanță
- Prezentare Tema 2
- Recuperări laborator
- Prezentare Tema 3
- Resurse: Redare text

Teme
- Regulament General
- Tema 1 - Bow and Arrow
- Tema 2 - Skyroads
- Tema 3 - Stylised Runner

Resurse
- Resurse Utile
- Notare

Table of Contents