Laboratorul 00

Framework laborator

Framework-ul utilizat ofera toate functionalitatile de baza ale unui motor grafic minimal, precum:

- Fereastra de desenare avand la baza un context **OpenGL 3.3+**
- Suport pentru incarcarea de meshe 3D
- Suport pentru incarcarea de texturi 2D pentru texturarea meshelor 3D
- Suport pentru definirea si incarcarea de shadere OpenGL

De asemenea, pe langa functionalitatile de baza framework-ul implementeaza un model generic pentru scrierea de aplicatii OpenGL. Astfel, sunt oferite urmatoarele aspecte:

- Control pentru fereastra de afisare
- Management pentru input de la tastatura si mouse
- Camera de vizualizare cu input predefinit pentru a usura deplasarea si visualizarea scenei
- Model arhitectural al unei aplicatii simple OpenGL, bazat pe toate aspectele prezentate

Functionalitatea framework-ului este oferita prin intermediul mai multor biblioteci (libraries):

- **GLFW**
  - **Site oficial** http://www.glfw/.org [http://www.glfw/.org] **Github** https://github.com/glfw/glfw [https://github.com/glfw/glfw]
  - oferta supportul de baza pentru API OpenGL precum context, fereastra, input, etc
- **GLEW**
  - **Site oficial** http://glew.sourceforge.net [http://glew.sourceforge.net] **Github** https://github.com/nigels-com/glew [https://github.com/nigels-com/glew]
  - Asigura suportul pentru extensiile de OpenGL suportate de placa video
- **GLM**
  - **Site oficial** http://glm.g-truc.net [http://glm.g-truc.net] **Github** https://github.com/g-truc/glm [https://github.com/g-truc/glm]
  - Functionalitati matematice bazate pe specificatiile limbajului GLSL (shadere OpenGL)
  - Asigura interoperabilitate simpla cu API OpenGL
- **ASSIMP**
  - **Site oficial** http://www.assimp.org [http://www.assimp.org] **Github** https://github.com/assimp/assimp [https://github.com/assimp/assimp]
  - Open Asset Import Library
  - Ofera suport pentru incarcarea de meshe si scene 3D
  - Suporta majoritatea formatele de stocarea 3D utilizate in industrie
- **STB**
  - **Github** https://github.com/nothings/stb [https://github.com/nothings/stb]
  - Ofera support pentru incarcare/decodare de imagini JPG, PNG, TGA, BMP, PSD, etc.

Standardul OpenGL

OpenGL este un standard(API) pe care il putem folosi pentru a crea aplicatii grafice real-time. Este aproape identic cu Direct3D, ambele avand o influenta reciproca de-a lungul anilor.

- Mai multe informatii despre istoricul OpenGL gasi la adresa: https://en.wikipedia.org/wiki/OpenGL [https://en.wikipedia.org/wiki/OpenGL]
- Explicatii complete prinvind API-ul OpenGL cat si utilizarea acestuia se pot gasi pe pagina oficiala a standardului: https://www.opengl.org/sdk/docs/man/ [https://www.opengl.org/sdk/docs/man/]

Incepand cu 2016 a fost lansat si API-ul Vulkan ce ofera access avansat low-level la capababilitatile grafice moderne ale placilor video. Standardul Vulkan este orientat dezvoltarii aplicatiilor de inalta performanta iar complexitatea acestuia depaseste cu mult aspectele de baza ce vor fi prezentate in cadrul cusului/laboratorului.

Structura framework-ului

- **/libs**
  - Bibliotecile utilizate in cadrul framework-ului
- **/Visual Studio**
  - Proiect Visual Studio 2013/2015 preconfigurat
- **/Resources**
  - Resurse necesare rularii proiecutului
  - **/Textures**
    - diverse imagini ce pot fi incarcate si utilizate ca texturi
  - **/Shaders**
    - exemple de programe shader - Vertex Shader si Fragment Shader
  - **/Models**
    - meshe 3D ce pot fi incarcate in cadrul framework-ului
- **/Source**
  - Surse C++
  - **/include**
    - o serie de headere predefinite pentru facilitarea accesului la biblioteci
    - gl.h
      - adauga supportul pentru API-ul OpenGL
    - glm.h
      - adauga majoritatea headerelor glm ce vor fi utilizate
      - printare usoara pentru `glm::vec2, glm::vec3, glm::vec4` prin intermediul operatorului C++ supraincarcat: `operator«`
    - math.h
      - simple definitii preprocesor pentru MIN, MAX, conversie radiani ⇔ grade
    - utils.h
      - simple definitii preprocesor pentru lucrul cu memoria si pe biti
  - **/Components**
    - diverse implementari ce facilitareaza lucrul in cadrul laboratoarelor
    - SimpleScene.cpp
      - model de baza al unei scene 3D utilizata ca baza a tuturor laboratoarelor
    - CameraInput.cpp
      - Imlpementare a unui model simplu de control FPS al camerei de vizualizare
  - **/Core**
    - API-ul de baza al framwork-ului
    - **/GPU**
      - GPUBuffers.cpp
        - Asigura supportul pentru definirea de buffere de date si incarcarea de date (tip mesh) pe GPU
      - Mesh.cpp
        - Loader de meshe 3D atat din fisier cat si din memorie
      - Shader.cpp
        - Loader de programe Shader pentru placa video
      - Texture2D.cpp
        - Loader de texturi 2D pe GPU
    - **/Managers**
      - ResourcePath.h
        - Locatii predefinite pentru utilizarea la incarcarea resurselor

- TextureManager.cpp
  - Asigura incarcare si management pentru texturile Texture2D
  - Incarca o serie de texturi simple predefinite
- **/Window**
  - WindowCallbacks.cpp
    - Asigura implementarea functiilor de callback necesare de GLFW pentru un context OpenGL oarecare
    - Evenimentele GLFW sunt redirectionate catre fereastra definita de Engine
  - WindowObject.cpp
    - Ofera implementarea de fereastra de lucru, support predefinite definire pentru callbacks, dar si un model de buffering pentru evenimente de input tastatura si mouse
  - InputController.cpp
    - Prin mostenire ofera support pentru implementarea callback-urilor de input/tastatura. Odata instantiat, obiectul se va atasa automat pe fereastra de lucru (pe care o obtine de la Engine) si va primi automat evenimentele de input pe care le va executa conform implementarii
    - In cadrul unui program pot exista oricate astefel de obiecte. Toate vor fi apelate in ordinea atasarii lor, dar si a producerii evenimentelor
  - Engine.cpp
    - Asigura initializarea contextului OpenGL si a ferestrei de lucru
  - World.cpp
    - Asigura implementarea modelului de functionare al unei aplicatii OpenGL pe baza API-ului oferit de Framework
- **/Laboratoare**
  - Implementarile pentru fiecare laborator SPG
  - Fiecare laborator va pleca de la baza oferita de SimpleScene

Modelul de functionare al aplicatiei de laborator

In cadrul unui laborator modelul aplicatiei grafice prezentat mai sus este implementat de catre clasa `World`.
Pasul 2 este tratat de catre instantele InputController in timp ce pasul 4 este asigurat de functiile `FrameStart()`, `Update(float deltaTime)`, si `FrameEnd()` mostenite de la clasa `World`. Clasa `World` extinde deja `InputController` pentru a usura munca in cadrul laboratorului.
Toate laboratoarele SPG vor fi implementate pe baza `SimpleScene` ce ofera urmatoarele facilitati:

- scena 3D cu randarea sistem de referinta in coordonate OpenGL
  - plan orizontal XOZ
  - evidentiarea spatiului pozitiv OX, OY, OZ)
- camera predefinita pentru explorarea scenei
- shadere predefinite pentru lucrul in primele laboratoare
- management pentru stocarea shaderelor si meshelor nou create prin nume

Etapele rularii aplicatiei

1. Se definesc proprietatile pentru fereastra de lucru (Main.cpp)
2. Se intializeaza Engine-ul astfel - `Engine::Init()`
   a. Se initializeaza API-ul OpenGL (`glfwInit()`)
   b. Se creeaza fereastra de lucru cu un context OpenGL 3.3+
      I. Se ataseaza evenimentele de fereastra prin intermediul WindowsCallbacks.cpp
   c. Se initializeaza managerul de texturi
3. Se creeaza si initializeaza o noua scena 3D de lucru avand la baza modelul de update prezentat anterior (Main.cpp)
4. Se porneste rularea scenei incarcate (`LoopUpdate()`)

OpenGL – Date

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

- date
- stari
- shadere

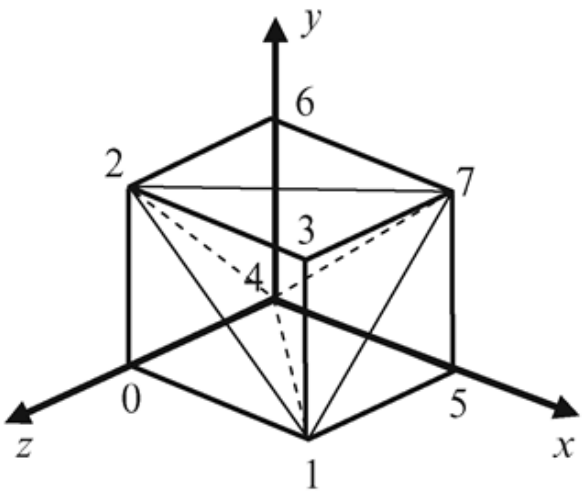
Datele contin informatiile ce definesc scena, precum:

- obiecte tridimensionale
- proprietati de material ale obiectelor (plastic, sticla, etc)
- pozitiile, orientarile si dimensiunile obiectelor lor in scena
- orice alte informatii necesare ce descriu proprietati de obiecte sau de scena

De exemplu pentru o scena cu un singur patrat avem urmatoarele date:

- varfurile patratului - 4 vectori tridimensionali ce 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(RGB)
- topologia patratului, adica metoda prin care legam aceste varfuri

Topologie



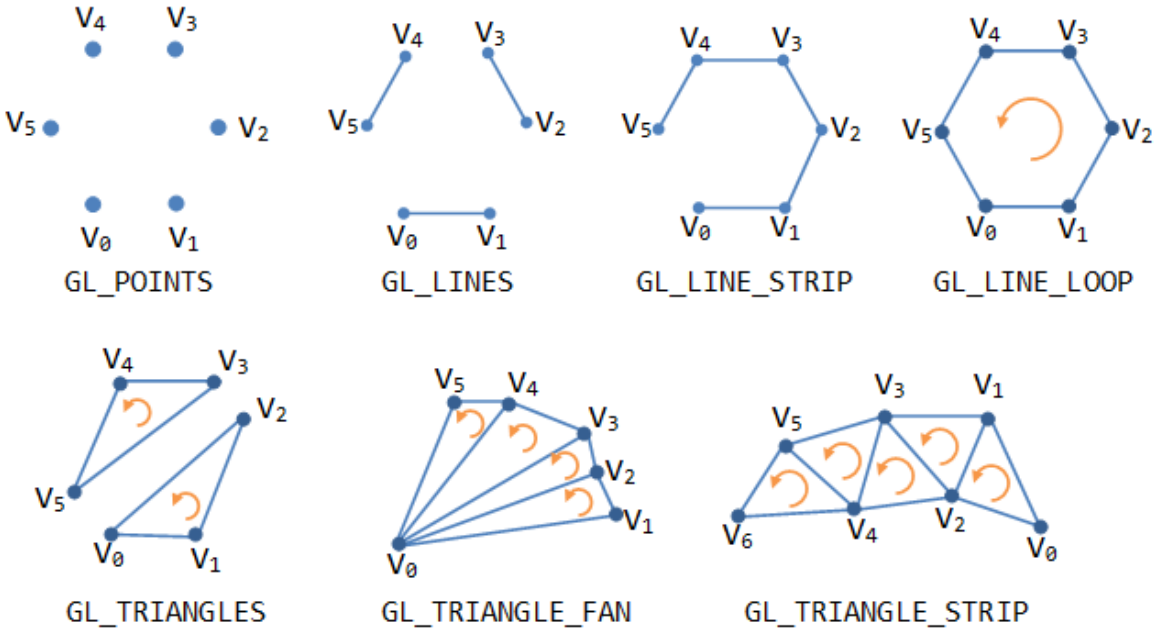
Lista Vertecsi		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

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 baza in OpenGL este triunghiul. Astfel asa cum se poate observa si in 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 varfuri si o lista de 12 triunghiuri care descrie modul in care trebuie unite varfurile specificate in lista precedenta pentru a forma fețele cubului. Folosind varfuri si indici putem descrie in mod discret orice obiect tridimensional.

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



OpenGL Primitives

Dupa cum se poate observa exista mai multe metode prin care geometria poate fi specificata:

- **GL_LINES** si **GL_TRIANGLES** sunt cele mai des utilizate primitive pentru definirea geometriei
- **GL_POINTS** este des utilizat pentru a crea sistemele de particule
- Celelalte modele reprezinta doar niste optimizari ale celor 3 primitive de baza, atat din perspectiva memoriei dar si a usurintei in a specifica anumite topologii insa utilitatea lor este deseori limitata intrucat obiectele mai complexe nu pot fi specificate decat prin utilizarea primitivelor simple

Suprafete 3D

Un „mesh” sau o suprafata 3D este un obiect tridimensional definit prin varfuri si indici.

Vertex Buffer Object (VBO)

Un vertex buffer object reprezinta un container in care stocam date ce tin de continutul varfurilor precum:

- pozitie
- normala
- culoarea
- coordonate de texturare
- etc…

Un vertex buffer object se poatea crea prin comanda OpenGL **glGenBuffers [https://www.opengl.org/sdk/docs/man/html/glGenBuffers.xhtml]**:

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

Asa cum se poate vedea si din explicatia API-ului, functia glGenBuffers [https://www.opengl.org/sdk/docs/man/html/glGenBuffers.xhtml] primeste numarul de buffere ce trebuie generate cat si locatia din memorie unde vor fi salvate referintele (ID-urile) generate.
In exemplu de mai sus este general doar 1 singur buffer iar ID-ul este salvat in variabila VBO_ID.

Pentru a distruge un VBO si astfel sa eliberam memoria de pe **GPU** se foloseste comanda **glDeleteBuffers [https://www.opengl.org/sdk/docs/man4/html/glDeleteBuffers.xhtml]**:

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

Pentru a putea pune date intr-un buffer trebuie intai sa legam acest buffer la un „target”. Pentru un vertex buffer acest „binding point” se numeste **GL_ARRAY_BUFFER**, si se poate specifica prin comanda **glBindBuffer [https://www.khronos.org/opengles/sdk/1.1/docs/man/glBindBuffer.xml]**:

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

In acest moment putem sa facem upload de date din memoria **CPU** catre **GPU** prin intermediul comenzii **glBufferData [https://www.opengl.org/sdk/docs/man4/html/glBufferData.xhtml]**:

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

- Comanda citeste de la adresa specificata, in exemplul de sus fiind adresa primului varf `&vertices[0]`, si copiaza in memoria video dimensiunea specificata prin parametrul al 2-lea.
- **GL_STATIC_DRAW** reprezinta un hint pentru driverul video in ceea ce priveste metoda de utilizare a bufferului. Acest simbol poate avea mai multe valori dar in cadrul laboratorului este de ajuns specificarea prezentata. Mai multe informatii gasiti pe pagina de manual a functiei glBufferData [https://www.opengl.org/sdk/docs/man4/html/glBufferData.xhtml]

Index Buffer Object (IBO)

Un index buffer object (numit si element buffer object) reprezinta un container in care stocam indicii vertex-ilor. 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(indices[0]) * indices.size(), &indices[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 indicilor toti vor fi de dimensiunea unui singur intreg.

Vertex Array Object (VAO)

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). In mod normal inainte de fiecare comanda de desenare trebuie specificate toate comenzile de „binding” pentru buffere sau atribute ce descriu datele ce doresc a fi randate. Pentru a simplifica acesta operatie se foloseste un vertex array object care tine minte toate aceste legaturi.

Un vertex array object este folosind comanda **glGenVertexArrays [https://www.opengl.org/sdk/docs/man4/html/glGenVertexArrays.xhtml]**:

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

Este legat cu **glBindVertexArray [https://www.opengl.org/sdk/docs/man4/html/glBindVertexArray.xhtml]**:

```
glBindVertexArray(VAO);
```

Inainte de a crea VBO-urile si IBO-ul necesar pentru un obiect se va leaga VAO-ul obiectului si acesta va tine minte automat toate legaturile specificate ulterior.

Dupa ce toate legaturile au fost specificate este recomandat sa dea comanda `glBindVertexArray(0)` pentru a dezactiva legatura catre VAO-ul curent, deoarece altfel riscam ca alte comenzi ulterioare OpenGL sa fie legate la acelasi VAO si astfel sa introducem foarte usor erori in program.

Inainte de comanda de desenare este suficient sa legam doar VAO-ul ca OpenGL sa stie toate legaturile create la constructia obiectului.

Transformari 3D

Obiectele 3D sunt definite intr-un sistem de coordonate 3D, de exemplu XYZ. Transformarile de baza sunt: translatii, rotatii si scalari. Aceastea sunt definite in format matriceal, in coordonate omgene, asa cum ati invatat deja la curs. Matricile acestor transformari sunt urmatoarele:

Translatia

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotatia

Rotatia fata de axa OX

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(u) & -sin(u) & 0 \\ 0 & sin(u) & cos(u) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotatia fata o axa paralela cu axa OX

Rotatia relativa la o axa paralela cu axa OX se rezolva in cel mai simplu mod prin:

1. translatarea atat a punctului asupra carui se aplica rotatia cat si a punctului in jurul caruia se face rotatia a.i. cel din urma sa se afle pe axa OX.
2. rotatia normala (in jurul axei OX),
3. translatarea rezultatului a.i. punctul in jurul caruia s-a facut rotatia sa ajunga in pozitia sa initiala

Similar se procedeaza si pentru axele paralele cu OY si OZ.

Scalarea

Scalarea fata de origine

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Daca $sx = sy = sz$ atunci avem scalare uniforma, altfel avem scalare neuniforma.

Scalarea fata de un punct oarecare

Scalarea relativa la un punct oarecare se rezolva in cel mai simplu mod prin:

1. translatarea atat a punctului asupra caruia se aplica scalarea cat si a punctului fata de care se face scalarea a.i. cel din urma sa fie originea sistemului de coordonate.
2. scalarea normala (fata de origine),
3. translatarea rezultatului a.i. punctul fata de care s-a facut scalarea sa ajunga in pozitia sa initiala

Lantul de transformari OpenGL

Spatiul Obiect

Spatiul obiect mai este denumit si **SPATIUL COORDONATELOR LOCALE**

Pentru a putea lucra mai eficient si a reutiliza obiectele 3D definite, in general fiecare obiect este definit intr-un sistem de coordonate propriu. Obiectele simple sau procedurale pot fi definite direct din cod insa majoritatea obiectelor utilizate in aplicatiile 3D sunt specificate in cadrul unui program de modelare gen **3D Studio Max**, **Maya**, **Blender**, etc. Definind independent fiecare obiect 3D, putem sa ii aplicam o serie de transformari de rotatie, scalare si translatie pentru a reda obiectul in scena 3D. Un obiect incarcat poate fi afisat de mai multe ori prin utilizarea unor ***matrici de modelare***, cate una pentru fiecare instanta a obiectului initial, ce mentin transformarile 3D aplicate acestor instante.

In general, fiecare obiect 3D este definit cu centrul (sau centrul bazei ca in poza de mai jos) in originea propriului sau sistem de coordonate, deoarece in acest fel pot fi aplicate mai usor transformarile de modelare. Astfel, rotatia si scalarea fata de centrul propriu sunt efectuate intotdeauna fata de origine.

Spatiul Lume

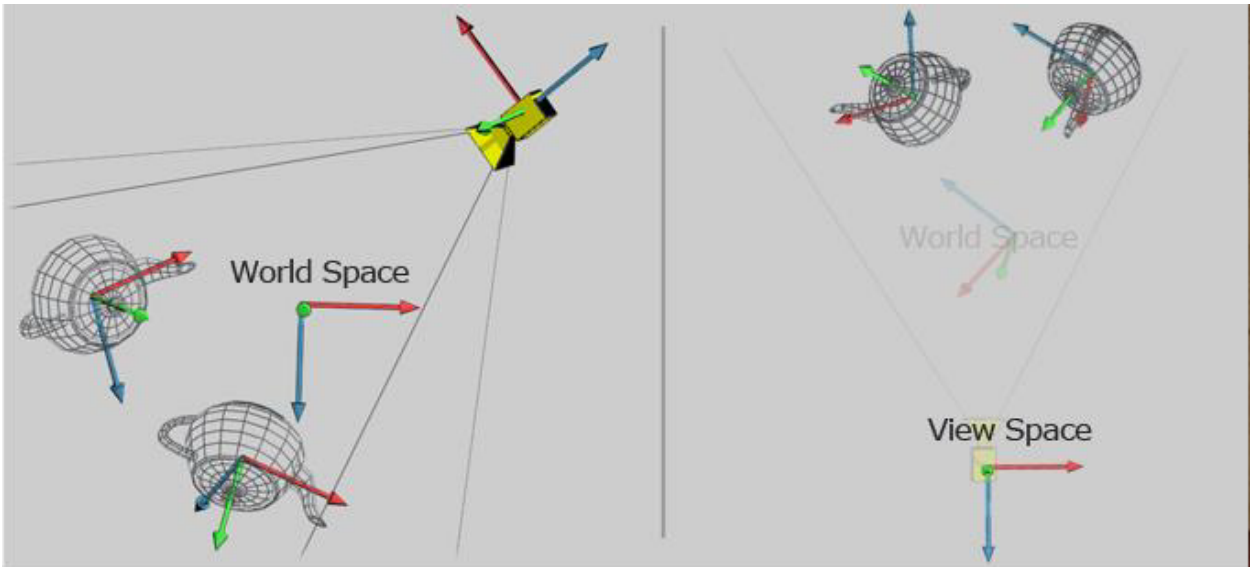
Spatiul lume sau **SPATIUL COORDONATELOR GLOBALE** este reprezentat prin intermediul ***matricii de modelare***, aceeasi despre care s-a vorbit sus. Matricea se obtine printr-o serie de **rotatii**, **scalari** si **translatii**. Prin multiplicarea fiecarui varf al unui obiect (mesh 3D) cu aceasta matrice, obiectul va fi mutat din spatiul local in spatiul lume, adica se face trecerea de la coordonate locale la coordonate globale.

Folosind matrici de modelare diferite putem amplasa un obiect in scena de mai multe ori, in locatii diferite, cu rotatie si scalare diferta daca este necesar. Un exemplu este prezentat in scena din dreapta.

Spatiul de Vizualizare

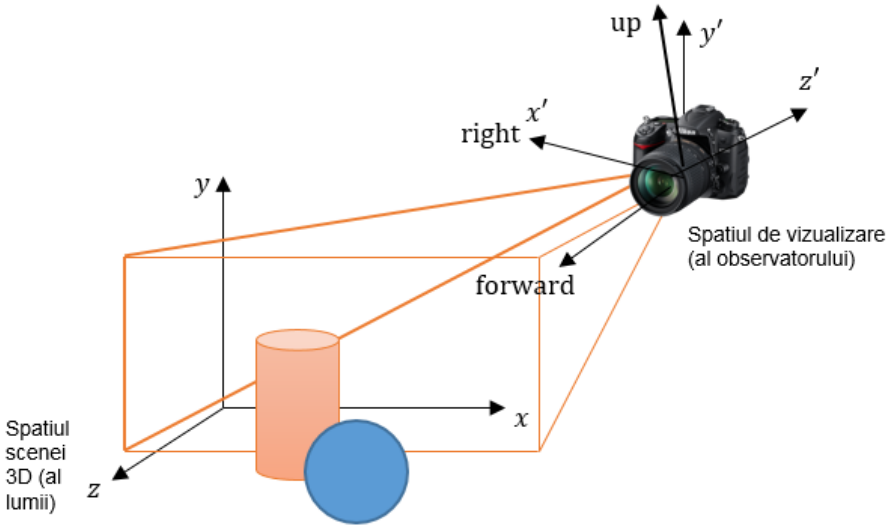
Spatiul de vizualizare sau **SPATIUL CAMEREI** este reprezentat de **matricea de vizualizare**.

Matricea de modelare pozitioneaza obiectele in scena, in spatiul lume. Dar o scena poate fi vizualizata din mai multe puncte de vedere. Pentru aceasta exista transformarea de vizualizare. Daca intr-o scena avem mai multe obiecte, fiecare obiect are o matrice de modelare diferita (care l-a mutat din spatiul obiect in spatiul lume), insa toate obiectele au aceeasi matrice de vizualizare. Transformarea de vizualizare este definita pentru intreaga scena.



In spatiul lume camera poate sa fie considerata ca un obiect avand cele 3 axe locale OX, OY, OZ (vezi poza). Matricea de vizualizare se poate calcula folosind functia `glm::lookAt`.

```
glm::mat4 View = glm::lookAt(glm::vec3 posCameraLume, glm::vec3 directieVizualizare, glm::vec3 cameraUP);
```



Ox,Oy,Oz sunt axele sistemului de coordonate al lumii (spatiul scenei 3D). Punctul O nu este marcat in imagine. O'x',O'y',O'z' sunt axele sistemului de coord. al observatorului (spatiul de vizualizare). Punctul O' nu este marcat in imagine (este inauntrul aparatului).

Vectorul **forward** este directia in care observatorul priveste, si este de asemenea normala la planul de vizualizare (planul fiind baza volumului de vizualizare, ce seamana cu o piramida si este marcat cu contur portocaliu). Vectorul **right** este directia dreapta din punctul de vedere al observatorului. Vectorul **up** este directia sus din punctul de vedere al observatorului.

In imagine, observatorul este un pic inclinat, in mod intentionat, in jos, fata de propriul sistem de axe. Cand observatorul este perfect aliniat cu axele, right coincide cu +x', up coincide cu +y', iar forward coincide cu -z'. In imagine, se poate vedea ca up nu coincide cu +y', iar forward nu coincide cu -z'.

Vectorul "up" se proiecteaza in planul de vizualizare, cu directia de proiectie paralela cu normala la planul de vizualizare. Proiectia acestuia da directia axei verticale a planului de vizualizare.

In spatiul lume camera poate fi considerata un simplu obiect 3D asupra caruia aplicam transformarile de rotatie si translatie. Daca in spatiul lume, camera poate fi pozitionata oriunde si poate avea orice orientare, in spatiul de vizualizare (spatiul observator) camera este intotdeauna pozitionata in (0,0,0), si priveste in directia OZ negativa.

Matricea de vizualizare contine transformari de rotatie si translatie, la fel ca si matricea de modelare. De aceea, daca tinem scena pe loc si mutam camera, sau daca tinem camera pe loc si rotim/translatam scena, obtinem acelasi efect:

„The engines don’t move the ship at all. The ship stays where it is and the engines move the universe around it.”
- Futurama

Totusi, cele doua matrici au scopuri diferite. Una este folosita pentru pozitionarea obiectelor in scena, iar cealalta pentru vizualizarea intregii scene din punctul de vedere al camerei.

Exemplu: Daca vrem sa ne uitam pe axa **OX**(lume) din pozitia (3, 5, 7) codul corespunzator pentru functia glm::lookAt este:

```
glm::lookAt(glm::vec3(3, 5, 7), glm::vec3(1, 0, 0), glm::vec3(0, 1, 0));
```

Spatiul de Proiectie

Dupa aplicarea transformarii de vizualizare, in spatiul de vizualizare camera se afla in origine si priveste inspre **–OZ**. Pentru a putea vizualiza pe ecran aceasta informatie este necesar sa se faca proiectia spatiului vizualizat de camera intr-un spatiu 2D. Cum spatiul vizibil al camerei poate fi de diferite feluri, cel mai adesea trunchi de piramida (**proiectie perspectiva**) sau paralelipiped (**proiectie ortografica**) in OpenGL este necesara trecerea intr-un spatiu final numit spatiu de Proiectie ce reprezinta un ***cub*** centrat in origine cu dimensiunea 2, deci coordonatele X, Y, Z intre +1 si -1.

Din spatiul de proiectie este foarte usor matematic sa obtinem proiectia finala 2D pe viewport fiind nevoie doar sa mapam informatia din cubul [-1,1] scalata corespunzator pe viewport-ul definit de aplicatie.

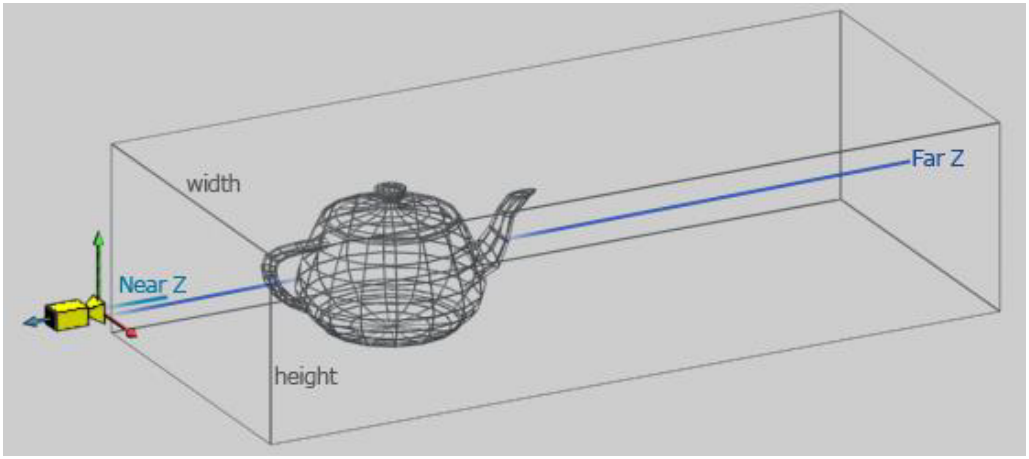
Matricea de Proiectie

Trecerea din spatiul de vizualizare in spatiul de proiectie se face tot utilizand o matrice, denumita **matrice de proiectie**, calculata in functie de tipul de proiectie definit. Biblioteca **GLM** ofera functii de calcul pentru cele mai utilizate 2 metode de proiecte in aplicatiile 3D, anume: proiectia **perspectiva** si **ortografica**

Datele (varfurile din spatiul de vizualizare) sunt inmultite cu **matricea de proiectie** pentru a se obtine pozitiile corespunzatoare din spatiul de proiectie.

Proiectia Ortografica

In proiectia ortografica observatorul este plasat la infinit. Distanta pana la geometrie nu influenteaza proiectia si deci nu se poate determina vizibil din proiectie. Proiectia ortografica pastreaza paralelismul liniilor din scena.



Proiectia ortografica este definita de latimea si inaltimea ferestrei de vizualizare cat si a distantei de vizualizare dintre planul **din apropiere** si planul **din departare**. In afara acestui volum obiectele nu vor mai fi vazute pe ecran.

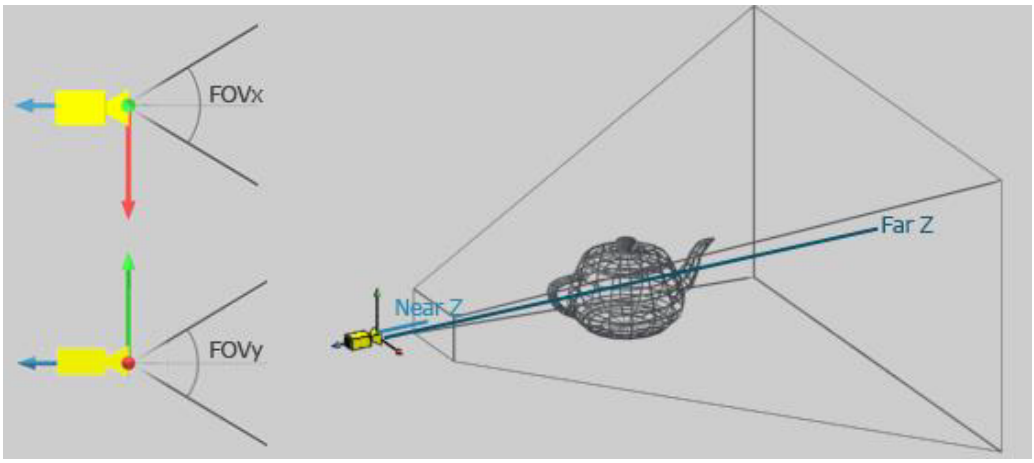
$$\begin{bmatrix} \frac{1}{width} & 0 & 0 & 0 \\ 0 & \frac{1}{height} & 0 & 0 \\ 0 & 0 & -\frac{2}{Z_{far} - Z_{near}} & -\frac{Z_{far} + Z_{near}}{Z_{far} - Z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matricea de proiectie poate fi calculata utilizand functia `glm::ortho` unde punctele **left, right, bottom, top** sunt relative fata de centrul ferestrei (0, 0) si definesc **inaltimea** si **latimea ferestrei de proiectie**

```
glm::mat4 Projection = glm::ortho(float left, float right, float bottom, float top, float zNear, float zFar);
```

Proiectia Perspectiva

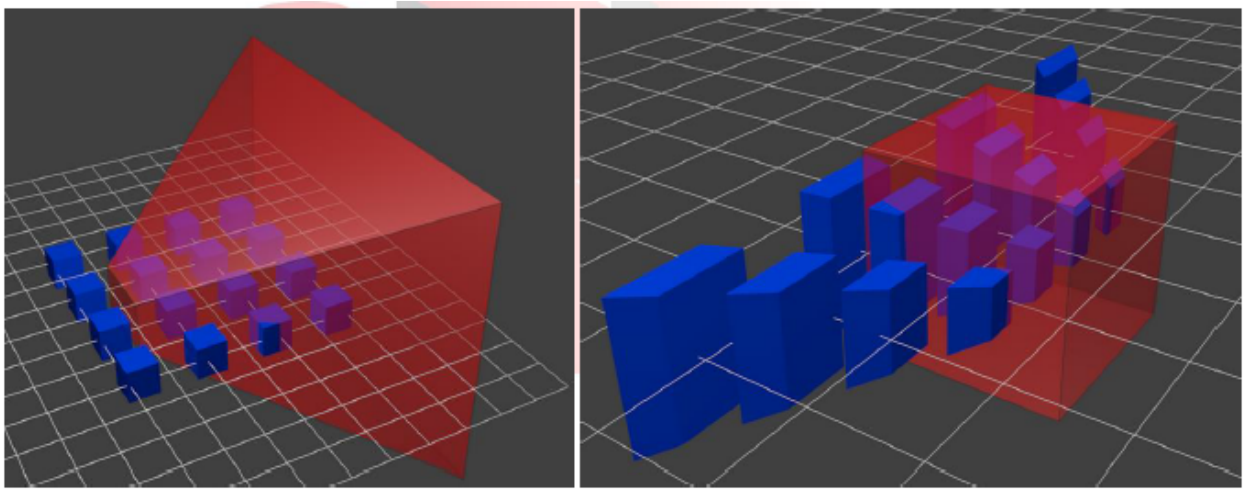
Proiectia perspectiva este reprezentata de un trunchi de piramida (frustum) definit prin cele 2 planuri, **cel din apropiere** si **cel din departare**, cat si de deschiderea unghiurilor de vizualizare pe cele 2 axe, OX si OY. In proiectia perspectiva distanta pana la un punct din volumul de vizualizare influenteaza proiectia.



Matricea de proiectie in acest caz poate fi calculata cu ajutorul funectiei `glm::perspective` ce primeste ca si parametri deschiderea unghiului de vizualizare pe orizontala (**Field of View - FoV**), raportul dintre latimea ai inaltimea ferestrei de vizualizare (**aspect ratio**), cat si distanta pana la cele 2 planuri zFar si zNear.

$$\begin{bmatrix} \tan^{-1}(\frac{FOVx}{2}) & 0 & 0 & 0 \\ 0 & \tan^{-1}(\frac{FOVy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{Z_{far} + Z_{near}}{Z_{far} - Z_{near}} & -\frac{2\,(Z_{near}\,Z_{far})}{Z_{far} - Z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

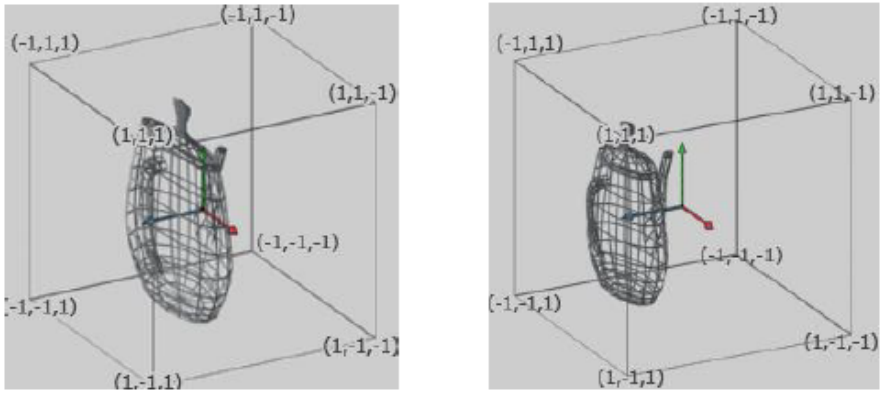
In cazul proiectiei perspectiva, dupa inmutirea coordonatelor din spatiul view, componenta w a fiecarui vertex este diferita, ceea ce inseamna ca spatiul de proiecte nu e acelasi pentru fiecare varf. Pentru a aduce toti vectorii in acelasi spatiu se imparte fiecare componenta a vectorului rezultat cu **componenta w**. Aceasta operatie este realizata automat de placa video, in cadrul unei aplicatii fiind nevoie doar de inmultirea cu matricea de proiectie.



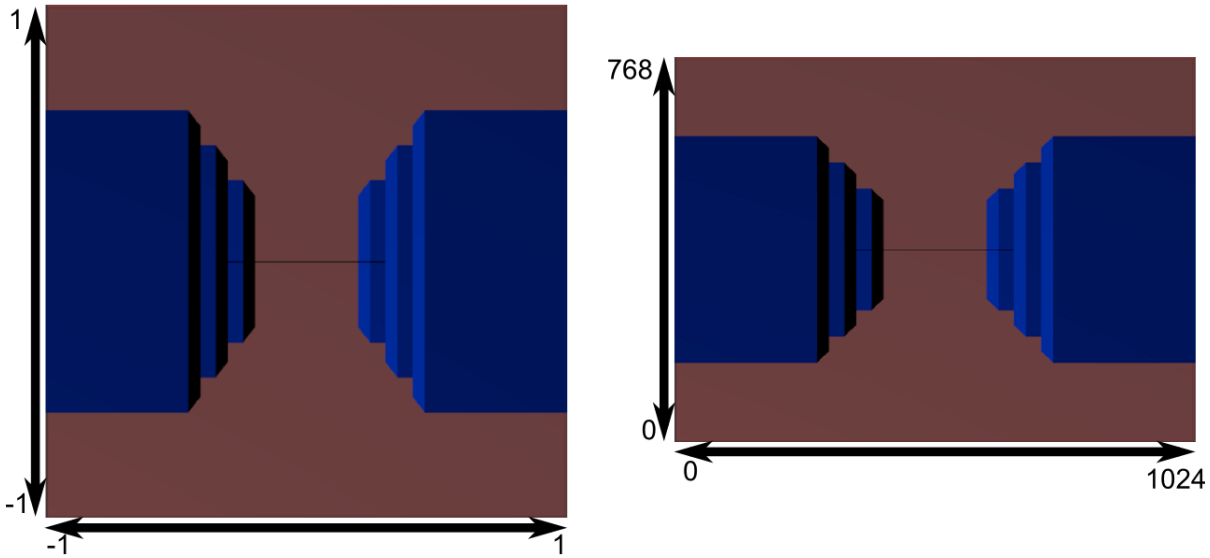
Volum de vizualizare perspectiva (**stanga**) si rezultatul obtinut (**dreapta**) in urma aplicarii transformarii de proiectie asupra geometriei din scena

Spatiul Coordonatelor de Dispozitiv Normalizate (NDC)

Dupa aplicarea transformarilor de **Modelare**, **Vizualizare** si **Proiectie** iar apoi **divizarea cu W** a vectorilor, se obtine spatiul de coordonate normalizate (**NDC**) reprezentat de un CUB centrat in origine (0, 0, 0) cu latura 2. Informatia din acest cub se poate proiecta foarte usor pe orice suprafata 2D de desenare definita de utilizator.



Exemplu rezultat al proiectiei in coordonate dispozitiv normalizate (**NDC**). Proiectie **ortografica** (stanga), **perspectiva** (dreapta)



Exemplu vizualizare **spatiu NDC** din directia camerei (**stanga**) si **proiectia** corespunzatoare pentru un anumit viewport (**dreapta**)

Aplicarea Transformarilor de Modelare, Vizualizare si Proiectie

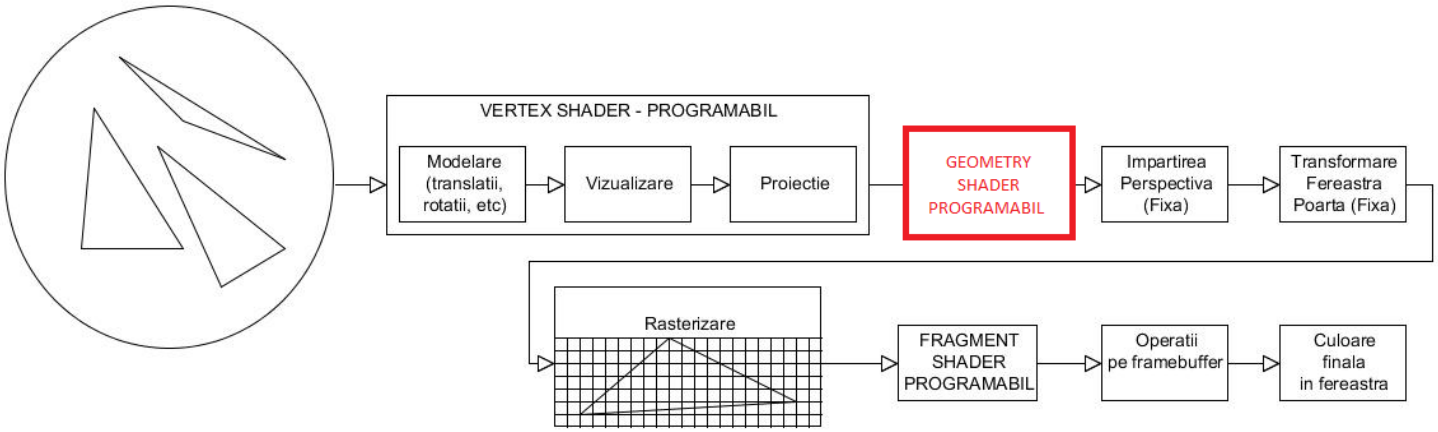
Aplicarea trasformarilor de **Modelare, Vizualizare si Proiectie** se face prin inmultirea fiecarui varf al geometriei din scena cu cele 3 matrici calculate.

```
pos_vertex = Projection * View * Model * pos_vertex
```

Banda Grafica

Banda Grafica este un lant de operatii executate de procesoarele GPU. Unele dintre aceste operatii sunt descrise in programe numite **shadere** (eng. ***shaders***), care sunt scrise de programator si transmise la GPU pentru a fi executate de procesoarele acestuia. Pentru a le deosebi de alte operatii executate in banda grafica, pe care programatorul nu le poate modifica, **shaderele** sunt numite „etape programabile”. Ele dau o mare flexibilitate in crearea de imagini statice sau dinamice cu efecte complexe redate in timp real (de ex. generarea de apa, nori, foc etc prin functii matematice).

Folosind OpenGL sunt transmise la **GPU**: coordonatele varfurilor, matricile de transformare a varfurilor (M: modelare, V: vizualizare, P: proiectie, MV: modelare-vizualizare, MVP: modelare-vizualizare-proiectie), topologia primitivelor, texturi si ale date.



1. In **etapa programabila VERTEX SHADER** se transforma coordonatele unui varf, folosind matricea MVP, din coordonate obiect in coordonate de decupare (eng. *clip coordinates*). De asemenea, pot fi efectuate si calcule de iluminare la nivel de varf. Programul VERTEX SHADER este executat in paralel pentru un numar foarte mare de varfuri.

2. Urmeaza o **etapa fixa**, in care sunt efectuate urmatoarele operatii:

- asamblarea primitivelor folosind varfurile transformate in vertex shader si topologia primitivelor;
- eliminarea fetelor nevizibile;
- decuparea primitivelor la frontiera volumului canonic de vizualizare (ce inseamna? [https://gamedev.stackexchange.com/q/6279]);
- impartirea perspectiva, prin care se calculeaza coordonatele dispozitiv normalizate ale varfurilor: xd = xc/w; yd = yc/w;zd = zc/w, unde [xc,yc,zc,w] reprezinta coordonatele unui varf in sistemul coordonatelor de decupare;
- transformarea fereastra–poarta: din fereastra (-1, -1) – (1, 1) in viewport-ul definit de programator.

3. Urmatoarea etapa este **Rasterizarea**. Aceasta include:

- calculul adreselor pixelilor in care se afiseaza fragmentele primitivelor (bucatele de primitive de dimensiune egala cu a unui pixel);
- calculul culorii fiecarui fragment, pentru care este apelat programul **FRAGMENT SHADER**
- in etapa programabila **FRAGMENT SHADER** se calculeaza culoarea unui fragment pe baza geometriei si a texturilor; programul **FRAGMENT SHADER** este executat in paralel pentru un numar mare de fragmente.
- testul de vizibilitate la nivel de fragment (algoritmul z-buffer);

- operatii raster, de exemplu pentru combinarea culorii fragmentului cu aceea existenta pentru pixelul in care se afiseaza fragmentul.

Rezultatul etapei de rasterizare este o **imagine** memorata intr-un tablou de pixeli ce va fi afisat pe ecran, numit ^^*frame buffer*^^.

Incepand cu a cincea generatie [https://en.wikipedia.org/wiki/List_of_Intel_graphics_processing_units#Fifth_generation] de procesoare video integrate si OpenGL 3.x, intre etapele 2 si 3 exista inca o etapa programabila, numita ***Geometry shader***.

GEOMETRY SHADER-ul este singura etapa programabila ce lucreaza direct la nivel de primitiva, avand access la toate informatiile din toti varfurile primitivei de intrare. Primitivele de input pot fi puncte, linii, triunghiuri sau variantele acestora cu adiacenta (pe care nu le vom folosi). Geometry shader-ul poate primi primitive de un tip si poate scoate primitive de un tip complet diferit!

Shader OpenGL

Pentru implementarea de programe SHADER in OpenGL se foloseste limbajul dedicat GLSL (GL Shading Language).

Legarea unui shader la programul care foloseste OpenGL este o operatie complicata, de aceea va este oferit codul prin care se incarca un shader.

Un **VERTEX SHADER** e un program care se executa pentru **FIECARE** vertex trimis catre banda grafica. Rezultatul transformarilor, care reprezinta coordonata post-proiectie a vertexului procesat, trebuie scris in variabila standard **gl_Position [https://www.opengl.org/wiki/Built-in_Variable_(GLSL)#Vertex_shader_outputs]** 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 v_position;

// Uniform properties
uniform mat4 Model;
uniform mat4 View;
uniform mat4 Projection;

void main()
{
    gl_Position = Projection * View * Model * vec4(v_position, 1.0);
}
```

Un **FRAGMENT SHADER** e un program ce este executat pentru **FIECARE** fragment generat in urma operatiei de rasterizare (ce inseamna? [https://graphicdesign.stackexchange.com/q/260]). Fragment shader are in mod obligatoriu 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);
}
```

Un **GEOMETRY SHADER** e un program ce este executat pentru **FIECARE** primitiva data ca intrare. Geometry shader are in mod obligatoriu o functie numita main. Un exemplu de geometry shader:

```
#version 440
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
in vec2 in_texcoord[];
out vec2 texcoord;

void main(){
     texcoord = in_texcoord[0];
     gl_Position = gl_in[0].gl_Position;
     EmitVertex();

     texcoord = in_texcoord[1];
     gl_Position = gl_in[1].gl_Position;
     EmitVertex();
```

```
    texcoord = in_texcoord[2];
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();
    EndPrimitive();
}
```

`layout(triangles) in` ne spune ca geometry shaderul citeste triunghiuri de la etapele precedente din banda grafica (Vertex shader) iar `layout(triangle_strip, max_vertices =3) out` ne spune ca geometry shaderul trimite mai departe la impartirea perspectiva si apoi rasterizare triangle strip-uri, cu un numar maxim de 3 varfuri.

Cu structura `gl_in[]` putem citi propietatile tinute default pentru fiecare vertex in banda grafica, printre care este si pozitia. Acest lucru nu este valabil pentru toate atributele, pe celelalte trebuie sa le trimitem manual. Dupa cum se poate observa in exemplu, inputul pentru atribute este de tip array iar iesirea este de valoare, consistent cu ideea ca geometry shaderul citeste primitive iar apoi pe baza lor creeaza noi varfuri si topologie noua.

`EmitVertex()` este o comanda ce emite un vertex cu atributele de iesire setate pana la comanda curenta. Ex: gl_Position (pe care trebuie sa il trimitem la iesire) si atributul manual texcoord.

`EndPrimitive()` este o comanda ce semnaleaza terminarea primitivei. Cu aceasta comanda putem crea topologie. In exemplul dat, geometry shaderul trimite la iesire exact varfurile si topologia primita, acest tip de geometry shader fiind numit si „pass-through”. Geometry shader-ul mai poate fi folosit si pentru instantiere, un proces prin care se deseneaza de mai multe ori acelasi obiect cu transformari diferite. Totusi ca geometry shader-ul nu este etapa programabila ideala pentru procesul de amplificare de geometrie, acest proces fiind mult mai eficient cu etapele de teselare (pe care nu le invatam).

Cum legam un obiect geometric la shader?

Legarea intre obiecte (mesh, linii etc.) si shadere se face prin atribute. Datorita multelor versiuni de OpenGL exista multe metode prin care se poate face aceasta legare. In laborator vom invata metoda specifica OpenGL 3.3 si OpenGL 4.1. Metodele mai vechi nu mai sunt utilizate decat in atunci cand hardware-ul utilizat impune restrictii de API.

API-ul OpenGL modern (3.3+) utilizeaza metoda de legare bazata pe layout-uri [https://www.opengl.org/wiki/Layout_Qualifier_(GLSL)]. In aceasta metoda se folosesc pipe-uri ce leaga un atribut din OpenGL de un nume de atribut in shader.

```
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(VertexFormat), (void*)0);
```

Prima comanda seteaza pipe-ul cu numarul 2 ca fiind utilizat. A doua comanda descrie structura datelor in cadrul VBO-ului astfel:

- pe pipe-ul **2** se trimit la shader 3 float-uri (argument 3) pe care nu le normalizam (argument 4)
- argumentul 5 numit si **stride**, identifica pasul de citire (in bytes) in cadrul VBO-ului pentru a obtine urmatorul atribut; cu alte cuvinte, din cati in cati octeti sarim cand vrem sa gasim un nou grup de cate 3 float-uri care reprezinta acelasi lucru
- argumentul 6 identifica offsetul inital din cadrul buffer-ul legat la GL_ARRAY_BUFFER (VBO); cu alte cuvinte, de unde plecam prima oara.

In Vertex Shader vom primi atributul respectiv pe pipe-ul cu indexul specificat la legare, astfel:

```
layout(location = 2) in vec3 vertex_attribute_name;
```

Mai multe informatii se pot gasi pe pagina de documentatie Vertex Shader attribute index [https://www.opengl.org/wiki/Layout_Qualifier_(GLSL)#Vertex_shader_attribute_index].

Pentru mai multe detalii puteti accesa:

- API-ul de OpenGL aici: https://www.opengl.org/sdk/docs/man/ [https://www.opengl.org/sdk/docs/man/]
- API-ul pentru GLSL aici: https://www.opengl.org/sdk/docs/manglsl/ [https://www.opengl.org/sdk/docs/manglsl/]

Un articol despre istoria complicata a OpenGL si competitia cu Direct3D/DirectX poate fi citit aici [https://softwareengineering.stackexchange.com/q/60544].

Cum trimitem date generale 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 obtinem locatia variabilei in programul shader cu functia glGetUniformLocation [https://www.opengl.org/sdk/docs/man4/html/glGetUniformLocation.xhtml]:

```
int location = glGetUniformLocation(int shader_program, "uniform_variable_name_in_shader");
```

- **shader_program** reprezinta ID-ul programului shader compilat pe placa video
- in cadrul framework-ului de laborator ID-ul se poate obtine apeland functia `shader->GetProgramID()` sau direct accesand vriabila membru `shader->program`

Apoi, dupa ce avem locatia (care reprezinta un offset/pointer) putem trimite la acest pointer informatie cu functii de tipul glUniform [https://www.opengl.org/sdk/docs/man4/html/glUniform.xhtml]:

```
//void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value)
glm::mat4 matrix(1.0f);
glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(matrix));

// void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3)
glUniform4f(location, 1, 0.5f, 0.3f, 0);

//void glUniform3i(GLint location, GLint v0, GLint v1, GLint v2)
glUniform3i(location, 1, 2, 3);

//void glUniform3fv(GLint location, GLsizei count, const GLfloat *value)
glm::vec3 color = glm::vec3(1.0f, 0.5f, 0.8f);
glUniform3fv(location, 1, glm::value_ptr(color));
```

Functiile **glUniform** sunt de forma **glUniform[Matrix?]NT[v?]** (regex) unde:

- Matrix - in cazul in care e prezent identifica o matrice
- N - reprezinta numarul de variabile de tipul **T** ce vor fi trimise:
  - **1, 2, 3, 4** in cazul tipurilor simple
  - pentru matrici mai exista si **2×3, 2×4, 3×2, 3×4, 4×2, 4×3**
- T - reprezinta tipul variabilelor trimise
  - **ui** - unsigned int
  - **i** - int
  - **f** - float
- v - datele sunt specificate printr-un vector, se da adresa de memorie a primei valori din vector

Comunicarea intre shadere-le OpenGL

In general pipeline-ul programat este alcatuit din mai multe programe shader. In cadrul cursului de EGC vom utiliza doar Vertex Shader si Fragment Shader. OpenGL ofera posibilitatea de a comunica date intre programele shader consecutive prin intermendiul atributelor **in** si **out**

In metoda specifica OpenGL 3.3 numele de atribut **attribute_name** trebuie sa fie acelasi atat in Vertex Shader cat si in Fragment Shader pentru a se stie legatura intre input/output.

Vertex Shader:

```
#version 330  // GLSL version of shader (GLSL 330 means OpenGL 3.3 API)

out vec3 attribute_name;
```

Fragment Shader:

```
in vec3 attribute_name;
```

In caz ca avem support pentru GLSL 410 (OpenGL 4.1) se poate specifica si locatia attributului astfel, caz in care doar locatiile vor fi folosite pentru a lega iesirea unui Vertex Shader de intrarea la Fragment Shader si nu numele atributului.

Mai multe detalii se pot obtine de la: Program separation linkage
[https://www.opengl.org/wiki/Layout_Qualifier_(GLSL)#Program_separation_linkage]

Vertex Shader:

```
#version 410  // GLSL 410 (OpenGL 4.1 API)

layout(location = 0) out vec4 vertex_out_attribute_name;
```

Fragment Shader:

```
#version 410

layout(location = 0) in vec4 fragment_in_attribute_name;
```

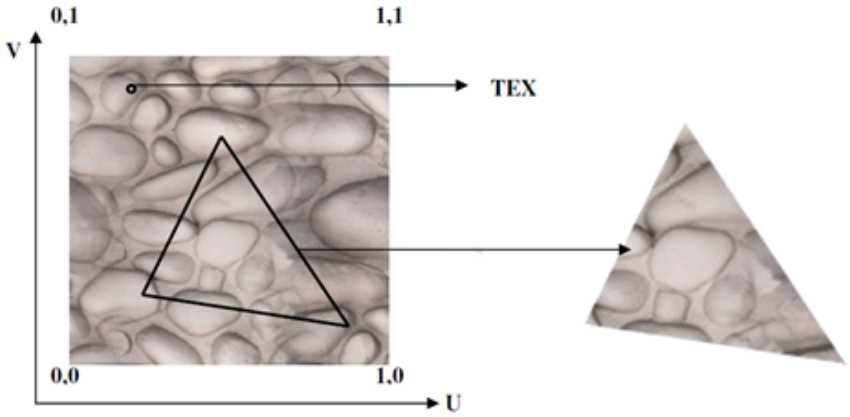
Texturi

Maparea texturilor

Pentru a mapa (impacheta) o textura peste un triunghi, trebuie sa specificam in ce parte din textura corespunde fiecare vertex. Asadar, fiecare vertex ar trebui sa aiba asociata un set de coordonate de textura (2D adica `glm::vec2`) care specifica partea din textura unde isi are locul. Interpolarea intre varfuri se face in *fragment shader*.

Coordonatele de textura se afla in intervalul $[0, 1]$ pentru axele x si y (in cazul 2D). Coordonatele texturii incep din punctul $(0, 0)$ pentru coltul din stanga jos a imaginii pana la punctul $(1, 1)$ care se afla in coltul din dreapta sus.

Un punct de pe imagine si care este in spatiul $[0, 1] \times [0, 1]$ se numeste **texel**, numele venind de la *texture element*. Dupa cum se poate vedea in imaginea de mai jos, in functie de coordonatele fiecarui vertex al triunghiului, partea din textura care este mapata peste triunghi poate fi diferita:



Adaugarea unei texturi

Pentru a construi o textura in OpenGL avem nevoie in primul rand de pixelii imaginii ce va fi folosita ca textura. Pixelii trebuie fie generati functional, fie incarcati dintr-o imagine, iar acest pas este independent de OpenGL. In laborator, pentru a citi texturi, se poate folosi clasa `Texture2D`:

```
Texture2D::Load2D(const char* fileName, GLenum wrapping_mode)
```

unde `wrapping_mode` poate fi:

- `GL_REPEAT`: textura se repeta pe toata suprafata obiectului
- `GL_MIRRORED_REPEAT`: textura se repeta dar va fi vazuta in oglinda pentru repetarile impare
- `GL_CLAMP_TO_EDGE`: coordonatele vor fi intre 0 si 1
- `GL_CLAMP_TO_BORDER`: asemanator cu clamp to edge, doar ca ceea ce se afla dincolo de marginea imaginii nu mai este texturat.

Dupa ce avem pixelii imaginii incarcate putem genera un obiect de tip textura de OpenGL folosind comanda:

```
unsigned int gl_texture_object;

glGenTextures(1, &gl_texture_object);
```

Similar cu toate celelalte procese din OpenGL, nu lucram direct cu textura ci trebuie sa o asociem unui punct de legare. Mai mult, la randul lor, punctele de legare pentru texturi sunt dependente de unitatile de texturare [https://en.wikipedia.org/wiki/Texture_mapping_unit]. O unitate de texturare e foarte similara ca si concept cu pipe-urile pe care trimitem atribute. Setam unitatea de texturare folosind comanda (o singura unitate de texturare poate fi activa):

```
glActiveTexture(GL_TEXTURE0 + nr_unitatii_de_texturare_dorite);
```

Iar pentru a lega obiectul de tip textura generat anterior la unitatea de textura activa folosim punctul de legare `GL_TEXTURE_2D`:

```
glBindTexture(GL_TEXTURE_2D, gl_texture_object);
```

Pentru a incarca datele efective in textura folosim comanda:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
```

- Primul argument specifica tipul de textura. Daca punem `GL_TEXTURE_2D`, inseamna ca aceasta functie va asocia obiectului de tip textura (trecut anterior prin `bind`) o textura 2D (deci daca avem bind-uite un `GL_TEXTURE_1D` sau `GL_TEXTURE_3D`, acestea nu vor fi afectate).
- Al 2-lea argument specifica nivelul de mipmap [https://en.wikipedia.org/wiki/Mipmap] pentru care vrem sa cream imaginea. Vom explica ce este un mipmap pe parcusul laboratorului. Pentru moment, putem sa lasam valoarea aceasta 0.
- Al 3-lea argument specifica formatul in care vrem sa fie stocata imaginea. In cazul nostru este RGB.
- Al 4-lea si al 5-lea argument seteaza marimea imaginii.
- Urmatorul argument ar trebui sa fie mereu 0 (legacy stuff)
- Argumentele 7 si 8 specifica formatul si tipul de date al imaginii sursa.
- Ultimul argument il reprezinta vectorul de date al imaginii.

Asadar, `glTextImage2D` incarca o imagine definita prin datele efective, adica un array de `unsigned chars`, pe obiectul de tip textura legat la punctul de legare `GL_TEXTURE_2D` al unitatii de texturare active la momentul curent, nivelul 0 (o sa luam aceasta constanta ca atare pentru moment), cu formatul intern `GL_RGB` cu lungimea width si cu inaltimea height, din formatul `GL_RGB`. Datele citite sunt de tip `GL_UNSIGNED_BYTE` (adica `unsigned char`) si sunt citite de la adresa data.

Utilizarea texturii

Pentru a folosi o textura in shader trebuie urmat acest proces:

```
glActiveTexture(GL_TEXTURE0);

glBindTexture(GL_TEXTURE_2D, texture1);

glUniform1i(glGetUniformLocation(ourShader.Program, "texture_1"), 0);

glActiveTexture(GL_TEXTURE1);
```

```
glBindTexture(GL_TEXTURE_2D, texture2);

glUniform1i(glGetUniformLocation(ourShader.Program, "texture_2"), 1);
```

Unitatea de texturare este folositoare in momentul in care vrem sa atribuim textura unei variabile uniforme din shader. Scopul acestui mecanism este de a ne permite sa folosim mai mult de 1 textura in shaderele noastre. Prin folosirea unitatilor de texturare, putem face bind la multiple texturi, atat timp cat le setam ca fiind active.

OpenGl are minim 16 unitati de texturare care pot fi activate folosind `GL_TEXTURE0` pana la `GL_TEXTURE15`. Nu este nevoie sa se specifice manual numarul, devreme ce unitatea de texturare cu numarul `X` poate fi activata folosind `GL_TEXTURE0 + X`.

Urmatorul cod este un exemplu de shader ce poate folosi legarea precedenta, unde `texcoord` reprezinta coordonatele de texturare primite ca atribute in vertex shader si apoi pasate catre rasterizer pentru interpolare:

```
#version 330

uniform sampler2D texture_1;

in vec2 texcoord;

layout(location = 0) out vec4 out_color;

void main()
{
        vec4 color = texture2D(texture_1, texcoord);
        out_color = color;
}
```

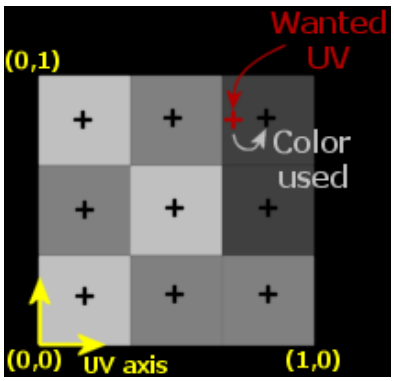
Filtrare

Coordonatele prin care se mapeaza varfurile obiectului pe textura nu depind de rezolutia imaginii, ci sunt valori `float` in intervalul $[0, 1]$, iar OpenGL trebuie sa-si dea seama ce texel (texture pixel) sa mapeze pentru coordonatele date. Pentru a rezolva aceasta problema se foloste filtrarea, care este o metoda de esantionare si reconstructie a unui semnal.

Reconstructia reprezinta procesul prin care, utilizand acesti pixeli, putem obtine valori pentru oricare din pozitiile din textura (adica nu neaparat exact la coordonatele din mijlocul pixelului, acolo unde a fost esantionata realitatea in spatiul post proiectie).

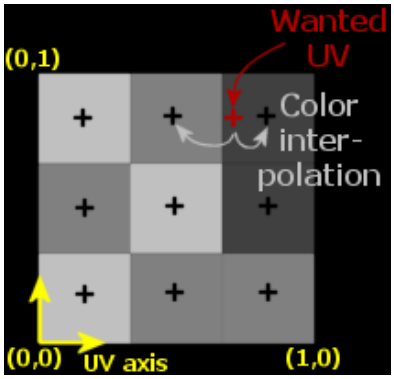
Pentru a face acest proces mai usor, OpenGL are o serie de filtre care pot fi folosite pentru a obtine maparea dorita, iar cele mai des utilizate sunt: `GL_NEAREST` si `GL_LINEAR`.

`GL_NEAREST` (care se mai numeste si *nearest neighbor filtering*) este filtrarea default pentru OpenGL. Cand este folosit acest filtru, OpenGL selecteaza pixelul al carui centru este cel mai aproape de coordonatele de texturare. Mai jos se pot vedea 4 pixeli unde crucea reprezinta exact coordonatele de texturare. Texelul din stanga sus are centrul cel mai aproape de coordonata texturii si astfel este ales:



`GL_LINEAR` (cunoscut drept filtrare biliniara) ia valoarea interpolata din texelii vecini ai coordonatei de texturare, aproximand astfel culoarea mai bine. Cu cat distanta de la coordonata de texturare pana la centrul texelului este mai

mica, cu atat contributia culorii acelui texel este mai mare. Mai jos putem vedea cum pixelul intors



Se pot folosi filtre diferite pentru cazul in care vrem sa marim imaginea si cand vrem sa o micsoram (*upscaling* si *downscaling*). Filtrul se specifica folosind metoda `glTextPameter`:

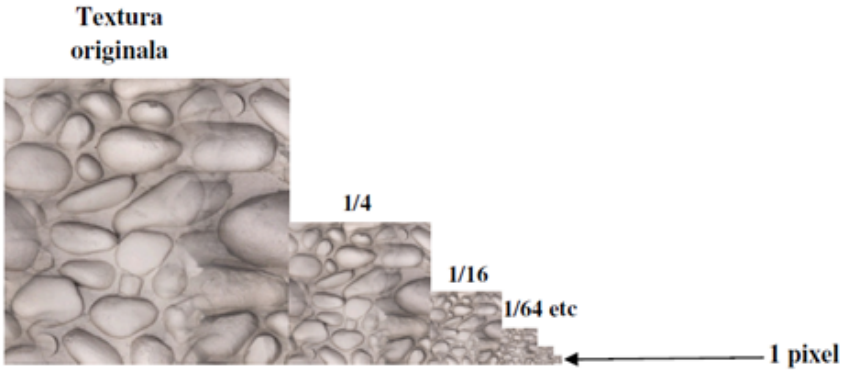
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Mipmaps

Imaginati-va cazul in care avem o camera plina cu obiecte ce folosesc aceeasi textura dar se afla in pozitii diferite. Cele care sunt mai indepartate vor aparea mai mici fata de cele care sunt mai apropiate, dar toate vor avea aceeasi textura de rezolutie mare.

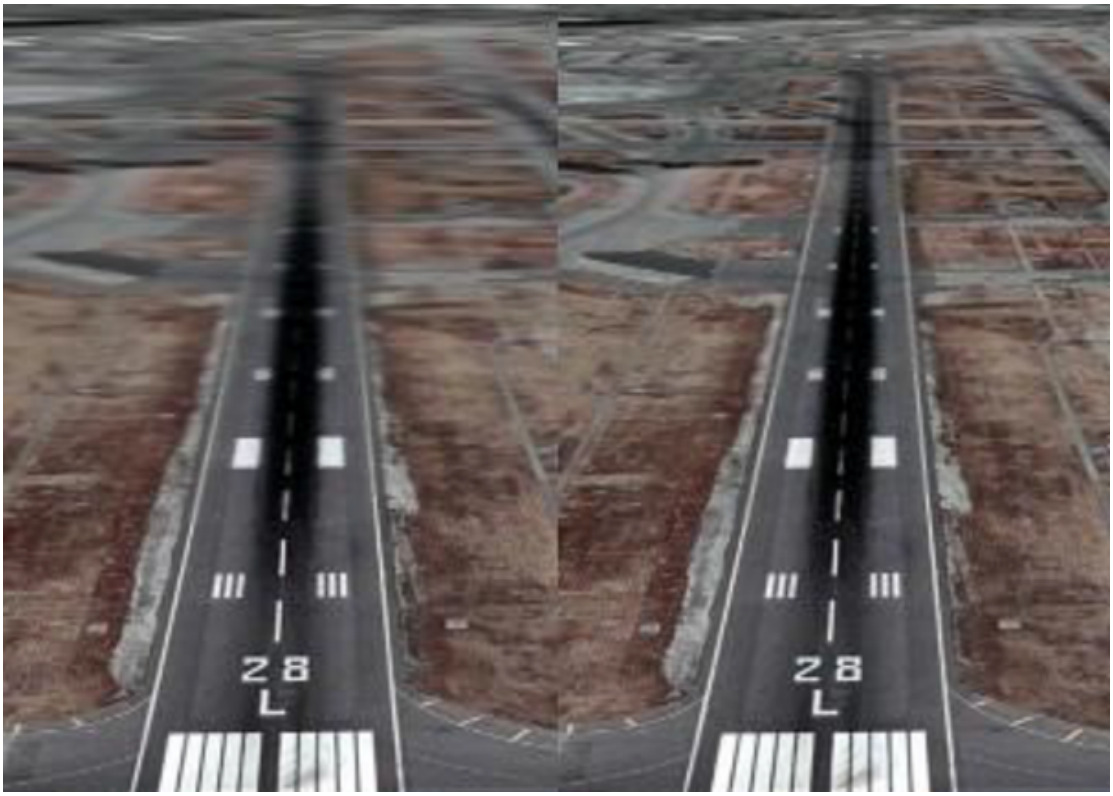
Deoarece obiectele care se afla la departare vor folosi probabil doar cateva fragmente din imaginea de baza, OpenGL va intampina dificultati in obtinerea culorii din textura de rezolutie mare deoarece trebuie sa aleaga culoarea care sa poata reprezenta o portiune foarte mare din textura. Comprimarea unei portiuni mari din textura intr-un singur frament poate duce la artefacte visible pentru obiectele mici, pe langa memoria irosita prin folosirea unei texturi mari pentru obiecte mici.

Pentru a rezolva aceasta problema, se foloseste un concept numit **mipmap**, care este de fapt o colectie de copii ale aceleiasi imagini, unde fiecare copie este de doua ori mai mica decat copia anterioara. Ideea din spatele conceptului de mipmap este destul de simpla: dupa un anumit prag de distanta, OpenGL va folosi o textura mipmap mai mica pentru acel obiect. Fiindca obiectul este la departare, faptul ca rezolutia este mai mica nu va fi observat de utilizator. O textura mipmap arata in felul urmator:



Crearea de texturi mipmaps este destul de anevoioasa de facut manual, asa ca OpenGL poate face tot acest proces in mod automat, folosind functia `glGenerateMipmap(GL_TEXTURE_2D);` dupa ce am creat textura.

Cand privim un obiect dintr-un anumit unghi, se poate ca OpenGL sa faca schimbarea intre diferite niveluri de texturi mipmap, ceea ce poate duce la artefacte asa cum se vede in imaginea de mai jos :



Exact ca filtrarea normala, este posibil sa folosim filtrare intre diferite niveluri de mipmaps folosind filtrare NEAREST si LINEAR atunci cand se produce schimbarea intre niveluri. Pentru a specifica acest tip de filtru, inlocuim filtrarea anterioara cu urmatoarele 4 optiuni :

- GL_NEAREST_MIPMAP_NEAREST : foloseste cea mai apropiata textura mipmap si foloseste interpolare nearest neighbor pentru a alege culoarea.
- GL_LINEAR_MIPMAP_NEAREST : foloseste cea mai apropiata textura mipmap si foloseste interpolare liniara pentru a obtine culoarea.
- GL_NEAREST_MIPMAP_LINEAR : interpoleaza liniar intre cele mai apropiate doua texturi mipmap si si foloseste interpolare nearest neighbor pentru a obtine culoarea
- GL_LINEAR_MIPMAP_LINEAR : interpoleaza liniar intre cele mai apropiate doua texturi mipmap si foloseste interpolare liniara pentru a obtine culoarea.

Exemplu de folosire:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

O greseala comuna este folosirea filtrului de mipmap pentru marimea imaginii. Acest filtru nu va avea niciun efect deoarece texturile mipmap sunt folosite in principal atunci cand obiectele devin **mai mici**. Marirea texturii nu foloseste mipmap si daca dam un astfel de filtru, vom primi o eroare de tipul GL_INVALID_ENUM.

Mai multe informatii si detalii despre filtrare se pot gasi pe pagina API-ului glTexParameter [https://www.opengl.org/sdk/docs/man4/html/glTexParameter.xhtml]