

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IASI

**FACULTATEA DE INFORMATICA**



LUCRARE DE LICENȚĂ

**Game Engine 3D**

propusă de

**Alexandru Ovidiu Scutaru**

**Sesiunea: Iulie 2019**

Coordonator științific

**drd.ing. Vlad Craciun, Prof.dr. Dorel Lucanu**

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IASI

**FACULTATEA DE INFORMATICA**

## **Game Engine 3D**

**Alexandru Ovidiu Scutaru**

**Sesiunea: Iulie 2019**

Coordonator științific

**drd.ing. Vlad Craciun, Prof.dr. Dorel Lucanu**

Avizat,  
Îndrumător lucrare de licență,  
drd.ing. Vlad Craciun, Prof.dr. Dorel Lucanu.

Data: ..... Semnătura: ..... .....

### **Declarație privind originalitatea conținutului lucrării de licență**

Subsemnatul **Scutaru Alexandru Ovidiu** domiciliat în **România, jud. Vaslui, mun. Vaslui, str. Husului, bl.153, sc. B, et. 3, ap. 6**, născut la data de **22.11.1994**, identificat prin CNP **1941122374523**, absolvent al **Universității "Alexandru Ioan Cuza" din Iași, Facultatea de informatică** specializarea **informatică**, promoția 2019, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Game Engine 3D** elaborată sub îndrumarea domnului **drd.ing. Vlad Craciun, Prof.dr. Dorel Lucanu**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: .....

Semnătura: .....

### **Declarație de consimțământ**

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Game Engine 3D**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Alexandru Ovidiu Scutaru**

Data: .....

Semnătura: .....

# Cuprins

<b>Motivație</b>	<b>2</b>
<b>Introducere</b>	<b>3</b>
<b>1 Engine</b>	<b>4</b>
1.1 Funcționalități core	4
1.1.1 Crearea ferestrei	4
1.1.2 Input-ul de la tastatură și mouse	6
1.1.3 Timers	6
1.1.4 Loader de resurse	6
1.2 Grafică	10
1.2.1 Shaders	11
1.2.2 Matrici	14
1.2.3 Skybox	18
1.2.4 Teren	18
1.3 Audio	20
1.4 Fizica	20
<b>2 Editor</b>	<b>22</b>
2.1 Toolbar	23
2.2 Attribute Editor	23
2.2.1 Object Creation	24
2.2.2 Object Placement	24
2.2.3 Skybox	30
2.2.4 Teren	30
2.3 Viewport	31

<b>3</b>	<b>Joc rezultat</b>	<b>36</b>
3.1	Structură . . . . .	36
3.2	Randare . . . . .	37
3.3	Gameloop . . . . .	39
	<b>Concluzii</b>	<b>41</b>

# Motivație

Proiectul de față s-a născut din dorința mea de a înțelege și învăța cum este realizată grafica pe calculator, în special cea 3D. Ca limbaj de programare am ales să folosesc C++ pentru că în primul rând îl stăpânesc bine iar în al doilea rând este mult mai potrivit pentru astfel de proiecte care necesită performanță crescută. Pentru interacțiunea cu procesorul grafic am ales OpenGL pentru că este un API cross-platform.

Inițial am realizat un vizualizator de obiecte 3D, care doar încărca un singur obiect și îl randa. Nu am simțit ca este îndeajuns și astfel am decis să adaug mai multe funcționalități, ajungându-se la o librărie care facilitează dezvoltarea unor jocuri 3D simple, atât din punctul de vedere al codului scris cât și al proiectării grafice a nivelurilor.

# Introducere

Am amintit mai sus că am creat o librărie cu care se pot realiza jocuri simple, acest lucru se numește Game Engine [7]. Ceea ce am realizat eu este doar un *minimum viable product*, deoarece dezvoltarea unui engine generic și complet este o sarcină dificilă și de durată, de exemplu Unreal Engine [44], care este dezvoltat de zeci chiar sute de oameni încă din anul 1995.

Un astfel de program este o agregare de componente complexe precum:

- modul core: creare fereastră, preia input-ul de la tastatură și de la mouse, încarcă informația pentru randare în memoria plăcii video etc
- grafică: desenarea lumii virtuale, a elementelor de interfață grafică
- fizică: simularea gravitației, coliziunea dintre obiecte
- sunet: muzica de fundal, efecte
- suport scriptare: unele acțiuni din joc (o platforma care urcă și coboară etc.) pot fi încărcate la runtime sub formă de scripturi, scrise în LUA [22] de exemplu.

Am încercat să trec prin toate modulele de mai sus, însă accentul a căzut în principal pe partea grafică.

Acest proiect este alcătuit din alte trei sub-proiecte:

- **Engine-ul propriu-zis**: oferă un api abstractizat pentru a facilita interacțiunea cu componentele de mai sus
- **WorldEditor**: este o aplicație cu interfață grafică pentru a crea hărți ce vor fi folosite de jocul final
- **Jocul rezultat**: inițial este un proiect de tip boilerplate care oferă un exemplu minimal de joc, care poate fi îmbunătățit.

În capitolele următoare voi prezenta fiecare sub-proiect în parte.



# Capitolul 1

## Engine

Acest sub-proiect ofera un API ce facilitează utilizarea componentelor majore: grafică, audio, fizică, loader de resurse, input handler etc. Acesta este compilat ca o librărie statică (.lib pe Windows) și este folosit de celelalte două sub-proiecte.

### 1.1 Funcționalități core

#### 1.1.1 Crearea ferestrei

[ceva] Pentru a afișa grafică pe ecran este nevoie de o fereastră. Se poate deschide o fereastră folosind API-ul platformei pe care rulează aplicația, dar dacă în viitor s-ar dori utilizarea și pe alte sisteme de operare este o idee bună să se folosească o librărie care oferă un API care face abstracție de sistem, precum cel oferit de librării ca SDL2, Glut sau GLFW. Eu am ales SDL2 (Simple DirectMedia Library ver. 2) [38].

Listing 1.1: creare fereastră

---

```
SDL_Init (SDL_INIT_VIDEO) ;

SDL_GL_SetAttribute (SDL_GL_RED_SIZE, 8) ;
SDL_GL_SetAttribute (SDL_GL_GREEN_SIZE, 8) ;
SDL_GL_SetAttribute (SDL_GL_BLUE_SIZE, 8) ;
SDL_GL_SetAttribute (SDL_GL_ALPHA_SIZE, 8) ;

SDL_GL_SetAttribute (SDL_GL_BUFFER_SIZE, 32) ;
SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 16) ;
SDL_GL_SetAttribute (SDL_GL_STENCIL_SIZE, 8) ;

SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1) ;

m_sdlWindow = SDL_CreateWindow (
```

```

    name.c_str(),
    SDL_WINDOWPOS_CENTERED,
    SDL_WINDOWPOS_CENTERED,
    width,
    height, a
    SDL_WINDOW_OPENGL
);

m_glContext = SDL_GL_CreateContext(m_sdlWindow);

SDL_GL_SetSwapInterval(1);

```

---

Mai sus este prezentat modul în care se folosește SDL2 pentru a crea o fereastră.

Se inițializează librăria SDL2 cu un apel al funcției *SDL\_Init*. Ca parametru este dat un flag ce specifică ce modul să folosească. Implicit este activat și cel ce se ocupă de input-ul de la tastatură (*SDL\_INIT\_EVENTS*).

Se setează attribute ce specifică dimensiunea în biți a fiecărei componente a culorii pixelilor (roșu, verde, albastru și alfa) rezultând 32 de biți per pixel. Analog se setează dimensiunea unor diverși bufferi, folosiți de OpenGL în procesul de randare.

În SDL2 trebuie apelată metoda *SDL\_CreateWindow* pentru a afișa o fereastră. Aceasta primește ca parametri numele ferestrei, poziția de pe ecran, dimensiunile și o mască de biți ce specifică proprietățile acesteia (borderless, fullscreen, context opengl).

Pentru ca OpenGL să fie folosit cu noua fereastră are nevoie de un *context OpenGL* [26]. Acest context conține informații despre o instanță de OpenGL, printre care și *framebuffer-ul* (mai multe informații în secțiunea despre grafică) pe care acesta va desena. Contextul este creat utilizând metoda *SDL\_GL\_CreateContext*, care ia ca parametru un pointer la fereastra creată mai sus.

Acum se poate afișa pe fereastră ceea ce procesorul grafic randează, dar acest lucru nu este dorit pentru că elementele apar pe ecran în mod secvențial în funcție de când au fost terminate de randat. Pentru a afișa tot în același timp se utilizează un proces numit *double buffering*, în care se alocă spațiu pentru două ferestre. Procesorul grafic desenează o imagine completă pe fereastra din spate apoi acestea sunt interschimbate și începe să deseneze pe cealaltă, astfel succesiunea de cadre (frame-uri) este fluidă.

Dar dacă frecvența cu care monitorul afisează cadrele pe ecran nu coincide cu rata de cadre a aplicației (framerate) apare fenomenul de *tearing* în care un cadru nou este afișat parțial peste cel anterior. SDL sare în ajutor cu metoda *SDL\_GL\_SetSwapInterval*, care pornește VSync-ul (sinconizarea verticală), astfel este limitat efectul de *screen tea-*

ring.

### 1.1.2 Input-ul de la tastatură și mouse

SDL2 oferă un API prin care se face polling (*SDL\_PollEvent* la evenimente de input, astfel la începutul fiecărei iterații ale buclei principale se verifică pe ce butoane a apăsător utilizatorul, în urma cărora se va lua acțiunea corespunzătoare.

Listing 1.2: Event polling

---

```
SDL_Event e;
while(SDL_PollEvent(&e)) {
    switch(e.type) {
        case SDL_QUIT:
            //utilizatorul a apasat "X" pe fereastra
            break;
        case SDL_MOUSEMOTION:
            //miscare a mouse-ului
            break;
        case SDL_KEYDOWN:
            //tasta apasata
            break;
        case SDL_WINDOWEVENT:
            //event legat de fereastra
            switch(e.window.event) {
                case SDL_WINDOWEVENT_RESIZED:
                    //fereastra a fost redimensionata
                    break;
            }
    }
}
```

---

### 1.1.3 Timers

Orice joc are nevoie de o modalitate de a număra cât timp a trecut de la un anumit eveniment sau de a declanșa o acțiune la un anumit interval sau pentru a seta o valoare ținta a numărului de cadre pe secunda cu care să ruleze aplicația.

Și în acest caz se poate folosi SDL2 pentru a cere numărul de milisecunde (ticks) de când a fost inițializată librăria, *SDL\_GetTicks*. Cu ajutorul acestei proprietăți se pot crea aceste temporizatoare și cronometre.

### 1.1.4 Loader de resurse

Pentru a afișa ceva pe ecran avem nevoie obiecte 3D. Acestea sunt structuri compuse din:

- geometrie (mulțimea poligoanelor din care este alcătuit, sunt folosite triunghiuri sau patrulatere, eu am lucrat cu triunghiuri)

- textură (imaginea cu care va fi desenată geometria)

**Textura** este o imagine simplă care este o mapare a unor coordonate 2D (u,v) la o anumită culoare (*texel*) [42]. Această culoare va fi folosită pentru a desena *fragmentul* [6] corespunzător de pe geometrie. Eu am ales să folosesc *PNG* [31] pentru că este un format compresat lossless. Parsarea este realizată cu o librărie scrisă de *Sean Barrett* și anume *stb\_image.h*[40] care încarcă imaginea într-un tablou de *unsigned char(byte)*.

**Geometria** este un fișier text în format *Wavefront OBJ* [47], ce descrie obiectul în funcție de originea acestuia (spațiu local).

Acesta specifică:

- vertecșii (vârfurile triunghiurilor în funcție de originea spațiului local),
- coordonatele (u,v) de pe textură, corespunzători vertecșilor (este nevoie doar pentru vertecși deoarece la randare se face interpolare pentru culoarea dintre cele trei puncte ce determină un triunghi),
- vectorii normali ce indică în ce direcție este fața unui poligon,
- fețele (trei grupuri ce reprezintă vertex / coordonată(u,v) / vector normal, pentru fiecare punct al triunghiului; sunt folosiți indcși din listele mai sus).

Obiecte de tipul .obj se pot modela în programe gratuite precum Blender.

Listing 1.3: dreptunghi.obj

---

```
# v x y z specifica vertecsi
v -1.0 1.0 0.0
v 1.0 1.0 0.0
v -1.0 -1.0 0.0
v 1.0 -1.0 0.0
# vt u v specifica coordonate ale texturii
vt 0.0 0.0
vt 1.0 0.0
vt 0.0 1.0
vt 1.0 1.0
# vn x y z specifica vectori normali
vn 0.0 0.0 1.0
# f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3 specifica fete
f 1/1/1 3/3/1 2/2/1
f 2/2/1 3/2/1 4/4/1
```

---

Mai sus am specificat că am lucrat doar cu triunghiuri. Dreptunghiul trebuie deci specificat cu două triunghiuri (șase vârfuri), dar totuși în document apar doar patru. Acest lucru se datorează faptului că două vârfuri sunt comune, deci se pot lua doar o

singură dată. Proces care se numește indexare și se folosește de specificatorii pentru fețe (f). Se observă că se refolosesc vertecsi 2 și 3 din primul triunghi la al doilea.

În urma parsării (parsarea acestui format a fost făcută după un tutorial de-al lui *TheBennyBox* [29]) se obține o structură de forma:

---

```
struct mesh_t {  
    std::vector<vec3> pos;  
    std::vector<vec2> uv;  
    std::vector<vec3> norm;  
    std::vector<uint> ind;  
} mesh;
```

---

Nici textura nici geometria nu pot fi procesate de către GPU încă pentru că acestea nu sunt încărcate în memoria video.

Pentru a încărca în memoria video utilizând OpenGL se folosesc următorii pași generali:

- generare resursă în memoria video, acțiune ce returnează un handle
- se leagă handle-ul la starea globală a OpenGL-ului (binding), astfel orice acțiune asupra unei resurse o va modifica pe cea activă
- se încarcă datele în zona de memorie respectivă

#### **Pentru texture:**

Listing 1.4: TextureLoader.cpp

---

```
unsigned char* data = stbi_load("img.png", &width, &height,  
                               &numComp, reqComp);  
glGenTextures(1, &id);  
glBindTexture(GL_TEXTURE_2D, id);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,  
             GL_RGBA, GL_UNSIGNED_BYTE, data);
```

---

Mai sus se poate vedea că imaginea este stocată într-un tablou de octeți, iar atunci când este încărcată în V-RAM folosind *glTexImage2D* este specificat faptul că este o textură 2D, nivelul în ierarhia de mipmap (imagini apropiate au calitate buna, cele depărtate mai slabă), că fiecare pixel este RGBA (are patru canale: roșu, verde, albastru și alfa), dimensiunile, grosimea unei borduri și că datele sunt stocate în bytes.

#### **Pentru geometrie:**

Lucrurile sunt puțin mai dificile. Toate informațiile legate de geometrie (structura *mesh\_t* definită mai sus) trebuie stocate într-un *Vertex Array Object (VAO)* [45]. În programul principal există doar un handle la acea zonă. Un VAO conține mai multe liste

de attribute, denumite *Vertex Buffer Object (VBO)* [46]. Aceste VBO-uri vor conține lista de vertecși, coordonatele texturii, vectorii normali, lista de indecși și orice alte date care vor fi folosite sub formă de tablouri de numere (eg. `float[SIZE]`, `std::vector<int>` etc).

Listing 1.5: ModelLoader.cpp

---

```
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);
glGenBuffers(NUM_BUFFERS, vboIDs);

glBindBuffer(GL_ARRAY_BUFFER, vboIDs[POSITION_VB]);
glBufferData(GL_ARRAY_BUFFER, sizeof(mesh.pos[0]) * mesh.pos.size(),
             &mesh.pos[0], GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, vboIDs[TEXCOORDS_VB]);
glBufferData(GL_ARRAY_BUFFER, sizeof(mesh.uv[0]) * mesh.uv.size(),
             &mesh.uv[0], GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, 0);

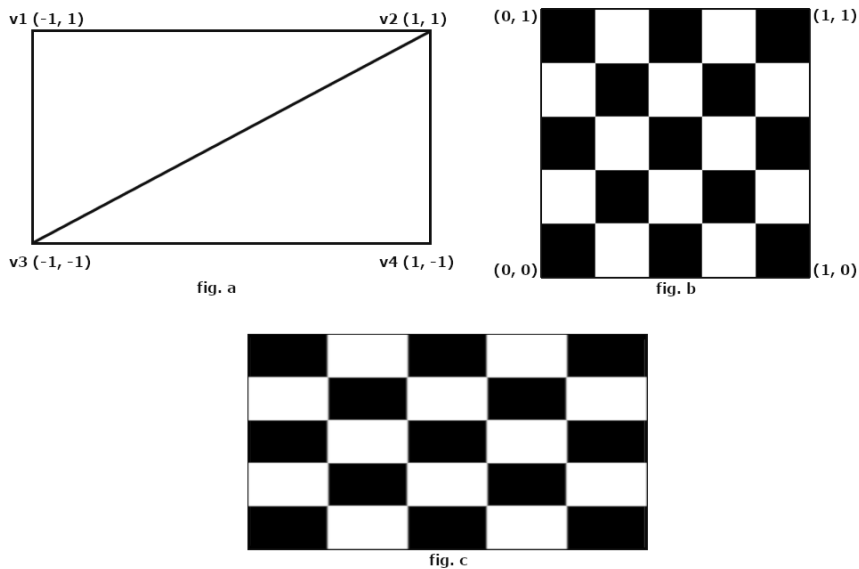
glBindBuffer(GL_ARRAY_BUFFER, vboIDs[NORMAL_VB]);
glBufferData(GL_ARRAY_BUFFER, sizeof(mesh.norm[0]) * mesh.norm.size(),
             &mesh.norm[0], GL_STATIC_DRAW);
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIDs[INDEX_VB]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(mesh.ind[0]) * mesh.ind.size(),
             &mesh.ind[0], GL_STATIC_DRAW);
```

---

Pentru o stoca un tablou de date se folosește `glBufferData` și se specifică tipul de VBO, dimensiunea totală în bytes, adresa primului element, și un flag, în cazul de față `GL_STATIC_DRAW` ce îi spune GPU-ului că datele acestea nu se vor modifica prea curând, astfel acesta le va stoca într-o zonă de memorie optimizată pentru operațiuni de citire.

Acum avem obiectul 3D încarcat în V-RAM și gata de randat, pași urmăriti aici [18].



Când geometriei dreptunghi.obj definită mai sus (fig. a) i se aplică o textură în carouri (fig. b) rezultă obiectul din fig. c.

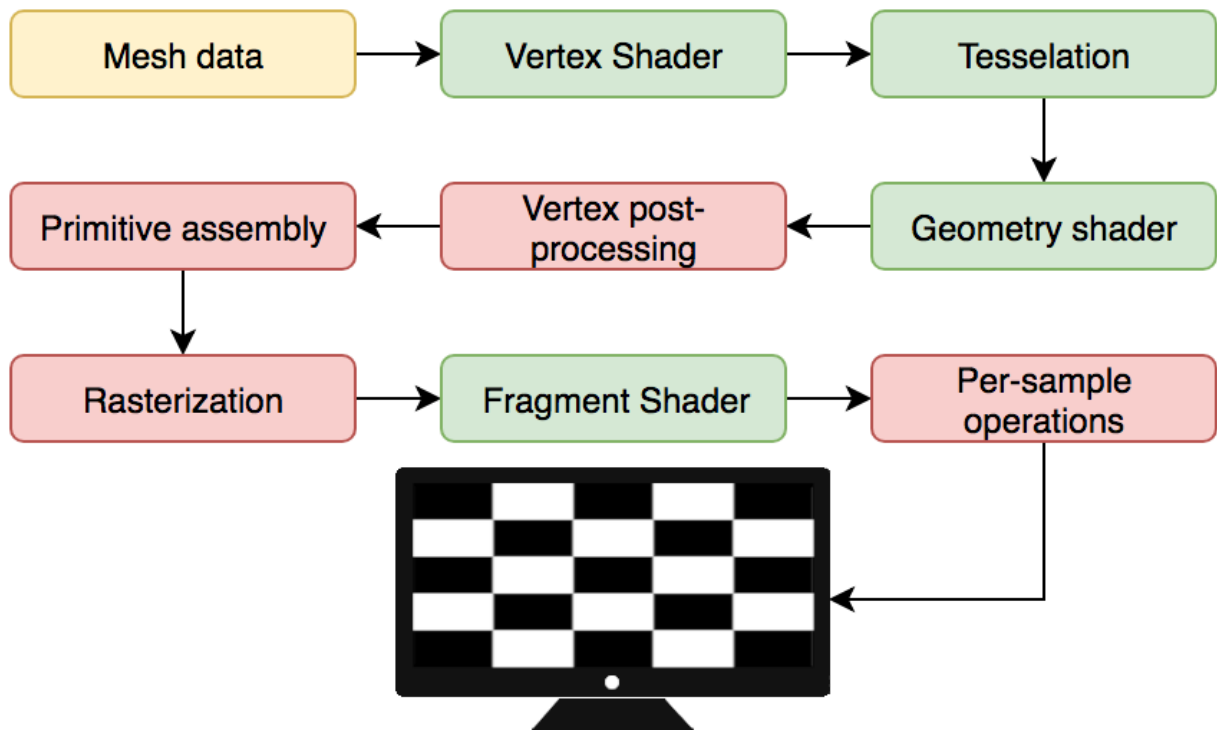
Se observă că textura aplicată obiectului este puțin alungită, este efectul dat de faptul că se face interpolare între pozițiile vertecșilor pentru a găsi culoarea corespunzătoare de pe textură.

## 1.2 Grafică

Partea care se ocupă de grafică se numește Renderer [3].

Până acum avem fereastră cu context, am încărcat obiecte 3D în memoria video, acum trebuie randate pe ecran. Pentru a putea interacționa cu procesorul grafic este nevoie de un API specializat. OpenGL-ul nu poate fi inclus ca un header normal în care găsim funcții, acestea trebuie încărcate manual prin apeluri specifice platformei. O alternativă este să se folosească o librărie care face acest lucru automat (glew, glut). Eu folosesc *GLEW* [13], care este o librărie header only.

Sarcina OpenGL-ului este aceea de a transforma mulțimea de obiecte 3D în reprezentări 2D pe ecran. Acest proces se realizează în mai mulți pași (**rendering pipeline**). Mai jos sunt evidențiați pe scurt acești pași, mai multe informații [34]



### 1.2.1 Shaders

În figura de mai sus, căsuța cu galben reprezintă datele încărcate deja în memoria video, cele verzi reprezintă programe pe care le putem modifica iar cele roșii sunt stagii în care nu putem interveni. Căsuțele verzi sunt programe care rulează pe procesorul grafic și se numesc *shaders* [19]. Limbajul de programare în care se scriu acestea se numește *GLSL (OpenGL Shading Language)* [28] și este asemănător cu C-ul.

Câteva particularități ale acestui limbaj:

Comunicarea dintre CPU și programele de pe GPU se realizează prin niște tipuri speciale declarate ca fiind *uniform* (variabile globale care își păstrează valoarea de la un apel de randare la altul și care pot fi modificate explicit din aplicația care rulează pe CPU). O variabilă de tipul *textituniform* dacă nu este folosită, sau dacă modul în care este folosită nu schimbă rezultatul shader-ului, atunci aceasta este eliminată la compilare, astfel orice apel care cere locația acesteia (în același sens cu care se poate lua adresa unei variabile din C++) va eșua.

Listing 1.6: Incărcarea unui bool in shader

---

```

//in shader
uniform bool my_boolean;
...
//in aplicatia C++
GLuint location = glGetUniformLocation(program, "my_boolean");
glUniform1i(location, (int)myBool);

```

---



Excepție fac tablourile de vertcși încărcăți deja în memoria video, care se declară astfel:

Listing 1.7: Layout qualifier

---

```
layout (location = 0) in vec3 pos;
layout (location = 1) in vec2 texCoords;
layout (location = 2) in vec3 normal;
```

---

*location* este indexul la care se află VBO-urile (tablourile încărcate mai sus) în VAO.

Shaderele comunică între ele doar în mod secvențial prin valori declarate ca fiind *in* sau *out*. Primul apelat este shader-ul vertex, acesta produce un anumit output care este trimis către geometry și apoi la fragment.

Acum că au fost notate câteva particularități ale limbajului GLSL, voi trece succint prin toate stagiile pipeline-ului. Eu am modificat doar programele de vertex și fragment (o aplicație care folosește OpenGL trebuie în mod obligatoriu să le implementeze pe acestea, programul geometry este opțional (este pass-through în mod implicit).

- **Vertex Shader**

Listing 1.8: Simple vertex shader

---

```
layout (location = 0) in vec3 pos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    gl_Position = projection * view * model * vec4(pos, 1.0);
}
```

---

Se observă că sunt declarate o matrice 3x1 (vec3) (matrice coloană) și trei matrice 4x4 (mat4). Acestea vor fi prezentate în secțiunea următoare. Pe scurt, în urma calculelor vertecșii vor fi mutați în lumea virtuală în relație cu transformările lor, cameră și proiecție(perspectivă, ortogonală).

- **Tesellation**

Acest stagiul poate fi modificat astfel încât să preia geometria și să creeze noi subdiviziuni ale poligoanelor în funcție de cât de apropiat este obiectul. Astfel obiectele cu cât sunt mai apropiate cu atât vor avea mai multe detalii.

- **Geometry Shader**

Acest program primește ca input o primitivă (punct, linie, poligon etc) și poate returna ca output (în funcție de cum este programat) mai multe astfel de primitive. De exemplu pentru firele de iarbă de pe jos se pot trimite doar niște puncte în spațiu iar în acest stadiu fiecare punct poate fi transformat într-un obiect 3D ce reprezintă firul de iarbă.

- **Vertex Post-processing și Primitive Assembly**

Până în acest punct procesorul grafic a lucrat doar cu vertecși. În aceste stagii aceștia vor fi asamblați în primitiva dorită. De exemplu ca input avem trei vertecși și dorim să randăm un triunghi. Acest stadiu ia punctele și le uneste formând triunghiul dorit.

- **Rasterization**

Primitivele procesate până în prezent sunt transformate în elemente discrete, numite fragmente (image rastru formată din pixeli).

- **Fragment Shader**

Listing 1.9: Simple fragment shader

---

```
in vec3 TexCoords;

out vec4 FragColor;

uniform sampler2D diffuse;

void main() {
    FragColor = texture(diffuse, TexCoords);
}
```

---

Se observă că primim coordonatele (u,v) de pe textură și returnăm culoarea fragmentului (pixelului).

Dar apare un nou tip de dată și anume *sampler2D*. În acesta se află imaginea 2D încărcată în memoria video. Iar funcția *texture* ia culoarea de pe imagine corespunzătoare coordonatelor. Programul de mai sus este unul foarte simplu, însă acesta poate deveni destul de complex când se vor calcula culorile în funcție de lumină, reflexivitate, umbrire și alte efecte vizuale.

- **Per-sample operation**

După construirea fragmentelor, rezultatul va fi salvat ca o imagine într-un buffer. Implicit ea va fi randată direct pe fereastra noastră, dar poate fi salvată într-un obiect

numit *framebuffer* peste care se pot realiza diverse procesari (tehnica numită post-processing).

Shader-ele sunt specifice jocului care este dezvoltat și sunt fișiere text care sunt încărcate la runtime. Engine-ul expune o clasă abstractă care le încarcă, validează și compilează [2] dar care oferă și metode prin care pot fi încărcate diferite valori în GPU care vor folosi la procesul de randare dorit.

## 1.2.2 Matrici

Pentru a înțelege cine va fi *gl\_Position* din vertex shader-ul de mai sus după efectuarea calculelor, trebuie vazut ce este fiecare matrice în parte, iar pentru aceasta trebuie puțină algebră liniară.

Formatul Wavefront OBJ specifică vertexii obiectului în funcție de originea acestuia. Acest lucru poartă numele de **Object Space**. Fiecare vertex este un vector (ca noțiune matematică, nu tablou de numere), în cazul de sus acesta este format din trei elemente (coordonatele x, y, z). Deoarece se lucrează cu matrice 4x4, acesta trebuie să aibă patru elemente pentru a putea fi înmulțit, astfel este adăugat un al patrulea element.

După fiecare înmulțire se face trecerea la un alt *spațiu de coordonate*.

- **Matricea *model***

La oricare moment, un anumit obiect va trebui să fie într-o anumită poziție, cu o anumită dimensiune și cu o anumită rotație undeva în lumea virtuală. Aici ne folosim de matrice 4x4 pentru a descrie aceste *transformări*.

Translatare:

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} t_x + x \\ t_y + y \\ t_z + z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & t_x + x \\ 0 & 1 & 0 & t_y + y \\ 0 & 0 & 1 & t_z + z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matricea coloană este vectorul ce specifică poziția vertexului relativ la originea spațiului local, elementele  $t_x$   $t_y$   $t_z$  din matrice sunt coordonatele ce specifică unde trebuie mutat (translatat vertexul).

Se observă că în urma înmulțirii rezultă un nou vector ce conține noua poziție. Elementele din acest rezultat se pot include într-o matrice identitate ca mai sus pentru alte eventuale calcule.

Rotatie:

Se pot folosi unghiuri Euler (trei valori ce specifică rotația pe fiecare axă), dar această prezentare suferă de un efect numit *gimbal lock* în care se pierde un grad de libertate. În Engine am folosit *quaternioni* [32] (numere complexe cu patru dimensiuni) însă acestia se pot converti la matrice pentru a calcula rotatia obiectului:

$$quat = \begin{pmatrix} x & y & z & w \end{pmatrix} \rightarrow \begin{pmatrix} 1 - 2 * (y^2 + z^2) & 2 * (xy + wz) & 2 * (xz - wy) & 0 \\ 2 * (xy - wz) & 1 - 2 * (x^2 + z^2) & 2 * (yz + wx) & 0 \\ 2 * (xz + wy) & 2 * (yz - wx) & 1 - 2 * (x^2 + y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scalare:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x * x \\ s_y * y \\ s_z * z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} s_x * x & 0 & 0 & 0 \\ 0 & s_y * y & 0 & 0 \\ 0 & 0 & s_z * z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Similară cu operația de translatare, doar că elementele ce descriu scalarea se află pe diagonala principală.

Înmulțirea matricilor nu este comutativă, deci trebuie o anumită ordine în care se vor efectua calculele.

*model = mat\_translatare \* mat\_rotatie \* mat\_scalare;*

Matricile fiind pre-înmulțite, ordinea transformării vertexului este următoarea: se face scalare, se rotește iar mai apoi se translatează. Acest lucru face posibilă rotirea obiectului având ca pivot originea locală a acestuia, fără a-l deforma. Iar odată rotit este mutat în noua locație. Schimbând ordinea se poate seta un alt pivot de rotație, rezultând efecte de orbitare etc.

Aceste transformări se fac per obiect de fiecare dată când acesta trebuie randat. După înmulțirea matricei model cu poziția relativă la originea vertexului se trece la **World Space**.

Acum toate obiectele se află în lumea virtuală la noile coordonate, cu transformările corespunzătoare.

- **Matricea *view***

În orice aplicație grafică este nevoie de o cameră care specifică unde este și în ce direcție privește utilizatorul. În OpenGL camera este tot timpul în centru iar lumea se mișcă/roteste în jurul acesteia în direcția opusă.

Pentru a realiza acest lucru este nevoie de o matrice 4x4 care definește un sistem de trei axe perpendiculare și poziția ce va deveni noua origine a sistemului.

Folosind funcția *lookAt(position, position + front, up)* din GLM cu parametrii: poziția camerei, direcția în care se uită și vectorul care specifică direcția în sus se obține matricea *view*. După cum se vede avem poziția, vectorul înainte și vectorul în sus. Lipsește al treilea vector, cel ce specifică direcția laterală dreapta a camerei. Acest lucru se calculează folosind produsul vectorial (în engleza cross product) dintre cei doi vectori, care returnează perpendiculara pe planul format de aceștia.

$$view = \begin{pmatrix} r_x & r_y & r_z & p_x \\ u_x & u_y & u_z & p_y \\ d_x & d_y & d_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Camera are 3 tipuri de rotații: pitch - rotație pe axa x, yaw - rotație pe axa y și roll - rotație pe axa z. În proiect cea din urmă nu poate fi alterată. Înmulțirea cu aceasta modifică poziția/rotația întregii lumi relativ la camera. Acum ne aflăm în **Eye Space**.

- **Matricea *projection***

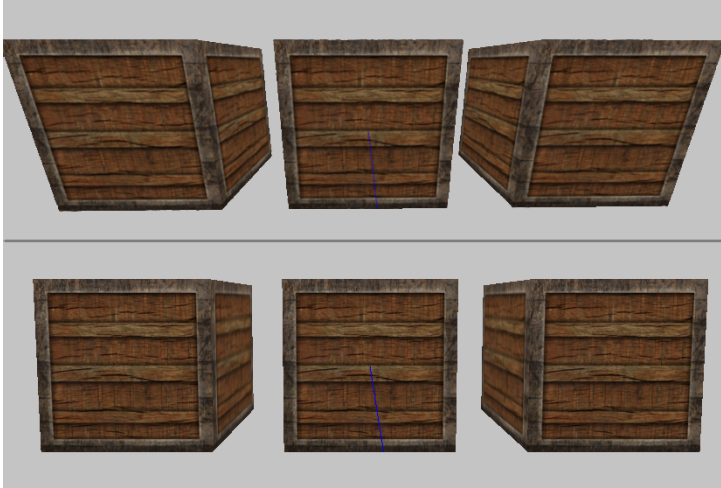
Acum trebuie aplicată o proiecție. Pentru obiecte este aplicată perspectiva, cea ortogonală pretându-se la alte tipuri de aplicații (modelare cad etc). Această proiecție face ca obiectele să pară mai mici cu cât sunt mai depărtate.

```
projection = glm::perspective(fov = radians(90.0f), aspect = width / height, near = 0.1f, far = 1000.0f);
```

$$projection = \begin{pmatrix} \frac{\cot(\frac{fov}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{fov}{2}) & 0 & 0 \\ 0 & 0 & \frac{far+near}{near-far} & \frac{2*far*near}{near-far} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Prin înmulțire rezultă un trunchi de piramidă numit *frustum*. Acum suntem în **Clip space**. Se numește *clip* deoarece tot ce este în interiorul frustum-ului va fi randat, iar ce nu, este *tăiat (clipped)*.

Pentru perspectivă avem nevoie de fov(unghi ce determină câmpul vizual). Folosind funcția de mai sus cu o valoare a fov-ului mai mare rezultă o distorsie de tipul *pincushion*, care se poate rezolva fie setând un unghi mai mic fie aplicând o distorsie inversă, de tipul *barrel* [35]. În figura de mai jos, imaginea din partea de sus evidențiază acest efect de distorsie *pincushion*, iar cea de dedesubt este cum ar trebui de fapt să arate.



Mai sus sunt expuse chestiunile teoretice, atât cât le-am înțeles eu, mai multe informații despre aceste transformări în bibliografie [24], [43].

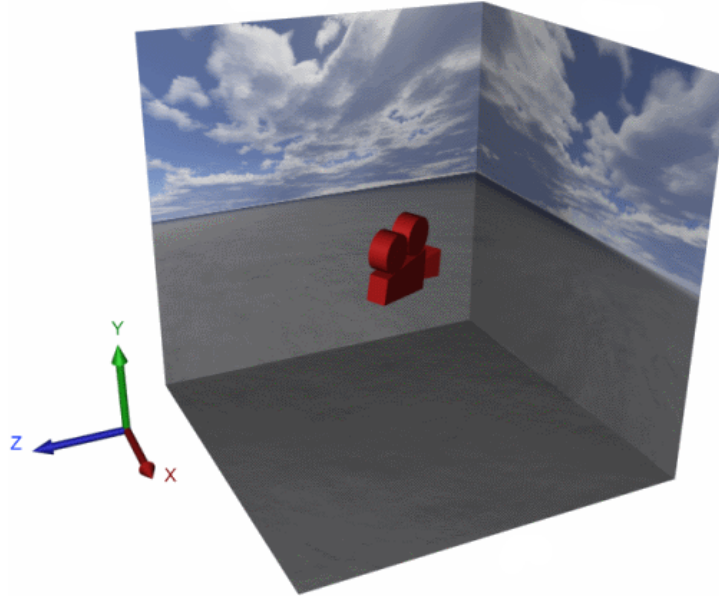
În practică folosesc librăria *GLM (Gl Mathematics)* [12], ce urmează aceeași denumire și mod de folosire al matricelor și vectorilor din limbajul GLSL.

Acum *gl\_Position* conține poziția curentă a unui vertex. Vertex shader-ul este rulat pentru fiecare vertex din lume în paralel.

Înainte ca Fragment Shader-ul să fie executat, OpenGL-ul intervine automat pentru a face încă o transformare și anume cea a spațiului **Normalized Device Coordinates (NDC)**. Are loc procesul de *perspective division* în care fiecare element al poziției este împărțit la componenta *w*(coordonata omogena, în urma calculelor din GLM va deveni *-z*), rezultând valori în intervalul  $[-1.0, 1.0]$ .

### 1.2.3 Skybox

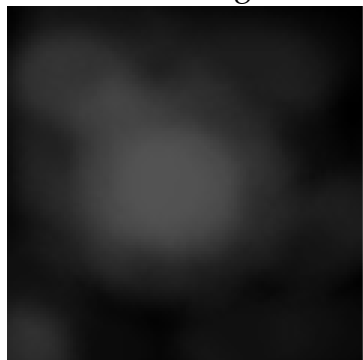
Skybox-ul este un cub cu originea la poziția camerei. Fiecare față din cele șase este texturată cu o imagine ce reprezintă o parte din cer. Este similar cu randarea unui cub normal, doar că de data aceasta se folosește un cubemap (o textură formată din șase imagini) iar camera se află în interiorul acestuia, astfel este simulat cerul în joc. Tehnica se numește *cube mapping* [39].



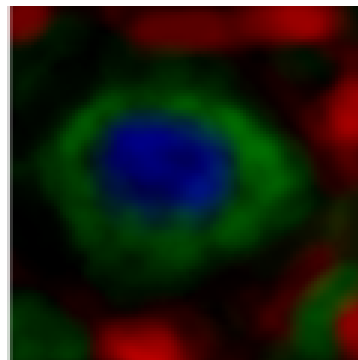
(image: <https://raptor.developpez.com/tutorial/opengl/skybox/>)

### 1.2.4 Teren

Ca orice obiect, terenul este la rândul lui constituit din două componente, textură și geometrie. Ambele sunt generate folosind două texturi.



Heightmap



Blendmap

*Heightmap* este o imagine grayscale care pentru fiecare pixel specifică înălțimea, iar *Blendmap* este o imagine ce specifică cum va fi texturat terenul.

#### 1. Textura

Terenul poate fi texturat folosind patru texturi diferite, corespunzătoare culorilor

canalelor (roșu, verde, albastru) și cea de bază folosită în locurile în care un amestec RGB nu este prezent în totalitate. La coordonatele (x,y) se verifică ponderile canalelor din blendmap apoi se înmulțesc cu valoarea culorii de pe fiecare textură.

Listing 1.10: blendmap

---

```
vec4 blend = texture(blendMap, TexCoords);
float baseAmount = 1 - (blend.r + blend.g + blend.b);
//deseneaza textura de mai multe ori (factor de repetare)
vec2 tiledCoords = TexCoords * tiling_factor;
vec4 base = texture(baseTexture, tiledCoords) * baseAmount;
vec4 red = texture(rTexture, tiledCoords) * blend.r;
vec4 green = texture(gTexture, tiledCoords) * blend.g;
vec4 blue = texture(bTexture, tiledCoords) * blend.b;
vec4 finalColor = base + red + green + blue;
```

---

## 1. Geometria

Se începe cu un pătrat care are numărul de vertecși egal cu numărul de pixeli de pe heightmap. Se parcurge imaginea, se citește valoarea pixelului și se stabilește înălțimea vertexului corespunzător de pe pătrat.

În final se dorește ca terenul să aibă centrul la coordonatele (0, 0, 0) și înălțimea aflată într-un interval [-max, max] specificat.

Listing 1.11: heightmap

---

```
//min, max, temp_h[w*h] calculate mai sus folosind heightmap-ul

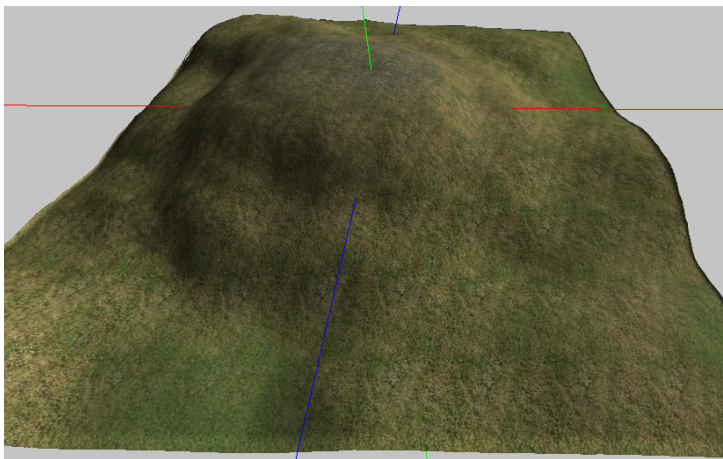
//normalizare inaltime in intervalul [-height_mult, height_mult]
float halfRangeVal = fabs(max - min) / 2.0f;
float medianVal = (max + min) / 2.0f;
min = FLT_MAX, max = FLT_MIN;

for(size_t i = 0; i < temp_h.size(); i++){
    float height = ((temp_h[i] - medianVal) / halfRangeVal) * height_mult;
    heights.push_back(height);
}
```

---

În final rezultă un obiect ca cel de mai jos. Acesta la randul lui poate fi încărcat în engine-ul de fizică.





Modul în care este adăugat terenul este inspirat dintr-un tutorial în Java realizat de *ThinMatrix* [10].

## 1.3 Audio

Pentru audio am folosit librăria *SDL\_Mixer* [36]. Care oferă funcții pentru încărcare de sunete (mp3, ogg etc), redare și setare a numărului de repetiții [37].

Listing 1.12: exemplu *sdl\_mixer*

---

```
Mix_Init(MIX_INIT_MP3 | MIX_INIT_OGG);
Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT, 2, 4096);

Mix_Chunk* chunk = Mix_LoadWAV(filePath);
Mix_PlayChannel(0, m_chunk, loops);

Mix_Music* loadedMusic = Mix_LoadMUS(filePath);
Mix_PlayMusic(m_music, loops);
```

---

## 1.4 Fizica

Simularea fizică este un subiect dificil pe care nu l-am abordat în profunzime. Am ales să folosesc un engine existent aflat în curs de dezvoltare, *ReactPhysics3D* [33]. L-am ales pentru că este ușor de folosit și vine cu o documentație destul de detaliată pentru versiunea la care se află, dar care este îndeajuns pentru nevoile acestui proiect.

În general un engine de fizica este responsabil de modul în care obiectele amintite mai sus interacționează în timp real în lumea virtuală creată. Unui obiect 3D îi este atribuit un *RigidBody* care este o descriere 3D simplificată a obiectului (alcătuită din una sau mai multe corpuri geometrice: cub, sferă, capsulă etc). La un anumit interval

de timp fix se fac mai multe verificări pentru detectare de coliziuni.

1. *broad phase*: fiecărui obiect îi este calculat un *Axis Aligned Bounding Box (AABB)* care este o prismă aliniată cu axele de coordonate, care înglobează întreg *RigidBody-ul*. Apoi este construită o structură de date (quadtree, octree etc) ce reprezintă compartimentarea spațială.

2. *narrow phase*: parsând structura rezultată mai sus se extrag cu ușurință obiectele care au sansă mare să fi avut coliziune. În această verificare nu se mai folosește *AABB-ul* ci se verifică fiecare corp geometric din care este construit *RigidBody-ul*. Dacă se găsește coliziune atunci se rezolvă contactul respectiv (obiectul este mișcat în sens opus în funcție de forța și elasticitate pentru a nu fi suprapunere).

Eu am scris un wrapper care simplifică modul în care este creat un astfel de *RigidBody* care mai apoi poate fi încărcat în acest engine de fizică și care periodic citește starea în care se află (datele care specifică poziția și rotația acestora) pentru a fi actualizate în lumea grafică și randate corespunzător.

# Capitolul 2

## Editor

Acesta este un subproiect care se folosește de Engine-ul prezentat mai sus. Este o aplicație care oferă utilizatorului o interfață grafică (GUI) prin care să își creeze lumea virtuală a jocului pe care vrea să îl dezvolte.

Inițial am folosit *Qt* pentru interfața grafică. Însă cautând și alte alternative am aflat de librăria *ImGui* (*Immediate Mode Graphical User Interface*) [14]. Am ales să o folosesc pe cea din urma pentru ca este destul de ușor de utilizat și integrat. Un alt motiv este acela că nu ea este cea care se ocupă de fereastră, ci doar desenează peste ceea ce Engine-ul randează. Asta înseamnă că atunci când nu este nevoie de GUI, acesta se poate ascunde. *Immediate Mode* nu se referă la modul în care versiunile vechi de OpenGL operau (randarea se făcea prin apeluri succesive de randare ale unor primitive) față de modul de acum în care se folosesc acele VAO-uri. Pentru *ImGui*, *Immediate Mode* se referă la faptul că apelurile de randare ale GUI-ului pot fi grupate într-un stack care este rulat iar rezultatul final este dat către GPU (sub formă de *draw calls batches*).

Listing 2.1: Exemplu dialog setări folosind ImGui

---

```
ImGui::Begin("Grid Settings");
{
    bool boolVal = Grid::isEnabled();
    ImGui::Checkbox("Show Grid", &boolVal);
    Grid::setEnabled(boolVal);

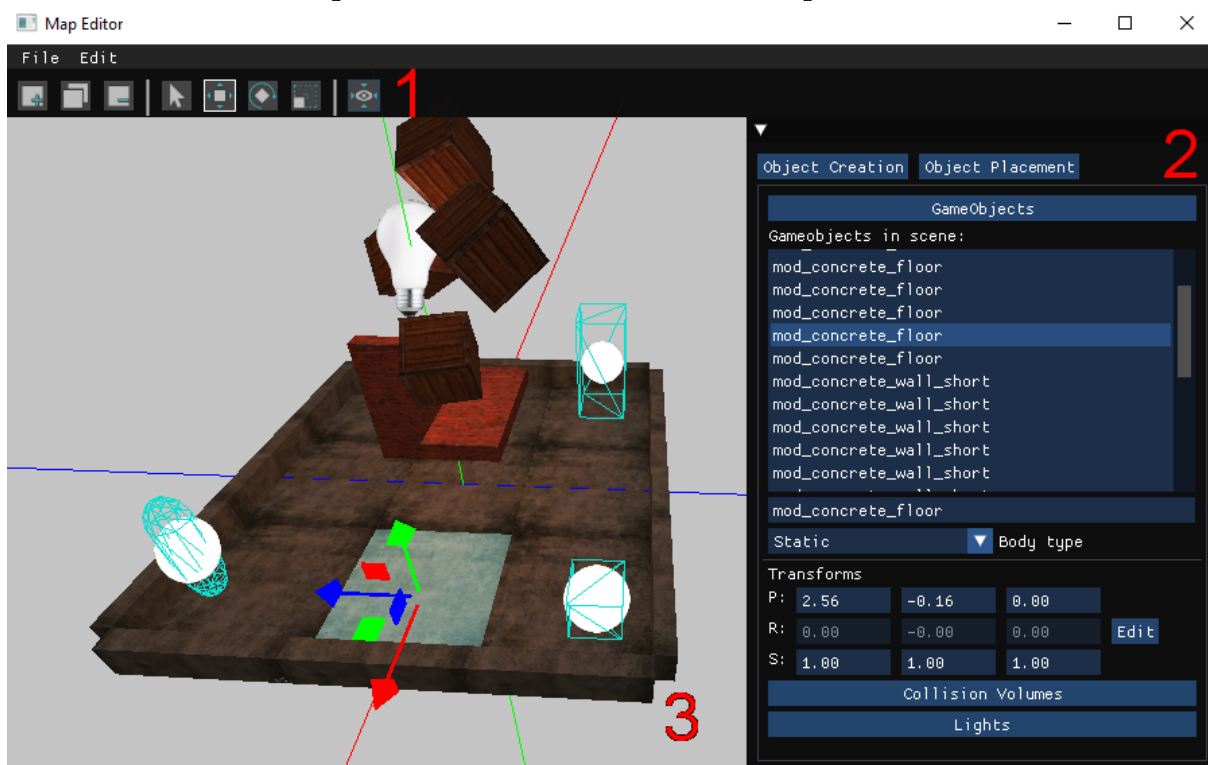
    static int step = 0;
    ImGui::Combo("Grid Step", &step, " 0.16 \0 0.32 \0 0.64 \0\0", 3);
    Grid::setStep(step);

    ImGui::Text("%.3f Grid Height", Grid::getHeight());
    if(ImGui::Button("Decrease")) Grid::decHeight();
    ImGui::SameLine();
    if(ImGui::Button("Increase")) Grid::incHeight();
}
```

```
ImGui::End();
```

Mai sus este prezentat modul general în care este folosită această librărie.

În continuare voi prezenta în ce constă acest editor pentru nivele.



Din imagine se poate vedea că este împărțit în trei secțiuni.

## 2.1 Toolbar

Conține butoane pentru access rapid la diferite funcționalități/unelte.

De la stânga la dreapta: adaugare element nou, duplicare elemente selectate, ștergere elemente selectate, mod doar selecție, mod selecție pentru mutare, mod selecție pentru rotire, mod selecție pentru scalare, centrare camera pe obiectul selectat.

## 2.2 Attribute Editor

Conține controale pentru a defini un obiect pentru joc, modifica atributele elementului pe care se lucrează, adăuga diferite tipuri de lumini, poziții de start și sfârșit, se pot adăuga scripturi pentru a defini comportamentul anumitor obiecte, etc.

Este alcătuit din mai multe pagini diferite.

## 2.2.1 Object Creation

În acest mod se randează un singur obiect ce poate fi creat/modificat și ulterior folosit de joc. Crearea constă în setarea unei geometrii(fișier de tipul OBJ), unei texturi diffuse ce conține culoarea, unei texturi speculare ce conține informații despre cum obiectul va reflecta lumina. Pe lângă parametri vizuali se pot altera și cei ce specifică cum se va comporta obiectul din punct de vedere fizic (elasticitate, coeficient de frecare, corpuri de coliziune pentru *RigidBody*-ul aferent etc.).

În urma salvării rezultă un fișier .JSON [20] ce conține această asociere de parametri cu numele noului obiect.

Listing 2.2: character

```
{
  "angularDamp": 0.0,
  "billboard": false,
  "bounciness": -1.0,
  "collision": [
    {
      "mass": 64.0,
      "pos": [ 0.0, 0.94, 0.16 ],
      "rot": [ 0.0, 0.0, 0.0, 1.0 ],
      "scale": [ 0.56, 1.14, 0.56 ],
      "shape": 4
    }
  ],
  "diff": "character_DIFF.png",
  "friction": -1.0,
  "gravity": true,
  "linearDamp": 0.0,
  "mesh": "character.obj",
  "rollingResist": -1.0,
  "sleep": true,
  "spec": "character_SPEC.png"
}
```

Unul din parametri se numește *billboard*. Aceasta este o tehnică care face ca un obiect să fie tot timpul cu fața spre camera (mai multe informații mai jos). Acest lucru va fi folositor când voi prezenta secțiunea *Viewport* din editor.

## 2.2.2 Object Placement

Elementele sunt împărțite în trei categorii. În funcție de ce grup este selectat, butoanele din bara de unelte amintită mai sus știu pe ce element să facă acțiunea respectivă.

- **Gameobjects**

Acestea sunt obiecte create ca mai sus. Acestea apar într-o listă iar numele din editor poate fi schimbat pentru o navigare mai ușoară prin ele.

Sub această listă se poate selecta tipul fizic de obiect:

-*Static*: sunt obiecte care fac coliziune cu alte obiecte dar nu se pot misca în urma acestora și nici gravitația nu are efect asupra lor (potrivite pentru pereți, podea etc.)

-*Dynamic*: sunt obiecte care sunt afectate de gravitație și care în urma coliziunii își modifică poziția și rotația (potrivite pentru cutii etc.)

Mai jos sunt controale care modifică poziția/rotația/scalarea obiectului selectat.

În imagine se poate observa că la rotație în editor este un caz special și trebuie explicit apasat butonul *Edit*. Acest lucru se datorează faptului că pentru utilizatori este mult mai ușor și natural să seteze rotația cu unghiuri în grade (Euler Angles). Dar acest lucru suferă de efectul de *gimbal lock* [11]. Prin apăsarea butonului edit se specifică unghiuri Euler care mai apoi vor fi transformate în radiani și apoi în Quaternioni [32]. Este utilă această transformare pentru că engine-ul de fizică lucrează tot cu quaternioni. Aceștia sunt numere complexe dar în patru dimensiuni ( $ai + bj + ck + d$ ) care specifică orientarea unui obiect, greu de vizualizat și de înțeles, dar foarte optimi în operații cu ei pentru computere.

În continuare este un câmp *generic*, valoarea acestuia poate fi orice întreg, este acolo pentru a putea diferenția anumite tipuri de Gameobjects (fie pentru a crea diferite tipuri de Gameobjects utilizând Abstract Factory, fie pentru a salva anumite informații legate strict de acel Gameobject, după cum spuneam Engine-ul doar oferă posibilitatea, implementarea este la latitudinea utilizatorului).

La final este o căsuță în care poate fi scris un script simplu în limbajul LUA, ce va fi interpretat de joc pentru a modifica starea Gameobject-ului respectiv, fără a fi nevoie de recompilare.

```

Script
initial_x = -2.56
distance = 2
speed = 1.2

update = function(obj, time)
    local pos = obj:getPos()
    pos.x = initial_x + distance *
        math.sin(time * speed)
    obj:setPos(pos)
end

```

Scriptul din imagine primește ca parametri obiectul pentru care trebuie rulat și timpul de când a fost inițializată librăria, iar în funcție de rezultatul calculului, obiectul va fi mutat de la stânga la dreapta cu viteza specificată.

Pentru a putea rula această funcție la runtime, programul trebuie să expună metodele folosite. Acest lucru se face ca în exemplul de mai jos:

Listing 2.3: c++ api pentru lua

---

```

lua_State* L = luaL_newstate();
luaL_openlibs(L);

//se expune structura vec3 si attributele acesteia
luabridge::getGlobalNamespace(L)
    .beginClass<glm::vec3>("vec3")
        .addData<float>("x", &glm::vec3::x)
        .addData<float>("y", &glm::vec3::y)
        .addData<float>("z", &glm::vec3::z)
    .endClass();

//se expun 2 metode ale clasei GameObject
luabridge::getGlobalNamespace(L)
    .beginClass<GameObject>("GameObject")
        .addFunction("getPos", &GameObject::getPosition)
        .addFunction("setPos", &GameObject::setPosition)
    .endClass();

//se "compileaza" scriptul de mai sus
luaL_dostring(L, obj->updateScript.c_str());
lua_pcall(L, 0, 0, 0);

//se cere functia "update" din tabelul global
luabridge::LuaRef updateFunc = luabridge::getGlobal(L, "update");

//se incearca rularea acesteia, daca nu reuseste va arunca o exceptie
try{
    updateFunc(&obj, SDL_GetTicks() / 1000.0f);
} catch(luabridge::LuaException& ex) {
    std::cout << ex.what() << std::endl;
}

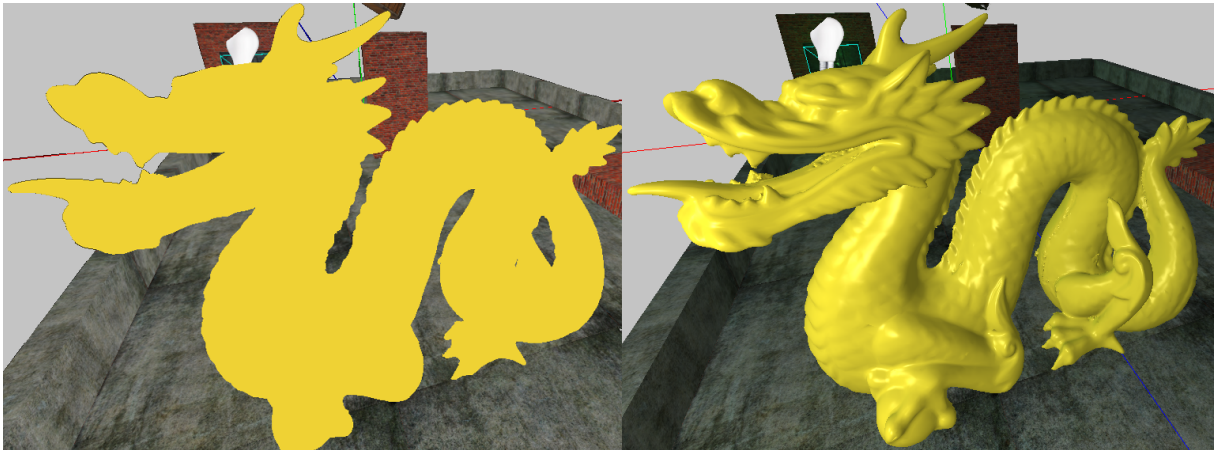
```

---

- **Lights**

O parte foarte importantă din grafica 3D este dată de modul în care este simulată lumina, pentru că schimbă total modul în care va arăta randarea.

Obiectele randate cu metodele de mai sus nu sunt foarte realiste, culoarea lor fiind uniformă. Mai jos se poate vedea în stânga randare simplă iar în dreapta randarea în care se ia în calcul lumina (shading).



Mai sus este evidențiat modelul de lumina *Phong* [30].

Am urmat tutorialul acesta [23] pentru a adauga acest mod de lumina.

Listing 2.4: Phong shading

```
vec3 lightDir = normalize(lightPos - fragmentPos);
vec3 normal = normalize(normal);
vec3 viewDir = normalize(viewPos - fragmentPos);
vec3 reflLightDir = reflect (-lightDir, normal);

vec3 baseColor = vec3 (1.0, 0.0, 0.0);

float amb = 0.1;
float diff = max (dot(normal, lightDir), 0.0);
float spec = pow (max(dot( viewDir, reflLightDir), 0.0), shine);

vec3 resultColor = vec4(amb * baseColor + diff * baseColor + spec * ba
```

Acesta este realizat din trei componente de lumina:

- **Ambient Lighting**

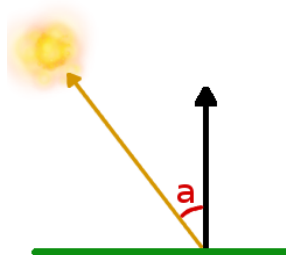
Această tehnică de iluminare se referă la faptul că în condițiile în care nu sunt surse de lumină în apropiere obiectul nu este complet negru, deci are un nivel de culoare minim.

- **Diffuse Lighting**

Modificarea intensității culorii (dă senzația de iluminare) în funcție de unghiul din-

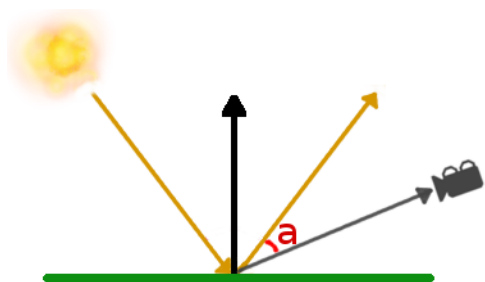


tre razele de lumină și normala feței obiectului. Astfel ne folosim de vectorul normal din fișierul .obj și vectorul care specifică direcția luminei (diferența dintre poziția luminei și poziția fragmentului(pixelul de colorat)). Având aceste informații, se calculează produsul scalar dintre ele care este fixat în intervalul  $[0, 1]$ . Vectorii cu care se lăurează sunt vectori unitate (au lungimea 1), astfel valoarea produsului este chiar  $\cos(\text{unghi})$ . Rezultatul este 1 cand direcția luminei este paralela și în același sens cu normala (unghi de 0 grade) și scade cu cat unghiul crește. Această valoare va fi înmulțită cu valoarea culorii curente pentru a calcula noua culoare a feței (factor de intensitate).



### - Specular Lighting

Acest model de lumină ia în considerare cât de lucios/reflectiv este un obiect. Este nevoie de direcția luminei calculata mai sus (dar în sens opus), de normala feței iar în plus avem nevoie de vectorul de la poziția fragmentului la camera. Direcția luminei este reflectată relativ la normală. Se calculează produsul scalar dintre reflecția luminei și direcția catre cameră. La fel valoarea este 1 cand sunt paralele în aceeași direcție și scade cu cât unghiul este mai mare.



Sunt cazuri în care un obiect nu este la fel de reflectiv pe toată suprafața (ex: o cutie din lemn cu margini din metal), pentru astfel de cazuri se folosește o textură 2D numită *specular map* care conține pixeli grayscale. Dacă este negru nu reflectă deloc iar dacă este alb este foarte reflectiv. Efect evidențiat în imaginea de mai jos.



Engine-ul oferă trei tipuri de lumini: direcțională, spot și point [41].

Toate luminile au trei câmpuri comune. Acestea specifică culoarea luminei, astfel: *ambient* este culoarea ambientală dată de lumina, *diffuse* este culoarea luminei propriu-zise iar *specular* este culoarea luminei reflectată de obiect. În editorul de attribute există controale (built-in de librăria ImGui) care ușurează selectarea culorii dorite. Se poate scrie la mână o anumită culoare (format RGB, HSV dar și HTML) dar are și selecție vizuală.

Însă pentru a putea defini lumini cu comportamente diferite este nevoie de parametri specifici unui anumit tip:

### 1. Direcțională

Aceasta simulează o sursă de lumină aflată la o distanță foarte mare de obiect astfel încât razele ce vin de la ea sunt tratate ca fiind parale. În practică se folosește un singur vector care specifică direcția luminei și care luminează orice obiect din scena din aceeași direcție și cu aceeași intensitate. De regulă se folosește o singură astfel de sursă de lumină (ex. soarele).

### 2. Spot

Este specificată de poziția la care se află, direcția în care luminează și raza unui cerc care reprezintă zona ce este luminată (orice este în afara acestuia nu va fi luminat). Aceasta nu este o lumină infinită, deci este nevoie să se specifice o anumită distanță până la care poate lumina. Se poate folosi o funcție liniară care să scadă intensitatea treptat însă efectul nu este unul foarte bun. Astfel s-a ajuns să se folosească următoarea formulă care atenuează progresiv intensitatea acesteia în funcție de distanță:

$$attenuation = 1.0 / (c + l * distance + q * distance * distance)$$

folosindu-se trei variabile *constant*, *linear*, *quadratic* [21]. Este folosită pentru lumina

de la lanterne, proiectoare etc. Pentru simplitate am folosit o singură astfel de lumină pentru lanternă.

### 3. Point

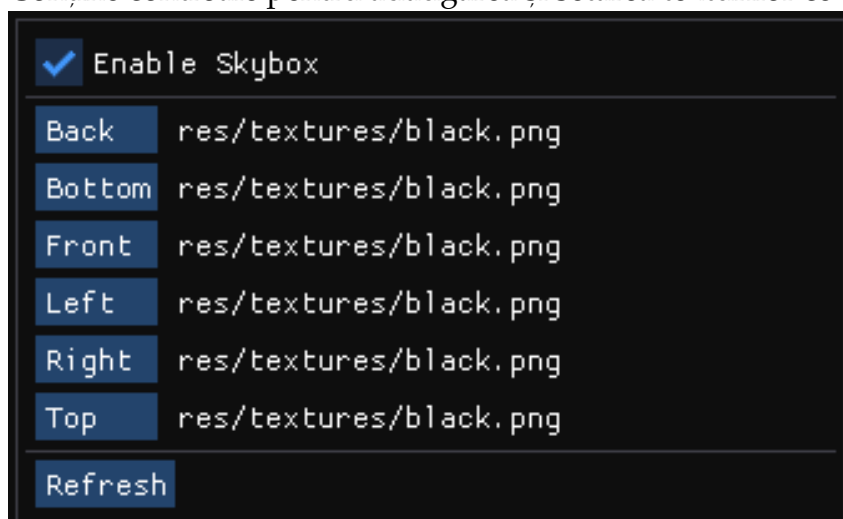
Este specificată de poziție și atenuare, aceasta va lumina în orice direcție atât cât îi permite atenuarea. Folosita pentru lumânări, torțe, becuri etc.

- **Collision Volumes**

Sunt momente în care se dorește ca atunci când player-ul ajunge într-o anumită zonă să se întâmple un anumit lucru (ex. a ajuns la final și trebuie afișat faptul că nivelul a fost completat). Acest lucru se realizează folosind engine-ul de fizică. Se pot crea obiecte 3D invizibile player-ului dar cu care acesta poate interacționa. Acestea se mai numesc și *triggers*. De obicei player-ul nu trebuie să fie obstrucționat de acestea ci engine-ul de fizica trimite un eveniment de coliziune fără a-l rezolva. Însa versiunea folosită de mine nu are această funcționalitate(ori e posibil sa nu știu eu exact cum trebuie folosit), ea trebuie realizată manual. În momentul în care se face coliziune cu un astfel de obiect, evenimentul apare dar player-ul se blochează în el, deci volumul de coliziune trebuie șters la mână din lumea fizica.

## 2.2.3 Skybox

Conține controale pentru adăugarea și setarea texturilor ce vor simula cerul.



## 2.2.4 Teren

Acest tab conține controale cu ajutorul cărora se încarcă resursele specifice terenului (heightmap, blendmap, texturi) și cele pentru a seta înălțimea și factorul de repetare

pentru texturi (tiling factor - un numar care specifică dacă o anumită textură este mapată fie unu la unu, fie parțial fie refolosită de mai multe ori). Se poate alege dacă se dorește simularea unui efect de ceață, care face ca obiectele cu cât sunt mai depărtate să fie mai puțin vizibile. Acest lucru nu este o optimizare, obiectele vor fi procesate dar vor fi randate cu o culoare apropiată de cea a cerului. Este o modalitate de a masca tehnicile de culling (despre care voi vorbi în capitolul următor). Când un obiect este la o distanță mare de cameră, acesta nu este trimis către randare, astfel această ceață face ca obiectele să nu se oprească brusc din randat ci să scadă vizibilitatea astfel încât cele depărtate să poată fi eliminate de la randare fără ca utilizatorul să observe.



## 2.3 Viewport

Această secțiune este pur 3D, exact ca un joc. Navigarea se realizează cu tastele WASD pentru mișcarea înainte-înapoi, stânga-dreapta, QE pentru sus-jos, click dreapta rotește camera iar roțița de la mouse schimbă viteza de deplasare.

Secțiunea *Attribute Editor* se poate ascunde (este collapsable), astfel toată fereastra rămâne doar pentru viewport, lucru care facilitează navigarea prin nivel.

Obiectele nu pot fi selectate, mutate/rotite doar din pagina de editat atributele pentru că acest lucru ar deveni destul de anevoios când se lucrează cu un număr mare de obiecte per nivel. Astfel am implementat și o modalitate de a interacționa cu acestea și în 3D fără folosirea elementelor de GUI.

### Seleție

Implementarea inițială arunca o rază (în engleza raycast) din cameră către poziția mouse-ului în lumea 3D (calcularea în sens invers a matricelor din capitolul anterior) și se verifică dacă pe o anumită distanță intersecta un anumit obiect, detalii de implementare urmate [17]. Asta înseamnă că pentru fiecare obiect din lume trebuia verificată intersecția cu această rază. Dar am avut o problemă, obiectele când sunt încărcate, li se calculează cea mai mică sferă care le înglobează. Pentru scene complexe cu obiecte cu forme mai alungite acest mod de selecție nu este foarte precis.

Am văzut un comentariu pe un forum care aducea ca alternativă randarea fiecărui obiect cu o culoare unică care îi este atribuită la creare. Pentru a selecta este nevoie ca fiecare obiect să aibă un ID unic (pentru editor a fost de ajuns ca de fiecare dată când un obiect randabil este creat, un contor să fie incrementat iar valoarea respectivă să îi fie atribuită). Când se dă click pe viewport se intră într-un caz special de randare și anume:

- se calculează la fel matricea mvp (model, view, projection)
- se folosește un fragment shader special care nu se mai folosește de textură și lumini pentru a desena obiectele ci de acel ID unic:

Listing 2.5: color decoding

---

```
glm::vec3 color = glm::vec3(  
    ((ID & 0x000000FF) >> 0) / 255.0f,  
    ((ID & 0x0000FF00) >> 8) / 255.0f,  
    ((ID & 0x00FF0000) >> 16) / 255.0f  
);
```

---

ID este un int pe 32 de biți, trei cei mai nesemnificativi bytes sunt tratați ca fiind componentele RGB pentru culoare. Mai sus se extrage byte-ul corespunzător iar valoarea este normalizată rezultând în valori în intervalul [0.0, 1.0]. Astfel fiecare obiect are o culoare unică.

- după ce toată scena a fost randată, folosind SDL2 se pot afla coordonatele de la care s-a dat click cu mouse-ul, iar cu aceste coordonate se poate afla culoarea de pe framebuffer (structura pe care OpenGL-ul randează).

## Listing 2.6: color encoding

---

```

unsigned char color[4];
GLint viewport[4];
glGetIntegerv(GL_VIEWPORT, viewport);
glReadPixels((GLuint)coords.x, (GLuint)(viewport[3] - coords.y),
             1, 1, GL_RGBA, GL_UNSIGNED_BYTE, &color);

int index = color[0] << 0 +
            color[1] << 8 +
            color[2] << 16;

```

---

Variabila *viewport* este un array ce conține informații despre bufferul în care OpenGL randează (x, y, width, height). *glReadPixels* citește de la coordonatele x, height - y (y de la SDL2 crește de sus în jos) o regiune de 1x1 pixeli RGBA în array-ul color. Culoarea este apoi recombina într-un int.

- exista o mapare  $\langle int, object \rangle$  și astfel se află în timp constant ce obiect a fost selectat.

- daca tasta CTRL este apăsată se pot selecta multiple obiecte, daca SHIFT este apăsat atunci obiectul pe care s-a dat click este deselectat.

Aici trebuie folosită tehnica de *billboarding* [15].

Acest lucru este ușor de realizat folosind matricele *model* și *view* amintite mai sus. Acestea sunt matrice 4x4 iar în urma calculelor, submatricea 3x3 cu colțul în poziția (0,0) conține informații despre rotație. Dacă înlocuim această submatrice din model cu oglindirea submatricei din view relativ la diagonala principală se obține rotația opusă camerei, astfel obiectul este rotit spre camera tot timpul.

$$model = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix} \quad view = \begin{pmatrix} v_{00} & v_{01} & v_{02} & v_{03} \\ v_{10} & v_{11} & v_{12} & v_{13} \\ v_{20} & v_{21} & v_{22} & v_{23} \\ v_{30} & v_{31} & v_{32} & v_{33} \end{pmatrix}$$

$$newModel = \begin{pmatrix} v_{00} & v_{10} & v_{20} & m_{03} \\ v_{01} & v_{11} & v_{21} & m_{13} \\ v_{02} & v_{12} & v_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix}$$

În plus, valorile din matricea view sunt de așa natură încât după calculul  $view * model$ , această submatrice va fi chiar matricea identitate  $I_3$ . Astfel se observă o a doua metoda pentru billboarding. În shader se schimbă acea submatrice cu identitatea, însă

acel shader îl folosesc și la alte lucruri care nu sunt billboarduri, prima metoda fiind cea aleasă.

Luminile, la fel și volumele de coliziune, nu au un obiect 3D vizibil în joc ci sunt doar niște date. Pentru a putea fi vizualizate în editor, acestora le este atribuit un obiect randabil sub forma de imagine 2D care este rotită tot timpul spre camera. Pentru lumina direcțională este folosită o imagine ce simbolizează un soare, pentru lumina point este folosit un bec iar pentru trigger un disc alb.

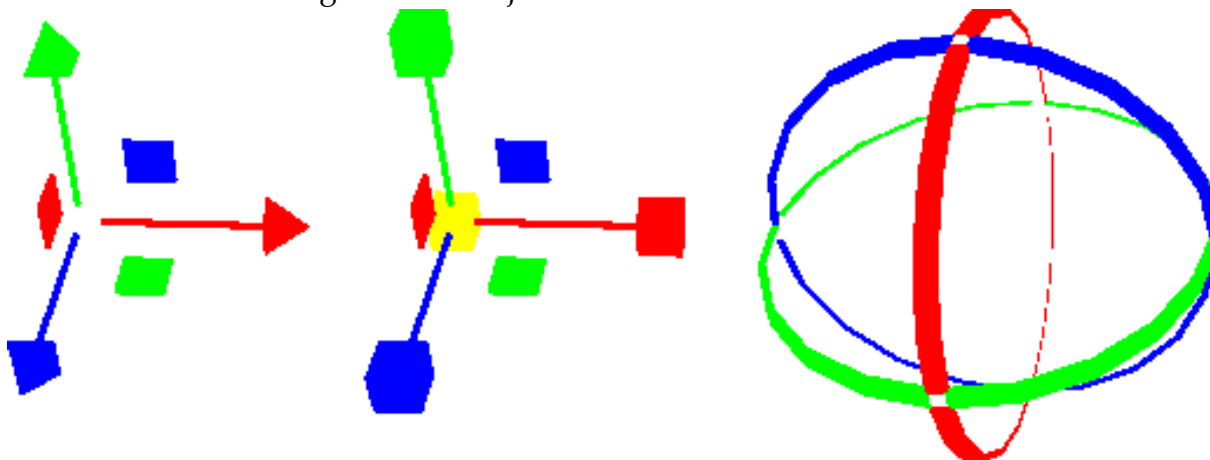
Obiectelor selectate le este adăugată o tentă cyan culorii de bază, fiind mai ușor de observat. În cazul obiectelor billboard, acestora le este adăugată o culoare roșie.

Doar faptul că se pot selecta obiectele direct din viewport nu este chiar folositor, este nevoie și de o modalitate de a le manipula.

### Manipulare

Am încercat să replic controalele din *Maya* (program de modelare și animație 3D) care poartă denumirea de *transform gizmos*, dar neștiind cum acestea sunt făcute am implementat o versiune destul de rudimentară.

În centrul obiectului selectat se vor afla trei obiecte 3D reprezentând axele de coordonate. Ca în imaginea de mai jos:



În funcție de modul de selecție din toolbar acestea au diferite forme: mutare - săgeți, scalare - săgeți cu vârful pătrat, rotație - cercuri.

Atunci când se dă click pe oricare din ele, selecția de mai sus știe că mouse-ul este pe un astfel de *gizmo*. În funcție de rotația camerei și de axa selectată, se calculează un produs scalar astfel încât obiectul să fie mutat/rotit/scalat în direcția în care mouse-ul este tras (funcționează în peste 80% din cazuri).

Gizmo-urile de mutare și scalare au și câte un pătrat pe fiecare plan determinat de două axe. Când acesta este selectat, acțiunea se va face pe cele două axe simultan.

În meniul *Edit -> Grid* se poate deschide un dialog prin care se poate activa și

configura un grid astfel încât în momentul în care sunt mutate obiectele în scenă, acestea să fie aliniate la acesta (grid snapping). În acest mod este ușurată munca alinierii obiectelor ce reprezintă pereți, podea etc.

Când nivelul este gata, acesta se poate salva accesând meniul *File -> Save* într-un fișier .JSON, care este structurat astfel:

Listing 2.7: structura nivel

---

```
{
  "collisionVolumes": [
    {
      "inEditorName": "trigger1",
      "pos": [...],
      "rot": [...],
      "scale": [...],
      "shape": 0,
      "type": 2
    }
  ],
  "gameobjects": [
    {
      "bodyType": 0,
      "inEditorName": "mod_concrete_floor",
      "name": "mod_concrete_floor",
      "pos": [...],
      "rot": [...],
      "scale": [...]
    }
  ],
  "lights": {
    "dirLight": {
      "amb": [...],
      "diff": [...],
      "dir": [...],
      "spec": [...]
    },
    "pointLights": [
      {
        "amb": [...],
        "att": [...],
        "diff": [...],
        "pos": [...],
        "spec": [...]
      }
    ]
  }
}
```

---



# Capitolul 3

## Joc rezultat

Acest proiect este un template care oferă un exemplu minimal de joc ce poate fi dezvoltat folosind Engine-ul. Jocul presupune traversarea unor obstacole (zone înalte peste care jucătorul nu poate sări în mod direct, platforme care se mișcă, găuri în podea etc.) pentru a ajunge la finalul nivelului (acesta este dat de o lumină verde deasupra unei uși), folosindu-se de cutii pe care le poate împinge, chei etc. Acesta este doar un exemplu, modul în care jocul va fi dezvoltat este la latitudinea utilizatorului. Am mai implementat un joc care generează în mod aleator un labirint 3D (parcurs în adâncime) pe care jucătorul trebuie să îl traverseze.

### 3.1 Structură

Structura template-ului este destul de simplă. Obiectele care fac parte din joc urmează modelul bazat pe moștenire.

*Actor*: conține informații de bază cu privire la localizare dar și un ID unic.

*Gameobject*: este un tip de obiect care este fizic și randabil, astfel acesta moștenește din *Actor* dar și din *renderer::RenderableEntity*. Este principalul element, tot ce se poate vedea în joc sunt instanțe ale acestei clase.

*CollisionVolume*: este un obiect doar fizic, ce nu poate fi văzut/randat. Folosit pentru a ști dacă player-ul a făcut coliziune de exemplu cu ușa de la finalul nivelului.

*Player*: obiect special ce poate fi controlat de utilizator dar care este afectat și de lumea fizică dată de engine-ul de fizică (eg. gravitație).

Toate cele de mai sus sunt instanțiate și coordonate de clasa *MainApp*. Aceasta conține mai multe metode care îndeplinesc sarcini specializate, precum:

- `initSystems`: inițializează toate modulele din Engine (grafică, audio, limita FPS(frames per second)).

- `initLevel`: încarcă hărți realizate cu editorul.

- `loop`: aceasta este bucla principală a jocului (*game loop*). Este un subiect puțin mai complex ce va fi descris mai jos, însa pentru acum, aceasta trebuie să realizeze sarcinile de mai jos

- `processInput`: la fiecare iterație a buclei, utilizând SDL2 se face polling și se populează o listă cu input-ul de la utilizator (tastatură și mouse).

- `update`: pe baza input-ului se updatează poziția jucătorului, rotația camerei etc.

- `drawGame`: după ce obiectele au fost updatate se vor randa cu noile transformări.

## 3.2 Randare

În capitolul 1 am precizat că programele care rulează pe GPU (*shaders*) sunt specifice aplicației, așa că Engine-ul oferă doar o interfață ce va trebui implementată de utilizator.

În acest exemplu este implementat un astfel de shader concret pentru *Gameobject*. Acesta conține minimul de funcționalitate pentru randare și Phong shading-ul prezentat mai sus (se poate îmbunătăți pentru rezultatul dorit).

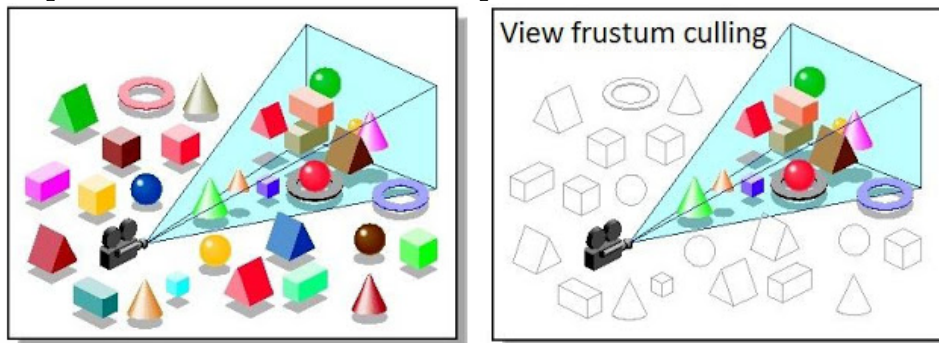
În exemplul de față obiectele se randează în *batch*-uri, nu individual unul după celălalt. OpenGL-ul funcționează pe baza unui *state global* [27], în funcție de care știe pentru ce obiect trebuie aplicate acțiunile ulterioare, astfel acesta știe să deseneze un singur VAO (Vertex Array Object) per apel. Dar apelul de binding (setarea în state-ul global) al VAO-ului ce va fi folosit este o operație costisitoare. Deși GPU-ului în sine este foarte rapid în procesarea datelor deja încărcate (din pricina faptului că acțiunile au loc paralelizat pentru fiecare vertex/fragment), aceste apeluri din CPU ce vizează schimbarea state-ului în repetate rânduri aduc un overhead foarte mare care impactează performanța aplicației. Din acest motiv, înainte de fiecare apel de randare, se procesează lista de obiecte ce urmează să fie randate și se grupează după VAO, construindu-se asocieri de tipul `unordered_map<TexturedModel*, std::vector<GameObject*>>` (care este de fapt un hashmap). La randare se face binding o singură dată pentru VAO-ul de la cheie iar apoi acesta este randat pentru fiecare *Gameobject* din vector cu trans-

formările corespunzătoare.

Listing 3.1: randare folosind batches

```
auto batches = Utils::BatchRenderables<GameObject>(objects);
for(auto const& batch : batches) {
    renderer::Renderer::BindTexturedModel(batch.first);
    for(auto const& gameObject : batch.second) {
        //apply matrix transformations to get the model matrix
        m_gameObjectsShader.loadModelMatrix(modelMatrix);
        renderer::Renderer::DrawTexturedModel(batch.first);
    }
}
```

Dacă scena conține foarte multe obiecte iar CPU-ul le trimite pe toate la GPU pentru randare acesta o să proceseze toți vertexii din lume. Deși GPU-ul se descurcă bine cu astfel de operații, în momentul în care se ajunge la clipospace, vertexii care sunt în afara aceluia spațiu sunt eliminați din procesul de randare. Astfel dacă după cum spuneam sunt foarte multe obiecte sunt șanse mari ca acestea să fie procesate în mod inutil. De aceea este o practică bună să nu se trimită decât ceea ce este vizibil pentru randare. Tehnica se numește *culling*. Sunt mai multe tipuri de culling: occlusion, frustum, back-face. Eu am implementat doar metoda *Frustum Culling*. Frustumul este trunchiul de piramidă dat de câmpul vizual al camerei. Orice obiect care nu este în interiorul acestuia nu este trimis către GPU pentru randare. Pentru a realiza acest lucru, în momentul în care se încarcă un obiect se calculează AABB-ul corespunzător (axis aligned bounding box). Cu ajutorul acestuia se va calcula centrul obiectului iar distanța de la acest centru la un vertex al AABB-ului este raza unei sfere numită bounding sphere. Se folosește sfera pentru că verificarea intersecției unui plan cu sferă este relativ rapidă. Mai multe informații despre această tehnică la articolul [16].



(image: <http://aduartegames.blogspot.com/2016/03/view-frustum-culling.html>)

### 3.3 Gameloop

O astfel de aplicație rulează în interiorul unei bucle cât timp fereastra este deschisă.

În această buclă se întâmplă multe acțiuni care sunt decuplate: polling pentru inputul de la utilizator, update-ul diferitelor obiecte (player, platformă, poziția soarelui (lumina direcțională) etc), simularea fizică a obiectelor, randarea în urma acestei simulări). De preferat este ca aceste lucruri să se întâmple în mod constant. Pe consolele vechi (NES) nu se pune problema pentru că jocul era programat în funcție de viteza procesorului și se știa că playerul merge cu 1 pixel per tick de exemplu. Dar în zilele de azi sunt multe CPU-uri diferite cu viteze diferite, deci nu se poate ca fiecare iterație să dureze exact X milisecunde pentru fiecare frame. Astfel ajungem la noțiunea de *timestep* (cu cât avansează un frame față de altul pentru a menține constantă experiența jucătorului).

Există trei tipuri de astfel de timestep-uri:

**variable** - se specifică o valoare țintă de durată pentru un frame, *desired\_time*. Se calculează raportul *current\_time/desired\_time* iar acesta este folosit ca factor pentru a avansa logica. Dacă acest factor este mare (framerate-ul scade mult sub cel țintă) pot să apară efecte neplăcute, unele coliziuni nu vor fi detectate etc.

**Semi-Fixed** - este o îmbunătățire față de cel variabil, nu se avansează într-un singur pas, ci se execută logica de update de câte ori este nevoie în cadrul aceleiași iterații. Deși pare că este o soluție bună, în cazul engine-urilor de fizică nu se poate folosi, pentru că pot să apară în continuare comportamente neașteptate.

**Fixed** - update-ul fizic se face independent de cel grafic. Acesta se face la un interval fix setat (1/30, 1/60) numit *physics step*. În bucla principală, în primă fază, obiectelor li se setează poziția din lumea fizică. Apoi cât timp se poate, se avansează fizica cu acel *physics step*.

Listing 3.2: update fizica, pas fix

---

```
accumulator += frameTime;
while(accumulator >= PHYSICS_STEP) {
    processInput();
    update(PHYSICS_STEP);
    accumulator -= PHYSICS_STEP;
}
interpolation = accumulator / PHYSICS_STEP;
```

---

La începutul buclei, obiectele din lumea grafică se află la pozițiile corespunzătoare din cea fizică de dinainte de update. După update se obține o valoare pentru interpolare, ce se integrează astfel:

$$new\_pos = current\_physics\_pos * interpolation + world\_pos * (1.0f - interpolation)$$

Acest lucru face ca mișcarea să fie fluidă, altfel obiectele ar fi părut că se teleportează.

Mai multe detalii la [9], [5] și [8].

# Concluzii

Acest proiect a reprezentat pentru mine un exercițiu bun, în urma căruia am învățat multe lucruri. Inițial nu știam ce înseamnă să dezvolti un game engine (fie și unul de bază cum este cel de față). Dar am trecut prin multe arii de la partea de design grafic (modele 3D, texturi), algebra liniară, lucru atât high level cât și low level, puțină fizică și nu în ultimul rând grafică pe calculator folosind direct GPU-ul. Mai mult de jumătate din timpul dezvoltării a fost ocupat documentare. Aici au intrat parcurgerile librăriilor, citirea diferitelor articole ce expuneau diferite tehnici fie pentru randare, lumină, simulare fizică etc, dar și experimentare cu diferite librării și alegerea celei potrivite.

Acest proiect este departe de a fi într-un stadiu final, deoarece sunt foarte multe lucruri care pot fi îmbunătățite sau adăugate.

**Arhitectura.** Este recomandată folosirea unui *ECS (Entity Component System)* [4], care favorizează compoziția peste moștenire. Astfel un obiect are unul sau mai multe componente precum sunet, networking, grafică, modul de scriptare etc, crescând astfel flexibilitatea.

**Animatie.** Sunt obiecte 3D în format *Collada* [1] descrise în XML care pe langa geometrie, au asociat un schelet (tehnica numita *rigging*) și o serie de keyframe-uri care descriu la un anumit moment de timp poziția fiecărui os, în final făcându-se interpolare între valori se pot anima geometriile legate de schelet.

**Scripting.** Ideal ar fi expunerea unui API complet din C++ către interpretorul de limbaj de scripting astfel încât totul să fie controlat integral prin scripturi.

**Sunet 3D.** În jocuri 3D sunetul este și el destul de important. Este nevoie ca localizarea spațială a unei surse de sunet să fie naturală și imediată. Astfel ar trebui folosită librăria *OpenAL* [25] care este similară cu OpenGL, doar ca vizează sunetul. Astfel sunetul se va auzi dintr-o anumită direcție, intensitatea acestuia va scădea în funcție de distanță (atenuare) ba chiar se poate simula și efectul Doppler.

# Bibliography

- [1] *Collada .dae format*. URL: <https://en.wikipedia.org/wiki/COLLADA>.
- [2] *Compiling the shaders*. URL: [https://www.khronos.org/opengl/wiki/Shader\\_Compilation](https://www.khronos.org/opengl/wiki/Shader_Compilation).
- [3] *Computer Graphics Rendering*. URL: [https://en.wikipedia.org/wiki/Rendering\\_\(computer\\_graphics\)](https://en.wikipedia.org/wiki/Rendering_(computer_graphics)).
- [4] *Entity Component System based architecture*. URL: [https://en.wikipedia.org/wiki/Entity\\_component\\_system](https://en.wikipedia.org/wiki/Entity_component_system).
- [5] *Fix your time step by GafferOnGames*. URL: <https://gafferongames.com/game-physics/fix-your-timestep/>.
- [6] *Fragment*. URL: [https://en.wikipedia.org/wiki/Fragment\\_\(computer\\_graphics\)](https://en.wikipedia.org/wiki/Fragment_(computer_graphics)).
- [7] *Game Engine*. URL: [https://en.wikipedia.org/wiki/Game\\_engine](https://en.wikipedia.org/wiki/Game_engine).
- [8] *Gamedev.StackExchange FixedTimestep*. URL: <https://gamedev.stackexchange.com/a/132835>.
- [9] *Gameloop*. URL: <https://gameprogrammingpatterns.com/game-loop.html>.
- [10] *Generare teren folosind textura heightfield de ThinMatrix*. URL: <https://www.youtube.com/watch?v=O9v6olrHPwI>.
- [11] *Gimbal Lock*. URL: [https://en.wikipedia.org/wiki/Gimbal\\_lock](https://en.wikipedia.org/wiki/Gimbal_lock).
- [12] *GL Mathematics*. URL: <https://glm.g-truc.net/0.9.9/index.html>.
- [13] *Graphics Language Extension Wrangler (GLEW)*. URL: <http://glew.sourceforge.net/basic.html>.
- [14] *Immediate Mode Graphical User Interface (ImGui)*. URL: <https://github.com/ocornut/imgui>.

- [15] *Implementare billboarding*. URL: <https://www.geeks3d.com/20140807/billboarding-vertex-shader-glsl/>.
- [16] *Implementare Frustum Culling*. URL: <https://www.gamedevs.org/uploads/fast-extraction-viewing-frustum-planes-from-world-view-projection-matrix.pdf>.
- [17] *Implementare selectie*. URL: <http://antongerdelan.net/opengl/raycasting.html>.
- [18] *Incarcare obiect in VRAM de LearnOpenGL*. URL: <https://learnopengl.com/Getting-started/Hello-Triangle>.
- [19] *Informatii introductive despre shadere de LearnOpenGL*. URL: <https://learnopengl.com/Getting-started/Shader>.
- [20] *JSON Format*. URL: <https://www.json.org/>.
- [21] *Light attenuation*. URL: <http://wiki.ogre3d.org/tiki-index.php?page=-Point+Light+Attenuation>.
- [22] *LUA Scripting*. URL: <https://www.lua.org/about.htm>.
- [23] *Lumina de LearnOpenGL*. URL: <https://learnopengl.com/Lighting/Basic-Lighting>.
- [24] *Matricea Model-View-Projection*. URL: [http://www.codinglabs.net/article\\_world\\_view\\_projection\\_matrix.aspx](http://www.codinglabs.net/article_world_view_projection_matrix.aspx).
- [25] *Open Audio Library*. URL: <https://en.wikipedia.org/wiki/OpenAL>.
- [26] *OpenGL Context*. URL: [https://www.khronos.org/opengl/wiki/OpenGL\\_Context](https://www.khronos.org/opengl/wiki/OpenGL_Context).
- [27] *OpenGL global state system*. URL: [https://www.khronos.org/opengl/wiki/Portal:OpenGL\\_Concepts/State](https://www.khronos.org/opengl/wiki/Portal:OpenGL_Concepts/State).
- [28] *OpenGL Shading Language (GLSL)*. URL: [https://www.khronos.org/opengl/wiki/Core\\_Language\\_\(GLSL\)](https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)).
- [29] *Parsare OBJ*. URL: [https://github.com/BennyQBD/ModernOpenGLTutorial/blob/master/obj\\_loader.cpp](https://github.com/BennyQBD/ModernOpenGLTutorial/blob/master/obj_loader.cpp).
- [30] *Phong lighting*. URL: [https://en.wikipedia.org/wiki/Phong\\_reflection\\_model](https://en.wikipedia.org/wiki/Phong_reflection_model).



- [31] *PNG format*. URL: [https://en.wikipedia.org/wiki/Portable\\_Network\\_Graphics](https://en.wikipedia.org/wiki/Portable_Network_Graphics).
- [32] *Quaternioni*. URL: <https://en.wikipedia.org/wiki/Quaternion>.
- [33] *ReactPhysics3D*. URL: <https://www.reactphysics3d.com/usermanual.html>.
- [34] *Rendering pipeline*. URL: [https://www.khronos.org/opengl/wiki/Rendering\\_Pipeline\\_Overview](https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview).
- [35] *Rezolvare distorsie pincushion*. URL: <http://www.decarpentier.nl/lens-distortion>.
- [36] *SDL Mixer*. URL: [https://www.libsdl.org/projects/SDL\\_mixer/](https://www.libsdl.org/projects/SDL_mixer/).
- [37] *SDL Mixer tutorial de LazyFoo*. URL: [http://lazyfoo.net/tutorials/SDL/21\\_sound\\_effects\\_and\\_music/index.php](http://lazyfoo.net/tutorials/SDL/21_sound_effects_and_music/index.php).
- [38] *Simple DirectMedia Library ver2 (SDL2)*. URL: <https://wiki.libsdl.org/FrontPage>.
- [39] *Simulare cer in joc*. URL: <http://ogldev.atSPACE.co.uk/www/tutorial25/tutorial25.html>.
- [40] *stb\_image library*. URL: [https://github.com/nothings/stb/blob/master/stb\\_image.h](https://github.com/nothings/stb/blob/master/stb_image.h).
- [41] *Surse de lumina de LearnOpenGL*. URL: <https://learnopengl.com/Lighting/Light-casters>.
- [42] *Texel - Texture Element*. URL: [https://en.wikipedia.org/wiki/Texel\\_\(graphics\)](https://en.wikipedia.org/wiki/Texel_(graphics)).
- [43] *Transformari folosind matrici*. URL: <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>.
- [44] *Unreal Engine4*. URL: <https://www.unrealengine.com/en-US>.
- [45] *Vertex Array Object (VAO)*. URL: [https://www.khronos.org/opengl/wiki/Vertex\\_Specification#Vertex\\_Array\\_Object](https://www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Array_Object).
- [46] *Vertex Buffer Object (VBO)*. URL: [https://www.khronos.org/opengl/wiki/Vertex\\_Specification#Vertex\\_Buffer\\_Object](https://www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Buffer_Object).
- [47] *Wavefront OBJ format*. URL: [https://en.wikipedia.org/wiki/Wavefront\\_.obj\\_file#File\\_format](https://en.wikipedia.org/wiki/Wavefront_.obj_file#File_format).