



UNIVERSITATEA DIN
BUCUREŞTI



FACULTATEA DE
MATEMATICA ŞI
INFORMATICA

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

SIMULAREA EFECTELOR DE ILUMINARE ÎN GRAFICA PE CALCULATOR

Absolvent
Bojici Valentin-Gabriel

Coordonator științific
Prof.dr. Mihai-Sorin Stupariu

București, iunie 2023

Rezumat

Scopul iluminării în grafică pe calculator este realismul prin simularea cât mai exactă a unei fotografii produsă de un aparat de fotografiat sau interpretarea artistică prin scene care nu arată realistic dar sunt plăcute din punct de vedere vizual. În funcție de cerințe, există metode pentru efecte de iluminare ce pot fi aplicate în timp real pentru interactivitate sau pentru scene statice. Pentru a exemplifica efectele metodelor de iluminare în timp real este creată o aplicație interactivă în limbajul C++ folosind API-ul OpenGL pentru comunicarea cu unitatea de procesare grafică.

Abstract

Illumination is used in computer graphics to closely mimic the appearance of a photograph produced by a real camera or to create scenes that are visually pleasing despite not being realistic. Depending on the requirements, there are methods for real-time, interactive illumination or for static scenes. In order to showcase the effects of real-time illumination, an application was developed, written in C++ and using OpenGL API for communicating with the graphics processing unit.

Cuprins

1	Introducere	5
1.1	Motivație	5
1.2	Scopul lucrării	5
1.3	Structura	5
1.4	Repere istorice	6
1.5	Contribuția proprie	6
2	Preliminarii	7
2.1	Rendering Equation	7
2.1.1	Formula originală	7
2.1.2	Notății	7
2.1.3	Varianta folosită	8
2.2	OpenGL	8
3	Modele de iluminare	9
3.1	Phong BRDF	9
3.2	Blinn-Phong BRDF	10
3.3	Cook-Torrance BRDF	11
3.3.1	Funcția D	11
3.3.2	Funcția G	12
3.3.3	Funcția F	15
4	Implementare	17
4.1	App	17
4.2	Scene	17
4.3	Buffers	20
4.4	Framebuffers	21
4.5	Meshes	22
4.6	Shaders	24
4.6.1	Clasa Shader	24
4.6.2	Vertex shader	25

4.6.3	Fragment shader	25
4.7	Evenimente	26
4.8	Camera	28
4.8.1	Clasa Camera	28
4.9	Surse de lumină	29
4.9.1	Atenuare	30
4.9.2	Spotlight	30
4.9.3	Clasa Light	31
4.9.4	Normal mapping	32
4.10	Umbre	32
4.10.1	Fragment shader	33
4.10.2	Directional light	33
4.10.3	Spot light	33
4.10.4	Point light	34
4.10.5	Compararea distanțelor	34
4.10.6	Optimizări	35
4.11	Texturi	36
4.11.1	Clasa TextureManager	37
4.12	Modele	38
4.13	Materiale	38
4.14	Post-processing	40
4.14.1	Gamma correction	40
4.14.2	Tone mapping	40
4.15	Toon shading	41
4.15.1	Two-tone shading	41
4.15.2	Detectarea liniilor de contur	42
5	Concluzii	45
5.1	Dezvoltări ulterioare	46
5.1.1	Ray tracing	46
5.1.2	Iluminare globală	46
5.1.3	Ambient occlusion	46
Bibliografie		47

Capitolul 1

Introducere

1.1 Motivație

Iluminarea joacă un rol important în jocurile video. Fiind familiar cu jocurile video am considerat interesant să afiu mai multe detalii despre modul de funcționare al unui astfel de efect dar și despre cum poate fi implementat. În plus, am dorit să mă concentrez și pe implementarea unei aplicații de tip desktop și pe concepte de programare orientată obiect.

1.2 Scopul lucrării

Scopul lucrării este de a cerceta modele de iluminare ce pot fi aplicate în timp real și de a crea o aplicație desktop interactivă pentru prezentarea lor. Utilizatorul are acces la o interfață grafică de unde poate controla diversi parametri utilizati în algoritmii de iluminare.

1.3 Structura

Lucrarea este structurată în câteva capitole:

Introducere: informații generale precum *motivație, scop, repere istorice*.

Preliminarii: se descrie pe scurt ecuația care stă la baza modelelor de iluminare și *OpenGL* (ce a permis crearea unei aplicații pentru exemplificarea iluminării).

Modele: sunt prezentate modelele de iluminare cercetate: *Phong, Blinn-Phong, Cook-Torrance*.

Implementare: se descriu principalele componente ale aplicației dar și modul în care a fost dezvoltată.

Concluzii: un scurt rezumat și câteva posibile dezvoltări ulterioare.

1.4 Repere istorice

Domeniul modelelor de iluminare nu este nou, cel mai cunoscut și folosit, pentru că este ușor de implementat, este cel introdus de Bui Tuong Phong în 1975 [16]. În 1982 Robert L. Cook și Kenneth E. Torrance au introdus un model nou bazat pe micro-fațete [5] care rămâne folosit și astăzi, cu diferite modificări (de exemplu de compania Disney [4]).

1.5 Contribuția proprie

Un exemplu de aplicație similară poate fi *BRDF Explorer*¹ creată de compania Disney [4] pentru a analiza diferențele dintre diferite modele. *BRDF Explorer* permite încărcarea de cod sursă (shader) pentru un model și schimbarea parametrilor utilizati dintr-o interfață grafică [4].

În aplicația prezentată în această lucrare am introdus câteva texturi pentru comparații între modelele Phong, Blinn-Phong și Cook-Torrance dar și o scenă cu modele 3D în care este folosit modelul Cook-Torrance pentru a exemplifica rezultatele realistice obținute de acest model. În plus am implementat mai multe variații ale funcțiilor folosite în modelul Cook-Torrance pentru a observa diferențele dintre acestea.

¹<https://github.com/wdas/brdf>

Capitolul 2

Preliminarii

2.1 Rendering Equation

2.1.1 Formula originală

Rendering Equation (sau Ecuația de Randare) este un rezultat publicat de James T. Kajiya în 1986 [13] care este foarte important și stă la baza modelelor de iluminare. Ecuația descrie cum interacționează razele de lumină într-un anumit punct și cum acest rezultat influențează ce vedem când observăm acel punct.

2.1.2 Notații

Se folosesc următoarele notății în formulele din această lucrare:

v : vector unitate de la punctul observat spre observator

n : vector unitate ce reprezinta normala în punctul observat

l : vector unitate de la punctul observat spre originea razei de lumină

r : vector unitate ce reprezintă reflexia vectorului **l** față de normala **n**

h : vector unitate astfel încât unghiul dintre **h** și **v** este egal cu unghiul dintre **h** și **l**¹

(**a** · **b**) : cosinusul unghiului dintre vectorii unitate **a** și **b**

¹**h** poate fi calculat astfel: $\frac{\mathbf{l} + \mathbf{v}}{|\mathbf{l} + \mathbf{v}|}$ și este mai eficient de calculat decât vectorul **r**.

2.1.3 Varianta folosită

Există diferite versiuni ale formulei prezentate de Kajiya. Varianta pe care am folosit-o este din *Real-Time Rendering* [10, p. 311].

$$L_e(x, \mathbf{v}) + \int_{\mathbf{l} \in \Omega} f(x, \mathbf{l}, \mathbf{v}) L_i(x, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l}) d\mathbf{l} \quad (2.1)$$

Când observatorul privește un punct x , ceea ce observă este suma a 2 elemente: lumina emisă (creată) de punctul respectiv în direcția \mathbf{v} (primul termen din formulă), și suma tuturor contribuțiilor razelor de lumină din mediul înconjurător în acel punct (al doilea termen din formulă) — se presupune că suprafața nu este transparentă și toate direcțiile razelor de lumină din mediul reprezintă o emisferă² (notată cu Ω) deasupra punctului.

$f(x, \mathbf{l}, \mathbf{v})$ este *funcția de distribuție a reflectivității bidirectional* (Bidirectional Reflectance Distribution Function), sau mai simplu BRDF, descrie procentul de lumină reflectată din direcția \mathbf{l} în direcția \mathbf{v} de către punctul x .

$L_i(x, \mathbf{l})$ este cantitatea de lumină ce ajunge în punctul x din direcția \mathbf{l}

Valoarea $(\mathbf{n} \cdot \mathbf{l})$ modelează efectul următor: dacă o suprafață primește o anumită cantitate de lumină din direcția dată de normală și suprafața este rotită, aceeași cantitate de lumină ajunge pe o suprafață mai mare deci va părea mai întunecată. Când suprafața este rotită cu 90° , suprafața este paralelă cu direcția razelor de lumină și nu va fi iluminată deloc.

2.2 OpenGL

OpenGL este o *interfață de programare a aplicației* (application programming interface) care permite comunicarea și rularea aplicațiilor folosind unitatea de procesare grafică (GPU). OpenGL permite scrierea unui program care rulează pe GPU numit *shader* [19]. Scopul acestor programe este desenarea unor primitive (punkte, segmente de dreaptă, triunghiuri) pe un ecran.

Fiindcă avem un control fin asupra obiectelor desenate (la nivel de pixeli) putem implementa diferite metode care simulează iluminarea.

²Este considerată o emisferă centrată în acel punct și orientată spre direcția normalei. Emisfera semnifică faptul că razele de lumină nu pot veni din spatele suprafeței (materialul este opac).

Capitolul 3

Modele de iluminare

Definirea unui model de iluminare constă, de fapt, în definirea funcției f — BRDF, din ecuația (2.1).

Un punct de pe o suprafață poate fi iluminat în două moduri: o rază de lumină de la o sursă lovește direct acel punct sau o rază de lumină lovește alt punct și, după una sau mai multe reflexii, lovește (indirect) punctul. Datorită acestui fapt putem despărți integrala din ecuația (2.1) într-o sumă de doi termeni [10, p. 311]:

$$\int_{\text{direct}} f(x, \mathbf{l}, \mathbf{v}) L_i(x, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l}) d\mathbf{l} + \int_{\text{indirect}} f(x, \mathbf{l}, \mathbf{v}) L_i(x, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l}) d\mathbf{l} \quad (3.1)$$

Prima integrală se referă la *iluminarea directă* (razele de lumină din emisfera de deasupra punctului vin de la sursele de lumină) și poate fi scrisă ca o sumă pentru că există un număr finit de surse de lumină:

$$\sum_{\mathbf{l}_k} f(x, \mathbf{l}_k, \mathbf{v}) L_i(x, \mathbf{l}_k) (\mathbf{n} \cdot \mathbf{l}_k), \quad (3.2)$$

unde \mathbf{l}_k este direcția de la punctul x către sursa de lumină k . Această sumă este ușor de calculat și este implementată în aplicație pentru fiecare model.

A doua integrală se referă la *iluminarea indirectă*. În cazul aplicațiilor interactive se aproximează cu o constantă pentru că necesită calcule complexe [7, p. 115].

3.1 Phong BRDF

Modelul Phong [16] este foarte cunoscut și folosit pentru simplitatea lui și pentru faptul că are rezultate bune. Modelul Phong este un model empiric și este suma a 3 termeni: **ambient**, **diffuse** și **specular**.

Ambient : reprezintă iluminarea indirectă (lumina reflectată de alte obiecte din jur care nu sunt surse de lumină).

Diffuse : reprezintă componenta difuză din iluminarea directă (lumina este „împrăștiată” în multe direcții). Ea nu depinde de poziția observatorului și este constantă.

Specular : reprezintă componenta speculară din iluminarea directă, este cea responsabilă pentru aspectul strălucitor al unor materiale (lumina este reflectată predominant într-o direcție — \mathbf{r}). Ea depinde de poziția observatorului.

BRDF-ul Phong pe care l-am folosit¹ (implementat în fișierul *phong.frag*) este extras din formula prezentată de Phong [16, p. 5] :

$$f(x, \mathbf{l}, \mathbf{v}) = k_d + k_s \frac{(\mathbf{r} \cdot \mathbf{v})^\alpha}{(\mathbf{n} \cdot \mathbf{l})}, \quad (3.3)$$

unde k_d , k_s , α sunt parametri ce depind de material: k_d și k_s controlează cât de puternică este componenta difuză respectiv speculară iar α controlează cât de strălucitor este materialul (1 reprezintă un material care nu este strălucitor iar ∞ reprezintă o oglindă perfectă). Când unghiul dintre \mathbf{v} și \mathbf{r} este 0 (observatorul se află pe direcția dată de \mathbf{r}) valoarea cosinusului este 1 deci componentă speculară este maximă.

3.2 Blinn-Phong BRDF

Blinn [2] a modificat formula lui Phong pentru reflexia speculară și a folosit bisectoarea unghiului dintre direcția luminii și direcția observatorului în locul direcției razei de lumină reflectate.

În același articol, Blinn descrie faptul că o suprafață poate fi gândită ca fiind alcătuită din micro-fațete microscopice orientate în direcții aleatoare ce se comportă ca niște oglinzi ideale. Din acest motiv o micro-fațetă trebuie să fie orientată în direcția \mathbf{h} pentru ca raza de lumină reflectată să ajungă de la sursă la observator. Se presupune că, cu cât unghiul dintre \mathbf{h} și \mathbf{n} este mai mic, cu atât sunt mai multe micro-fațete orientate „corect” (în direcția \mathbf{h}), deci componenta speculară este mai mare.

De aceea s-a folosit unghiul dintre \mathbf{h} și \mathbf{n} în locul unghiului dintre \mathbf{v} și \mathbf{r} . BRDF-ul Blinn-Phong pe care l-am folosit (implementat în fișierul *blinn.frag*) este extras din formula prezentată de Blinn [2, p. 2]:

$$f(x, \mathbf{l}, \mathbf{v}) = k_d + k_s \frac{(\mathbf{h} \cdot \mathbf{n})^\alpha}{(\mathbf{n} \cdot \mathbf{l})} \quad (3.4)$$

¹Componenta speculară conține o împărțire la $(\mathbf{n} \cdot \mathbf{l})$ pentru a anula înmulțirea cu acest termen din formula 3.2.

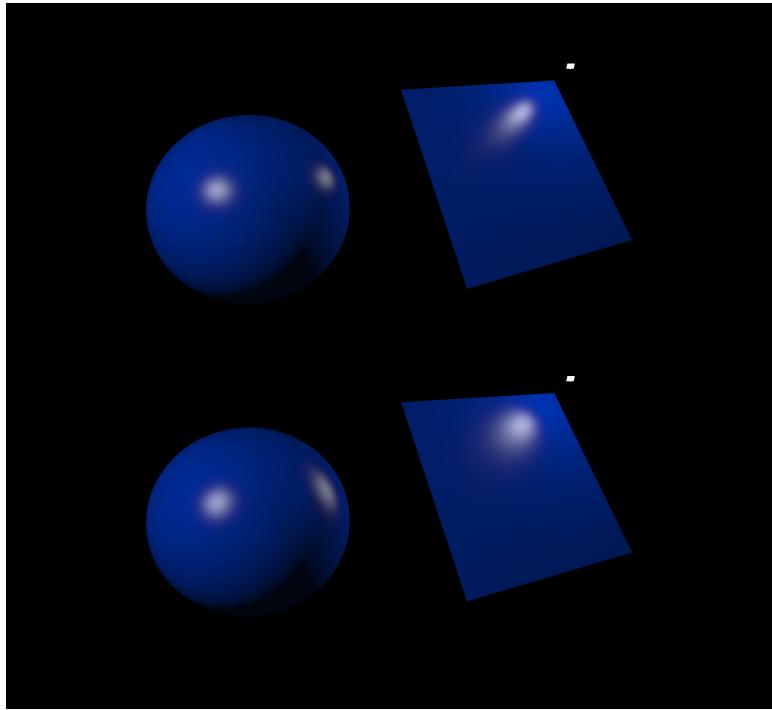


Figura 3.1: Comparație între modelele Blinn-Phong (sus) și Phong (jos) în aplicația dezvoltată. Pentru a avea rezultate similare cu modelul Phong parametrul α din BRDF-ul Blinn-Phong este aproximativ de 4 ori mai mare.

3.3 Cook-Torrance BRDF

În 1982, Cook și Torrance [5] au descris un model ce aproximează o suprafață cu o distribuție de micro-fațete, idee ce a fost descrisă de Blinn [2]. Acest BRDF este mai complex și necesită mai multă putere de calcul decât BRDF-ul Blinn-Phong dar poate modela mai multe tipuri de materiale și este bazat pe fenomene fizice.

Funcția BRDF introdusă de Cook și Torrance ² [5, pp. 4–5] (implementată în fișierul *cook-torrance.frag*):

$$dR_d + s \frac{DFG}{4(\mathbf{n} \cdot \mathbf{v})(\mathbf{n} \cdot \mathbf{l})}, \quad (3.5)$$

unde $d + s \leq 1$ pentru a se respecta o constrângere legată de conservarea energiei [10, p. 312], R_d este componenta difuză (care nu depinde de poziția observatorului deci poate fi constantă) iar funcțiile D , F , G sunt prezentate mai jos.

3.3.1 Funcția D

Funcția D reprezintă procentajul (funcție masă de probabilitate) de micro-fațete care au normala la suprafață în direcția \mathbf{h} într-un punct [5] (se presupune că micro-fațetele

²În formula originală numitorul din formulă conține π în loc de 4, în urma verificărilor calculelor numitorul conține 4. [11] [10, p. 337]

sunt oglinzi ideale ce vor reflecta lumina doar dacă normala la suprafață este în direcția \mathbf{h}).

Formulele folosesc un parametru $\alpha \in [0, 1]$ care reprezintă cât de neted este un material la nivel de micro-fațete (valori apropiate de 0 pentru materiale foarte netede, de exemplu metale lustruite și valori apropiate de 1 pentru materiale cu proeminente la nivel microscopic, de exemplu lemn).

Distribuția Beckmann

Cook și Torrance au folosit distribuția Beckmann [5][10, p. 338] (implementată în funcția `D_Beckmann` din fișierul `cook-torrance.frag`):

$$D_{\text{Beckmann}} = \frac{1}{\pi\alpha^2(\mathbf{n} \cdot \mathbf{h})^4} \exp\left(\frac{(\mathbf{n} \cdot \mathbf{h})^2 - 1}{\alpha^2(\mathbf{n} \cdot \mathbf{h})^2}\right) \quad (3.6)$$

Distribuția Blinn-Phong

Altă distribuție menționată [10, p. 340] (implementată în funcția `D_Phong` din fișierul `cook-torrance.frag`):

$$D_{\text{Blinn-Phong}} = \frac{\alpha_{\text{phong}} + 2}{2\pi} (\mathbf{n} \cdot \mathbf{h})^{\alpha_{\text{phong}}} \quad (3.7)$$

Parametrul α_{phong} are valori între 1 și ∞ (pentru o oglindă ideală). Pentru a utiliza valori între 0 și 1 am folosit formula $\alpha_{\text{phong}} = 2\alpha^{-2} - 2$ [24, p. 7] (unde $\alpha \in [0, 1]$).

Distribuția GGX

Blinn [2, p. 4] prezintă distribuția Trowbridge-Reitz [22, p. 5] care a fost redescoperită în 2007 [24, p. 7] sub numele de GGX care a devenit numele utilizat în practică [10, p. 340] (implementată în funcția `D_GGX` din fișierul `cook-torrance.frag`):

$$D_{\text{GGX}} = \frac{\alpha^2}{\pi(1 + (\mathbf{n} \cdot \mathbf{h})^2(\alpha^2 - 1))^2} \quad (3.8)$$

Burley [4] propune ca valoarea α să fie ridicată la patrat înainte de a se folosi formula. Se menționează că astfel schimbările par mai liniare, în special pentru materiale foarte netede. Din acest motiv am implementat și eu în aplicație acest lucru.

3.3.2 Funcția G

Funcția G (funcția de mascare) reprezintă procentajul de micro-fațete cu normala la suprafață \mathbf{h} care sunt vizibile din direcția observatorului [10, p. 333]. Funcția simulează

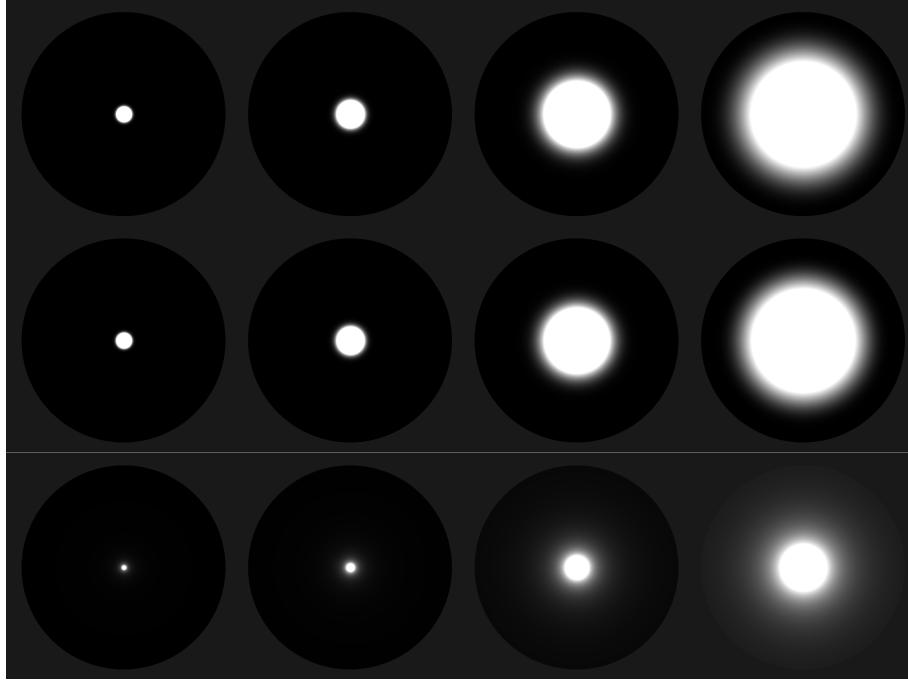


Figura 3.2: Comparație ale funcțiilor D , în aplicație. Primul rând: Phong, al doilea rând: Beckmann, ultimul rând: GGX. Parametrul α ia valorile 0.025, 0.05, 0.15, 0.3 (de la stânga spre dreapta)

faptul că, în practică, nu toate micro-fațetele sunt vizibile neapărat. Din anumite unghiuri unele micro-fațete proeminente le ascund pe celelalte.

Funcția G folosită de Cook și Torrance [5, p. 5] (implementată în funcția `G_Cook` din fișierul `cook-torrance.frag`):

$$G_{\text{Cook-Torrance}} = \min \left\{ 1, \frac{2(\mathbf{n} \cdot \mathbf{h})(\mathbf{n} \cdot \mathbf{v})}{(\mathbf{v} \cdot \mathbf{h})}, \frac{2(\mathbf{n} \cdot \mathbf{h})(\mathbf{n} \cdot \mathbf{l})}{(\mathbf{v} \cdot \mathbf{h})} \right\} \quad (3.9)$$

O altă funcție pe care am implementat-o este funcția G_2 Smith (varianta necorelată) ce ține cont și de faptul că unele micro-fațete mai puțin proeminente nu sunt iluminate din cauza altor micro-fațete [10, p. 335] (implementată în funcțiile `G2_U_Beckmann` și `G2_U_GGX`):

$$G_2(\mathbf{v}, \mathbf{l}) = G_1(\mathbf{v})G_1(\mathbf{l}), \quad (3.10)$$

unde $G_1(v) = \frac{1}{1+\Lambda(a_v)}$ iar funcția Λ diferă în funcție de ce distribuție D este folosită.

Formula presupune că efectele de umbră și mascare sunt independente ceea ce nu este adevărat în realitate și cauzează rezultate mai întunecate [10, p. 335].

Funcția G'_2 Smith (varianta corelată) ține cont de faptul că cele două efecte au legătură (în funcție de înălțime). Dacă un punct este la o înălțime mai mică față de cele din jur,

sunt mai multe şanse să nu fie vizibil [10, p. 335]:

$$G'_2(\mathbf{v}, \mathbf{l}) = \frac{1}{1 + \Lambda(a_v) + \Lambda(a_l)} \quad (3.11)$$

Se poate observa din Figura 3.3 că variantele funcției G'_2 care țin cont de faptul că mascarea și lipsa iluminării sunt corelate au valori puțin mai mari decât valorile funcțiilor G_2 , în special în zona care delimită partea iluminată de cea lipsită de lumină.

Funcția Λ

Funcțiile G_2 și G'_2 de mai sus folosesc funcția Λ ce utilizează parametrul a [10, p. 339]:

$$a_s = \frac{(\mathbf{n} \cdot \mathbf{s})}{\alpha \sqrt{1 - (\mathbf{n} \cdot \mathbf{s})}}, \quad (3.12)$$

unde \mathbf{s} este \mathbf{l} sau \mathbf{v} .

Funcția Λ pentru D_{Beckmann}

Formula originală este mai greu de calculat aşa că este folosită o aproximare [10, p. 339]

$$\Lambda(a) \approx \begin{cases} \frac{1-1.259a+0.396a^2}{3.535a+2.181a^2} & , \text{ dacă } a < 1.6 \\ 0 & , \text{ altfel} \end{cases} \quad (3.13)$$

Această definiție este utilizată în formulele 3.10 și 3.11 și este implementată în funcțiile `G2_Beckmann` și `G2_U_Beckmann`.

Funcția Λ pentru D_{GGX}

Formula pe care am folosit-o este [10, p. 341]:

$$\Lambda(a) = \frac{-1 + \sqrt{1 + \frac{1}{a^2}}}{2} \quad (3.14)$$

Această definiție este utilizată în formulele 3.10 și 3.11 și este implementată în funcțiile `G2_GGX` și `G2_U_GGX`.

Funcția Λ pentru D_{Phong}

Nu există o formulă pentru funcția Λ [24, p. 7]. Se recomandă folosirea funcției Λ pentru distribuția Beckmann împreună cu parametrul $\alpha_{\text{phong}} = 2\alpha^{-2} - 2$ [24, p. 7].

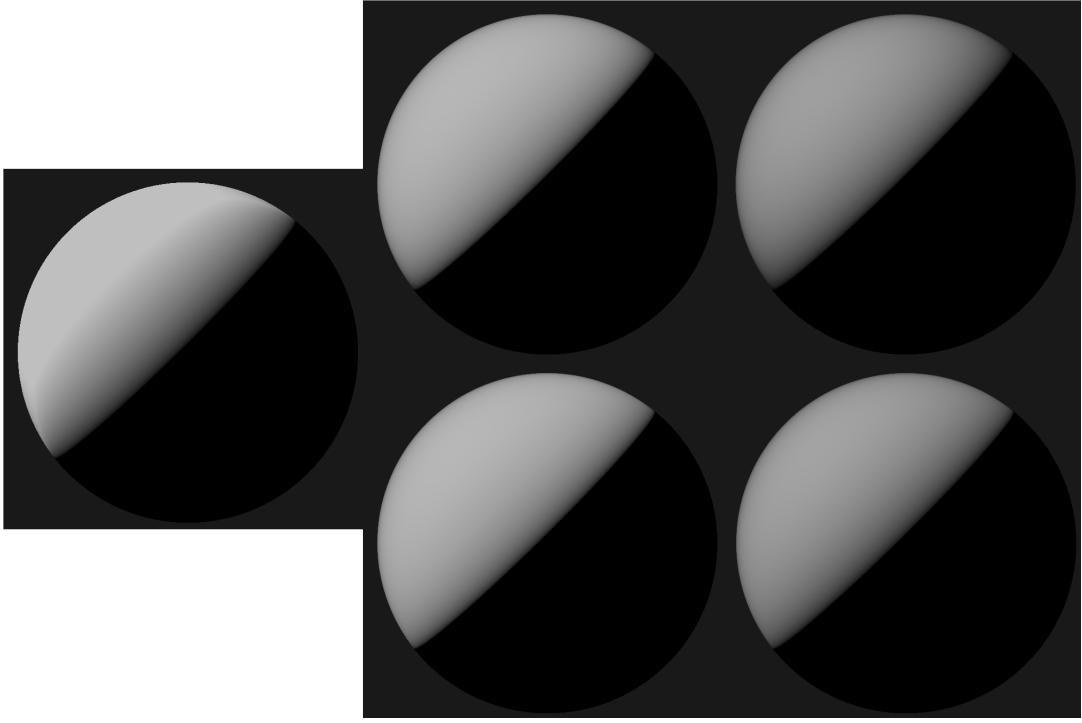


Figura 3.3: Comparație ale funcțiilor G , în aplicație cu $\alpha = 0.6$. În stânga: varianta $G_{\text{Cook-Torrance}}$, mijloc: $G_{2_{\text{Beckmann}}}$, dreapta: $G_{2_{\text{GGX}}}$ (varianta necorelată sus și cea corelată jos)

3.3.3 Funcția F

Funcția F reprezintă efectul Fresnel. Acest efect se referă la faptul că o suprafață reflectă mai multă lumină cu cât unghiul dintre normala la suprafață și direcția razelor de lumină este mai aproape de 90° . Un exemplu poate fi un lac: dacă privim apa, fiind aproape de mal și privind în jos, putem vedea ce este în apă. În schimb, dacă privim în depărtare spre malul opus, vedem reflexia lucrurilor din jurul lacului.

Formulele Fresnel sunt complexe și astfel în practică se folosește aproximarea Schlick [10, p. 320]. Formula găsită de Schlick este [18, p. 8] :

$$F(\theta) = F_0 + (1 - F_0)(1 - \cos \theta)^5, \quad (3.15)$$

unde $F_0 \in [0, 1]$ reprezintă factorul de reflexie când raza de lumină este perpendiculară pe suprafață. F_0 se poate calcula și cu indici de refracție [10, p. 321].

În [10, p. 322] sunt date câteva exemple de valori pentru F_0 . Pentru dielectrice (non-metale) F_0 are valori mici: ≈ 0.04 . Metalele au valori mai mari și reprezintă „culoarea” lor pentru că ele nu reflectă în mod difuz lumina.

În aplicație am permis utilizatorului să aleagă valori (culoare) pentru F_0 dar pentru

a fi mai ușor de folosit am oferit și opțiunea de a calcula F_0 astfel:

$$F_0 = \text{mix}(0.04, \text{diffuse}, \text{metalness})$$

Am interpolat liniar între 0.04 și culoarea difuză (specificată în interfață) în funcție de parametrul *metalness* $\in [0, 1]$ unde 0 înseamnă că materialul este dielectric și 1 că este metal.

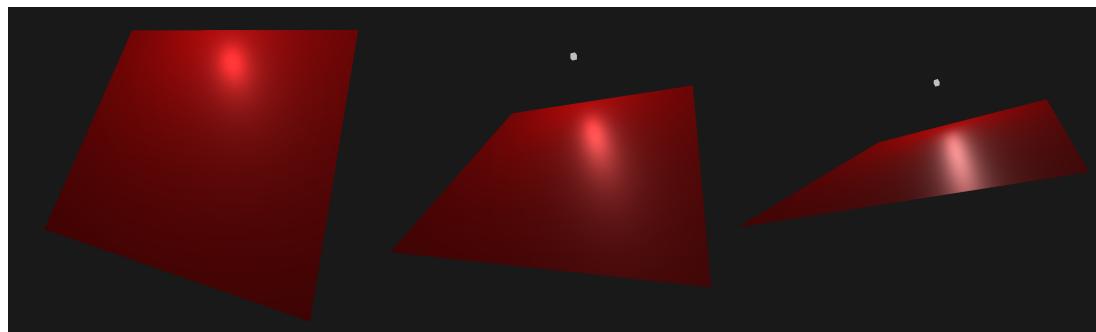


Figura 3.4: Efectul Fresnel, în aplicație cu $\alpha = 0.5$ și $\text{metalness} = 0.5$.

Capitolul 4

Implementare

Aplicația este dezvoltată pentru Windows folosind C++ și librăriile: **Dear ImGui**[6] — pentru interfața grafică, **Open Asset Import Library (ASSIMP)**[14] — pentru încărcarea modelelor, **stb_image** și **stb_image_write** [1] — pentru încărcarea imaginilor și salvarea lor, **The OpenGL Extension Wrangler Library**[12] — pentru a utiliza extensiile de OpenGL, **GLFW**[15] — pentru crearea unei ferestre, **OpenGL Mathematics** [9] — pentru funcții matematice.

Interfața grafică în care utilizatorul poate modifica diferenți parametrii este o fereastră separată.

Aplicația conține 3 scene: *Box scene* — o cameră și 3 corpuri geometrice: un con, un cub, o sferă și 3 surse de lumină, *Texture scene* — 2 obiecte (plan sau sferă) și 2 surse de lumină, pentru comparații între modele de iluminare, *Model test scene* — o cameră cu diferenite modele: masă, scaun, bibliotecă. Este folosit modelul Cook-Torrance pentru a exemplifica realismul pe care modelul de iluminare îl aduce unei scene.

4.1 App

Clasa `App` este cea responsabilă pentru construirea ferestrei cu ajutorul funcțiilor *GLFW*, pentru configurarea OpenGL, *GLEW* și *Dear ImGui*. De asemenea în clasa `App` este bucla principală care rulează cât timp aplicația este deschisă.

Clasa `App` semnalează scenei active dacă dimensiunile ferestrei sunt modificate (pentru ca *framebuffer-ul* să fie modificat corespunzător).

4.2 Scene

Clasa `Scene` este o interfață pentru restul scenelor care pot supraîncărca funcțiile `onRender()`, `onRenderImGui()` și `updateWidthHeight(width, height)`. Funcția `onRender()` este responsabilă pentru afișarea conținutului scenei, `onRenderImGui()` este pentru responsabilă

pentru afișarea meniului iar `updateWidthHeight(width, height)` pentru a actualiza variabilele atunci când dimensiunea ferestrei se schimbă. Configurarea pentru OpenGL și *Dear ImGui* este făcută deja în clasa `App`, iar scenele trebuie doar să deseneze scena cu ajutorul funcțiilor OpenGL și să afișeze meniul cu ajutorul funcțiilor din *Dear ImGui*.

Am folosit şablonul de proiectare comportamental (behavioral design pattern) *State* [21, p. 353].¹

Clasa conține o dată membră `m_currentScene` care reprezintă scena activă în acest moment. Ea este definită în constructorul clasei `App` și inițial reprezintă scena „Meniu”:

```
1   m_scene = std::make_unique<SceneMenu>(m_scene, m_windowWidth,
                                             m_windowHeight);
```

și este modificată de alte clase care moștenesc clasa „Scene” atunci când se dorește schimbarea stării curente. Funcția `setScene` este apelată când o altă scenă devine activă (când starea este schimbată).

Meniu

Scena „Meniu” este doar interfață grafică prin care utilizatorul alege o altă scenă. Nu este desenat nimic în fereastră.

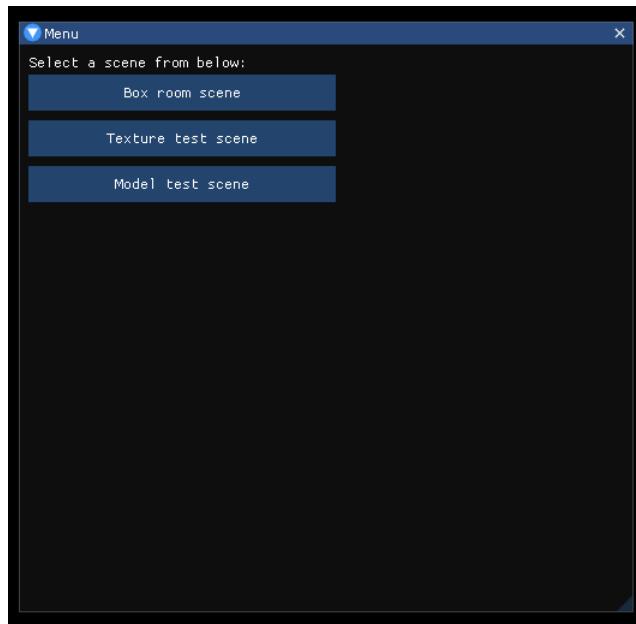


Figura 4.1: Interfață grafică din scena „Meniu”

Box scene

Scena „Box room” este o cameră cu 3 coruri geometrice. În acest mod se poate observa cum diferite modele de iluminare modifică aspectul scenei.

¹Un obiect își schimbă comportamentul atunci când starea se schimbă. În acest caz, când scena se modifică, rezultatul funcțiilor `onRender()` și `onRenderImGui()` se schimbă.

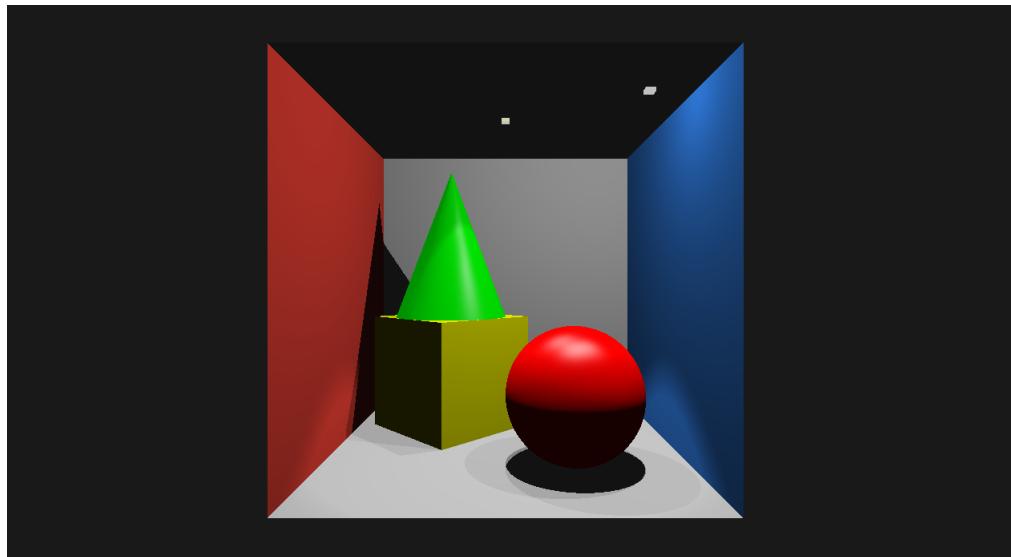


Figura 4.2: Scena „Box room”

Interfața grafică permite modificarea parametrilor surselor de lumină și ale obiectelor.

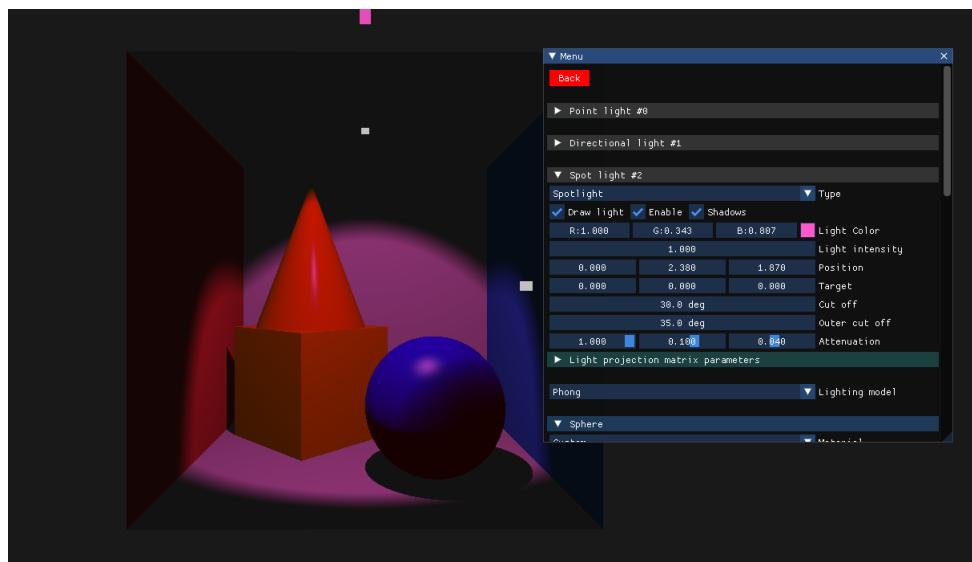


Figura 4.3: Scena „Box room” cu diferenți parametri modificați

Texture scene

Scena „Texture scene” conține doar 2 obiecte (un plan sau o sferă). Fiecare obiect poate avea un model diferit de iluminare. În acest mod se pot observa diferențe între modele. De asemenea, există posibilitatea de a folosi texturi pentru a obține un efect realist.

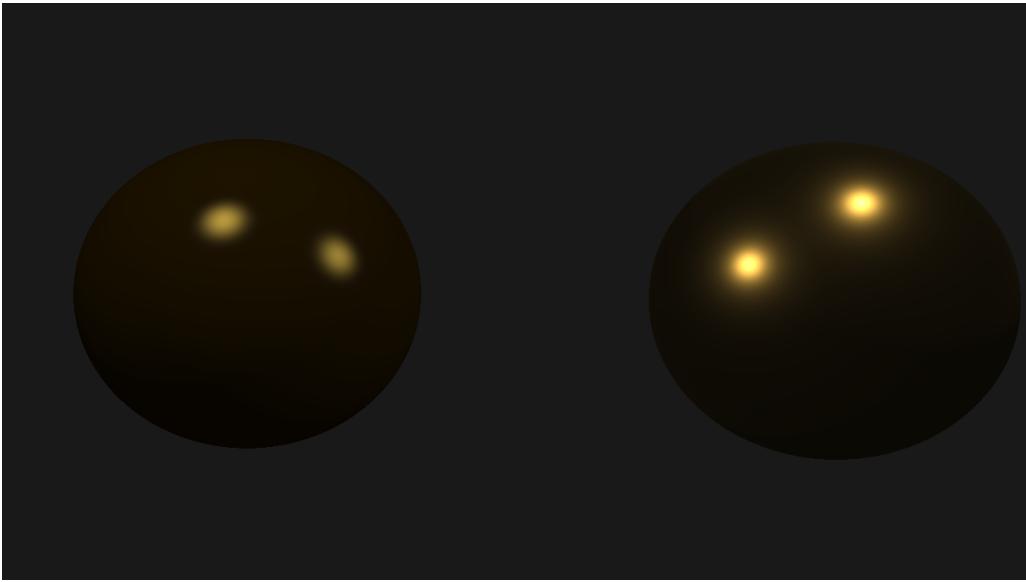


Figura 4.4: Scena „Texture scene” — comparație între modelul Phong (stânga) și Cook-Torrance (dreapta) pentru două sfere din aur.

Model scene

Scena „Model scene” este o cameră cu diferite modele pentru a evidenția rezultatele ce pot fi obținute cu modelul Cook-Torrance. Nu s-au utilizat modelele Phong și Blinn-Phong pentru că s-au folosit texturi pentru parametrii modelului Cook-Torrance ce nu pot fi transformați cu ușurință în parametri compatibili cu modelele Phong și Blinn-Phong.

4.3 Buffers

Pentru a desena primitive trebuie specificat cu ajutorul API-ului OpenGL datele pe care dorim să le încărcăm pe GPU dar și ce tipuri de date am folosit. Fiecare vârf are următoarele caracteristici: poziție, coordonate de texturare², normală în acel punct și tangentă³ în acel punct.

Am creat clase pentru fiecare dintre cele 3 tipuri de *buffers* folosite pentru a abstractiza crearea și utilizarea lor: *Vertex Buffer Objects - clasa *vbo**, *Element Buffer Objects - clasa *ebo** (pentru a specifica indicii vâfurilor pentru primitivele pe care le voi desena), *Vertex Array Objects - clasa *vao** (după ce datele sunt încărcate, trebuie specificat cum sunt organizate: câți octeți are un vârf, câte atrbute există și ce dimensiune dar și ce tip de date au fiecare)

²Aceste coordonate sunt folosite pentru a afișa texturile (imaginile) folosite. O textură 2D are coordonatele, începând din colțul stânga jos și în sens trigonometric: (0,0);(1,0);(1,1);(0,1).

³Folosită pentru *normal mapping*, o metodă prin care se folosesc normalele extrase dintr-o textură pentru realism.

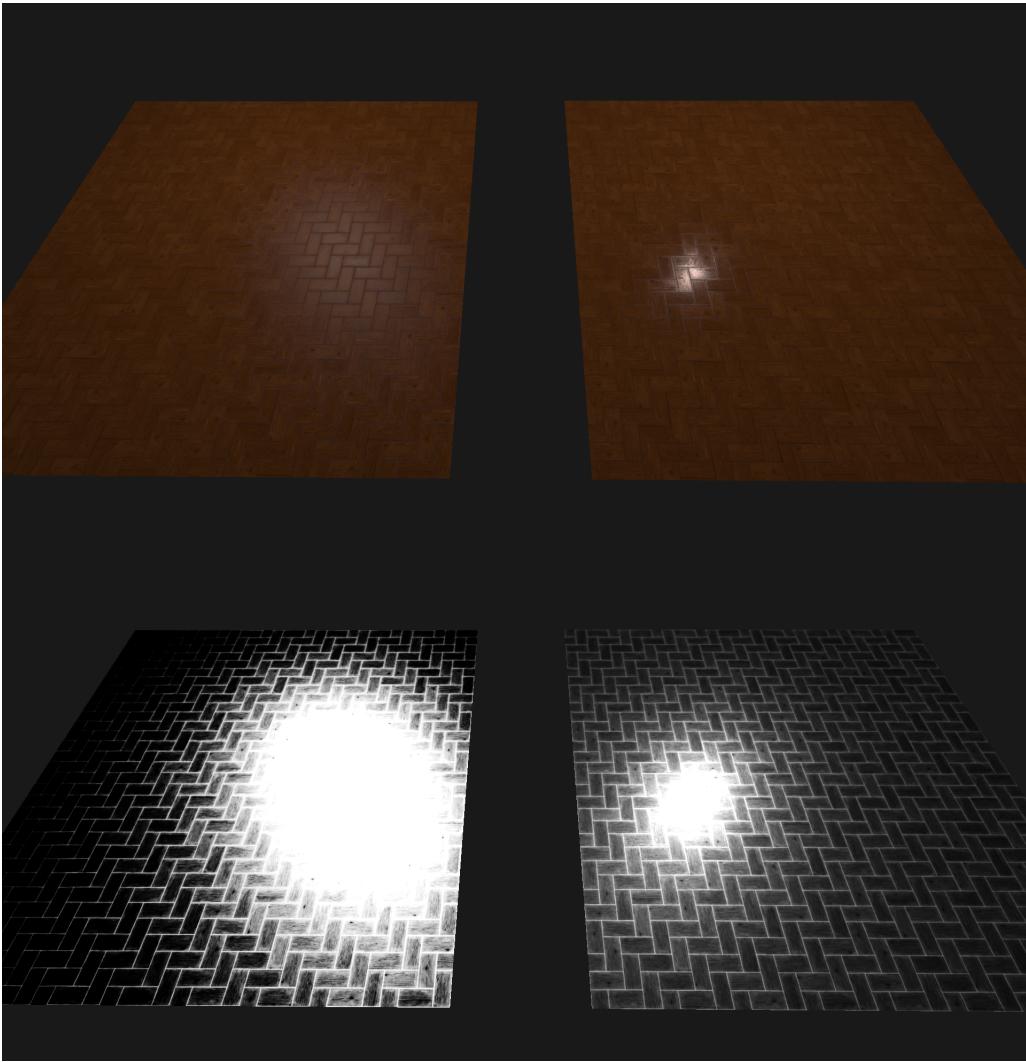


Figura 4.5: Scena „Texture scene” — comparație între distribuția Beckmann (stânga) și GGX (dreapta) pentru modelul Cook-Torrance. Sus este rezultatul final iar jos este doar valoarea funcției D

4.4 Framebuffers

După procesul de randare, rezultatul este stocat într-un *framebuffer*, o listă 2D de pixeli ce conțin mai multe informații [19, p. 321]. Informațiile ce pot fi stocate sunt *culoare*, *adâncime*, *valoare „stencil”*.

Există un *framebuffer* implicit al cărui conținut (culori) vor fi afișate pe ecran în final, dar un *framebuffer* poate fi creat manual. Este necesară crearea unor *atașamente* pentru *culoare*, *adâncime*, *stencil* în funcție de nevoi. Adâncimea poate fi folosită în *testul de adâncime*: obiectele care se află în spatele altora sunt ascunse (nu se desenează). Valoarea *stencil* poate fi folosită pentru a masca părți din *framebuffer*, însă nu am folosit aceasta funcționalitate în aplicație.

Un atașament poate fi o textură sau un *renderbuffer*. Diferența este că valorile dintr-o textură pot fi accesate într-un *shader* iar din acest motiv nu este la fel de rapid [7, p. 203].



Figura 4.6: Scena „Model scene” cu funcția D Beckmann și functia G Cook-Torrance

Clasa `Framebuffer` se ocupă de crearea *framebuffer-ului* dar și de atașamente. Ea conține metode precum `addColorAttachment` sau `addDepthAttachment` pentru adăugarea de atașamente dar și `getColorAttachment`, `getDepthAttachment` pentru a obține id-urile atașamentelor (textură sau *renderbuffer*) și pentru a le putea transmite *shader-ului* (în cazul texturilor), `saveColorAttachmentToPNG` pentru a salva un atașament de culoare într-o imagine.

Utilizare

Utilizarea *framebuffer-urilor* a fost necesară pentru obținerea umbrelor dar și pentru post-procesare. De exemplu în cazul umbrelor scena a fost randată de 2 ori, prima dată folosind un *framebuffer* creat anterior pentru a reține distanța de la o sursă de lumină la cel mai apropiat obiect. Aceste informații au fost salvate în texturi apoi transmise *shader-ului* pentru randarea scenei cu umbre.

4.5 Meshes

Pentru a utiliza diferite corpuri geometrice (con, sferă, cub) am creat o clasa `Mesh`. Constructorul primește ca argumente o listă de vârfuri, indici și texturi, apoi folosește clasele menționate mai sus (VBO, EBO, VAO) pentru a încărca datele pe GPU și a le desena folosind triunghiuri.

Un *mesh* reprezintă în aplicație un lucru ce poate fi desenat cu o singură apelare a funcției `glDrawElements` (funcția folosită pentru a desena cu ajutorul indicilor). Așadar, un *mesh* este un obiect simplu, modelele complexe (ultima scenă) sunt alcătuite din mai multe obiecte *mesh*.

Există metode statice `getPlane()`, `getSphere()`, `getCone()`, `getCube()` care abstractizează crearea unui plan, sferă, con, cub.

Generarea unei sfere

Pentru generarea unei sfere am folosit un icosaedru regulat convex ale cărui triunghiuri le-am împărțit recursiv în mai multe triunghiuri iar coordonatele vârfurilor noi generate au fost normalizate pentru a le projecța pe o sferă cu raza 1. Metoda este descrisă în [20, p. 118]. Am folosit această metodă pentru a obține o distribuție mai echilibrată a triunghiurilor (dacă foloseam reprezentarea parametrică a sferei obțineam mai multe vârfuri spre „poli”).

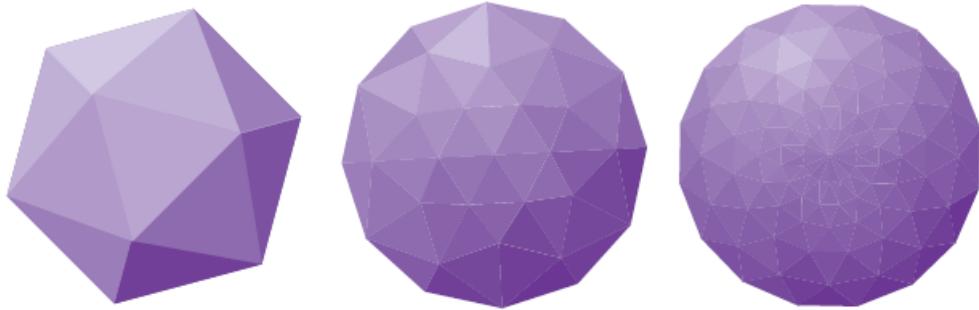


Figura 4.7: Ilustrare a procesului de subdiviziune a unui icosaedru — *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1* (p. 118)

Coordonatele vârfurilor unui icosaedru cu latura 2 sunt: $(\pm 1, 0, \pm \phi)$, $(0, \pm \phi, \pm 1)$, $(\pm \phi, \pm 1, 0)$, unde $\phi = \frac{1+\sqrt{5}}{2}$. Am folosit ordinea vârfurilor și indicilor din [20, pp. 115–116] pentru icosaedru.

Pentru a realiza 1 subdiviziune, am parcurs lista de indici (câte 3 la fiecare pas, pentru 1 triunghi — i_1, i_2, i_3), pentru fiecare latura am calculat mijlocul și l-am adăugat la lista de vârfuri. La final am precizat indicii ce alcătuiesc cele 4 triunghiuri care înlocuiesc triunghiul curent (folosind indicii noilor vârfuri create: t_1, t_2, t_3):

```

1 std::vector<unsigned int> newTriangles{
2     i1, t1, t3,
3     t1, t2, t3,
4     t1, i2, t2,
5     t3, t2, i3
6 };

```

Generarea unui con

Pentru a genera un con am folosit reprezentarea parametrică [3]:

$$\begin{cases} x = v \sin(u) \\ y = v & u \in [0, 2\pi], v \in [0, \text{inaltime}]. \\ z = v \cos(u) \end{cases}$$

Am găndit conul ca fiind alcătuit din mai multe părți orizontale (nivele) și verticale (sectoare). Conul este astfel alcătuit din mai multe trapeze. Pentru fiecare sector am parcurs fiecare nivel (de jos în sus) și am generat un punct la acele coordonate.

Dacă indicele vârfului din stânga jos al unui trapez este i , atunci indicele celui din stânga sus este $i + 1$, indicele celui din dreapta jos este $i + stacks + 1$ (unde $stacks$ reprezintă numărul de nivale) pentru că sunt $stacks + 1$ vârfuri pe fiecare sector (vârful conului se repetă) iar indicele celui din dreapta sus este $i + stacks + 2$. Acești indici au fost folosiți pentru a desena 2 triunghiuri ce formează un trapez.

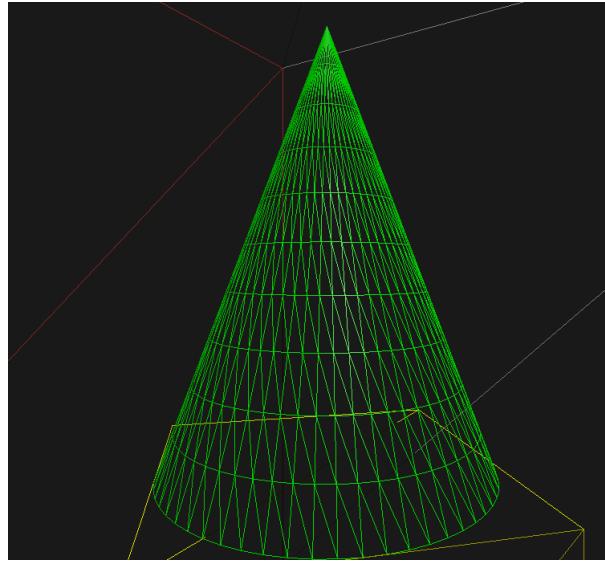


Figura 4.8: Con alcătuit din triunghiuri

4.6 Shaders

Am folosit cele 2 tipuri de programe de tip shader — *vertex* și *fragment*. Pentru fiecare model de iluminare am folosit un alt *fragment shader* pentru că iluminarea se face pentru fiecare pixel, și un singur *vertex shader*, pentru că parametrii vârfurilor obiectelor nu se modifică pentru fiecare model de iluminare.

4.6.1 Clasa Shader

Clasa *Shader* se ocupă de citirea și încărcarea în GPU a programelor de tip shader dar și de setarea variabilelor de tip *uniform*⁴ prin metode precum *setInt*, *setFloat* etc.

Un *program* este alcătuit din *vertex shader* și *fragment shader* (optional și *geometry shader* pe care nu l-am folosit). Codul sursă pentru *vertex shader* și *fragment shader* este încărcat în memorie — *glShaderSource* , compilat — *glCompileShader*, atașat unui

⁴Aceste variabile pot fi modificate din programul principal. Pentru a le identifica ușor, am adăugat „*u_*” ca prefix.

program creat anterior — *glAttachShader*, apoi are loc procesul de *linking* pentru program — *glLinkProgram*. [19, p. 87]

Acste lucruri sunt abstractizate în metoda *load* din clasa *Shader*. Astfel, un program shader este încărcat într-o linie de cod:

```
1 Shader shader;  
2 shader.load("source.vert", "source.frag");
```

Templates

Fiindcă sunt mai multe fișiere pentru diferite *fragment shaders* care au în comun multe linii, am implementat o preprocesare a codului sursă pentru a combina conținutul fișierelor. Metoda este inspirată din framework-ul *Laravel*⁵.

Am folosit *@include "<fișier>"* pentru a include conținutul unui alt fișier.

Am folosit *@has "<nume_sectiune>"* și *@extends "<nume_fișier>"*, *@section "<nume_sectiune>"* pentru a folosi un fișier ca „șablon” și a include părți din alt fișier în anumite locuri (sectiuni).

4.6.2 Vertex shader

În *vertex shader* se primesc parametrii pentru un vârf (poziție, coordonate texturare etc.). Se calculează poziția vârfului în spațiu (cu matricele de modelare, proiecție și vizualizare), se calculează bitangenta ca fiind produsul vectorial între normală și tangentă iar cu ajutorul celor 3 vectori se calculează o matrice de transformare pentru procesul de *normal mapping* [7, p. 314]. Tot aici se calculează și poziția vârfului din perspectiva fiecarei lumini, lucru ce va fi folosit pentru umbre.

Acste informații sunt transmise către *fragment shader*.

4.6.3 Fragment shader

Fișierul *base_shader.frag* este şablonul folosit. Lucrurile principale care au loc aici sunt:

- se calculează vectorul unitate de la fragment (pixel) către fiecare sursă de lumină și către observator
- este apelată funcția *BRDF(geometryTerm, lightDir, normal, viewDir)* a cărei definiție este în fișiere separate, pentru fiecare model (*@section "BRDF_implementation"*).
- se adaugă componența ambientală la rezultatul final (aproximare constantă)

⁵<https://laravel.com/docs/10.x/blade>

Phong

Am folosit formula 3.3 pentru BRDF-ul Phong.

În primele linii am calculat componenta difuză, am extras componenta difuză (culoarea) din textură (dacă este folosită) sau am folosit-o pe cea din material. Pentru culorile specificate în material am aplicat *gamma correction* (mai multe detalii în secțiunea *Gamma correction 4.14.1*).

Termenul difuz este această culoare extrasă înmulțită cu o constantă (din interfața grafică) pentru mai mult control.

Pentru componenta speculară am avut nevoie de direcția reflexiei razei de lumină în raport cu normala și am folosit o funcție deja existentă `reflect`. Am calculat cosinusul unghiului dintre această direcție și direcția către observator folosind produsul scalar. Valori mai mici ca 0 înseamnă că unghiul este mai mare ca 90° deci componenta speculară nu este vizibilă.

Am extras parametrii α (*alpha*) și k_s (*specFactor*) din texturi (dacă este cazul).

Blinn-Phong

Codul sursă pentru modelul Blinn-Phong 3.4 este identic cu cel pentru modelul Phong. Diferența este la calculul componentei speculare, unde se folosește vectorul care împarte unghiul dintre direcția către sursa de lumină și direcția către observator în părți egale:

```
vec3 halfway = normalize(lightDir + viewDir);
```

Cook-Torrance

Pentru modelul Cook-Torrance, am creat mai multe funcții pentru funcțiile D (`D_Beckmann`, `D_Phong`, `D_GGX`), G (`G2_Beckmann`, `G2_U_Beckmann`, `G2_GGX`, `G2_U_GGX`) și F (`F_Schlick`).

În funcția *BRDF* am folosit funcțiile D și G alese din interfața grafică apoi am calculat termenul specular (termenul difuz este luat din material/textura și este o culoare):

```
specular = (fresnel * slope_distribution * geometrical_attenuation) / (4 * NL * NV);
```

Rezultatul final este: `mix(diffuse, specular, metallic)` unde `metallic` este parametrul s din formula 3.5 ($d = 1 - s$). Variabila `metallic` reprezintă cât de „metalic” este materialul, valoarea 1 însemnând că doar componenta speculară este vizibilă. Am limitat valoarea la 0.005 pentru ca efectul Fresnel să fie vizibil și pentru dielectrice.

4.7 Evenimente

Pentru a detecta apăsarea tastelor sau mișcarea mouse-ului am folosit funcții din librăria *GLFW*[15]. Funcțiile folosite sunt:

glfwSetFrameBufferSizeCallback : pentru a detecta redimensionarea ferestrei

glfwSetKeyCallback : pentru a detecta faptul ca o tastă este apasată

glfwSetCursorPosCallback : pentru a detecta poziția mouse-ului

glfwSetMouseButtonCallback : pentru a detecta faptul că un buton al mouse-ului este apăsat

Pentru controlul interfeței grafice, librăria *Dear ImGui* [6] prelucrează evenimentele automat însă pentru restul aplicației ele au trebuit tratate.

Pentru acest lucru am folosit şablonul de proiectare comportamental (behavioral design pattern) *Observer*. Clasa `EventManager` este cea care primește evenimentele (trimise de GLFW cu funcțiile menționate anterior), creează obiecte ce reprezintă evenimentele și își întrează obiectele care doresc să primească evenimente.

Clasa Event

Această clasă reprezintă un eveniment din aplicație și poate fi de diferite tipuri în funcție de ce reprezintă (buton apăsat, mouse mișcat etc). Se rețin date despre eveniment cum ar fi: ce tastă a fost apăsată, care este poziția mouse-ului etc.

Clasa EventHandler

`EventHandler` este o interfață, ce conține doar o metodă abstractă `handleEvent`. Dacă o clasă dorește să prelucreze evenimente, trebuie să implementeze această metodă.

Clasa EventManager

Această clasă este responsabilă de centralizarea funcțiilor care prelucrează evenimente și distribuirea lor altor obiecte interesante ce s-au înregistrat anterior (se menține o listă cu obiectele interesante — `std::vector<EventHandler*> m_handlers`). Funcțiile `set*Callback` din librăria *GLFW* primesc ca argument o funcție ce este apelată când apare un eveniment iar aceste funcții fac parte din clasa `EventManager`. Un eveniment este transmis tuturor obiectelor înregisterate indiferent de ce tip este evenimentul, ceea ce în alte aplicații poate nu este de dorit însă în acest caz a fost suficient.

Librăria *Dear ImGui* are nevoie de evenimente pentru a funcționa însă ele sunt procesate independent. Totuși, în cazul în care interfața grafică este deasupra ferestrei principale, am dorit ca evenimentele să fie tratate doar de *Dear ImGui*, astfel am folosit o funcționalitate a librăriei care mi-a permis să verific dacă evenimentul curent este tratat de *Dear ImGui*.

Pentru că nu are sens să existe mai multe obiecte de tipul `EventManager` am folosit şablonul de proiectare creaţional (creational design pattern) *Singleton*, care presupune existenţa unui singur obiect de tipul unei anumite clase.

Operatorul de atribuire şi constructorul de copiere sunt şterse iar pentru a se obține obiectul se apelează metoda statică `getInstance`.

4.8 Camera

Pentru a vizualiza o scenă 3D pe un ecran 2D este nevoie, pe lângă altele, de transformări ale obiectelor 3D în reprezentări 2D. Sunt utilizate 3 matrice cu denumirile *model*, *view*, *projection*.

Matricea *model* este folosită pentru translaţii, scalări, rotaţii ale obiectelor care de obicei sunt centrate în originea sistemului de coordonate atunci când sunt create.

Matricea *view* este folosită pentru a simula utilizarea unei camere. Convenţia este că există o „camera” plasată în origine care priveşte în direcţia $-Z$ pentru a simplifica calculele. Folosind această matrice, schimbăm sistemul de coordonate pentru a simula orientarea camerei într-o direcţie.

Matricea *projection* este folosită pentru a trece de la o reprezentare 3D la o reprezentare 2D care poate fi apoi afişată pe un ecran.

Pentru crearea matricelor *model*, *view* şi *projection* am folosit librăria *GLM*[9].

4.8.1 Clasa Camera

Această clasă este responsabilă pentru obținerea unei noi matrice *view* atunci când poziţia sau orientarea sa este schimbată de utilizator prin apăsarea tastelor sau mişcarea mouse-ului.

Fiindcă trebuie să trateze evenimente, clasa implementează interfaţa *EventHandler*. Pentru a se deplaza se folosesc tastele *W*, *A*, *S*, *D* pentru deplasare în faţă, stânga, înapoi, dreapta şi tastele *R*, *F* pentru deplasare în sus şi în jos. Pentru a orienta camera se mişcă mouse-ul în timp ce este apăsat butonul din stânga al mouse-ului.

Matricea *view*

La un moment dat sunt cunoscute poziţia camerei şi direcţia în care observă. Știind acestea se pot calcula 3 vectori unitate ce reprezintă noul sistem de coordonate: `m_direction`, `m_right`, `m_cameraUp`.

Vectorul `m_UP` reprezintă direcţia verticală a scenei, care de obicei (şi în acest caz) este $(0, 1, 0)$. Vectorii `m_direction`, `m_cameraUp` reprezintă, din perspectiva camerei, în ce

direcție este orientată camera și în ce direcție este „sus” iar vectorul `m_right` este produsul vectorial dintre cei doi vectori.

Inițial `m_direction` este $(0, 0, -1)$ — camera este orientată spre $-Z$, `m_cameraUp` este $(0, 1, 0)$ — direcția verticală din punctul de vedere al camerei este $+Y$ și `m_right` este $(1, 0, 0)$. Acești vectori sunt modificați atunci când utilizatorul mișcă mouse-ul.

Matricea `view` este [7, p. 96]:

$$\underbrace{\begin{bmatrix} m_right.x & m_right.y & m_right.z & 0 \\ m_cameraUp.x & m_cameraUp.y & m_cameraUp.z & 0 \\ -m_direction.x & -m_direction.y & -m_direction.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{rotatie / schimbare reper}} \cdot \underbrace{\begin{bmatrix} 1 & 0 & 0 & -m_position.x \\ 0 & 1 & 0 & -m_position.y \\ 0 & 0 & 1 & -m_position.z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{translatie}}$$

Această matrice⁶ este creată în metoda `update`.

Actualizarea poziției și orientării

Când una din tastele W, A, S, D, R, F este apăsată, variabila `m_position` (poziția camerei) este modificată corespunzător.

Când poziția mouse-ului este actualizată se scad coordonatele poziției anterioare și se obțin 2 valori: Δ_x și Δ_y ce reprezintă cât de mult a fost mișcat mouse-ul pe orizontală și verticală.

Orientarea „sus-jos” a camerei este modificată în funcție de Δ_y prin aplicarea unei rotații pe axa dată de `m_right`: `glm::rotate(glm::radians(deltaY), m_right);`; iar orientarea „stânga-dreapta” este modificată în funcție de Δ_x prin aplicarea unei rotații pe axa dată de `m_UP` (direcția verticală a scenei): `glm::rotate(-glm::radians(deltaX), m_UP);`.

Pentru ca `m_direction` să nu fie coliniar cu `m_UP` (acest lucru este necesar pentru ca `m_right` să fie calculat corect), verific dacă cosinusul unghiului dintre acești vectori este mai mic ca 0.995 și mai mare ca -0.995 (în caz contrar nu efectuez rotația).

4.9 Surse de lumină

În aplicație am utilizat 3 tipuri de surse de lumină. Sursele sunt reprezentate de un singur punct în spațiu și sunt de mai multe tipuri:

Point (punctuală) : lumina este emisă în toate direcțiile (ex. un bec)

Directional (direcțională) : sursa de lumină este foarte departe încât razele de lumină pot fi considerate paralele (ex. Soarele)

⁶În libraria `GLM` și în OpenGL matricele sunt stocate pe coloane. Astfel `glm::vec4(m_right.x, m_cameraUp.x, -m_direction.x, 0.0f)` reprezintă prima coloană.

Spot : un caz particular de sursă de lumină punctuală, lumina este emisă predominant într-o direcție și scade din intensitate (până la 0) cu cât ne abatem de la aceasta direcție (ex. o lanternă)

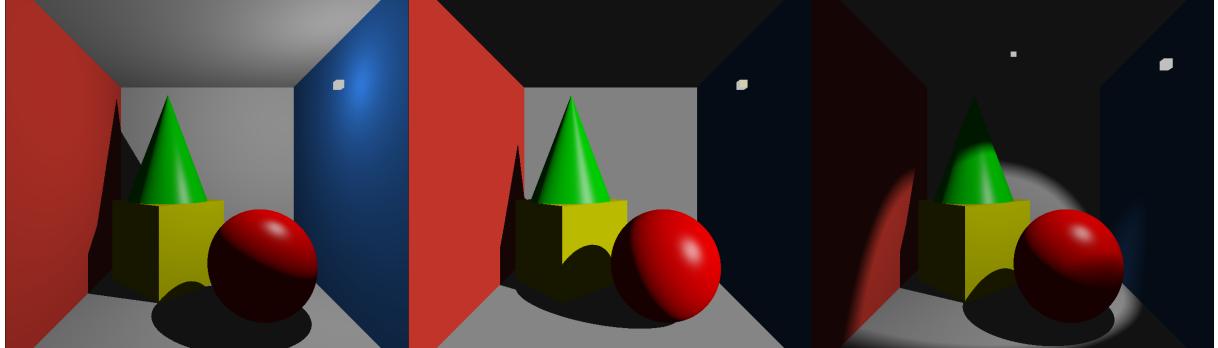


Figura 4.9: Tipuri de surse de lumină, stânga: *point light*, centru: *directional light*, dreapta: *spot light*

4.9.1 Atenuare

Pentru surse de lumină punctuale (inclusiv spot), intensitatea luminii scade cu pătratul distanței [10, p. 111]. Pentru a implementa acest efect am folosit formula [7, p. 141]:

$$\frac{1}{c_0 + c_1 d + c_2 d^2},$$

unde c_1 și c_2 sunt parametri ce controlează cât de rapid scade intensitatea luminii în funcție de distanță și pătratul ei iar c_0 asigură că numitorul este mai mare ca 1. Această formulă permite schimbare cu ușurință a efectului de atenuare în funcție de preferințe.

4.9.2 Spotlight

Pentru a realiza efectul *spotlight*, am folosit formula [10, p. 115]:

$$\left[\left(\frac{\cos \theta_s - \cos \theta_u}{\cos \theta_p - \cos \theta_u} \right)_{[0,1]} \right]^2,$$

unde $x_{[0,1]}$ înseamnă că valorile mai mici decât 0 devin 0 iar valorile mai mari ca 1 devin 1. θ_s este unghiul dintre direcția în care sursa de lumină este orientată și punctul pentru care este calculat efectul. Pentru valori mai mici decât θ_p punctul este iluminat în mod normal iar pentru valori mai mari ca θ_u punctul nu este iluminat.

Dacă $\theta_s < \theta_p$ rezultă că $\cos \theta_s > \cos \theta_p$ deci numărătorul din formulă este mai mare ca numitorul iar rezultatul este 1 (se păstrează intensitatea curentă).

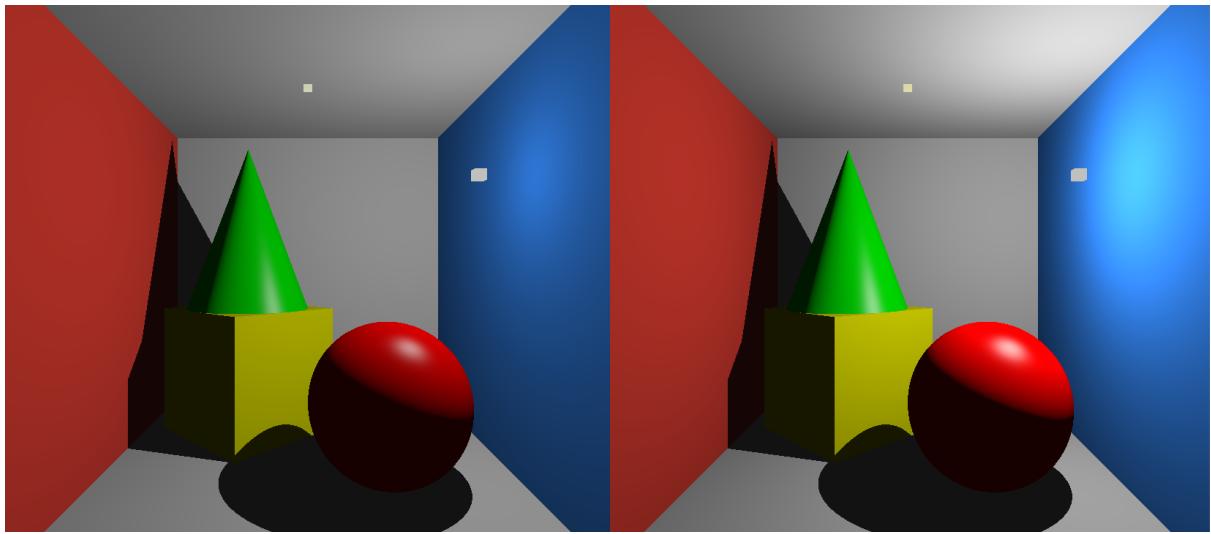


Figura 4.10: Comparație a efectului de atenuare. Stânga: $\text{intensitate}=1.0$, $c_1 = 0.1$, $c_2 = 0.04$. Dreapta: $\text{intensitate}=8.0$, $c_1 = 0$, $c_2 = 1$. (c_0 este 1 în ambele cazuri)

Dacă $\theta_s > \theta_u$ rezultă că $\cos \theta_s < \cos \theta_u$ deci numărătorul din formulă este negativ și rezultatul final este 0 (punctul nu este iluminat).

Pentru alte unghiuri funcția ia valori între 0 și 1 pentru a exista o trecere treptată între lipsa iluminării și prezența ei.

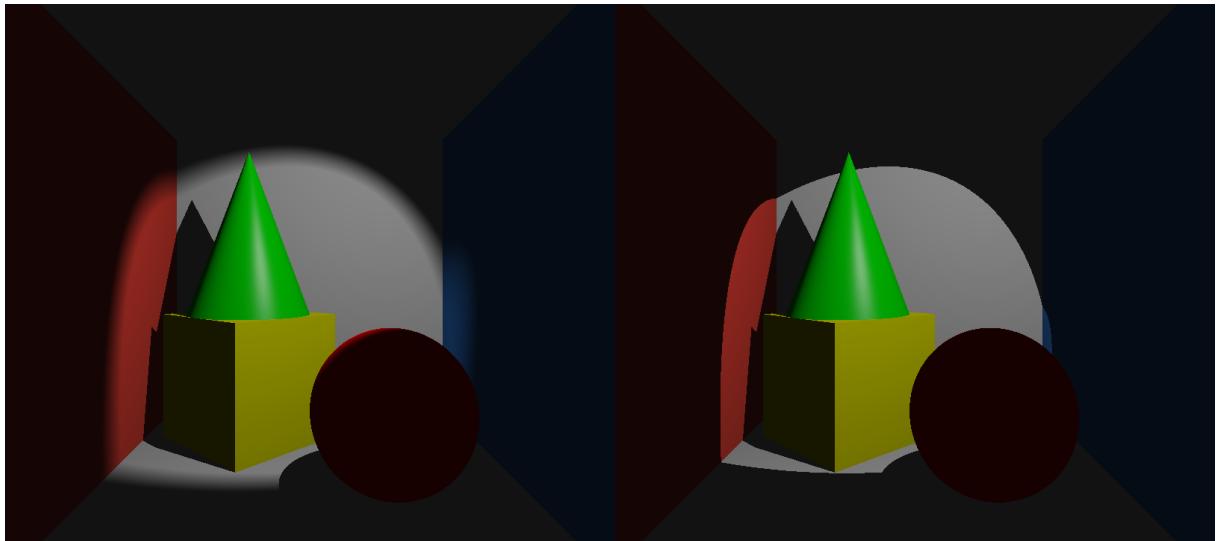


Figura 4.11: Efectul *spotlight*, în stânga $\theta_p = 30^\circ$ și $\theta_u = 35^\circ$, în dreapta $\theta_p = \theta_u = 30^\circ$

4.9.3 Clasa Light

Există o clasă abstractă `Light`, care este moștenită de clasele corespunzătoare celor 3 tipuri de surse de lumină. În această clasă există variabile comune celor 3 tipuri, cum ar fi poziție, culoare, intensitate.

Clasele corespunzătoare tipurilor de surse de lumină au variabile în plus cum ar fi (c_0 , c_1 , c_2 pentru atenuare sau θ_u , θ_p pentru efectul *spotlight*). În plus fiecare clasă conține și o metodă `imGuiRender` pentru redarea interfeței grafice care diferă în funcție de tip.

4.9.4 Normal mapping

Normal mapping[7, p. 314] este un mod de a simula o suprafață complexă (cu denivelări) prin perturbarea normalei folosind valori dintr-o textură. Normalele sunt stocate în textură în *tangent space* (un spațiu local), iar ele trebuie transformate în *world space* (spațiul în care sunt realizate calculele). Pentru această transformare este nevoie de o matrice TBN formată din cei 3 vectori: tangentă, bitangentă (produs scalar între normală și tangentă), normală. [7, p. 318].

Noua normală este extrasă din textură și transformată folosind matricea TBN (care este calculată folosind informațiile din vârful respectiv).

4.10 Umbre

Umbrele sunt un efect ce oferă mult realism unei scene.

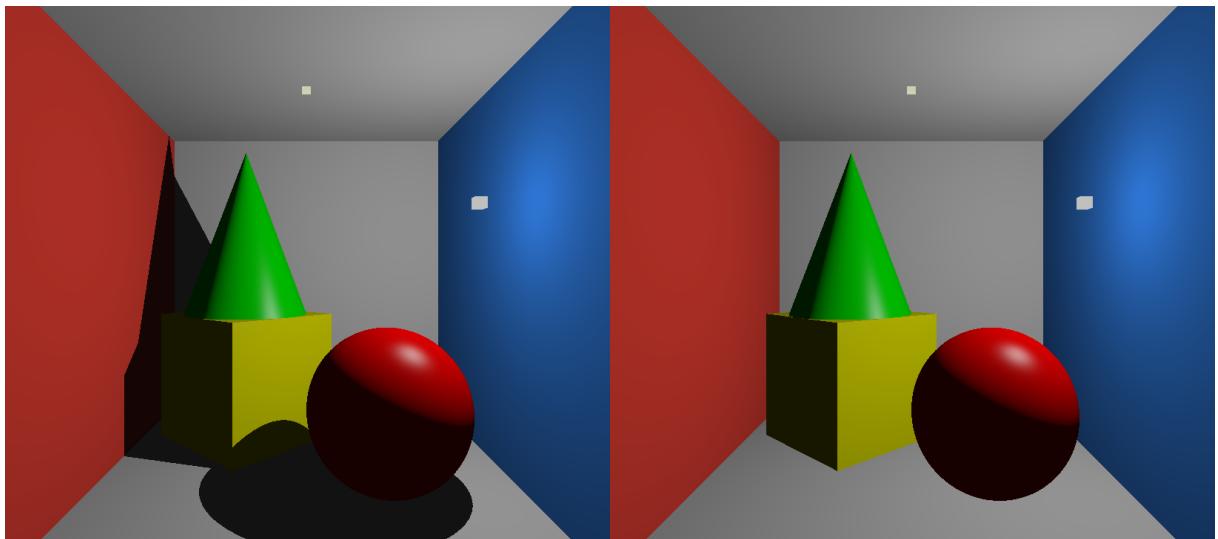


Figura 4.12: Comparație între o scenă cu umbre și fără umbre

Tehnica folosită a fost *shadow mapping*[10, p. 234][7, p. 285]. Ea presupune randarea scenei din perspectiva fiecărei sursă de lumină pentru a reține distanța de la sursa de lumină la obiectele cele mai apropiate.

Pentru fiecare scenă cu umbre a fost creat un *framebuffer* folosit pentru obținerea distanțelor. Fiindcă nu este nevoie de alte informații, acest framebuffer conține doar atașamentul de adâncime (textură).

Este nevoie de noi matrice *view* și *projection* în funcție de ce tip de sursă de lumină este folosit. Matricele *view* au fost create cu ajutorul funcției *lookAt* din librăria *GLM*. Funcție primește 3 argumente: poziția observatorului (sursei de lumină), punctul/direcția în care observatorul privește (pentru surse de lumină direcționale este $(0, 0, 0)$, pentru cele spot este direcția în care sursa de lumină este orientată iar pentru cele punctuale sunt 6 valori în funcție de direcțiile $(\pm 1, 0, 0)$, $(0, \pm 1, 0)$, $(0, 0, \pm 1)$).

4.10.1 Fragment shader

Codul sursă din *fragment* shader-ul folosit conține doar câteva linii și este adaptat după codul sursă din [7, p. 306]

```
1 // ...
2 vec3 dist = fragPos.xyz - u_lightPos;
3 gl_FragDepth = dot(dist, dist) / pow(u_farPlane, 2);
```

În prima linie am calculat distanța de la fragment către sursa de lumină. Variabila *u_farPlane* reprezintă limita maximă de vizualizare a unei scene, dacă un obiect este la o adâncime mai mare de această valoare el nu este vizibil. În acest caz am dorit să rețin în atașamentul de adâncime distanța dintre sursa de lumină și obiect.

Pentru a obține o valoare între 0 și 1⁷ am dorit să memorez valoarea $\frac{\sqrt{\text{dot}(\text{dist}, \text{dist})}}{\text{u_farPlane}}$ însă pentru a evita folosirea radicalului am stocat $\frac{\text{dot}(\text{dist}, \text{dist})}{\text{u_farPlane}^2}$.

4.10.2 Directional light

Matricea *projection* este creată cu ajutorul librăriei *GLM*. Am folosit funcția *ortho* pentru o proiecție ortogonală.

Pentru a randa scena din perspectiva sursei de lumină este nevoie de poziția sursei însă pentru acest tip de sursă nu există poziție. Pentru poziție am ales un punct (pe direcția razelor de lumină) astfel încât scena să fie vizibilă.

În figura 4.10.2 culoarea negru din atașamentul de adâncime reprezintă distanța 0 iar roșu reprezintă distanța maximă. Culoarea roșie apare pentru că atașamentul are doar o valoare și s-a folosit componența roșie a unui pixel (care are 3 valori pentru roșu, verde, albastru).

4.10.3 Spot light

Matricea *projection* este creată cu ajutorul librăriei *GLM*. Am folosit funcția *perspective* pentru o proiecție în perspectivă.

⁷Un element din textura este un număr între 0 și 1.

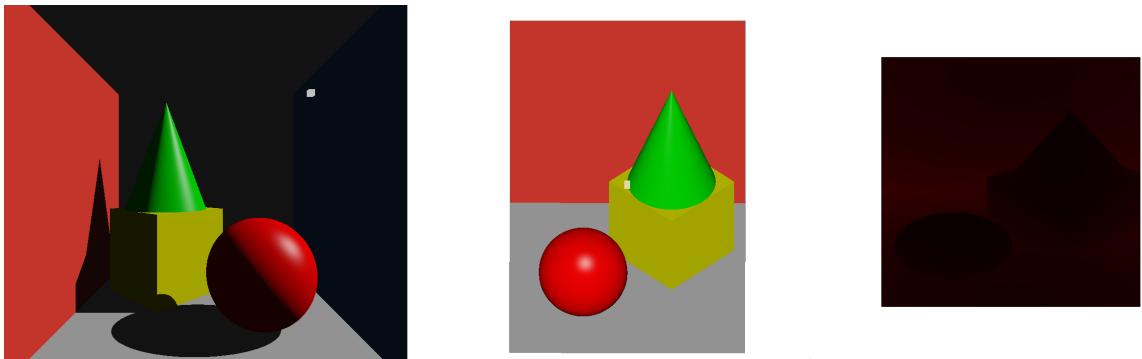


Figura 4.13: Scena cu umbre și o sursă de lumină direcțională. Stânga: scena cu o sursă de lumină direcțională, mijloc: aceeași scenă dar vizualizată cu o proiecție ortogonală din perspectiva sursei de lumină, dreapta: conținutul atașamentului de adâncime.

Procesul este similar cu cel pentru *directional light*. Diferența este că în acest caz se știe poziția sursei de lumină iar sursa de lumină este orientată într-o anumită direcție (posibil diferită de $(0, 0, 0)$).

4.10.4 Point light

Utilizarea umbrelor pentru acest tip de sursă de lumină este mai dificilă pentru că lumina este emisă în toate direcțiile. Pentru celelalte tipuri de surse de lumină atașamentul de adâncime a fost o textură 2D însă aici am folosit un *cubemap* — o textură sub formă de cub. Scena a fost randată de 6 ori, pentru fiecare direcție $(\pm X, \pm Y, \pm Z)$ iar distanțele au fost reținute în *cubemap* (fiecare latură corespunde unei direcții).

Matricea *projection* este asemănătoare cu cea folosită pentru surse de lumină spot, o proiecție în perspectivă. Diferența este că aici este folosit un câmp vizual suficient de mare încât scena să fie complet vizibilă (într-o anumită direcție).

4.10.5 Compararea distanțelor

După ce distanțele de la sursele de lumină până la cel mai apropiat obiect sunt reținute în texturi, acestea sunt utilizate la randarea scenei cu umbre. În shader este nevoie de matricele *view* și *projection* folosite în randarea scenei din perspectiva sursei de lumină pentru a aplica aceeași transformare (aplicată în *vertex shader* și interpolată automat) și a putea obține poziția fragmentului în noul sistem de coordonate, utilizată pentru a extrage distanță din textură.

Distanța curentă este calculată folosind aceeași metodă descrisă mai devreme din *fragment shader*.

Dacă distanța curentă este mai mare decât cea reținută în textură înseamnă că fragmentul curent este în spatele altui fragment (din perspectiva sursei de lumină) și nu este iluminat: `in_shadow = currentDepth > shadowMapDepth + bias ? true : false`

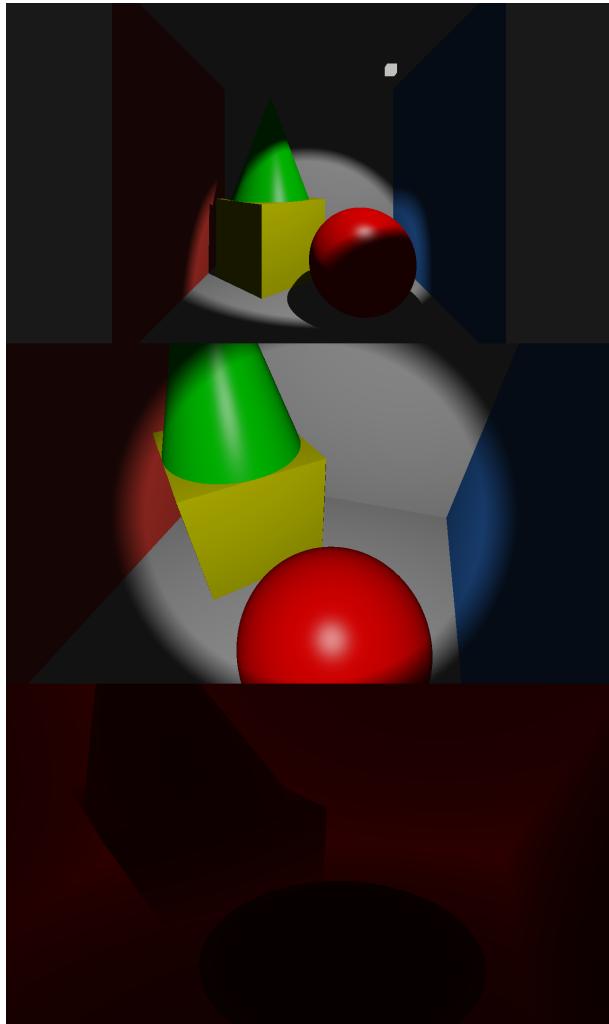


Figura 4.14: Scena cu umbre și o sursă de lumină spot. Sus: scena cu o sursă de lumină spot, centru: aceeași scenă dar vizualizată din perspectiva sursei de lumină, jos: conținutul atașamentului de adâncime

Se adaugă o valoare foarte mică ($bias = 0.001$) valorii din textură din cauza erorilor de precizie. Rezoluția texturii nu este suficient de mare și unele fragmente din scenă cu distanțe diferite dar foarte apropiate pot fi reprezentate de aceeași valoare (același „pixel” din textură). Soluția este să se adauge o mică valoare ca aceste distanțe să fie mărite artificial pentru a contracara efectul erorilor.

Dacă distanța curentă (ridicată la pătrat) este mai mare ca valoarea planului îndepărtat (ridicat la pătrat) utilizat pentru randarea scenei din perspectiva luminii am presupus că fragmentul este iluminat. Acest lucru se poate întâmpla doar dacă fragmentul este prea îndepărtat pentru a fi randat din perspectiva sursei de lumină.

4.10.6 Optimizări

Procesul de *shadowmapping* este costisitor când există mai multe surse de lumină și foarte costisitor dacă se folosesc surse de lumină punctuale pentru că scena trebuie randată

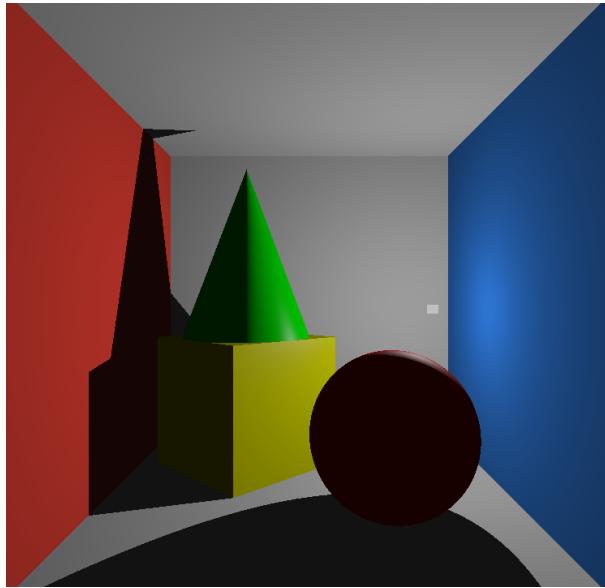


Figura 4.15: Scena cu umbre și o sursă de lumină punctuală

de multe ori.

Pentru a minimiza costul procesului de *shadowmapping* am implementat următoarele optimizări:

Salvarea matricelor *view* și *projection* : aceste matrice sunt salvate și recrate doar atunci când parametrii sursei de lumină sunt modificați (de ex. poziția).

Amânarea randării scenei : scena este randată din perspectiva sursei de lumină și texturile ce rețin distanțele sunt recrate doar când acest lucru este necesar — atunci când parametrii sursei de lumină sunt modificați.

4.11 Texturi

Texturile sunt imagini ce surprind modul în care arată un obiect. Sunt mai multe tipuri de texturi în funcție de utilizarea lor:

albedo/diffuse : aceste texturi reprezintă culorile obiectului când este iluminat în totalitate

roughness : texturi ce reprezintă parametrul α din modele precum Cook-Torrance (0 inseamnă perfect neted, 1 reprezintă prezența multor denivelări microscopice).

metallic : texturi ce reprezintă parametrul *metallic*⁸ din modele precum Cook-Torrance (0 inseamnă dielectric, 1 inseamnă metal).

⁸ *metallic* este parametrul s din formula 3.5. Formula devine: $(1 - metallic) \cdot R_d + metallic \cdot R_s$

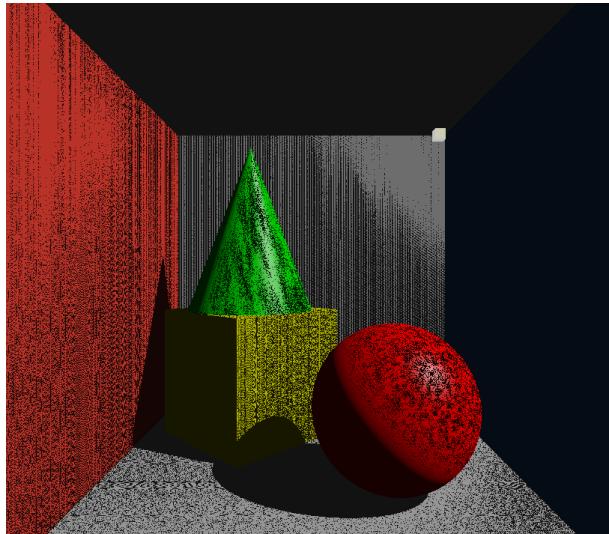


Figura 4.16: Efectul erorilor asupra umbrelor atunci când nu se adaugă *bias*. Fragmentele sunt identificate greșit ca fiind mai aproape de sursa de lumină, prin adăugarea valorii 0.001 efectul este anulat.

normal : texturi ce reprezintă normala în acel punct. Sunt folosite pentru a perturba normala "reală" în acel punct, astfel prin iluminare este simulată o suprafață cu denivelări (asemănător cu o iluzie optică).

emmisiive : texturi ce reprezintă surse de lumină (dacă valoarea pixel-ului este diferită de negru atunci face parte dintr-o sursă de lumină).

Utilizarea texturilor permite un control fin al caracteristicilor obiectelor (culoare, nerezime etc). Texturile au fost preluate de pe platforma *3dtextures.me*⁹.

Pentru încărcarea texturilor am folosit librăria *stb_image* [1].

4.11.1 Clasa TextureManager

Pentru a utiliza diferite texturi am creat o clasă ce se ocupă cu gestionarea lor. A fost utilizat şablonul de proiectare *Singleton*. Am folosit și un sistem de *caching*, pentru a nu stoca aceeași textură de mai multe ori în memorie.

Texturile sunt stocate în data membră *m_textures*:

```
1 std::unordered_map<std::string, std::weak_ptr<Texture>> m_textures;
```

Fiecare textură are asociat un nume (calea spre acel fișier) și un *std::weak_ptr*, un pointer ce poate fi *null* sau poate indica spre un obiect de tip *Texture*.

Când se apelează metoda *getTexture* obțin întâi pointerul:

```
1 std::weak_ptr<Texture> storedTexture = m_textures[path];
```

Încerc să accesez obiectul indicat de el:

⁹<https://3dtextures.me>

```
1 std::shared_ptr<Texture> texture = storedTexture.lock();
```

În cazul în care obiectul nu există (este `null`), este creat:

```
1 texture = std::make_shared<Texture>(path, type, flipY);
```

Citirea și încărcarea texturilor se realizează în clasa `Texture`. Aici imaginea este încărcată în memorie folosind librăria `stb_image`, și încărcată în GPU folosind funcții OpenGL precum `glTexImage2D`.

4.12 Modele

Pentru încărcarea modelelor am folosit librăria *ASSIMP* [14]. Modelele sunt alcătuite din mai multe componente (*meshes*) și texturi. *ASSIMP* încarcă informațiile într-o structură de date de tip arbore. Nodul rădăcină reține informații despre texturi și *meshes* iar fiecare nod reține informații despre ce *meshes* au fost folosite. Un *mesh* conține informații despre vârfuri (poziție, normală, tangentă) și despre indicii (pentru vârfuri) folosiți.

Modelele au fost preluate de pe platforma *sketchfab*¹⁰.

4.13 Materiale

Pentru fiecare model de iluminare am creat o clasă responsabilă pentru gestionarea parametrilor și afișarea lor în interfața grafică dar și de setarea variabilelor de tip *uniform* în shader.

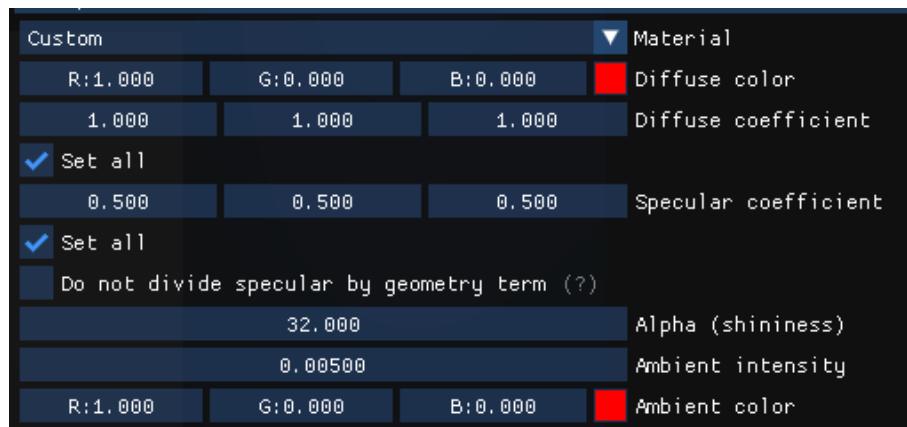


Figura 4.17: Interfața grafică pentru un material ce folosește modelul de iluminare Phong

Am creat câteva exemple de materiale: aur, cupru și fier. Pentru modelul Cook-Torrance am folosit parametrii *F0* din [10, p. 323] iar pentru modelele Phong și Blinn-Phong am configurat parametrii manual pentru a obține un aspect asemănător.

¹⁰<https://sketchfab.com>

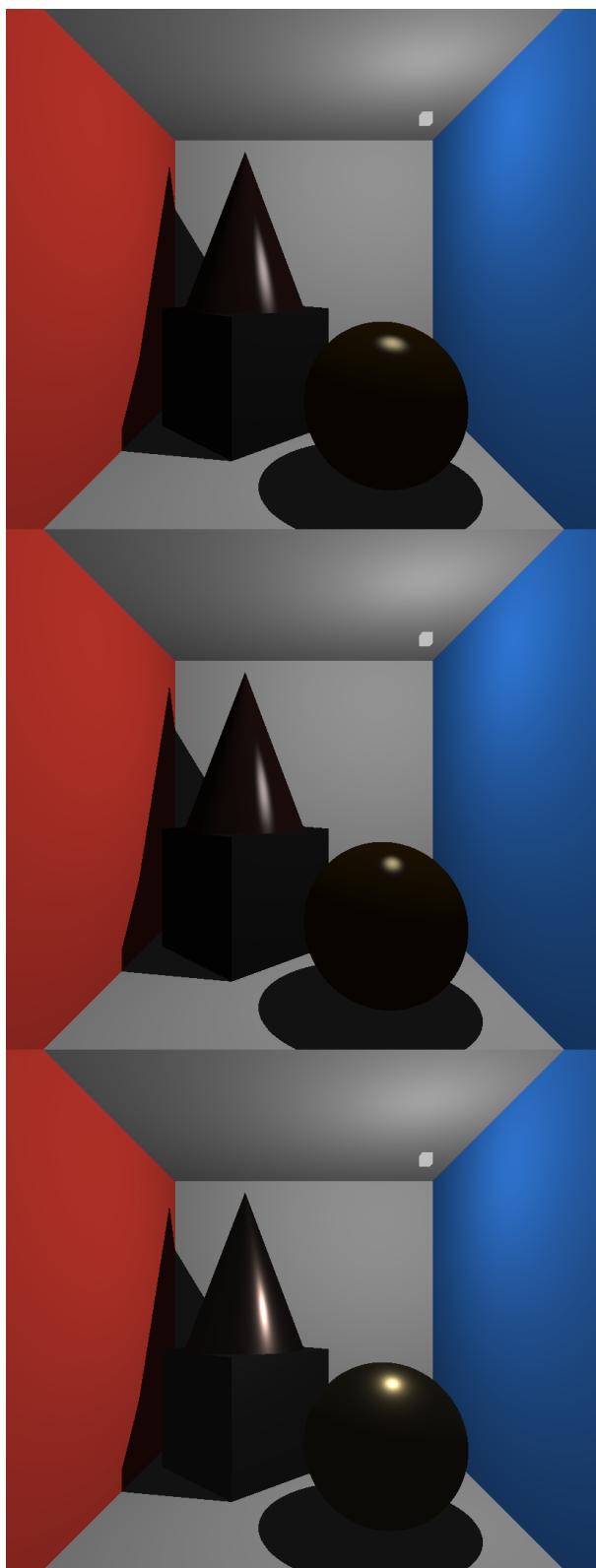


Figura 4.18: Comparație între modele Phong (sus), Blinn-Phong (mijloc) și Cook-Torrance (jos). Materialele folosite pentru sferă, con și cub sunt *aur*, *cupru* respectiv *fier*. Se poate observa că modelul Cook-Torrance arată cel mai realist.

4.14 Post-processing

Efectele de post-procesare sunt aplicate la finalul procesului de randare, asupra unei imagini (texturi). După randarea scenei într-o textură este folosit un alt shader care aplică diferite operații de post-procesare pentru fiecare pixel iar rezultatul final este afișat pe ecran.

4.14.1 Gamma correction

Intensitățile culorilor nu sunt percepute în mod liniar de ochiul uman. În trecut, monitoarele CRT funcționau astfel: prin dublarea voltajului unui pixel intensitatea percepătă de ochiul uman nu este dublată, ci este aproximativ egală cu ridicarea la puterea 2.2[10, p. 161]. Dacă un obiect are culoarea RGB $(0.5, 0.5, 0.5)$ (gri) pe ecran noi vom observa culoarea $\approx (0.21, 0.21, 0.21)$ care este o culoare mai întunecată.

Problema este că modelele folosite (Phong, Blinn-Phong, Cook-Torrance) funcționează în spațiul liniar. Soluția ar fi să anulăm efectul de mai sus prin ridicarea rezultatului la puterea $\frac{1}{2.2}$. Acest procedeu se numește *gamma correction*.

Mai mult, texturile au fost create în aşa fel încât să arate „corect” când sunt afișate pe ecran¹¹, deci a fost aplicată deja transformarea care anulează ridicarea la puterea 2.2. Pentru a obține valori liniare, valorile RGB ar trebui ridicate la puterea 2.2. Acest lucru este posibil și la încărcarea imaginii pe GPU prin funcția `glTexImage2D` și parametrul `GL_SRGB`:

Dacă este aplicat *gamma correction* atunci efectul trebuie anulat și pentru culorile alese din interfața grafică (pentru că s-a aplicat deja *gamma correction*).

4.14.2 Tone mapping

Implicit, un atașament de culoare al unui *framebuffer* retine valori între 0 și 1 pentru componentele RGB (se pot afișa pe ecran doar valori între 0 și 1). Dacă o sursă de lumină este puternică și valorile RGB ale culorilor din scenă sunt mai mari ca 1 ele vor fi pierdute. O soluție ar fi să stocăm valorile chiar dacă sunt mai mari ca 1 apoi, cunoșcând aceste valori, să le transformăm pentru a fi în intervalul $[0, 1]$.[7, p. 339].

Reinhard descrie următoarea formulă pentru a obține valori mai mici ale luminozității și a păstra detaliile din imagine [17]:

$$L_d(x) = \frac{L(x) \left(1 + \frac{L(x)}{L_{\text{white}}^2} \right)}{1 + L(x)}, \quad (4.1)$$

¹¹Este vorba despre texturi precum cele *albedo*, realizate pentru aspect (culori), **nu** despre texturi precum *roughness* care sunt realizate pentru a stoca informație.

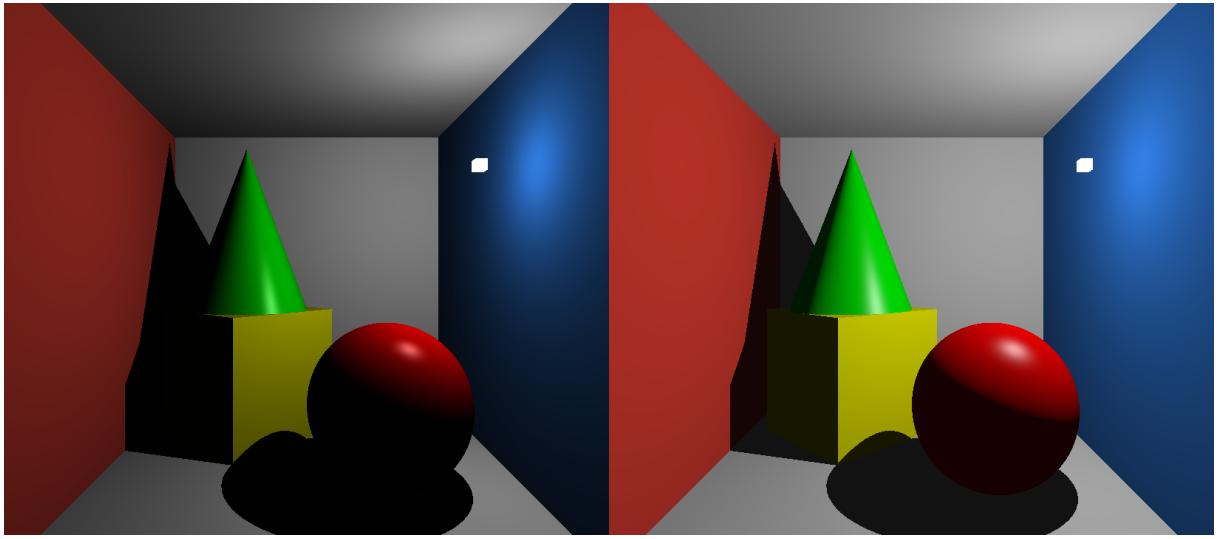


Figura 4.19: Aplicarea procesului *gamma correction*. Stânga: fără *gamma correction*. Dreapta: cu *gamma correction*

unde $L(x)$ reprezintă luminozitatea pixel-ului și L_{white} reprezintă cea mai mică luminozitate care va fi reprezentată de culoarea alb (valoare aleasă în funcție de efectul dorit).

Pentru extragerea luminozității din cele 3 valori RGB am utilizat formula [23, p. 4]:

$$L(x) = 0.2126x_R + 0.7152x_G + 0.0722x_B \quad (4.2)$$

În final, valorile RGB ale unui pixel au fost înmulțite cu raportul dintre rezultatul formulei Reinhard 4.1 și luminozitatea pixelui curent.

4.15 Toon shading

Pe lângă modelele de iluminare care oferă realism unei scene, am implementat și un model de iluminare de tip desen animat. Două caracteristici des întâlnite este utilizarea unui număr redus de culori și aplicarea unor linii de contur obiectelor.

4.15.1 Two-tone shading

Am folosit procedeul *two-tone shading*[10, p. 652] ce constă în colorarea unui obiect folosind 2 culori.

Două variabile tip contor au fost folosite ¹²: `isFullyLit` și `isPartiallyLit`. Pentru un fragment, dacă valoarea cosinusului unghiului dintre normală și direcția spre sursa de lumină este mai mic ca 0.5 `isPartiallyLit` este incrementat altfel `isFullyLit` este incrementat.

¹²Variabilele au fost folosite pentru ca efectul să funcționeze și cu mai multe surse de lumina, deși rezultatele nu sunt foarte satisfăcătoare în aceste cazuri.

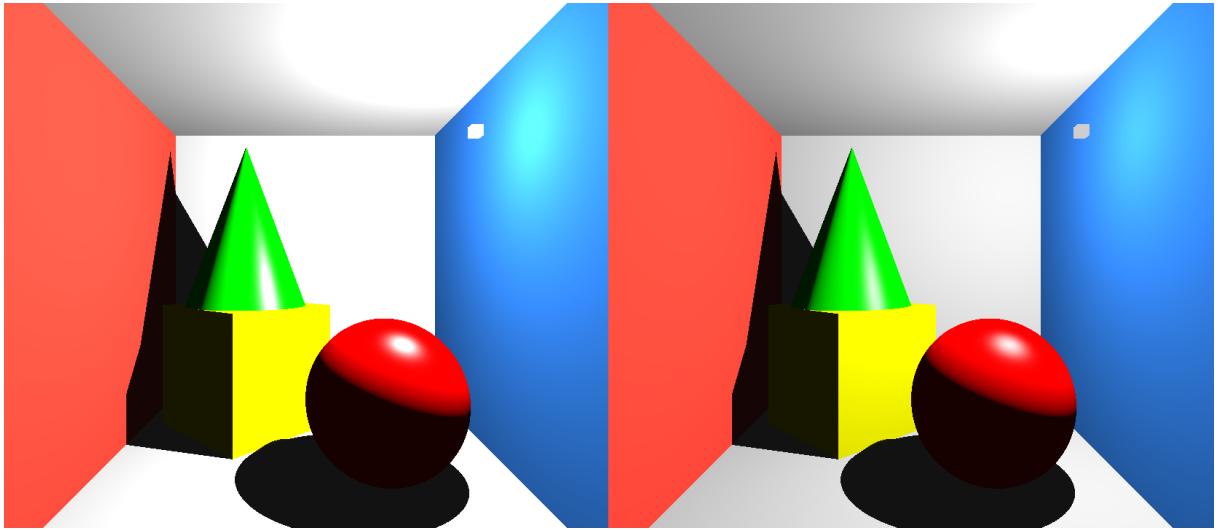


Figura 4.20: Utilizarea procesului *tone mapping*. Stânga: nu este folosit *tone mapping*. Dreapta: este folosit *tone mapping* cu $L_{\text{white}} = 2$.

Dacă `isFullyLit > isPartiallyLit` atunci fragmentul este considerat complet iluminat (și este folosită prima culoare). În caz contrar, fragmentul este considerat parțial iluminat (și este folosită cea de-a doua culoare).

4.15.2 Detectarea liniilor de contur

Detectarea și desenarea liniilor de contur a fost realizat cu metode de post-procesare descrise în [10, pp. 662–663]. Scena a fost randată în 2 două texturi: prima textură conține culorile scenei, folosind *two-tone mapping* iar a doua textură conține normalele fragmentelor respective (în componenta RGB) și distanța față de observator (în componentă A — *alpha*). Distanța a fost folosită pentru a putea extrage în mod corect valorile din texturi (cu cât distanța este mai mare cu atât textura este mai mică).

Normalele au fost folosite pentru procesul efectiv de detectare a liniilor de contur. Folosirea normalelor în locul culorilor permite detectarea liniilor de contur chiar dacă obiectele din scenă au aceleași culori. Desigur această soluție are neajunsuri, dacă două obiecte au aceleași normale nu vor fi detectate liniile de contur. Există metode pentru a îmbunătăți procesul, descrise în [10], însă nu le-am implementat.

Operatorul Sobel

Pentru a detecta liniile de contur am folosit *operatorii sobel*: 2 filtre de dimensiune 3×3 (*convolution kernels*) [8, p. 136]:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (4.3)$$

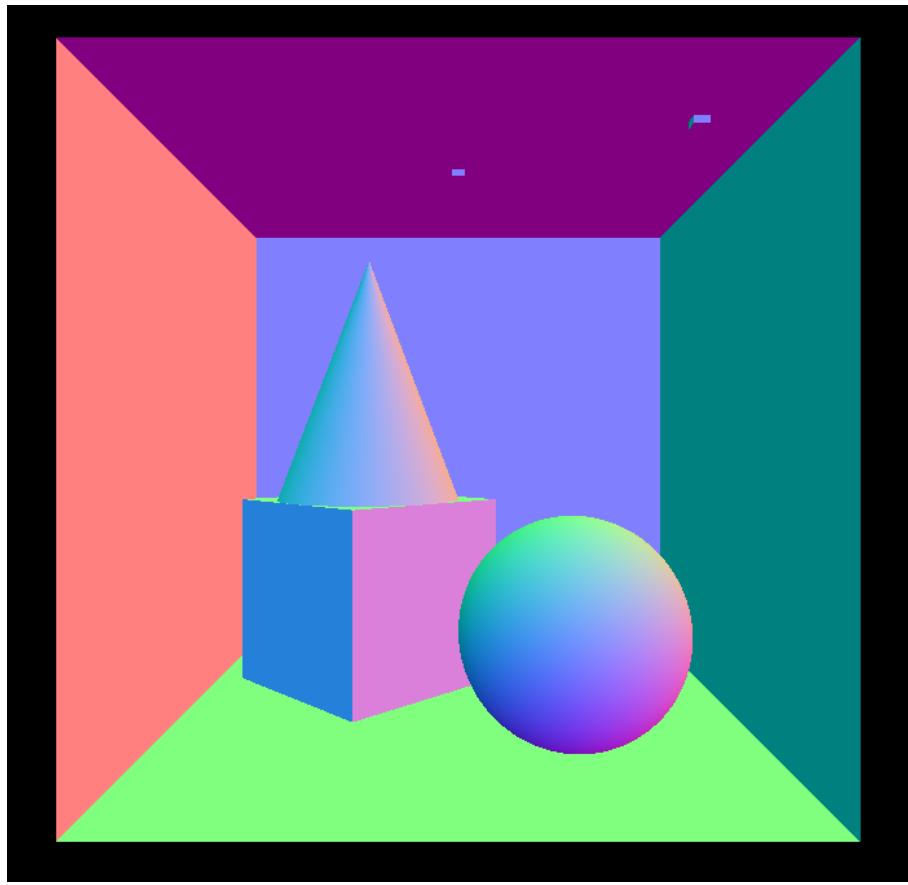


Figura 4.21: Scenă randată folosind normalele în fiecare punct

Prin utilizarea acestor filtre pentru fiecare pixel se obțin 2 valori g_x și g_y , care se combină folosind formula: $g = |g_x| + |g_y|$ [8, p. 135].

Pentru a verifica dacă pixelul curent este o linie de contur am verificat dacă una din valorile RGB este mai mare ca 0.6 (valori mari reprezintă şanse mai mici pentru un rezultat fals pozitiv).

În final, rezultatul este conținutul texturii ce conține culorile scenei fiind aplicate și liniile de contur.

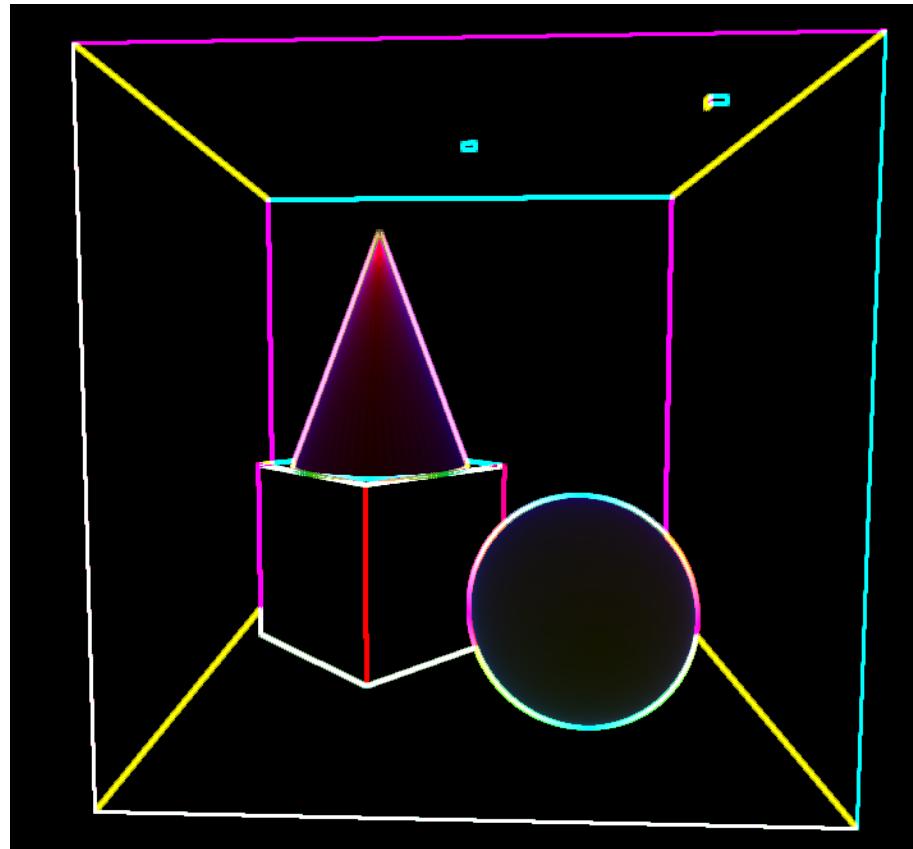


Figura 4.22: Detectarea liniilor de contur folosind *sobel operator*

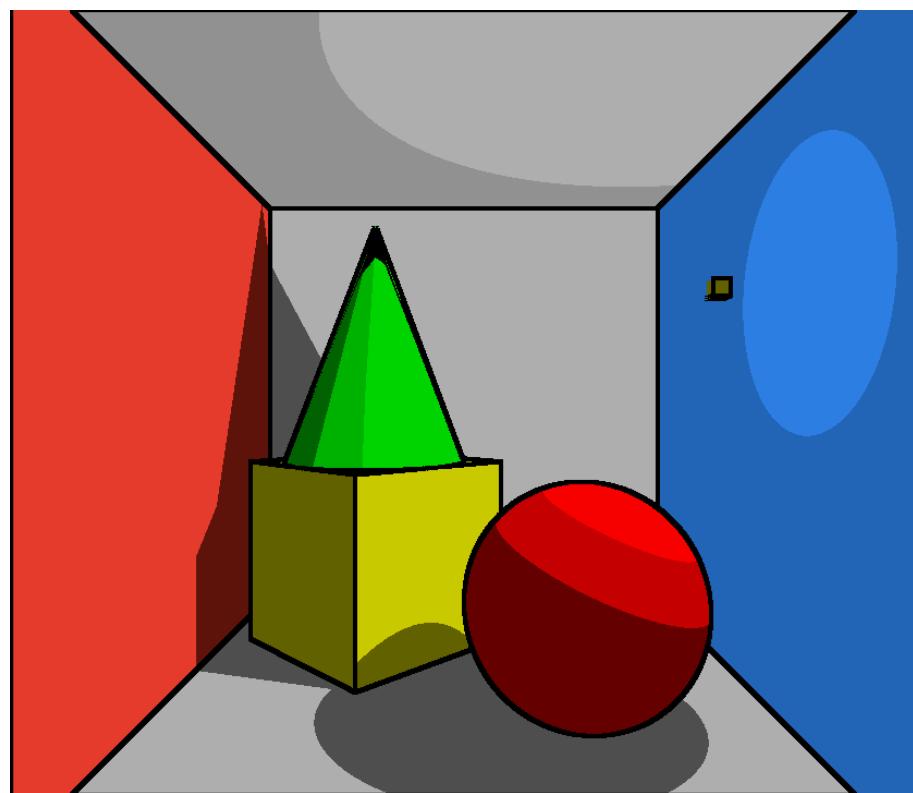


Figura 4.23: Scenă randată cu un model de iluminare tip desen animat

Capitolul 5

Concluzii

Aplicația dezvoltată a explorat diferite modele de iluminare având ca scop realismul dar și un model de iluminare tip desen animat. Modelele Phong și Blinn-Phong sunt modele empirice și eficiente în timp ce Cook-Torrance este un model ce necesită calcule complexe dar este bazat pe fenomene fizice și poate obține rezultate mai bune. Modelul Blinn-Phong este similar cu modelul Phong și puțin mai eficient deoarece vectorul \mathbf{h} este mai ușor de calculat decât vectorul \mathbf{r} și astfel poate fi folosit când resursele sunt limitate (rezultatele fiind acceptabile). Modelul Cook-Torrance poate fi folosit când se dorește obținerea unui rezultat mai realist și când sunt disponibile resurse pentru calculele necesare. În tabelul 5.1 se pot observa diferențele în funcție de timpul necesar randării unui singur cadru din aplicație.

Timp (ms.)	Model
10	Phong
9.5	Blinn-Phong
14	Cook-Torrance
10.5	Toon shading

Tabela 5.1: Comparație a timpilor de randare (în medie) pentru un cadru din aplicație (în scena *Box*) folosind modelele prezentate. Aplicația a fost rulată cu un procesor de tip *Intel i5-8265* și un GPU integrat *Intel UHD Graphics 620*.

În plus au fost implementate tehnici precum *normal mapping*, *shadow mapping* și diferite efecte de post-procesare. De asemenea, am urmărit implementarea folosind concepte de *programare orientată obiect* și separarea funcționalităților în clase.

5.1 Dezvoltări ulterioare

5.1.1 Ray tracing

Ray tracing presupune simularea unor „raze” de lumină pentru a determina iluminarea unei scene [10, p. 443]. Acest procedeu poate obține rezultate mult mai bune întrucât este foarte similar cu modul în care razele de lumina interacționează cu mediul înconjurător.

5.1.2 Iluminare globală

Iluminarea globală este o componentă importantă din iluminarea unei scene. Ea este aproximată în aplicație cu o constantă. Utilizarea unor tehnici bazate pe *radiosity* sau *ray tracing* [10, p. 442] pot obține rezultate realiste.

5.1.3 Ambient occlusion

Ambient occlusion este un alt efect ce amplifică realismul. El simulează prezența umbrelor în zone în care mai multe suprafețe sunt apropiate, cum ar fi în colțurile unei camere.

Bibliografie

- [1] Sean Barrett, *stb*, <https://github.com/nothings/stb>, 2023.
- [2] James F Blinn, „Models of light reflection for computer synthesized pictures”, în *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, 1977, pp. 192–198.
- [3] Mary Boas, *Mathematical Methods In The Physical Sciences*, John Wiley & Sons, 2006.
- [4] Brent Burley și Walt Disney Animation Studios, „Physically-based shading at disney”, în *Acm Siggraph*, vol. 2012, vol. 2012, 2012, pp. 1–7.
- [5] Robert L Cook și Kenneth E. Torrance, „A reflectance model for computer graphics”, în *ACM Transactions on Graphics (ToG)* 1.1 (1982), pp. 7–24.
- [6] Aymar Cornut, *Dear ImGui*, <https://github.com/ocornut/imgui>, 2023.
- [7] Joey De Vries, *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*, Kendall & Welling, 2020.
- [8] Rafael C Gonzales și Paul Wintz, *Digital Image Processing (2nd Edition)*, Prentice Hall, 2001.
- [9] Christophe Groovounet, *OpenGL Mathematics*, <https://github.com/g-truc/glm>, 2023.
- [10] Eric Haines, Naty Hoffman et al., *Real-time rendering*, Crc Press, 2018.
- [11] Eric Heitz, „Understanding the masking-shadowing function in microfacet-based BRDFs”, în *Journal of Computer Graphics Techniques* 3.2 (2014), pp. 32–91.
- [12] Milan Ikits și Marcelo Megallon, *The OpenGL Extension Wrangler Library*, <https://github.com/nigels-com/glew>, 2023.
- [13] James T Kajiya, „The rendering equation”, în *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.
- [14] Kim Kulling et al., *Open Asset Import Library*, <https://github.com/assimp/assimp>, 2023.
- [15] Camilla Löwy, *GLFW*, <https://github.com/glfw/glfw>, 2023.

- [16] Bui Tuong Phong, „Illumination for computer generated pictures”, în *Communications of the ACM* 18.6 (1975), pp. 311–317.
- [17] Erik Reinhard, Michael Stark, Peter Shirley și James Ferwerda, „Photographic tone reproduction for digital images”, în *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 267–276.
- [18] Christophe Schlick, „An inexpensive BRDF model for physically-based rendering”, în *Computer graphics forum*, vol. 13, 3, Wiley Online Library, 1994, pp. 233–246.
- [19] Mark Segal și Kurt Akeley, *The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile) - May 5, 2022)*, The Khronos Group, Mai 2022.
- [20] Dave Shreiner, Bill The Khronos OpenGL ARB Working Group et al., *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*, Pearson Education, 2009.
- [21] Alexander Shvets, „Dive Into Design Patterns”, în *Refactoring. Guru* (2018).
- [22] TS Trowbridge și Karl P Reitz, „Average irregularity representation of a rough surface for ray reflection”, în *JOSA* 65.5 (1975), pp. 531–536.
- [23] International Telecommunication Union, „Parameter values for the HDTV standards for production and international programme exchange”, în *Recommendation ITU-R BT* (2015), pp. 709–6.
- [24] Bruce Walter, Stephen R Marschner, Hongsong Li și Kenneth E Torrance, „Microfacet models for refraction through rough surfaces”, în *Proceedings of the 18th Eurographics conference on Rendering Techniques*, 2007, pp. 195–206.