

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

Cursul 1 - Introducere

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

3 martie 2022

Content



Organizare

Grafică pe calculator

- Istoric
- Aplicații
- Clasificări

Hardware și software pentru grafică

- Placa video și monitorul
- DirectX, OpenGL and Vulkan

Scan Conversion

Bibliografie



Semenstrul 6: 12 săptămâni

- ▶ 10 cursuri
- ▶ 7 laboratoare
- ▶ ultimele două săptămâni - prezentare proiecte

contact: sala C332
tudor.mihoc@ubbcluj.ro



- ▶ Laboratoarele sunt notate fiecare de la 1 la 10. Nota pe laboratoare va fi media aritmetică a notelor obținute pe laborator.
- ▶ Un referat și proiect notat de la 1 la 10.
- ▶ Examen scris notat de la 1 la 10.

Nota finală va fi:

$$\text{Nota} = 0.4 * \text{examen} + 0.3 * \text{laborator} + 0.2 * \text{referat/proiect}$$



- ▶ strâns legată de evoluția calculatoarelor în general
- ▶ creșterea puterii de calcul → grafică tot mai complexă
- ▶ în 1960 William Fetter a introdus conceptul de *computer graphics* [1]

era Supervisor of Advanced Design Graphics la Wichita Branch of the Military Aircraft Systems Division of Boeing

se căutau tehnici ce să permită desenul în perspectivă pe calculator pentru a ajuta la designul avioanelor



Rapid instrumentele grafice au fost preluate de artiști!

- ▶ singurele limite au fost **imaginea și limita computațională**
- ▶ în anii '80 au apărut calculatoarele personale → grafica în industria de enterainment
 - animații, jocuri video, efecte CGI – Star Wars de exemplu
 - de la simple desene 2d la fotorealism

- ▶ vizualizări – pentru a studia tendințe și modele;
- ▶ educație – pentru a înțelege și explica sisteme complexe, sau a antrena în condiții sigure experți în simulatoare;
- ▶ industria de entertrainment – este folosită pentru a genera conținut grafic în filme, videoclipuri, televiziune, etc.

Clasificarea metodelor



- * **Raster Graphics** – vom folosi de pixeli (puncte)
– o imagine bitmap (formată dintr-o secvență de pixeli)

- * **Vector Graphics** – se folosesc formule matematice (din geometria analitică)

Metodele moderne combină tehnici din aceste categorii.



Avem 2 tipuri de algoritmi:

- ▶ **Image order algorithms** – trec prin toate punctele ce trebuie reprezentate pe ecran
 - dacă scena este mult mai complexă decât imaginea (ultimele jocuri, filme etc.)
 - exemplu de algoritm: *ray casting*
- ▶ **Object order algorithms** – tratează obiectele care formează scena ce trebuie reprezentată
 - scene tipice, simple, care conțin mult mai puține elemente geometrice decât pixeli



card de expansiune – funcția principală: generarea de imagini ce pot fi afișate pe un monitor

- ▶ alte funcționalități: redarea accelerată de scene 3D, grafică 2D, adaptor TV tuner, decodare MPEG-2/MPEG-4, capacitatea de a utiliza mai multe periferice pentru redare, etc.
- ▶ unele suportă tehnologii avansate precum ray-tracing (perțul lor e încă mare)
- ▶ procesorul de pe această placa se numește Graphics Processing Unit (GPU) – specializat în manipularea extrem de rapidă a informațiilor pentru a crea imagini
- ▶ unele pot fi integrate în placa de bază (nivel scăzut de performanță versus costuri reduse; consumă resurse RAM)
- ▶ multiple nuclee – axată pe calcul paralel, extrem de rapid motiv pentru care a avut un impact profund în alte domenii din informatică (AI, cripto-currency, etc.)



O varietate imensă de tipuri cu diverse caracteristici.

A evoluat ca tehnologie de la cele cu *Tuburi catodice* la cele cu *Thin Film Transistor*.

Proprietăți:

- ▶ luminozitatea – candele pe metru pătrat (cd/m^2)
- ▶ diagonală – mărimea imaginii vizualizabile
- ▶ rezoluția ecranului – numărul de pixeli pe fiecare dimensiune
- ▶ distanța dintre subpixeli de aceeași culoare (milimetri)
- ▶ rata de reîmprospătare
- ▶ timpul de răspuns (milisecunde)
- ▶ raportul de contrast
- ▶ consumul de energie
- ▶ raportul de aspect – lungimea orizontală : lungimea cea verticală
- ▶ unghiul de vizualizare



- ▶ *DirectX* – colecție de API-uri pentru controlul funcțiilor multimedia dezvoltate de Microsoft (mai lent, creat pentru platformele Microsoft);
- ▶ *OpenGL* – API ce permite crearea imaginilor 2D și 3D (varietate imensă de aplicații, pe diverse platforme și sisteme) dezvoltat de Khronos Group;
- ▶ *Vulkan* – API ce permite programatorului să lucreze direct cu procesorul grafic dezvoltat de asemenea de Khronos Group.

Vulkan se va impune probabil în viitorul apropiat însă deocamdată plăcile grafice comune oferă un suport relativ limitat (extrem de scumpe).



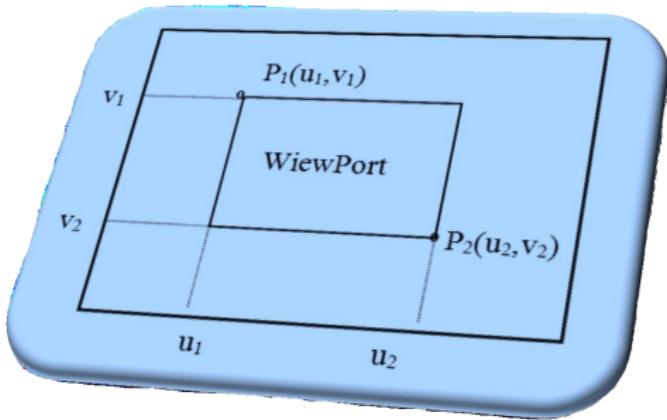
format dintr-o matrice de puncte – **pixeli**

Un Pixel:

- ▶ poate fi aprins (colorat) într-o anumită culoare
- ▶ poziția sa este definită prin coordonatele sale u și v
 - ▶ $0 \leq u < D_{mx}$, D_{mx} = dimensiunea maximă pe coloană
 - ▶ $0 \leq v < D_{my}$, D_{my} = dimensiunea maximă pe linie

coordonatele aflate într-un domeniu real \rightarrow coordonate întregi din domeniul $[0, D_{mx}] \times [0, D_{my}]$

putem desena pe un subdomeniu $[u_1, u_2] \times [v_1, v_2]$ al domeniului maxim $[0, D_{mx}] \times [0, D_{my}]$ numit **fereastra fizică (ViewPort)** – $P_1(u_1, v_1)$ și $P_2(u_2, v_2)$.



un punct din fereastra reală:

$$P(x, y) \in [a, b] \times [c, d] \subset R^2$$

$$P(x, y) \rightarrow M(u, v) \in ViewPort$$

$$u := Round((x - a) * (u_2 - u_1)/(b - a)) + u1$$

$$v := Round((y - d) * (v_2 - v_1)/(c - d)) + v1$$

Este procesul prin care obiectele grafice sunt reprezentate ca o colecție de pixeli.

- ▶ Obiectele grafice sunt continue, pixelii sunt discreți;
- ▶ pentru simplificare un pixel poate fi activat sau nu (alb/negru);
- ▶ exemple de obiecte grafice: puncte, linii, cercuri, elipse;
- ▶ pentru a genera obiecte grafice sunt o multitudine de algoritmi.

Scan Conversion

Punctul

Un pixel pe ecran este o regiune ce conține puncte matematice.

Prin Scan-Converting un punct se înțelege iluminarea acelui pixel ce conține acel punct.

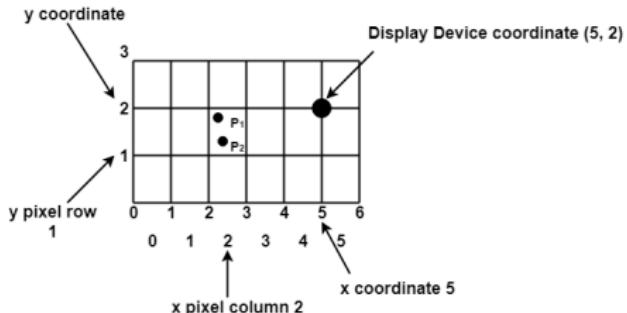
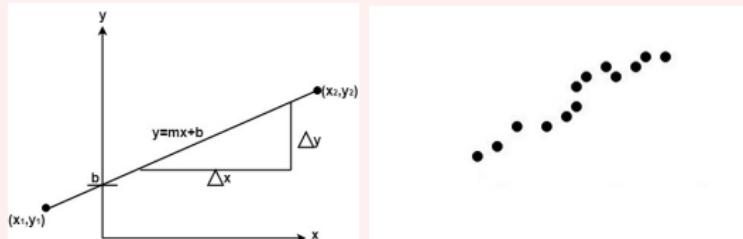


Figura: punctele P_1 și P_2 vor fi reprezentate de pixelul (2, 1)

punctul $P(x, y) \rightarrow$ pixelul $([x], [y])$

exemplu de transformare a unei linii



Algoritmi de scan-conversion pentru segmente:

- ▶ utilizarea directă a ecuației dreptei
- ▶ DDA (Digital Differential Analyzer)
- ▶ algoritmul lui Bresenham

Algoritmul lui Bresenham

Segment

caracteristică

următorul pixel selectat este cel ce e mai apropiat de segmentul dorit, mergând pe axa Ox

Desenăm un segment P_0P_F ce apartine dreptei $y = m * x + n$ cu panta $m \leq 1$:

- 0 pixelul de pornire $P(x, y) = P_0(x_0, y_0)$
- 1 Avem noii candidați $P_1(x + 1, y)$ și $P_2(x + 1, y + 1)$
- 2 determinăm distanța pe axa Oy de la pixel la punctul pe unde trece dreapta
- 3 alegem din candidati pe cel ce are distanța cea mai mică
- 4 dacă nu am ajuns la P_F repetăm de la pasul 1

Algoritmul lui Bresenham

Segment

Avantaje:

- ▶ este ușor de implementat
- ▶ este rapid și folosește doar incrementări
- ▶ este mai lent dar mai precis ca algoritmul DDA
- ▶ folosește doar puncte fixe (lucrează cu întregi)

Algoritmul lui Bresenham

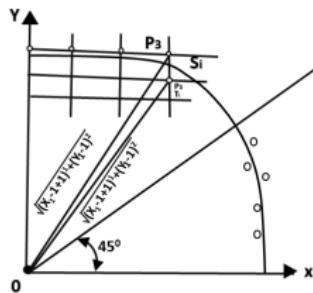
Cerc



20

caracteristică (similar cu algoritmul pentru linie)

următorul pixel selectat este cel ce e mai apropiat de cerc, mergând în sens crescător pe axa Ox , descrescător pe axa Oy , de la 90° la 45°



restul pixelilor se obtin prin simetrie (1 pixel \rightarrow 8 pixeli)

Bibliografie



[1] Robin Oppenheimer.

William Fetter, E.A.T., and 1960s Computer Graphics Collaborations in Seattle.

at HistoryLink.org, 2018.

<https://www.historylink.org/File/20542>.

The background features a large, stylized graphic element composed of several thick, curved lines in shades of blue, teal, and light orange. These lines curve from the left side towards the right, creating a sense of motion. Interspersed among the lines are small, glowing white dots that resemble stars or particles, particularly concentrated along the right edge.

Mulțumesc pentru atenție!

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

**Cursul 2 - Transformări geometrice uzuale
în grafica 2D și 3D. Sisteme de coordonate.
Vizualizarea 3D.**

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

4 martie 2022

Content



Sistemul de coordinate

Vectori

Matrici

Transformări geometrice

Biblioteca de funcții GLM

Introducere



ACEL CAPITOL DETESTAT!!

Sistemul de coordonate



Grafica → reprezentarea unor forme în spațiul 3D

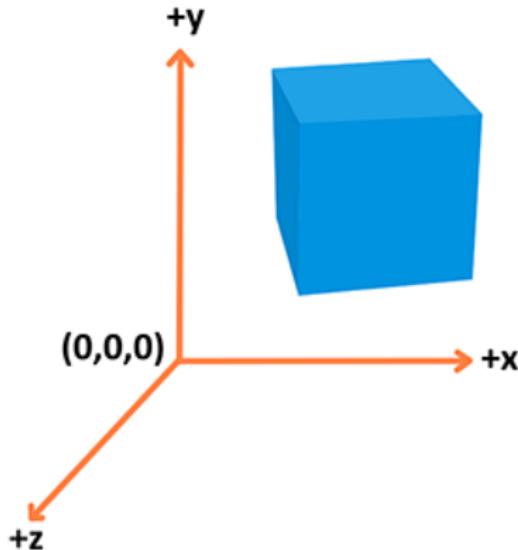


Figura: Sistemul de coordonate cartezian

Coordonatele punctelor



regula mâinii \implies direcțiile sistemului de coordonate

- ▶ Ox este îndreptată spre dreapta,
- ▶ Oy în sus,
- ▶ Oz arată spre noi (deciiese din ecran).

originea sistemului de coordonate – mijlocul ferestrei

un punct $P(c_1, c_2, c_3)$ va avea $c_i \in [-1.0, 1.0]$

Vertexuri

Obiectele vor fi construite în spațiul 3D folosind vertex-uri.

Un cub:

- ▶ 8 vârfuri și 6 fețe;
- ▶ fiecare vârf – un vertex;
- ▶ o față construită cu 4 vertex-uri, un vector normal;
- ▶ definit de textură

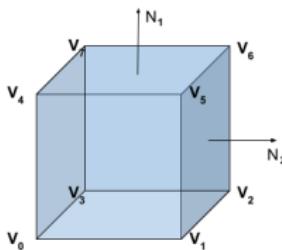


Figura: Un cub în spațiu, vâfurile sunt determinate de cele 8 vertex-uri, avem reprezentate și două normale din cele 6 (câte una pentru fiecare față a cubului).



Un vector este în esență o direcție în spațiu.

Definition

Se numește segment de dreaptă orientat orice pereche ordonată de puncte din spațiu.

Definition

Se numește vector (liber) mulțimea tuturor segmentelor de dreaptă orientate care au aceeași direcție, aceeași lungime și același sens cu un segment de dreaptă orientat. Notăm un vector folosind un simbol săgeată deasupra numelui (exemplu: \vec{a}).



În grafica pe calculator lucrăm cu dimensiuni de la 2 la 4

- ▶ 2 dimensiuni → direcție în plan
- ▶ 3 dimensiuni → orice direcție în 3D

Definition

Orice vector liber care are lungimea egală cu 1 se numește vesor.

o poziție din spațiu – vectori aplicați în originea sistemului de coordonate cu vârful în poziția dorită



Fie vectorii $\vec{A} = \begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} = A_x \vec{i} + A_y \vec{j} + A_z \vec{k}$ și

$$\vec{B} = \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = B_x \vec{i} + B_y \vec{j} + B_z \vec{k}.$$

► **Vesorii axelor de coordonate:**

$$\vec{i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \vec{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \vec{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

► **Modulul lui \vec{A} :**

$$|\vec{A}| = (A_x^2 + A_y^2 + A_z^2)^{\frac{1}{2}}$$

Vectori

Operații între un vector și un scalar



sumă/diferență

$$\vec{A} \pm s = \begin{pmatrix} A_x \pm s \\ A_y \pm s \\ A_z \pm s \end{pmatrix}$$

produs

$$\vec{A} * s = \begin{pmatrix} A_x * s \\ A_y * s \\ A_z * s \end{pmatrix}$$

împărțire

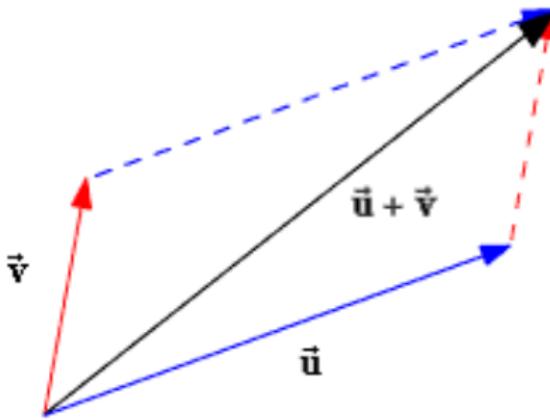
$$\vec{A}/s = \begin{pmatrix} A_x/s \\ A_y/s \\ A_z/s \end{pmatrix}$$

Vectori

Adunarea și scăderea a doi vectori



$$\vec{A} + \vec{B} = \begin{pmatrix} A_x + B_x \\ A_y + B_y \\ A_z + B_z \end{pmatrix}$$



În mod similar se face și scăderea.

Produs scalar și produs vectorial

- ▶ *Produsul scalar:*

$$\vec{A} \cdot \vec{B} = A_x B_x + A_y B_y + A_z B_z = |\vec{A}| |\vec{B}| \cos\theta$$

unde θ este unghiul făcut de cei doi vectori.

- ▶ *Produs vectorial:*

$$\vec{A} \times \vec{B} = (A_y B_z - A_z B_y) \vec{i} + (A_z B_x - A_x B_z) \vec{j} + (A_x B_y - A_y B_x) \vec{k}$$

$$\vec{A} \times \vec{B} = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{bmatrix}$$

$\vec{A} \times \vec{B}$ este întotdeauna perpendicular pe ambii vectori \vec{A} și \vec{B} .

► *Combinăție liniară de vectori*

$$\vec{w} = a_1 \vec{v}_1 + a_2 \vec{v}_2 + \dots + a_m \vec{v}_m$$

► *Combinăție afină de vectori:*

când în combinația liniară avem $a_1 + a_2 + \dots + a_m = 1$.

De exemplu: $(1 - t)\vec{A} + (t)\vec{B}$

► *Combinăție convexă de vectori:*

când avem $a_1 + a_2 + \dots + a_m = 1$ și $a_i \geq 0$ pentru $i = 1, \dots, m$.



Fie a, b – scalari și $\vec{P}, \vec{Q}, \vec{R}$ – vectori 3D

$$\vec{P} \times \vec{Q} = -(\vec{Q} \times \vec{P})$$

$$(a\vec{P}) \times \vec{Q} = a(\vec{P} \times \vec{Q})$$

$$\vec{P} \times (\vec{Q} + \vec{R}) = \vec{P} \times \vec{Q} + \vec{P} \times \vec{R}$$

$$\vec{P} \times \vec{P} = \vec{0} = (0, 0, 0)^T$$

$$(\vec{P} \times \vec{Q}) \cdot \vec{R} = (\vec{R} \times \vec{P}) \cdot \vec{Q} = (\vec{Q} \times \vec{R}) \cdot \vec{P}$$

$$a(\vec{P} + \vec{Q}) = a\vec{P} + a\vec{Q}$$

$$(\vec{P} + \vec{Q}) + \vec{R} = \vec{P} + (\vec{Q} + \vec{R})$$

O matrice este un sistem de numere grupate într-un tablou dreptunghiular care are un anumit număr de coloane, linii sau rânduri.

Definition

Fie $m, n \in \mathbb{N}^*$. Se numește matrice cu m linii și n coloane un tablou cu m linii și n coloane.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & & & \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}$$

unde elementele $a_{ij} \in \mathbb{R}$, $i = \overline{1, m}, j = \overline{1, n}$.

Matrici

Adunarea și scăderea matricilor

Operațiile de adunare și de scădere a matricilor se execută element cu element, aşadar se folosesc regulile operațiilor între scalari, aplicate între elementele cu același index ale matricilor.

În consecință pentru a executa aceste operații matricile vor trebui să aibă aceleași dimensiuni.

Exemplu:

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ -1 & 5 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+1 \\ 2+(-1) & 4+5 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 1 & 9 \end{bmatrix}$$

Similar se face și scăderea lor:

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ -1 & 5 \end{bmatrix} = \begin{bmatrix} 1-0 & 3-1 \\ 2-(-1) & 4-5 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & -1 \end{bmatrix}$$

Matrici

Produsul dintre o matrice și un scalar

Produsul dintre o matrice și un scalar este matricea obținută din elementele matricii initiale înmulțite fiecare cu scalarul dat.

De exemplu:

$$3 \cdot \begin{bmatrix} 0 & 1 \\ -1 & 5 \end{bmatrix} = \begin{bmatrix} 3 \cdot 0 & 3 \cdot 1 \\ 3 \cdot (-1) & 3 \cdot 5 \end{bmatrix} = \begin{bmatrix} 0 & 3 \\ -3 & 15 \end{bmatrix}$$



pre-condiție – regulă: numărul de coloane a matricii din stânga să fie egal cu numărul de linii a matricii din dreapta.

O consecință directă a acestei reguli este faptul că înmulțirea matricilor **nu este comutativă** ($A \cdot B \neq B \cdot A$).

Fie $A \in \mathcal{M}_{I,n}$ și $B \in \mathcal{M}_{n,m}$. Produsul lor $C = A \cdot B$ va avea elementele:

$$c_{i,j} = \sum_{k=1,n} a_{i,k} \cdot b_{k,j}$$

unde $i = \overline{1,I}$ și $j = \overline{1,m}$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 3 \cdot 5 + 4 \cdot 7 \\ 1 \cdot 6 + 2 \cdot 8 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix}$$



Transformările geometrice – scalare, translație, rotație, proiecție – pot fi descrise simplu, matematic, folosind matrici.

Transformări geometrice

Scalarea



Scalarea unui vector constă în multiplicarea mărimii unui vector, păstrând direcția.

Scalarea poate fi făcută pe oricare din cele trei direcții ale axelor de coordonate.

Să considerăm un vector $\vec{a} = (1, 3, 4)$.

Vom scala acest vector de-a lungul axei O_x cu un factor de 1.5, pe O_y cu un factor de 2.0 și pe O_z cu un factor de 0.7.

Vectorul nou va fi practic

$$\vec{a}' = (1.5 \cdot 1, 2.0 \cdot 3, 0.7 \cdot 4) = (1.5, 6.0, 2.8)$$

Transformări geometrice

Scalarea



Intuitiv construim următoarea matrice ce execută această transformare:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x \cdot a_x \\ S_y \cdot a_y \\ S_z \cdot a_z \\ 1 \end{pmatrix}$$

unde S_x , S_y și S_z sunt factorii cu care dorim să scalăm vectorul \vec{a} de-a lungul axelor O_x , O_y și respectiv O_z .

Notări în 4 dimensiuni



Pentru a simplifica notațiile și modul de calcul vectorilor 3D le mai atașăm o dimensiune.

Componenta w , a patra, a unui vector 3D se numește componentă omogenă.

Dimensiunea matricii unde codificăm transformarea va crește și ea, devenind o **matrice pătratică** de 4×4 .

Coordinatele omogene rezolvă problema reprezentării translației ca operație matricială.

Transformări geometrice

Translația



deplasează fiecare punct al unei figuri cu aceeași distanță într-o direcție dată

poate fi văzută ca adăugarea unui vector la fiecare punct translatat sau ca deplasarea originii sistemului de coordonate

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} T_x + a_x \\ T_y + a_y \\ T_z + a_z \\ 1 \end{pmatrix}$$

T_x , T_y și T_z sunt componentele vectorului cu care facem translația

fără această coloană nu am putea descrie translația ca un produs între o matrice și un vector.

Transformări geometrice

Rotatia în jurul axelor de coordonate

Rotatia în jurul axei O_x :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} a_x \\ \cos\theta \cdot a_y - \sin\theta \cdot a_z \\ \sin\theta \cdot a_y + \cos\theta \cdot a_z \\ 1 \end{pmatrix}$$

Rotatia în jurul axei O_y :

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot a_x + \sin\theta \cdot a_z \\ a_y \\ -\sin\theta \cdot a_x + \cos\theta \cdot a_z \\ 1 \end{pmatrix}$$

Rotatia în jurul axei O_z :

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot a_x - \sin\theta \cdot a_y \\ \sin\theta \cdot a_x + \cos\theta \cdot a_y \\ a_z \\ 1 \end{pmatrix}$$

Transformări geometrice

Gimbal lock

24

Teoretic combinând cele trei transformări putem să executăm orice rotație.

Practic, prin anumite rotații, se poate pierde un grad de libertate

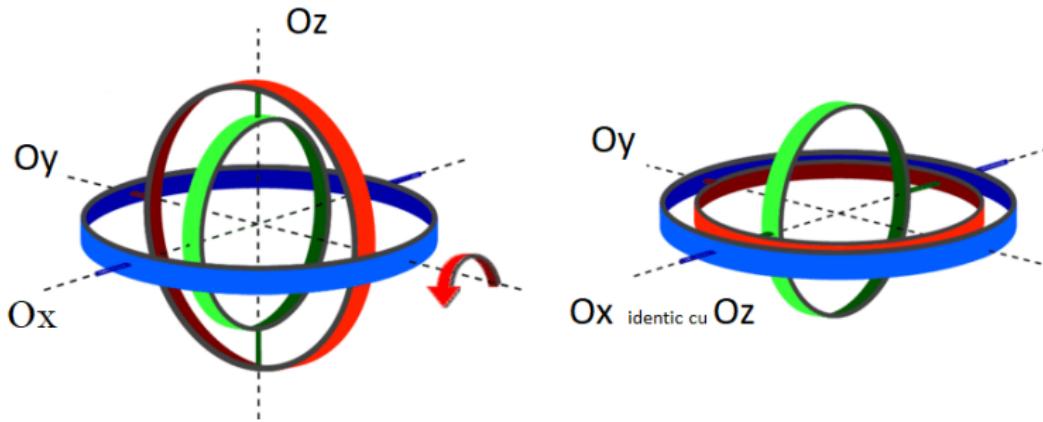


Figura: Pierderea unui grad de libertate printr-o rotație cu 90° în jurul axei O_y .

Transformări geometrice

Rotație în jurul unui vesor



Rotația se face în jurul unei vesor arbitrar, direct fără a mai combina matricile de rotație după axe.

dacă vesorul are componentele (R_x, R_y, R_z) :

$$\begin{bmatrix} \cos \theta + R_x^2(1 - \cos \theta) & R_x R_y(1 - \cos \theta) - R_z \sin \theta & R_x R_z(1 - \cos \theta) + R_y \sin \theta & 0 \\ R_y R_x(1 - \cos \theta) + R_z \sin \theta & \cos \theta + R_y^2(1 - \cos \theta) & R_y R_z(1 - \cos \theta) - R_x \sin \theta & 0 \\ R_z R_x(1 - \cos \theta) - R_y \sin \theta & R_z R_y(1 - \cos \theta) + R_x \sin \theta & \cos \theta + R_z^2(1 - \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Chiar și cu această transformare această problemă poate să mai apară, deși mai rar.

Pentru a evita complet acest fenomen se folosesc *cuaternioni*.

Transformări geometrice

Compunerea transformărilor

Scrierea transformărilor geometrice ca produs de matrici cu vectori permite compunerea lor, putând combina într-o singură matrice mai multe transformări.

Să considerăm un vector $\vec{a} = (a_x, a_y, a_z)$ pe care dorim să îl scalăm cu **3** și apoi să îl translatăm cu **(0, 2, 1)**. Vom folosi pentru asta două matrici, una pentru scalare și una pentru translatie.

$$\text{Translatie} \cdot \text{Scalare} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 2 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformări geometrice

Componerea transformărilor

Ordinea în care scriem matricile este importantă deoarece înmulțirea matricilor nu este comutativă.

Când aplicăm asupra unui vector transformările prima înmulțire efectuată este între vectorul inițial și ultima matrice din dreapta.

Ordinea aplicării transformărilor este de la dreapta la stânga.

Este indicat în practică să efectuăm întâi scalările, apoi rotațiile și la final translațiile. E posibil altfel ele să se influențeze în mod negativ una pe cealaltă.

Transformări geometrice

Compunerea transformărilor

Revenind la exemplul anterior, avem:

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 2 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \cdot a_x + 0 \\ 3 \cdot a_y + 2 \\ 3 \cdot a_z + 1 \\ 1 \end{pmatrix}$$

Asupra vectorului \vec{a} am aplicat o scalare și apoi o translație.

Transformări geometrice

Componerea transformărilor

Dacă am fi aplicat invers: întâi translația și apoi scalarea, translația la rândul ei ar fi fost scalată:

$$\begin{aligned} \text{Scalare} \cdot \text{Translație} \cdot \vec{a} &= \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \\ &= \begin{pmatrix} 3 \cdot (a_x + 0) \\ 3 \cdot (a_y + 2) \\ 3 \cdot (a_z + 1) \\ 1 \end{pmatrix} \end{aligned}$$

În acest exemplu se vede de ce este indicat să avem grijă la ordinea aplicărilor transformărilor.



bibliotecă header-only de clase și funcții, scrisă în C++

pentru a o folosi ea trebuie doar copiată și inclusă în proiectele noastre

Exemplu cum se include GLM:

```
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp>
    // glm::translate, glm::rotate, glm::scale,
    // glm::perspective
```

The background features a large, stylized graphic element composed of several thick, curved lines in shades of blue, teal, and light orange. These lines curve from the left side towards the right, creating a sense of motion. Interspersed among the lines are small, glowing white dots that resemble stars or particles, adding a magical or celebratory feel to the design.

Mulțumesc pentru atenție!

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

Cursul 3 - Limbajul GLSL

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

17 martie 2022



Generalități ale limbajului GLSL

Tipuri de date și variabile în GLSL

Attributes, Uniforms și Varyings

Variabile și tipuri predefinite

Obiective



După ce ați parcurs aceast capitol ar trebui să știți:

- ▶ Care sunt tipurile variabilelor în GLSL, cum se declară variabilele și cum sunt accesate.
- ▶ Care este sintaxa funcțiilor și a principalelor structuri repetitive și alternative în GLSL.
- ▶ Care sunt principalele funcții predefinite din GLSL.
- ▶ Ce este, cum se declară și cum se compilează un obiect de tip shader într-un program OpenGL.
- ▶ Ce este, cum se declară, cum se crează și cum se folosește un obiect de tip program într-o aplicație OpenGL.
- ▶ Cum se transmit valori variabilelor de intrare din programul C++ către shaderele scrise în GLSL.

Limbajul GLSL

Variante



Limbajul GLSL are mai multe variante, toate evoluând în paralel cu API-ul OpenGL.

În practică, ținând cont desigur și de necesitățile aplicației dezvoltate, nu e indicat a se folosi ultimele variante.

OpenGL version	GLSL version
2.0	110
2.1	120
3.0	130
...	...
3.3	330
...	...
4.5	450

Tabela: versiuni de OpenGL cu GLSL asociat lor.

Limbajul GLSL

Tipuri de date



avem patru tipuri de date principale: **float**, **int**, **bool** și **sampler**

float	int
vec2	ivec2
vec3	ivec3
vec4	ivec4
mat2, mat3, mat4	
sampler2D, bool	



tipul vector (observați modul în care se compune numele tipului):

- * **vec2, vec3, vec4** – 2D, 3D și 4D vectori de float
- * **ivec2, ivec3, ivec4** – 2D, 3D și 4D vector de int
- * **bvec2, bvec3, bvec4** – 2D, 3D și 4D vectori booleani

Pentru floats avem și matrici:

- * **mat2, mat3, mat4** – 2x2, 3x3, 4x4 matrici de float.

Samplers sunt tipuri de date folosite pentru texturi:

- * **sampler1D, sampler2D, sampler3D** – texturi 1D, 2D și 3D;
- * **samplerCube** – textură *Cube Map*;
- * **sampler1Dshadow, sampler2Dshadow** – componentă textură, de adâncime 1D și 2D.



Funcțiile sunt construite similar cu cele din C:

```
float myFunction ( float x, float y )
{
    float a = x * x;
    float b = y * y;

    return a + b;
}
```

Recursivitatea nu este permisă.

Nu avem pointers:

```
void myFunction ( float x, float y, float result )
{
    float a = x * x;
    float b = y * y;

    result = a + b;
}
```



Clasele au constructori similari cu cei din C++:

```
void myFunction ( float x )
{
    vec3 color3 = vec3 ( 0.5, 0.2, 0.8 );
    vec3 color3a = vec3 ( 0.5 ); // 3 valori sunt 0.5
    vec4 color4 = vec4 ( color3, 1.0 ); // alfa este 1.0

    //Vectori:

    vec2[3] arr = vec2[3] ( vec2(1.0, 2.0), vec2(3.0,4.0),
                           vec2(5.0, 6.0) );
}
```



Accesul la elementele unui vector poate fi făcut folosind x, y, z, sau w, făcând referință la primul, la al doilea, la al treilea, respectiv, la al patrulea element al lui.

```
void myFunction ( vec3 position )
{
    float x = position.x;
    float y = position[1];
}
```

Sunt trei tipuri de *swizzle masks*. Avem aşadar următoarele modalități de acces:

- ▶ accesul similar cu al elementelor unui vector: [0] [1] [2] [3];
- ▶ după axele de coordonate: .x .y .z .w;
- ▶ după canalul de culoare: .r .g .b .a;
- ▶ după dimensiunile texturii: .s .t .p .q.

Nu se pot combina aceste modalități de adresare.
Exemplu ".xrs" nu este o combinație *swizzle* validă.

Limbajul GLSL

Exemple valide:



```
vec2 p1 = position.xy;  
vec2 p2 = position.yz;  
  
vec4 colorAlpha;  
vec3 color = colorAlpha.rgb;  
vec3 colorbgr = colorAlpha.bgr;  
vec4 colorgray = colorAlpha.bbba;
```

Când folosim *swizzling* pentru a accesa componentele unui vector nu vom putea folosi același *swizzle* de două ori, deci **nu este admis** să folosim ceva de genul

someVec.xx = vec2(4.0, 4.0);

Limbajul GLSL

Swizzling pe scalari

În OpenGL 4.2 sau ARB_shading_language_420pack scalarii pot fi *swizzled* la rândul lor.

Evident aceştia au doar o componentă și atunci va fi acceptată o formulare de genul:

```
float aFloat;  
vec4 someVec = aFloat.xxxx;
```



Operatorii sunt similari cu cei din C: *vector * matrix*, *matrix * vector*, *matrix * matrix*.

Atenție!! Acesta nu este produsul scalar!

$$\text{vec2}(x_1, y_1) * \text{vec2}(x_2, y_2) = \text{vec2}(x_1 * x_2, y_1 * y_2)$$

Pentru produsul scalar se folosește funcția *dot()* și pentru produsul vectorial funcția *cross()*.



Avem o serie de funcții predefinite:

*abs, mod, exp, log, sin, cos, tan, acos, asin, atan, pow, min, max,
floor, ceil, fract, sqrt*

Funcțiile pot fi aplicate și vectorilor:

$$\sin(\text{vec2}(x, y)) = \text{vec2}(\sin(x), \sin(y))$$



- ▶ $\text{clamp}(x, \text{minVal}, \text{maxVal})$ – restricționează valoarea lui x în intervalul $[\text{minVal}, \text{maxVal}]$ (este echivalentă cu expresia: $\text{min}(\text{max}(x, \text{minVal}), \text{maxVal})$).
- ▶ $\text{step}(\text{threshold}, x)$ – crează o funcție prag:

$$\text{step}(\text{threshold}, x) = \begin{cases} 0.0 & , \text{ if } x < \text{threshold} \\ 1.0 & , \text{ otherwise.} \end{cases}$$

- ▶ $\text{smoothstep}(\text{start}, \text{end}, x)$ – face o interpolare Hermite între 0 și 1 când $\text{start} < x < \text{end}$. Extrem de utilă când se dorește o funcție prag cu o tranziție fină între 0 și 1. Valoarea rezultată ar fi dată de secvența:

```
genType t; /* Or genDType t; */
t = clamp((x - start) / (end - start), 0.0, 1.0);
return t * t * (3.0 - 2.0 * t);
```

Rezultatele sunt nedefinite pentru $\text{start} \geq \text{end}$.



- ▶ $\text{mix}(a, b, ratio)$ – execută o interpolare liniară între a și b cu ponderea $ratio$. Valoarea calculată va fi:

$$a \times (1 - ratio) + b \times ratio.$$

- ▶ $\text{length}(\text{vec2}(x, y))$ – returnează lungimea unui vector, adică $\sqrt{x^2 + y^2}$.
- ▶ $\text{distance}(\text{vec2}(A), \text{vec2}(B)) = \text{length}(B - A)$ – returnează distanța între două puncte A și B .
- ▶ $dFdx(p)$, $dFdy(p)$ – pot fi apelate doar în fragment shader, ele returnează derivatele parțiale ale expresiei p în raport cu variabila x (pentru $dFdx$) și y (pentru $dFdy$).
- ▶ $\text{texture}(\text{sampler2D } anImage, \text{vec2 } texCoord)$ – returnează *texels* (unitatea de bază a unei texturi) dintr-o textură *anImage* de la coordonatele *texCoord*.

Mecanismul din GLSL (numite *call by value-return to pass parameters*).

Parametrii funcțiilor sunt clasificați în trei categorii; de intrare (default), de ieșire sau intrare/ieșire.

Variabilele returnate sunt copiate înapoi în funcțiile ce le-au apelat și parametrii de intrare sunt copiați din procedurile ce le apelează.

Attributes, Uniforms și Varyings

Sunt trei tipuri de intrări și de ieșiri dintr-un shader:

- ▶ Uniforms
- ▶ Attributes
- ▶ Varyings

Attributes, Uniforms și Varyings

Uniforms



- ▶ *Uniforms* –valorile lor nu se schimbă la o procesare de imagine

Prin variabilele Uniform avem un mecanism de a partaja informații între programul cpp și shadere.

Attributes, Uniforms și Varyings

Attributes



- ▶ *Attributes* – se găsesc doar în vertex shader. Sunt folosite pentru variabile care își modifică valoarea cel mult odată pe vertex în shader.

Attributes sunt read-only. Sunt de două tipuri:

- ▶ definite de utilizator:

```
attribute float x;  
attribute vec3 velocity;
```

- ▶ variabile predefinite ce conțin informații precum culoare, poziție, normale, etc.

```
gl_Vertex  
gl_Color
```

Attributes, Uniforms și Varyings

Varyings



- ▶ *Varyings* – sunt folosite pentru a trimite date de la vertex shader la fragment shader.

Definite pentru fiecare vertex vor fi interpolate într-o primitivă în procesul de rasterizare.

Ele trebuie declarate ca variabile globale și pot fi predefinite sau definite de programator.

În GLSL se pot specifica variabile de tipul attributes, uniforms și varyings de către utilizator.

De exemplu dacă dorim să trimitem un vector tangent pentru fiecare vertex din aplicație către vertex shader putem defini o variabilă "tangent" de tip attribute.

```
attribute vec3 Tangent;
```

Variabile și tipuri predefinite

Attributes



o serie de variabile predefinite ce au specificații exacte

În vertex shader se găsesc câteva variabile de tip attributes predefinite:

- ▶ *gl_Vertex* – vector 4D reprezentând vertexul de poziție;
- ▶ *gl_Normal* – vector 3D reprezentând vertexul normalei;
- ▶ *gl_Color* – vector 4D reprezentând vertexul de culoare;
- ▶ *gl_MultiTexCoordX* – vector 4D reprezentând coordonatele unității X a texturii;
- ▶ sunt și alte atribute predefinite.

Variabile și tipuri predefinite

Uniforms

Variabile de tip uniforms:

- ▶ *gl_ModelViewMatrix* – matrice de 4×4 reprezentând matricea model-view;
- ▶ *gl_ModelViewProjectionMatrix* – matrice de 4×4 Matrix reprezentând matricea model-view-projection;
- ▶ *gl_NormalMatrix* – matrice de 3×3 reprezentând transpunerea inversă matricii model-view, e folosită pentru transformarea normală;
- ▶ sunt și alte variabile de acest tip predefinite, precum cele legate de lumini, materiale, etc.

Variabile și tipuri predefinite

Varyings



Variabile de tip Varyings:

- ▶ *gl_FrontColor* – un vector 4D reprezentând culoarea primitivelor din planul frontal;
- ▶ *gl_BackColor* – un vector 4D reprezentând culoarea primitivelor din planul din spate;
- ▶ *gl_TexCoord[X]* – un vector 4D reprezentând a X-a coordonată a texturii;
- ▶ sunt și alte variabile predefinite de acest tip.

Variabile și tipuri predefinite



Variabile predefinite folosite pentru ieșirile din shader:

- ▶ *gl_Position* – vector 4D reprezentând vertexul de poziție final, poate fi accesat doar în vertex shader;
- ▶ *gl_FragColor* – vector 4D reprezentând vertexul de culoare final, poate fi accesat doar în fragment shader;
- ▶ *gl_FragDepth* – float reprezentând adâncimea care e scris în bufferul de adâncime, poate fi accesat doar în fragment shader.

Aceste variabile predefinite sunt extrem de importante pentru ca fac legătura cu parametrii din OpenGL. De exemplu dacă apelăm

```
glLightfv(GL_LIGHT0, GL_POSITION, my_light_position)
```

această valoare va fi disponibilă într-un shader ca variabila *gl_LightSource[0].position*.

The background features a large, stylized graphic element composed of several thick, curved lines in shades of blue, teal, and light orange. These lines curve from the left side towards the right, creating a sense of motion. Interspersed among the lines are small, glowing white dots that resemble stars or particles, particularly concentrated along the right edge.

Mulțumesc pentru atenție!

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

Cursul 4 - Modalități de transmitere a informațiilor către shadere. Modelarea suprafețelor și curbelor

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

20 martie 2022



Obiecte în OpenGL

Obiecte de tip Buffer

Crearea obiectelor de tip buffer

Legarea obiectelor Buffer de context

Destinații pentru obiectele de tip buffer

Vertex Array Object

Pași pentru a crea o imagine

Vertex Specification

Exemplu utilizare VAO și VBO

Reprezentarea curbelor, suprafețelor și a corpurilor

Obiective



După ce ați parcurs aceast capitol ar trebui să știți:

- ▶ care sunt tipurile de obiecte în OpenGL.
- ▶ cum să generați, legați sau sterge obiectele de tip buffer.
- ▶ pașii necesari pentru a crea o imagine.
- ▶ cum să executați un Vertex Specification.
- ▶ Cum se transmit variabilele uniforms.
- ▶ Modalități de reprezentare a suprafețelor curbe și de rotație

Obiecte în OpenGL



Sunt structuri compuse din stări și date responsabile de transmiterea datelor de la CPU către GPU.

Datorită faptului că sunt colecții de stări, pentru a lucra cu ele, trebuie **legate** de un context OpenGL.

Obiectele în OpenGL pot fi:

- ▶ obiecte regulare (Regular Objects);
- ▶ obiecte container (Container Objects);
- ▶ obiecte non-standard – sync objects, shader și program Objects.

Regular Objects



Aceste tipuri de obiecte vor conține date.

Lista obiectelor regulaře este:

- ▶ Buffer Objects
- ▶ Renderbuffer Objects
- ▶ Texture Objects
- ▶ Query Objects
- ▶ Sampler Objects

Container Objects



Nu conțin date – containere pentru obiectele OpenGL obișnuite.

Lista obiectelor container:

- ▶ Framebuffer Objects
- ▶ Vertex Array Objects
- ▶ Transform Feedback Objects
- ▶ Program Pipeline Objects

Obiecte non-standard



- ▶ Obiecte de sincronizare (Sync Objects) – folosite pentru a sincroniza activitatea între GPU și aplicația noastră. Aceste obiecte permit un control fin.

- ▶ Obiectele de tip program reprezintă un cod complet procesat pentru a fi executabil, în OpenGL Shading Language, pentru unul sau mai multe etape din pipeline. Un obiect program este obiect OpenGL deși nu respectă modelele standard ale API ului.

"Legarea" obiectelor în OpenGL



- ▶ obiect OpenGL – trebuie *creat* și *legat* de un context OpenGL.
- ▶ când se face legătura – trebuie specificat ce tip de obiect va fi trimis către GPU.
- ▶ tipul datelor – definesc modul în care se comportă aceste obiecte.
- ▶ **Punct de Legătură** (Binding Point) specifică comportamentul obiectului.
- ▶ punctele de legătură (Targets) permit să fie folosite în diverse scopuri.

Exemple de puncte de legătură

Cele mai folosite puncte de legătură sunt:

- ▶ *GL_ARRAY_BUFFER*
- ▶ *GL_TEXTURE_BUFFER*
- ▶ *GL_ELEMENT_ARRAY_BUFFER*

Exemplu: un buffer cu punct de legătură *GL_ARRAY_BUFFER* se va comporta ca un VBO (Vertex Buffer Object)
– folosit deobicei pentru a memora datele legate de vertexuri.

Punche de legătură



Lista completă de Punche de legătură:

- ▶ *GL_ARRAY_BUFFER*
- ▶ *GL_COPY_READ_BUFFER*
- ▶ *GL_COPY_WRITE_BUFFER*
- ▶ *GL_ELEMENT_ARRAY_BUFFER*
- ▶ *GL_PIXEL_PACK_BUFFER*
- ▶ *GL_PIXEL_UNPACK_BUFFER*
- ▶ *GL_TEXTURE_BUFFER*
- ▶ *GL_TRANSFORM_FEEDBACK_BUFFER*
- ▶ *GL_UNIFORM_BUFFER*

Obiecte de tip Buffer

Sunt obiecte OpenGL care referă o secvență de memorie neformatată alocată de contextul OpenGL (aka GPU).

Sunt folosite să rețină o multitudine de date legate de:

- ▶ vertexuri
- ▶ informații legate de pixeli
- ▶ texturi, etc.

Au un alias sau un "nume" (de tipul *GLint*) prin care sunt referite.

Crearea obiectelor de tip buffer

```
void glGenBuffers( GLsizei n, GLuint * buffers)
```

returnează n nume de obiecte de tip buffer în $\textit{buffers}$

Parametrii:

- ▶ n – specifică numărul de nume de obiecte buffer ce vor fi generate
- ▶ $\textit{buffers}$ – un array unde vor fi generate numele obiectelor buffer generate.

Observație: nici un obiect de tip buffer nu este asociat cu numele generate până nu e apelată funcția

glBindBuffer

```
void glBindBuffer( GLenum target, GLuint buffer)
```

Se leagă obiectul de tip buffer de contextul OpenGL

Parametrii:

- ▶ *target* – specifică destinația bufferului
- ▶ *buffer* – specifică aliasul obiectului de tip buffer

Observație: când un obiect buffer este legat de destinația lui, legătura existentă pentru acea destinație este distrusă.

Destinații pentru obiectele de tip buffer

<code>GL_ARRAY_BUFFER</code>	Vertex attributes
<code>GL_ATOMIC_COUNTER_BUFFER</code>	Atomic counter storage
<code>GL_COPY_READ_BUFFER</code>	Buffer copy source
<code>GL_COPY_WRITE_BUFFER</code>	Buffer copy destination
<code>GL_DISPATCH_INDIRECT_BUFFER</code>	Indirect compute dispatch commands
<code>GL_DRAW_INDIRECT_BUFFER</code>	Indirect command arguments
<code>GL_ELEMENT_ARRAY_BUFFER</code>	Vertex array indices
<code>GL_PIXEL_PACK_BUFFER</code>	Pixel read target
<code>GL_PIXEL_UNPACK_BUFFER</code>	Texture data source
<code>GL_QUERY_BUFFER</code>	Query result buffer
<code>GL_SHADER_STORAGE_BUFFER</code>	Read-write storage for
<code>GL_TEXTURE_BUFFER</code>	Texture data buffer
<code>GL_TRANSFORM_FEEDBACK_BUFFER</code>	Transform feedback buffer
<code>GL_UNIFORM_BUFFER</code>	Uniform block storage

`void glDeleteBuffers(GLsizei n, const GLuint * buffers)`

șterge bufferele specificate

Parametrii:

- ▶ *n* – specifică numărul de buffere ce vor fi șterse
- ▶ *buffers* – șirul cu nume de obiecte buffer ce vor fi șterse.

Observații:

- ▶ după ce un buffer este șters nu va avea conținut și numele poate fi reutilizat.
- ▶ comanda ignoră zerourile și numele ce nu corespund unor buffere.

Vertex Array Object (VAO)

- ▶ este un obiect OpenGL de tip container pentru obiecte de tipul Vertex Buffer
- ▶ pentru a folosi un VAO:
 - ▶ el trebuie creat – glGenVertexArrays
 - ▶ el trebuie legat de contextul OpenGL – glBindVertexArray

Vertex Array Object (VAO)

- ▶ conține toate informațiile despre vertex-uri;
- ▶ conține formatul datelor din obiectele Buffer;
- ▶ el NU conține datele efective (acestea sunt în buffere);
- ▶ se crează și se distrugă ca orice alt obiect (`glGenVertexArrays`, `glDeleteVertexArrays`);
- ▶ `glBindVertexArray` funcționează puțin diferit, în sensul că nu există parametrul *target* – există o singură destinație pentru acest obiect.

Vertex Attribute Generic Array

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type,  
GLboolean normalized, GLsizei stride, const void * pointer);
```

- ▶ *index* specifică indexul vectorului de atribute;
- ▶ *size* specifică numărul de componente al unui atribut (1, 2, 3, 4);
- ▶ *type* specifică tipul componentelor unui atribut (*GL_BYTE*,
GL_UNSIGNED_BYTE, *GL_SHORT*, *GL_UNSIGNED_SHORT*, *GL_INT* și
GL_UNSIGNED_INT);
- ▶ *normalized* specifică dacă datele reprezentate în virgula fixă
trebuie normalize sau se convertesc direct în virgulă mobilă;
- ▶ *stride* specifică *the byte offset* între două vertex-uri consecutive;
- ▶ *pointer* specifică un offset pentru prima componentă a atributului.

Vertex 1			Vertex 2			Vertex 3		
<i>x</i> ₁	<i>y</i> ₁	<i>z</i> ₁	<i>x</i> ₂	<i>y</i> ₂	<i>z</i> ₂	<i>x</i> ₃	<i>y</i> ₃	<i>z</i> ₃
Byte:	0	4	8	12	16	20	24	28
Posiția:	<u>STRIDE: 12</u>		→					
- offset: 0								

Pasi pentru a crea o imagine – I

1. generăm un VAO (*glGenVertexArrays*).
2. legăm VAO de contextul OpenGL (*glBindVertexArray*) – facem legătura între VAO și context.
3. generăm un (*glGenBuffers()*).
4. legăm obiectul VBO (*glBindBuffer()*) – specificăm că acest buffer va fi folosit pentru operațiile ce urmează.
5. precizăm Buffer Data (*glBufferData()* sau *glBufferSubData()*) – alocăm și inițializăm suficientă memorie pentru bufferul curent legat de context.

7. obtinem locatia atributelor (*glGetAttribLocation()*).
8. obtinem locatia variabilelor uniform din programul shader activ (*glGetUniformLocation()*).
9. activam locatia atributelor gasite in shader (*glEnableVertexAttribArray()*).
10. informam OpenGL despre tipul de date si offsetul lor in bufferul legat curent (*glVertexAttribPointer()*).
11. cream imaginea scenei folosind datele din bufferele curente si activate (*glDrawArrays()* sau *glDrawElements()*).
12. stergem bufferele generate si eliberam resursele asociate (*glDeleteBuffers()*).

Vertex Specification - procesul prin care sunt pregătite obiectele necesare pentru a crea imaginea folosind un shader.

- ▶ datele sunt trimise într-un stream; totodată se informează OpenGL cum să interpreze acest stream.
- ▶ avem nevoie de un *program shader* care include un *Vertex Shader*.
- ▶ datele trimise către acest shader sunt o listă de attribute – **Vertex Attributes** – unde fiecare atribut este mapat ca o variabilă de intrare definită de utilizator.

- * pentru fiecare atribut trebuie trimis un vector de date, acești vectori sunt de lungime egală.
- * ordinea vertex-urilor în fluxul trimis e foarte importantă – ea determină cum sunt procesate și desenate primitivele.

sunt două modalități de a desena folosind vertexuri:

- ▶ fie sunt folosite în ordinea în care sunt transmise în stream (flux) (vezi exemplul din curs)
- ▶ fie se folosește o listă de indici care va defini ordinea lor (folosind EBO, vezi exemple la laborator).

Exemplu utilizare VAO și VBO



Să presupunem că dorim să desenăm două obiecte.

Informațiile necesare pentru a le desena:

* obiectul 1

- array ce conține coordonatele vertex-urilor și culoarea lor:
 $vector < float > v1 = \{x_0, y_0, z_0, r_0, g_0, b_0, x_1, y_1, \dots\}$

* obiectul 2

- un array cu coordonatele vertex-urilor:
 $vector < float > v2 = \{x_0, y_0, z_0, , x_1, y_1, \dots\}$
- unul cu culoarea vertex-urilor:
 $vector < float > c2 = \{r_0, g_0, b_0, r_1, g_1, \dots\}$

Vertex-urile sunt aranjate pentru a le folosi cu primitiva *GL_TRIANGLES*
– sunt ordonate astfel încât trei câte trei vertexuri formează un triunghi.

Exemplu utilizare VAO și VBO

- ▶ Declarăm doi vectori de două elemente de tip *GLuint* și un scalar de tip *GLuint*:
 - 1 *GLuint VAO[2], VBO1, VBO2[2];*
- ▶ Vom crea 2 Vertex Array Objects (VAO) câte unul pentru fiecare obiect și le punem într-un array:
 - 2 *glGenVertexArrays(2, VAO);*
- ▶ Precizăm că folosim primul obiect din acest sir:
 - 3 *glBindVertexArray(VAO[0]);*
- ▶ Creăm un array buffer și îl legăm de contextul Opengl (va conține atributele primului obiect) :
 - 4 *glGenBuffers(1, &VBO1);*
 - 5 *glBindBuffer(GL_ARRAY_BUFFER, VBO1);*
- ▶ Copiem în buffer vectorul v1
 - 6 *glBufferData(GL_ARRAY_BUFFER, v1.size() * sizeof(float), &v1.front(), GL_STATIC_DRAW);*

Exemplu utilizare VAO și VBO

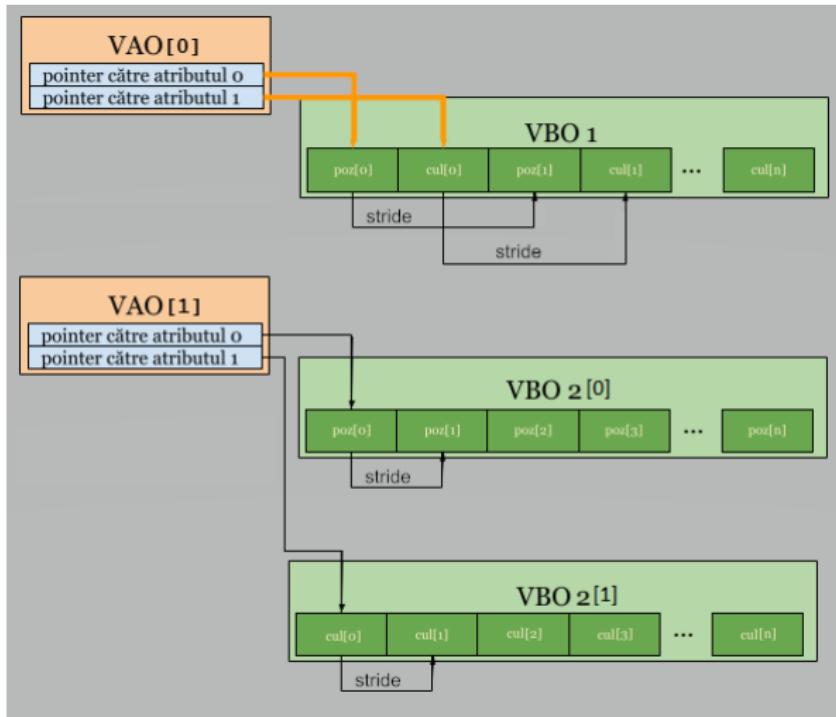


- ▶ Specificăm cum sunt aranjate atributele în acest buffer:
 - 7 `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);`
 - 8 `glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));`

semnificația argumentelor funcției de la pc. 7 și 8

- ▶ avem două attribute (coordonate – 0 și culoare – 1);
- ▶ fiecare este format din 3 elemente de tip *float*;
- ▶ nu sunt normalizează
- ▶ distanța între două attribute de același tip în acest buffer este de 6 mărită cu mărimea unui element – `sizeof(float)`;
- ▶ sirul atributelor începe de la adresa poziției 0 pentru atributul 1 – `(void*)0`, și de pe adresa poziției 3 pentru atributul 2 – `(void*)(3 * sizeof(float))`.

Exemplu utilizare VAO și VBO



Exemplu utilizare VAO și VBO

- ▶ Activăm cele două array-uri de atribute specificate pentru acest Vertex Array (aka VAO[0]):

```
9 glEnableVertexAttribArray(0);  
10 glEnableVertexAttribArray(1);
```

Vertex array-ul pentru primul obiect este pregătit pentru desenare:

- ▶ în main loop se leagă VAO[0] de contextul OpenGL
`glBindVertexArray(VAO[0])`
- ▶ se desenează obiectul 1 folosind comanda
`glDrawArrays(GL_TRIANGLES, 0, n).`

Exemplu utilizare VAO și VBO

- ▶ Pentru obiectul 2 folosim al doilea obiect din sirul VAO:
11 `glBindVertexArray(VAO[1]);`
- ▶ Creăm un sir de array buffere și legăm primul din ele de contextul Opengl (va conține primul atribut al celui de al doilea obiect):
12 `glGenBuffers(2, VBO2);`
13 `glBindBuffer(GL_ARRAY_BUFFER, VBO2[0]);`
- ▶ Copiem în buffer vectorul v2
14 `glBufferData(GL_ARRAY_BUFFER, v2.size() * sizeof(float), &v2.front(), GL_STATIC_DRAW);`
- ▶ Specificăm cum sunt aranjate atributele în acest buffer:
15 `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);`
- ▶ Activăm primul array de atribut specificat pentru acest Vertex Array (aka VAO[1]):
16 `glEnableVertexAttribArray(0);`

Exemplu utilizare VAO și VBO

- ▶ Repetăm operațiile de la 13 la 16 pentru cel de al doilea buffer din sir (va conține al doilea atribut al celui de al doilea obiect):
13' `glBindBuffer(GL_ARRAY_BUFFER, VBO2[1]);`
- ▶ Copiem în buffer vectorul *c2*
14' `glBufferData(GL_ARRAY_BUFFER, c2.size() * sizeof(float), &c2.front(), GL_STATIC_DRAW);`
- ▶ Specificăm cum sunt aranjate atributele în acest buffer:
15' `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);`
- ▶ Activăm al doilea array de atribut specificat pentru acest Vertex Array (aka VAO[1]):
16' `glEnableVertexAttribArray(q);`

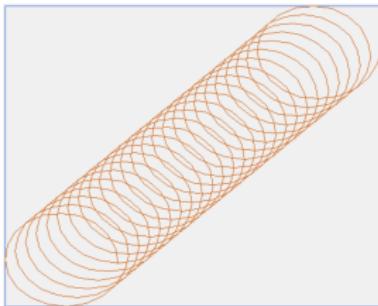
Al doilea obiect este astfel pregătit pentru a fi desenat în mod similar cu primul obiect.

Reprezentarea Curbelor

Reprezentarea unei curbe se poate realiza prin unirea proiecțiilor unui sistem de puncte de pe aceasta.

$$x = f(t); y = g(t); z = h(t)$$

De exemplu:



$$f(t) = \cos(t);$$

$$g(t) = \sin(t);$$

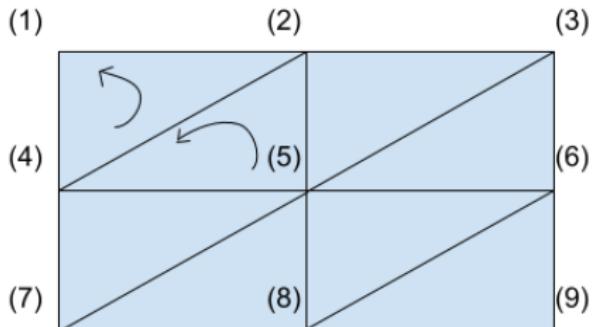
$$h(t) = t/25;$$

$$t \in [a, b]$$

Reprezentarea suprafetelor

- ▶ pentru o suprafață definită de o funcție $z : [a, b] \times [c, d] \rightarrow \mathbb{R}$
- ▶ vom crea un *mesh* (o rețea) divizând echidistant în n respectiv în m părți intervalele $[a, b]$ și respectiv $[c, d]$.
- ▶ în fiecare punct al rețelei calculăm valoarea lui z .
- ▶ multimea punctelor (x_i, y_j, z_i) vor furniza vertexurile necesare pentru reprezentarea suprafetei.

De exemplu pentru un *mesh* de 3×3 vom forma triunghiuri cu vertexurile grupându-le ca în figura:



The background features a large, stylized graphic element composed of several thick, curved lines in shades of blue, teal, and light orange. These lines curve from the left side towards the right, creating a sense of motion. Interspersed among the lines are small, glowing white dots that resemble stars or particles, adding a magical or celebratory feel to the design.

Mulțumesc pentru atenție!

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

**Cursul 5 - Modelarea obiectelor (solide) 3D.
Modelare geometrică și ierarhii.**

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

28 aprilie 2022



Modelarea de obiecte 3D

Exemple de formate de fișiere pentru obiecte 3D

Încărcarea modelelor 3D

Folosirea variabilelor GLSL Uniforms

Obiective



După ce ați parcurs aceast capitol ar trebui să știți:

- ▶ Unde puteți modela sau de unde puteți procura obiecte 3D.
- ▶ Încărcarea de obiecte 3D.
- ▶ Cum se transmit variabilele uniforms.
- ▶ Cum se folosesc matricile Model, View, Projection.
- ▶ Diverse facilități oferite de biblioteca GLM
- ▶ Cum putem folosi variabilele uniforms pentru a crea simple animații



Există o multitudine de softuri care permit modelarea de obiecte 3D.

Exemple:

- ▶ Blender – program software liber de grafică 3D
crearea modelelor 3D, mapare UV, texturare, rigging, simularea apei, animatie, randare, particule și alte simulări computerizate, editare non-lineară, compositing și crearea aplicațiilor interactive.
- ▶ Unity – un game engine
crearea de jocuri și experiențe 2D și 3D (filme – cinematice, interactive, virtuale...)

alte exemple: Cinema 4D, 3ds Max Design, Modo, Maya, Animaker, Adobe Premiere Pro, Adobe After Effects.

Formate de fișiere pentru obiecte în 3D



modelul 3D creat este salvat în fișiere de diverse formate:

- ▶ Collada – .dae
- ▶ Alembic – .abc
- ▶ Universal Scene Description – .usd, .usdc, .usda
- ▶ Grease Pencil as SVC
- ▶ Stanford – .ply)
- ▶ STL – .stl
- ▶ FBX – .fbx
- ▶ glTF 2.0 – .glb, .gltf
- ▶ WaveFront – .obj
- ▶ X3d Extensible 3D – .x3d

Exemplu: formatul glTF



formatul glTF™ (GL Transmission Format) este folosit pentru transmisia și încărcarea modelelor 3D în aplicații web

- reduce mărimea modelelor și timpii de procesare a lor (încărcare și desenare);
- este acceptat de diverse motoare 3D precum: Unity3D, Unreal Engine 4, and Godot;
- permite transmisia următoarelor date:
 - ▶ Meshes,
 - ▶ Materiale (Principled BSDF) și Shadeless (Unlit),
 - ▶ Texturi,
 - ▶ Camere,
 - ▶ Lumini punctuale (puncte, spot și direcționale),
 - ▶ Animații.

Exemplu: formatul .obj

Generalități



A fost dezvoltat inițial de Wavefront Technologies și a fost preluat și de alte companii.

E simplu și accesibil:

- ▶ Comentariile sunt marcate cu #
- ▶ Poate conține date legate de vertexuri, curbe, suprafete, corpuri, moduri de grupare etc.

Elementele cele mai comune sunt cele geometrice:

vertex-uri, coordonatele texturilor, normalele și fețele poligonale.

Exemplu: formatul .obj

Exemplu de elemente din fișier



```
# List of geometric vertices, with (x, y, z [,w]) coordinates,  
#           w is optional and defaults to 1.0.  
v 0.123 0.234 0.345 1.0  
v ...  
...  
# List of texture coordinates, in (u, [v ,w]) coordinates,  
#           these will vary between 0 and 1. v, w are optional  
#           and default to 0.  
vt 0.500 1 [0]  
vt ...  
...  
# List of vertex normals in (x,y,z) form; normals might not be  
#           unit vectors.  
vn 0.707 0.000 0.707  
vn ...  
...
```

Exemplu: formatul .obj

Exemplu de elemente din fișier



```
# Parameter space vertices in ( u [,v] [,w] ) form; free form
#           geometry statement
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
# Line element
l 5 8 1 2 4 9
```

Exemplu: formatul .obj

Descrierea formatului



Câteva elemente din formatul fișierelor *.obj*:

fiecare linie are la început o etichetă ce precizează tipul elementului de pe acea linie:

- ▶ **v** – vertex-uri, **vt** – coordonate texturi, **vn** – vectori normali
- ▶ **f** – fețe (pot avea mai multe formate):
 - listă de vertex-uri:
f v1 v2 v3;
 - listă de vertex/texturi:
f v1/vt1 v2/vt2 v3/vt3;
 - listă de vertex/texturi/normale:
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
 - lista de vertex//normale:
f v1//vn1 v2//vn2 v3//vn3

procesul prin care se preiau informațiile din fișiere și se pun în buffere

deobicei se folosesc diverse biblioteci sau se scriu efectiv funcțiile necesare

Exemplu:

Open Asset Import Library (ASSIMP) – bibliotecă de import de modele 3D multiplatformă extrem de populară



- ▶ sunt variabile globale în pipeline.
- ▶ se comportă ca niște parametrii ce sunt trimiși din programul nostru spre shadere.
- ▶ valorile lor sunt reținute în obiectul OpenGL de tip program.
- ▶ valorile lor nu se schimbă între diversele apeluri ale programelor shader (cum o fac de exemplu pozițiile punctelor din vertexuri).

Declararea variabilelor uniforms în shadere

Variabilele uniforms pot fi de orice tip. Ele sunt implicit constante.

Exemplu în GLSL:

```
struct TheStruct
{
    vec3 first;
    vec4 second;
    mat4x3 third;
};

uniform vec3 oneUniform;
uniform TheStruct aUniformOfType;
uniform mat4 matrixArrayUniform[25];
uniform TheStruct uniformArrayOfStructs[10];
```

Interface Bloc

este un grup GLSL de variabile de intrare, ieșire, uniforme, sau buffere de memorare. Acestea au o sintaxă specială care permite lucrul cu ele.

Variabilelor uniforms care nu aparțin unui interface block le pot fi precizate direct locația

```
layout(location = 3) uniform mat4 viewMatrix;
```

Observație: această sintaxă ne asigură că locația variabilei e 3, dar poate lipsi specificarea respectivă.

Accesul la variabilele uniforms

Obținerea locației



Pentru a avea locația unei variabile uniforme dintr-un program de tip shader folosim comanda:

```
GLint glGetUniformLocation( GLuint program,  
                           const GLchar * name);
```

Parametrii:

- ▶ *program* – specifică programul unde căutăm variabila uniformă
- ▶ *name* – referință la un string ce conține numele variabilei

funcția returnează –1 dacă variabila precizată nu există în acel program.

Erori posibile:

- ▶ *GL_INVALID_VALUE* dacă nu e nici un program cu acel alias.
- ▶ *GL_INVALID_OPERATION* dacă aliasul nu e a unui obiect program, sau programul nu a fost compilat.

Accesul la variabilele uniforms

Modificarea valorii

Există aproximativ **34** de funcții OpenGL care trimit valori variabilelor uniforms din shadere!

Formatul lor este *glUniform* + specificații tip variabilă!

Exemple:

- ▶ `void glUniform1f(GLint location, GLfloat v0)` – trimitem o variabilă de tip float;
- ▶ `void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3)` – trimitem patru variabile de tip float;

...

The background features a large, stylized graphic element composed of several thick, curved lines in shades of blue, teal, and light orange. These lines curve from the top left towards the bottom right, creating a sense of motion. Interspersed among the lines are small, glowing white dots that resemble stars or particles, adding to the dynamic feel of the design.

Mulțumesc pentru atenție!

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

Cursul 6 - Texturi (constante, variabile, aleatoare)

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

28 aprilie 2022



Obiecte de tip textură

Crearea și Inițializarea Texturilor

Tipuri și puncte de destinație pentru texturi

Accesul texturilor din shadere

Loading Textures from Files

Controlling How Texture Data Is Read

Using Multiple Textures



După ce ați parcurs aceast capitol ar trebui să știți:

- ▶ cum se crează o textură
- ▶ cum se încarcă o textură
- ▶ cum se transmite o textură unui shader

Obiecte de tip textură



Este un obiect OpenGL ce conține una sau mai multe imagini (cu același format)

O textură poate fi folosită în două moduri:

- ▶ poate fi ca sursă pentru textură într-un shader;
- ▶ poate fi folosită ca destinație pentru crearea imaginii.

Ca orice obiect OpenGL sunt generate cu *glGenTextures* și trebuie legate de context cu *glBindTexture*.

Crearea și Inițializarea Texturilor



- ▶ `glGenTextures()` – crearea texturilor;
- ▶ `glBindTexture()` – legarea lor în contextul curent de un punct de destinație;
- ▶ `glTexStorage2D()` – se alocă spațiu pentru o textură (2D în cazul acesta);
- ▶ `glTexSubImage2D()` – specificăm datele ce se pun în spațiul alocat.

Crearea și Inițializarea Texturilor



```
GLuint texture;

// Generăm un nume pentru textură
glGenTextures(1, &texture);

// Legăm textura în contextul curent de
//      punctul GL_TEXTURE_2D
glBindTexture(GL_TEXTURE_2D, texture);

// Specificăm mărimea memoriei folosită pentru textură
glTexStorage2D(GL_TEXTURE_2D, // 2D texture
                1, // 1 mipmap level
                GL_RGBA32F, // 32-bit floating-point RGBA data
                256, 256); // 256 x 256 texels

// definim ceva date pentru a le încărca
float * data = new float[256 * 256 * 4];
```

Crearea și Inițializarea Texturilor



```
// generate_texture() - o funcție definită de noi
// pentru a pune o imagine în variabila data
generate_texture(data, 256, 256);

// trimitem datele spre contextul curent
// presupunem că textura e legată de GL_TEXTURE_2D
glTexSubImage2D(GL_TEXTURE_2D, // 2D texture
                 0, // Level 0
                 0, 0, // Offset 0, 0
                 256, 256, // 256 x 256 texels
                 GL_RGBA, // Four channel data
                 GL_FLOAT, // Floating-point data
                 data); // Pointer to data

// eliberăm memoria alocată
delete [] data;
```

Tipuri și puncte de destinație pentru texturi



Texture Target (<i>GL_TEXTURE_*</i>)	Description
1D	One-dimensional texture
2D	Two-dimensional texture
3D	Three-dimensional texture
<i>RECTANGLE</i>	Rectangle texture
<i>1D_ARRAY</i>	One-dimensional array texture
<i>2D_ARRAY</i>	Two-dimensional array texture
<i>CUBE_MAP</i>	Cube map texture
<i>CUBE_MAP_ARRAY</i>	Cube map array texture
<i>BUFFER</i>	Buffer texture
<i>2D_MULTISAMPLE</i>	Two-dimensional multi-sample texture
<i>2D_MULTISAMPLE_ARRAY</i>	Two-dimensional array multi-sample texture

Accesul texturilor din shadere



după ce am creat obiectul de tip textură și l-am populat cu date îl putem folosi în shadere

în shadere texturile sunt variabile de tip sampler (care au corespondent cu fiecare tip de textură)

pentru exemplul de mai sus textura bi-dimensională va fi sampler2D.

pentru a citi datele din textură vom folosi funcția predefinită:

texelFetch(...) — aduce un singur textel dintr-o textură

- ▶ *sampler* – specifică variabila de unde se iau textel-ii;
- ▶ *P* – specifică coordonatele texturii de unde se ia informația;
- ▶ *lod* – dacă există, specifică nivelul de detaliu din textura de unde e luat textel-ul;
- ▶ *sample* – pentru cereri multisample, se specifică din care sample se aduce informația.

Accesul texturilor din shadere

Exemplu:



```
#version 430 core

uniform sampler2D s;

out vec4 color;

void main(void)
{
    color = texelFetch(s, ivec2(gl_FragCoord.xy), 0);
}
```

Tipurile de Texturi și corespondența lor

Texture Target	Sampler Type
<code>GL_TEXTURE_1D</code>	<code>sampler1D</code>
<code>GL_TEXTURE_2D</code>	<code>sampler2D</code>
<code>GL_TEXTURE_3D</code>	<code>sampler3D</code>
<code>GL_TEXTURE_RECTANGLE</code>	<code>sampler2DRect</code>
<code>GL_TEXTURE_1D_ARRAY</code>	<code>sampler1DArray</code>
<code>GL_TEXTURE_2D_ARRAY</code>	<code>sampler2DArray</code>
<code>GL_TEXTURE_CUBE_MAP</code>	<code>samplerCube</code>
<code>GL_TEXTURE_CUBE_MAP_ARRAY</code>	<code>samplerCubeArray</code>
<code>GL_TEXTURE_BUFFER</code>	<code>samplerBuffer</code>
<code>GL_TEXTURE_2D_MULTISAMPLE</code>	<code>sampler2DMS</code>
<code>GL_TEXTURE_2D_MULTISAMPLE_ARRAY</code>	<code>sampler2DMSArray</code>

Texture maps

- ▶ Culoarea de bază/ Alberto /diffuse map – ne dă culoarea efectivă
- ▶ Metallic – proprietatea metalică a materialului de a reflecta lumina
- ▶ Roughness – proprietatea materialului de a absorbi sau reflecta lumina albă (alb e aspru, negru e neted)
- ▶ Normal – crează iluzia de adâncime
- ▶ Height – crează iluzia adâncime/înălțime (practic îmbunătățește ceea ce crează textura normală)
- ▶ Occlusion maps – determină ce zone sunt întunecate pentru a crea iluzia de umbră

The background features a large, stylized graphic element composed of several thick, curved lines in shades of blue, teal, and light orange. These lines curve from the left side towards the right, creating a sense of motion. Interspersed among the lines are small, glowing white dots that resemble stars or particles, particularly concentrated along the right edge.

Mulțumesc pentru atenție!

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

Cursul 7 - Modelarea luminii

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

28 aprilie 2022

Content



Modelarea luminii Modelul Phong



- ▶ În realitate lumina e extrem de complicată – aproape imposibil de simulat perfect
- ▶ folosim aproximări pe modele simplificate ce permit procesarea și arată similar cu realitatea
- ▶ Categorii de modele:
 - 1 **Modele globale** – ex: ray tracing, lumen (Unreal5)
extrem de costisitoare computațional, rezultate spectaculoase
 - 2 **Modele locale** – ex: modelul Phong
"ieftine" computațional, rezultate adecvate

Putem avea două tipuri de surse de lumină

- ▶ punctuale – surse de lumină locale
- ▶ direcționale – surse de lumină aflate la infinit

Matricea Model-view-projection afectează lumina

Diferite efecte apar când sunt specificate:

- ▶ coordonatele camerei,
- ▶ coordonatele globale,
- ▶ coordonatele modelului.

Lumina punctuală

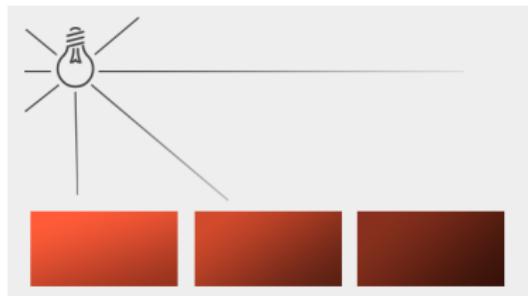


Figura: Lumina punctuală (spot light)

Lumina direcțională

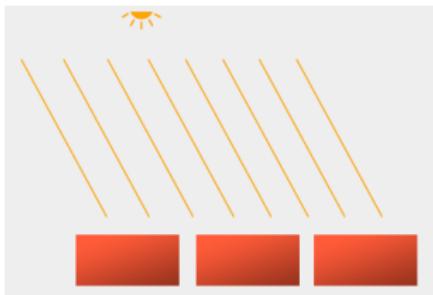


Figura: Lumina direcțională (directional Light)



Se numesc modele locale pentru că nu folosesc pentru a calcula luminozitatea unui punct alte informații decât cele locale (poziție, normală, tangentă, textură, etc.) și cele legate de sursele de lumină (poziții, tipuri, etc.)

Aventaje:

- ▶ folosesc doar informație locală;
- ▶ nu consumă multe resurse.

Dezavantaje:

- ▶ nu țin cont de obiectele din jur – dificil de realizat direct: transparentă, reflectia, refractia și umbrele.



Este o combinație de trei componente principale:

► *Ambient lighting* – lumină ambientală

chiar și pe întuneric vedem ceva (întunericul nu este absolut), undeva există o sursă de lumină. Simulăm acest lucru folosind o constantă care dă întotdeauna ceva culoare obiectelor.

► *Diffuse lighting* – lumină difuză

simulează impactul ce îl are lumina asupra unui obiect ce se găsește pe direcția ei. E cea mai importantă componentă a acestui model. Practic cu cât o parte a unui obiect se găsește în dreptul luminii cu atât e mai luminos.

► *Specular lighting* – lumina speculară

simulează acel punct luminos care apare pe obiectele scliptoare. Aceste accente au mai mult culoarea luminii decât culoarea obiectului.

Modelul Phong

Dezvoltat de *Bui Tuong Phong* la universitatea din Utah (1973) și extins de *James F. Blinn* (1977).

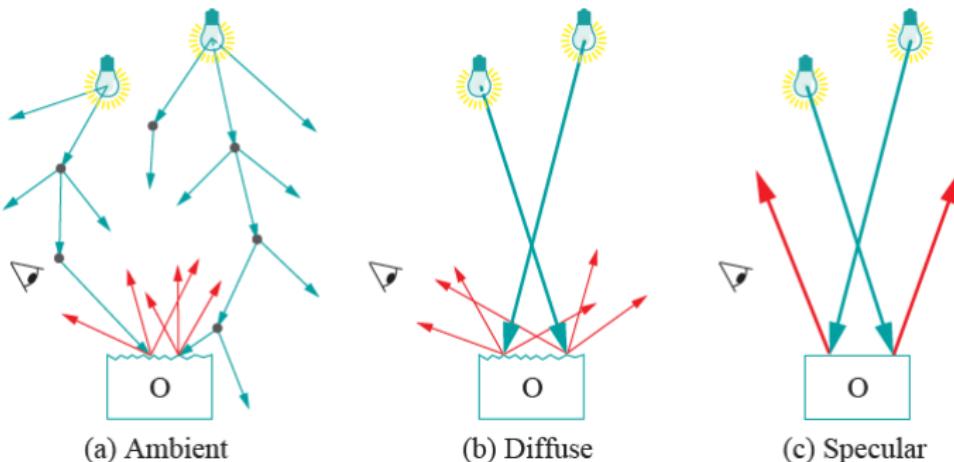
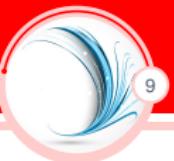


Figura: Cele trei componente ale modelului lui Phong.

Ambient Lighting

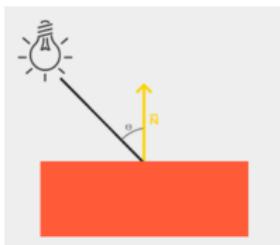


- ▶ simulează lumina slabă care se găsește peste tot
- ▶ se pune în fragment shader:

```
// ambient
float ambientStrength = 0.2;
vec3 ambient = ambientStrength * lightColor;

...
FragColor = vec4(ambient + diffuse + specular, 1.0) * aColor;
```

Diffuse Lighting



- ▶ În din vertex shader se transmit normalele în fiecare nod
- ▶ În fragment shader se folosesc aceste normale:

```
// diffuse
vec3 norm = normalize(nCoord);
vec3 lightDir = normalize(lumina - bPos);
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor;
...
FragColor = vec4(ambient + diffuse + specular, 1.0) * aColor;
```

- ▶ lumina – coordonatele luminii, bPos coordonatele nodurilor, nCoord normalelor
- ▶ normalele într-un nod A la suprafața triunghiulară ABC pot fi determinate prin produsul vectorial

$$BA \times CA$$

Atenție la direcția normalei (trebuie să fie spre exteriorul obiectului)!!

Normalele la un punct

Probleme



12

În urma transformărilor efectuate de matricea *model* normalele transmise în vertex shader **nu mai corespund** normalelor reale.

Soluție: transformarea normalelor în mod similar cu cea a vertexurilor

```
gl_Position = projection*view*model*vec4(aPos, 1);  
nCoord = mat3(transpose(inverse(model))) * normale;
```

practic transformăm normalele cu transpusa inversei matricii *model*.

Specular Lighting

- În fragment shader se transmite pozitia camerei și se adaugă componenta speculară

```
// specular
float specularStrength = 0.5;
vec3 viewDir = normalize(viewPos - bPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

The background features a large, stylized graphic element composed of several thick, curved lines in shades of blue, teal, and light orange. These lines curve from the left side towards the right, creating a sense of motion. Interspersed among the lines are small, glowing white dots that resemble stars or particles, adding a magical or celebratory feel to the design.

Mulțumesc pentru atenție!

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai, Cluj Napoca

Grafică pe calculator

Cursul 8 - Materiale

Tudor Dan Mihoc
tudor.mihoc@ubbcluj.ro

28 aprilie 2022

Content



Materiale și Lumină

Reflecția luminii

Strălucirea (luciul)

Modelarea Luminii

Ecuată Luminii

Modelarea Materialului

Componentele culorii materialului

Exemple de materiale

Exemple de cod

Lighting Maps



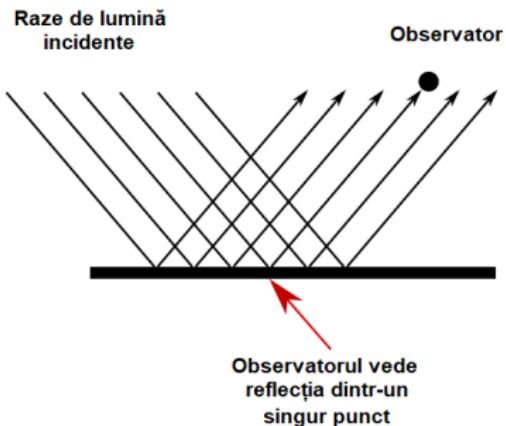
Orice obiect real este constituit dintr-un material:

- ▶ lemn brut – se văd fibrele, este rugos, nu strălucește, ...
- ▶ sticlă – este lucioasă, transparentă, netedă, ...
- ▶ metal – are luciu specific, ...
- ▶ ceramică – este netedă, stălucire medie (în funcție de finisaj), ...
- ▶ etc.

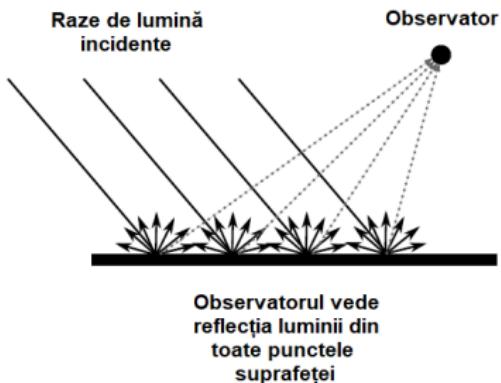
Fiecare material are proprietățiile lui care pentru o imagine cât mai realistă trebuie luată în considerare.

Reflectia luminii

Reflectia Speculară



Reflectia Difuză

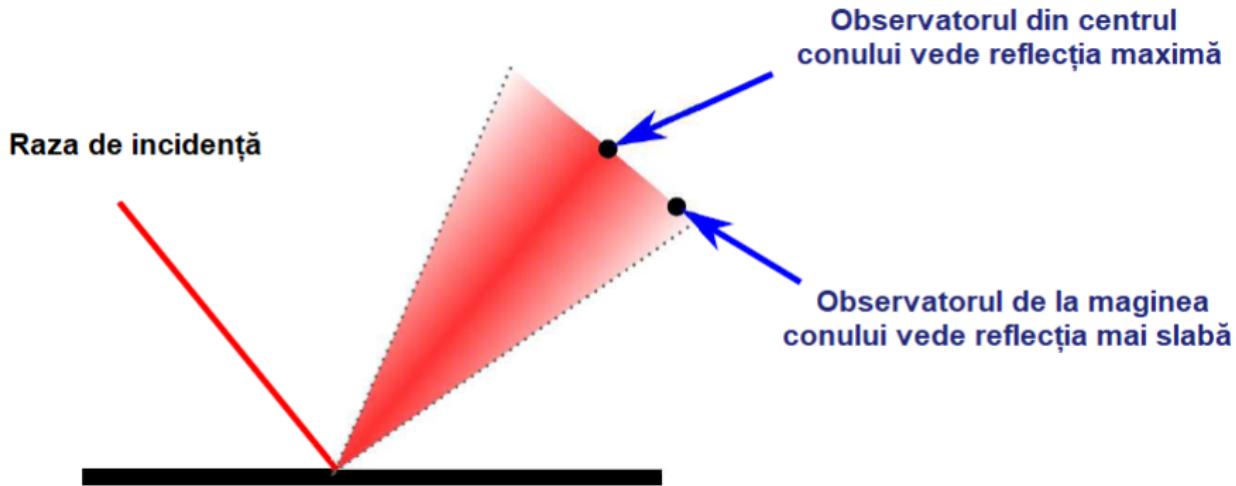


Conul de reflectie

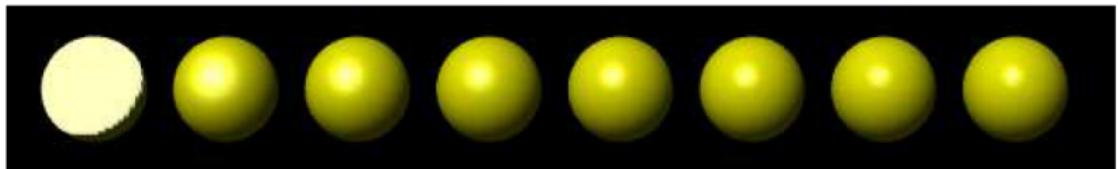


Modelarea accentelor speculare este făcută practic cu un con:

Conul de reflectie speculară



dacă materialul este mai lucios conul este mai îngust, dacă materialul este mai mat conul este mai larg.



- ▶ plecând de la stânga (strălucire 0) – o imagine nereușită
- ▶ mergând spre dreapta de la materiale mate spre strălucitoare
- ▶ ultimul element de la dreapta (strălucire maximă) – componenta speculară e aproape un punct



Culoarea luminii va fi modelată compunând-o din 4 componente:

- ▶ **culoarea difuză** (diffuse color) – culoarea luminii reflectată difuz de material
- ▶ **culoarea speculară** (specular color) – culoarea luminii reflectată specular de material
- ▶ **culoarea ambientală** (ambient color) – culoarea luminii ambientală reflectată de material
- ▶ **culoarea emisă** (emission color) – nu e o reflecție efectivă a luminii ci se referă la capacitatea materialului de a emite o lumină de o anumită culoare (deobicei este neagră (0.0, 0.0, 0.0)).

Ecuatia Luminii



Ignorăm deocamdată transparenta și considerăm doar componentele RGB din modelarea culorii:

- ▶ (md_r, md_g, md_b) culoarea difuză a materialului;
- ▶ (ms_r, ms_g, ms_b) culoarea speculară a materialului;
- ▶ (ma_r, ma_g, ma_b) culoarea ambientală a materialului;
- ▶ (me_r, me_g, me_b) culoarea emisă a materialului;

- ▶ (Id_r, Id_g, Id_b) culoarea difuză a luminii incidente;
- ▶ (Is_r, Is_g, Is_b) culoarea speculară a luminii incidente;
- ▶ (Ia_r, Ia_g, Ia_b) culoarea ambientală a luminii incidente;

Ecuatia luminii

- ▶ mh coeficient ce caracterizează proprietatea materialului de a străluci;
- ▶ f este 0 dacă suprafața nu e în lumină și 1 dacă este în lumină;
- ▶ L locația luminii;
- ▶ N normala la suprafață;
- ▶ V locația privitorului;
- ▶ R raza reflectată (centrul conului).

$$I_r = la_r * ma_r + f * (ld_r * md_r * (L \cdot N) + ls_r * ms_r * \max(0, V \cdot R)^{mh})$$

$$I_g = la_g * ma_g + f * (ld_g * md_g * (L \cdot N) + ls_g * ms_g * \max(0, V \cdot R)^{mh})$$

$$I_b = la_b * ma_b + f * (ld_r * md_b * (L \cdot N) + ls_r * ms_b * \max(0, V \cdot R)^{mh})$$

Modelarea materialului



În practică proprietățile materialului de a reacționa la lumină (culoarea materialului) sunt date de 4 parametri:

- ▶ ambient – (vector de 3 elemente float)
- ▶ diffuse – (vector de 3 elemente float)
- ▶ specular – (vector de 3 elemente float)
- ▶ shininess – (float)

Cu ajutorul acestor culori putem face materialului un 'fine tunning'

Exemple de materiale

Name	Ambient	Diffuse
emerald	(0.02, 0.17, 0.02)	(0.07, 0.61, 0.07)
jade	(0.13, 0.22, 0.15)	(0.54, 0.89, 0.63)
obsidian	(0.05, 0.05, 0.06)	(0.18, 0.17, 0.22)
pearl	(0.25, 0.20, 0.20)	(1.00, 0.82, 0.82)
ruby	(0.17, 0.01, 0.01)	(0.61, 0.04, 0.04)

Name	Specular	Shininess
emerald	(0.63, 0.72, 0.63)	0.6
jade	(0.31, 0.31, 0.31)	0.1
obsidian	(0.33, 0.32, 0.34)	0.3
pearl	(0.29, 0.29, 0.29)	0.088
ruby	(0.72, 0.62, 0.62)	0.6

Tabela: culorile și strălucirea câtorva materiale pe componente

Exemple de materiale

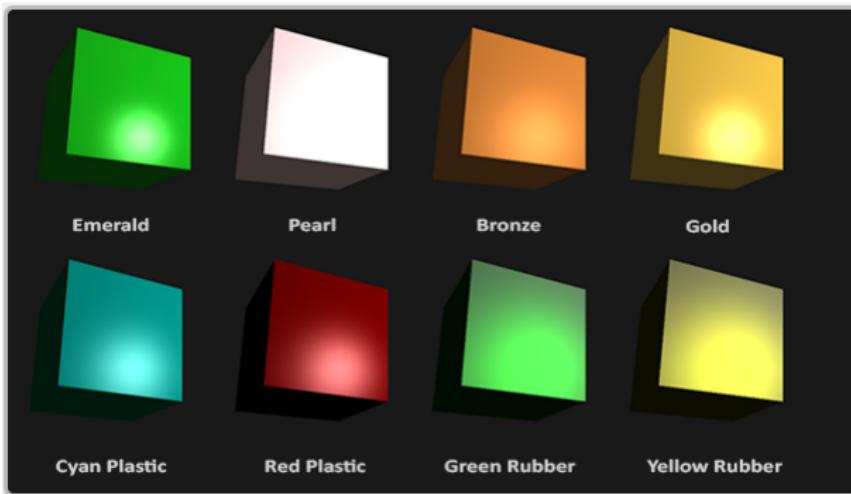


Figura: Câteva materiale. Primele 5 sunt construite conform tabelului anterior

Exemplu de cod

În fragment shader se definește structura Material și se initializează o variabilă uniformă de acest tip:

```
#version 330 core
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

uniform Material material;
```

Exemplu de cod

Avem apoi în funcția main din fragment shader:

```
void main()
{
    // ambient
    vec3 ambient = lightColor * material.ambient;

    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = lightColor * (diff * material.diffuse);

    // specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    vec3 specular = lightColor * (spec * material.specular);
}
```

Exemplu de cod

```
    vec3 result = ambient + diffuse + specular;  
    FragColor = vec4(result, 1.0);  
}
```

Pentru a transmite din programul nostru C++ valorile vom determina în mod normal locațiile ca la orice variabilă uniformă. De exemplu:

```
locatie = glGetUniformLocation(programul_nostru,  
                               "material.ambient");
```

După care se initializează ca orice altă variabilă uniformă.

Lighting Maps

- ▶ obiectele modelate pot avea mai multe materiale
- ▶ fiecare material va avea proprietățile lui
- ▶ apare necesitatea de a specifica rapid și eficient proprietățile materialelor

Soluția → folosirea unor texturi speciale: *diffuse* și *specular maps*.



Mulțumesc pentru atenție!