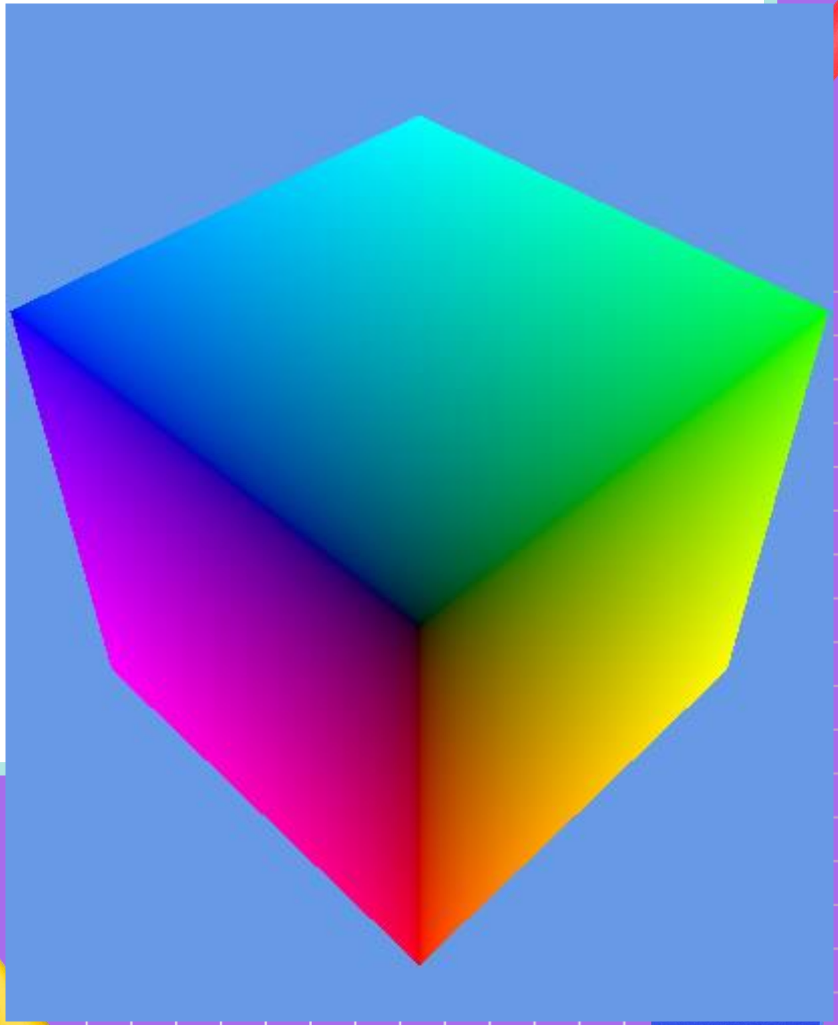
Programarea aplicațiilor de simulare

Curs 3 – Transformări și Randare



Conținut

01. Transformări

- Transformări 2D
- Transformări afine
- Coordonate omogene
- Transformări 3D

02. Geometrie

- Poligoane
- Transformarea geometriei
- Vizualizare & Proiecție

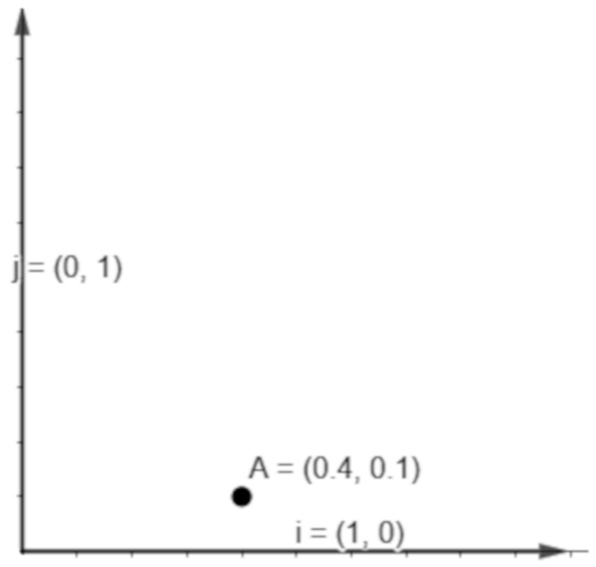
03. Randare

- Culori
- Rasterizare

04. OpenGL Pipeline

- Pipeline simplificat
- Shadere

Transformări

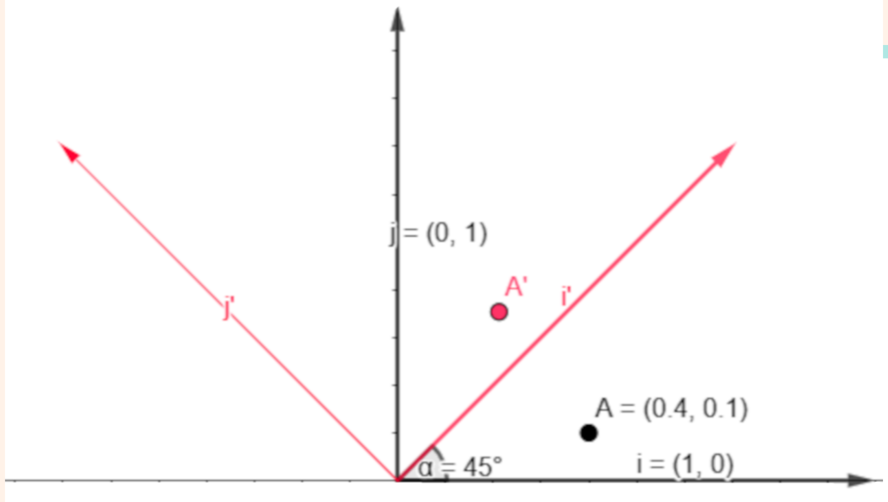


Coordonate carteziente

- Putem consideră puncte de forma $\mathbf{P} = (\boldsymbol{x}, \boldsymbol{y})$.
- $\vec{\boldsymbol{\imath}} = [\mathbf{1}, \mathbf{0}]$
- $\vec{\boldsymbol{\jmath}} = [\mathbf{0}, \mathbf{1}]$
- $\boldsymbol{P} = \boldsymbol{x}\vec{\boldsymbol{\imath}} + \boldsymbol{y}\vec{\boldsymbol{\jmath}}$

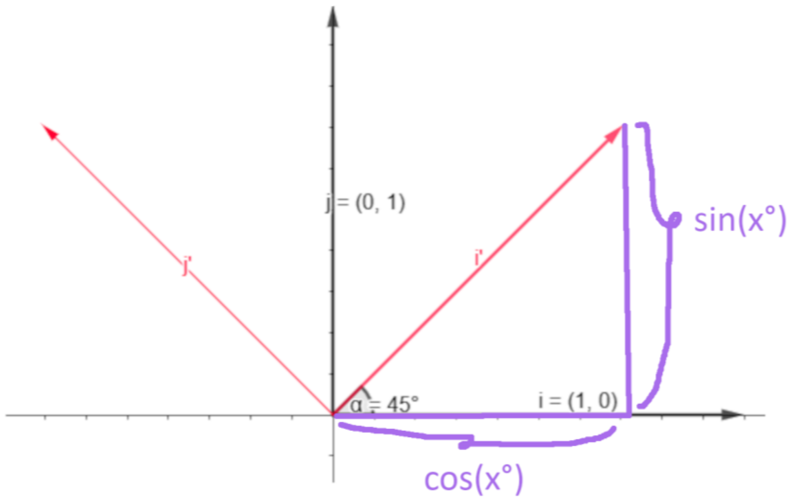
Exemplu

Fie $A = (0.4, 0.1)$



Coordonate carteziente

- Fie vectorii $\vec{i'}$ și $\vec{j'}$ doi vectori care corespund lui $\vec{i}$ și $\vec{j}$ rotiți cu $45°$ în sens trigonometric.
- $\vec{i'} = \left[\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]$
- $\vec{j'} = \left[-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]$
- Pentru a-l roti pe A în sens trigonometric cu $45°$, se poate face $A' = A_x\vec{i'} + A_y\vec{j'}$
- $A' = 0.4\left[\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right] + 0.1\left[-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]$
- $A' = \left(\frac{0.3\sqrt{2}}{2}, \frac{0.5\sqrt{2}}{2}\right)$



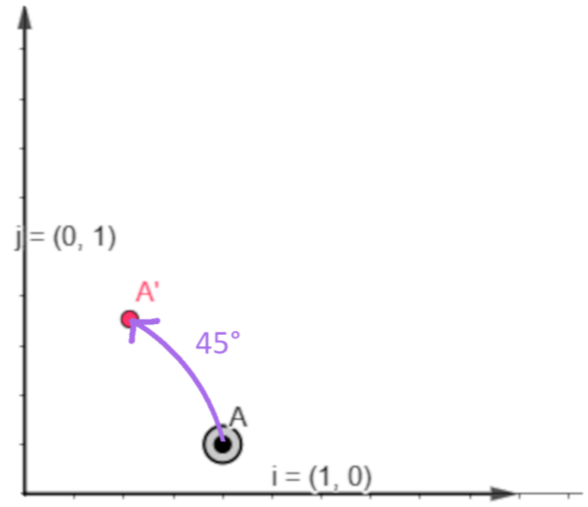
Baze vectoriale

- $\vec{i'}$ și $\vec{j'}$ formează o bază vectorială
- Folosind $\vec{i'}$ și $\vec{j'}$ poate fi descris orice punct din spațiul 2D
- Ei determină o transformare liniară
- Putem deduce vectorii $\vec{i'}$ și $\vec{j'}$ care determină o rotație de $x°$ în sens trigonometric.
- $\vec{i'} = [\cos(x°), \sin(x°)]$
- $\vec{j'} = [-\sin(x°), \cos(x°)]$
- Pornind de la un punct $P = [x, y]$, acesta poate fi rotit cu un unghi de $x°$ în sens trigonometric în jurul originii în felul următor: $P' = P_x\vec{i'} + P_y\,\vec{j'}$
- $P' = (P_x\cos(x°) - P_y\sin(x°), P_x\sin(x°) + P_y\cos(x°))$

Matrice

- Putem codifica o operație de tip $\boldsymbol{P'} = \boldsymbol{P_x}\vec{\boldsymbol{i'}} + \boldsymbol{P_y}\,\vec{\boldsymbol{j'}}$ folosind matrice $\boldsymbol{M_{ij}}$ de dimensiune $\mathbf{2} \times \mathbf{2}$
- Fie $\boldsymbol{P} = (\boldsymbol{P_x}, \boldsymbol{P_y}), \vec{\boldsymbol{i'}} = [\mathbf{i'_x}, \mathbf{i'_y}]$ și $\mathbf{j'} = [\boldsymbol{j'_x}, \boldsymbol{j'_y}]$, atunci:

- $P' = M_{ij}P = \begin{bmatrix} i'_x & j'_x \\ i'_y & j'_y \end{bmatrix}\begin{pmatrix} P_x \\ P_y \end{pmatrix} = (\mathrm{P_x} i'_x + \mathrm{P_y} j'_x, P_x i'_y + \mathrm{P_y} j'_y)$



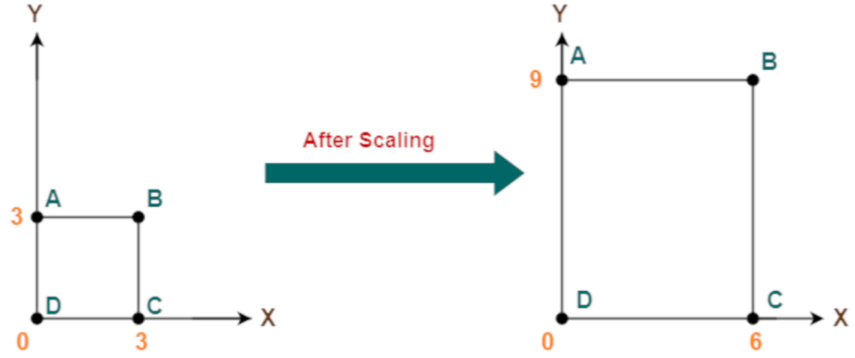
Tipuri de Transformări

- Pentru orice transformare liniară în 2D putem construi o matrice de dimensiune **2×2** care să codifice operația.

Rotație

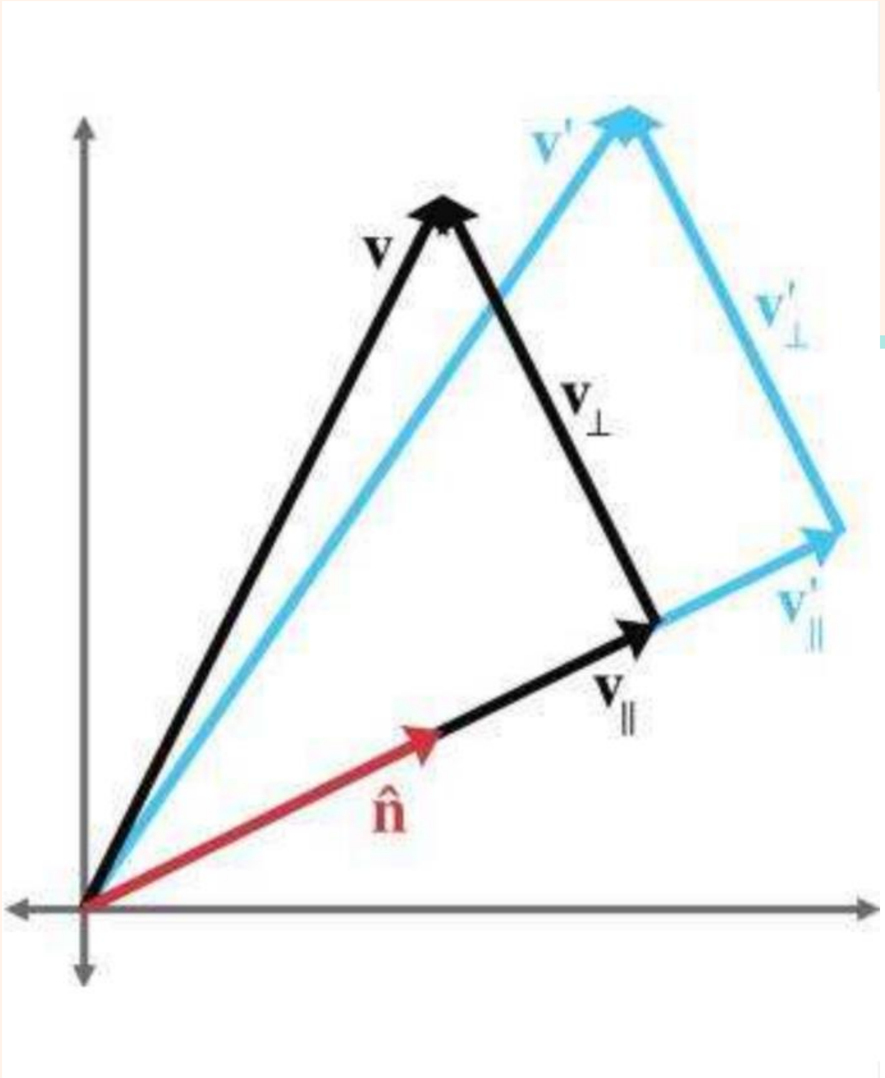
- Rotație în sens trigonometric de $x°$ în jurul originii
- $M = \begin{bmatrix} cos(x°) & -sin(x°) \\ sin(x°) & cos(x) \end{bmatrix}$
- $P' = MP = \begin{bmatrix} cos(x°) & -sin(x°) \\ sin(x°) & cos(x) \end{bmatrix} \begin{pmatrix} P_x \\ P_y \end{pmatrix} = \begin{pmatrix} P_x cos(x°) - P_y sin(x°) \\ P_x sin(x°) + P_y cos(x°) \end{pmatrix}$

Tipuri de Transformări



Scalare

- Scalare cu un factor s_x pe axa x și s_y pe axa y
- $P' = (P_x s_x, P_y s_y)$
- Exemplu: $B' = (B_x s_x, B_y s_y) = (2B_x, 3B_y) = (2 \cdot 3, 3 \cdot 3) = (6, 9)$
- $M = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$
- $P' = MP = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{pmatrix} \mathrm{P_x} \\ \mathrm{P_y} \end{pmatrix} = \begin{pmatrix} P_x s_x \\ P_y s_y \end{pmatrix}$

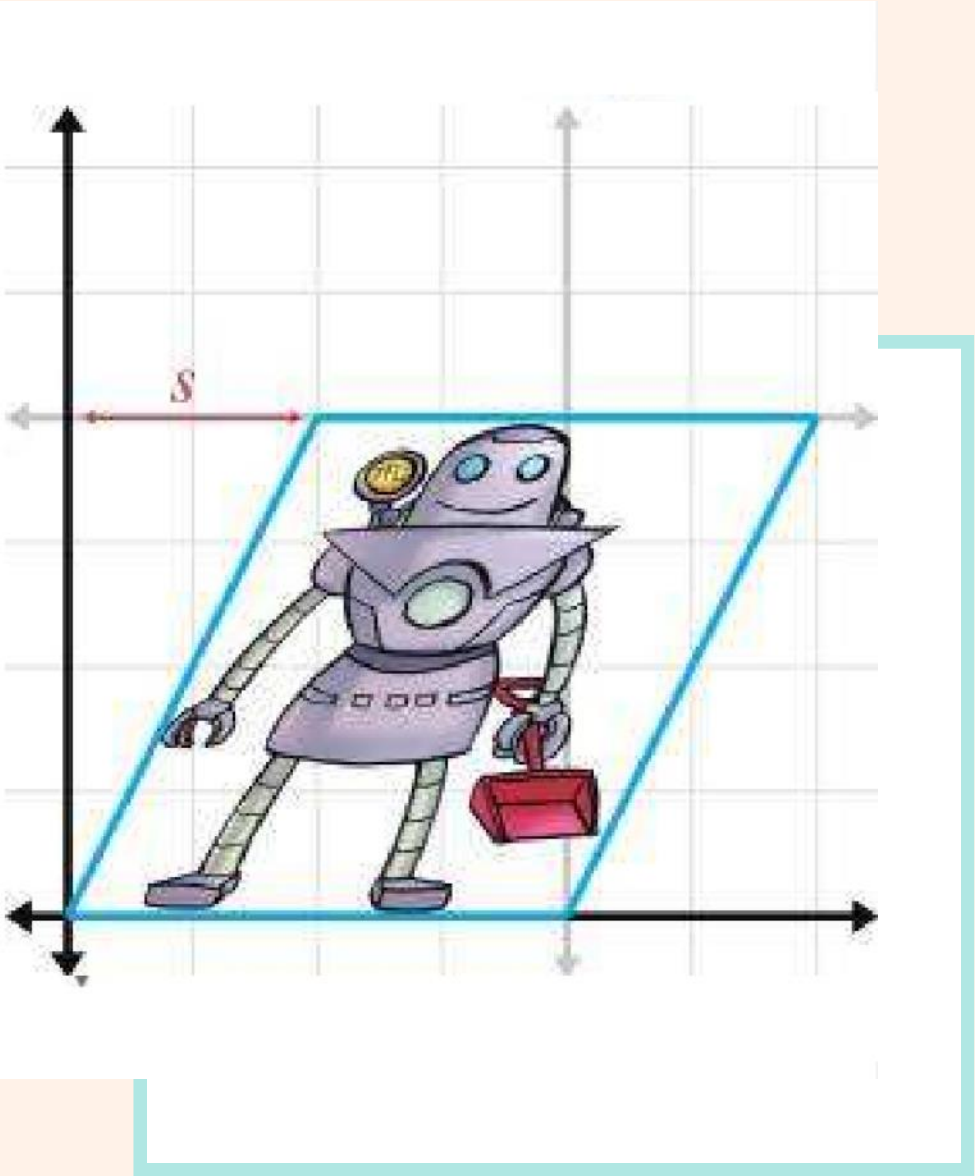


Tipuri de Transformări

Scalarea de-alungul unei axe

- Fie un vector normalizat $\vec{n}$ și un factor de scalare s, atunci putem calcula matricea $S(\vec{n}, s)$ care scalează un punct de-alungul axei determinate $\vec{n}$ cu un factor s.

$$S(\vec{n}, s) = \begin{bmatrix} 1 + (k-1)n_x^2 & (k-1)n_x n_y \\ (k-1)n_x n_y & 1 + (k-1)n_y^2 \end{bmatrix}$$



Tipuri de Transformări

Shearing (skew)

- Operație în care se adaugă un multiplu al unei coordonate celeilatle coordonate.
- Exemplu: $\boldsymbol{x' = x + sy}$
- $\boldsymbol{H_x(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}}$
- $\boldsymbol{H_y(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}}$

Transformări liniare

- În exemplele anterioare am folosit transformări liniare de forma $\boldsymbol{P'} = \boldsymbol{MP}$, unde $\boldsymbol{M}$ este o matrice de dimensiune $\mathbf{2} \times \mathbf{2}$, iar $\boldsymbol{P}$ un vector în spațiul 2D.

Translație

- Operația de translație deplasează un punct folosind un offset fix.
- $\boldsymbol{P'} = \boldsymbol{P} + \boldsymbol{T} = (\boldsymbol{P_x}, +\boldsymbol{T_x}\, \boldsymbol{P_y} + \boldsymbol{T_y})$

Putem construi o matrice de dimensiune 2×2 care să codifice operația de translație a unui punct în 2D?


NU PUTEM!!!

Transformări afine

- Transformările afine sunt de forma $\boldsymbol{P'} = \boldsymbol{MP} + \boldsymbol{T}$
- Toate transformările liniare sunt și transformări afine. Reciproca este în general falsă
- Translația este o transformare afină

Putem codifica transformările afine folosind matrice?


PUTEM!!!

Transformări afine

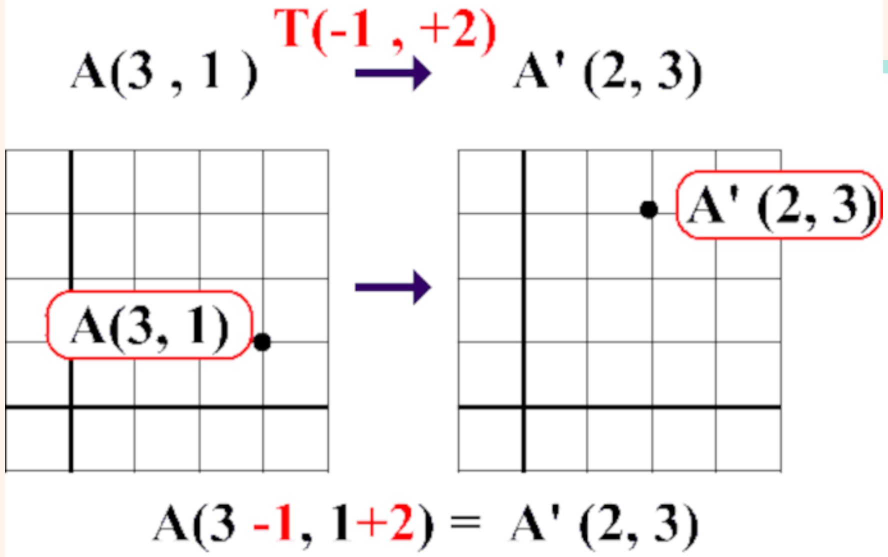
- Adăugăm încă o coordonată

$$\begin{pmatrix} P'_x \\ P'_y \end{pmatrix} = MP + T = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \begin{pmatrix} \mathrm{P_x} \\ \mathrm{P_y} \end{pmatrix} + \begin{pmatrix} T_\mathrm{x} \\ T_\mathrm{y} \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} P'_x \\ P'_y \\ 1 \end{pmatrix} = \begin{bmatrix} M_{11} & M_{12} & 0 \\ M_{21} & M_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} + \begin{pmatrix} T_x \\ T_y \\ 1 \end{pmatrix} =$$

$$\begin{bmatrix} M_{11} & M_{12} & T_x \\ M_{21} & M_{22} & T_y \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$



https://www.mathwarehouse.com/transformations/translations-in-math.php

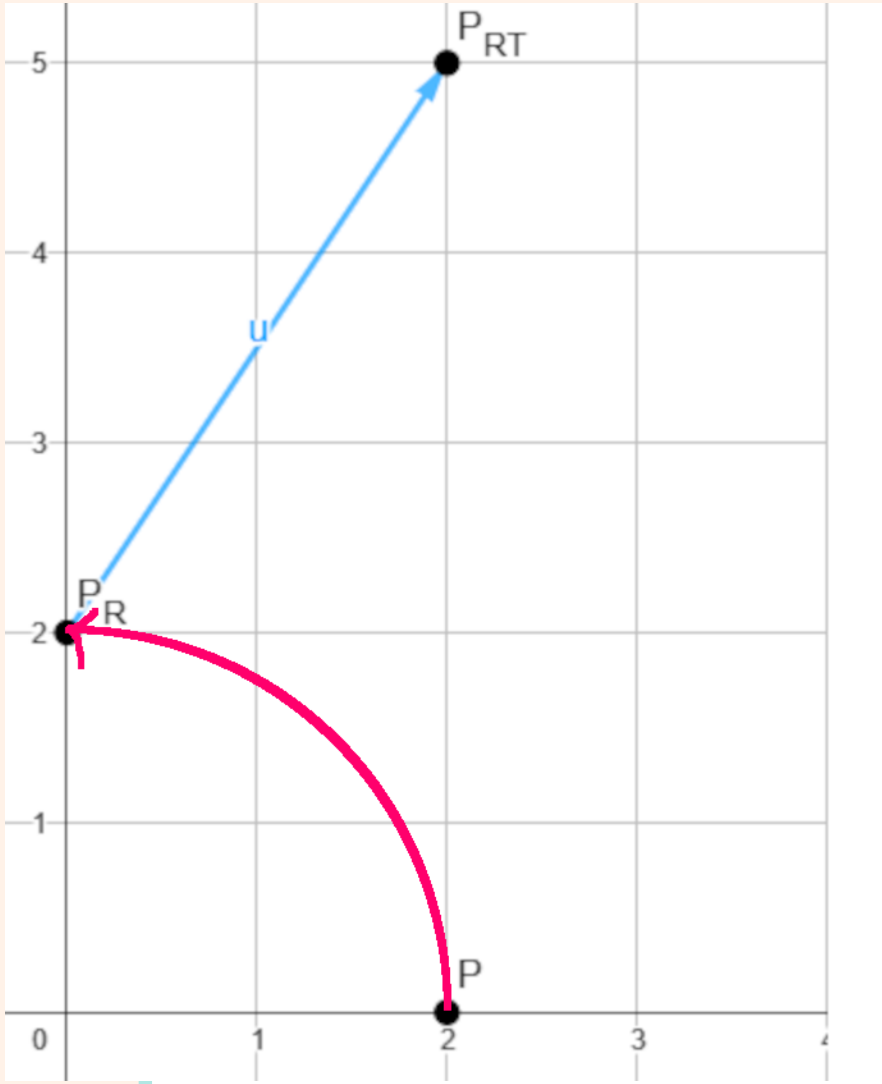
Tipuri de Transformări

Translație

- Este o transformare afină, deci trebuie să adăugăm o coordonată în plus
- Este o transformare afină, deci trebuie să adăugăm o coordonată în plus
- $M = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$

Combinarea transformărilor

- Toate transformările prezentate anterior pot fi codificate folosind matrice de dimensiune 3×3.
- Putem combina mai multe transformări prin înmulțirea matricelor aferente operațiilor.



Combinarea transformărilor

Exemplu

- Avem un punct $P = (2, 0)$ pe care vrem să-l rotim cu $90°$ în jurul originii în sens trigonometric, iar apoi să-i aplicăm o translație de $(2, 3)$
- Matricea de rotație

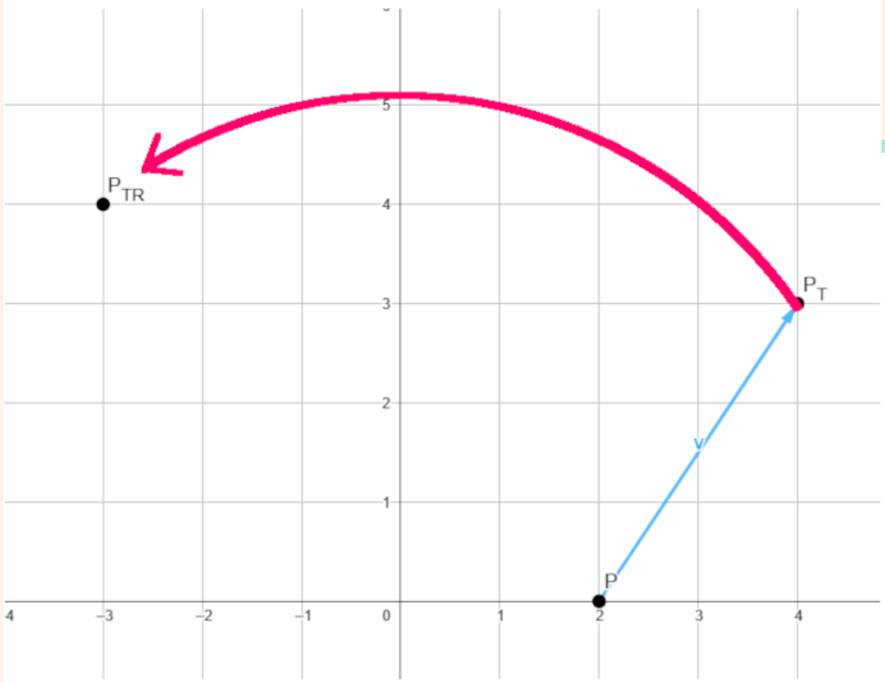
$$M_R = \begin{bmatrix} cos(90°) & -sin(90°) & 0 \\ sin(90°) & cos(90°) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Matricea de translație

$$M_T = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

- $P' = M_T M_R P = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} =$

$$\begin{bmatrix} 0 & -1 & 2 \\ 1 & 0 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}$$



Combinarea transformărilor

Ordinea operațiilor

- Ordinea în care sunt scrise operațiile contează!
- În cazul nostru folosim vectori coloană, deci operațiile se efectuează de la dreapta la stânga!
- O rotație urmată de o translație nu este același lucru o translație urmată de o rotație!

Exemplu

- Folosim exemplul anterior, dar prima dată efectuăm translația, apoi rotația
- $P' = M_R M_T P = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} =$

$$\begin{bmatrix} 0 & -1 & -3 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -3 \\ 4 \\ 1 \end{pmatrix}$$

Coordonate Omogene

- Coordonata inclusă anterior pentru a putea realiza transformări afine poate fi folosită și cu alte scopuri.

- Ne permite să folosim coordonate omogene. Coordonatele omogene sunt de forma (x, y, w)

- Coordonatele omogene au următoarea proprietate: $(x, y, 1) \leftrightarrow (wx, wy, w)$

- De regulă, dacă primim o coordonată (wx, wy, w) dorim să îi împărțim componentele la w pentru a obține un punct cu care putem lucra în coordonate carteziene.

Ce se întâmplă când $w = 0$?

Coordonate Omogene

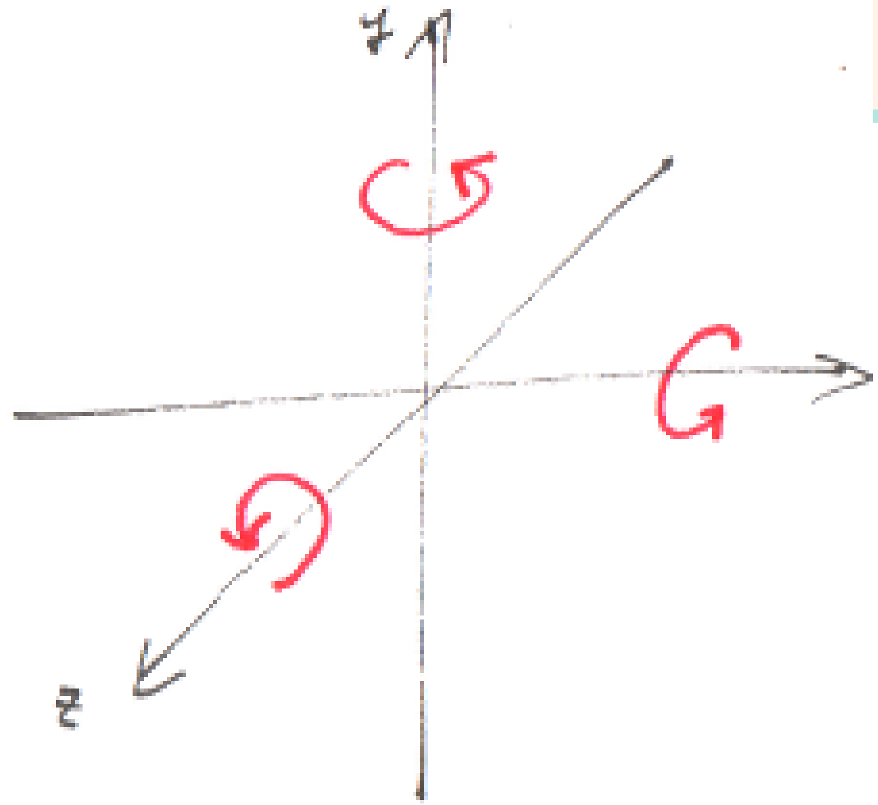
- Putem interpreta coordonatele de tip $(\boldsymbol{x}, \boldsymbol{y}, \mathbf{0})$ ca fiind direcții în loc de poziții concrete, deoarece împărțirile la 0 are ca rezultat $\pm\infty$.

- Approach-ul acesta de a considera vectorii ca fiind de forma $(\boldsymbol{x}, \boldsymbol{y}, \mathbf{0})$ funcționează excelent împreună cu operațiile matriceale, deoarece toate transformările rămân valide, în afară de translație, care este ignorată. Comportamentul este normal, deoarece translatarea vectorilor nu are sens întrucât aceștia nu sunt constrânși la o anumită origine.

Transformări 3D

- Toate transformările definite anterior se pot defini și în cazul coordonatelor 3-dimenionale.

- Și în acest caz se vor folosi coordonate omogene, lucrându-se cu vectori de dimensiune 4 și matrice de dimensiune $\mathbf{4} \times \mathbf{4}$.



Transformări 3D

Rotație

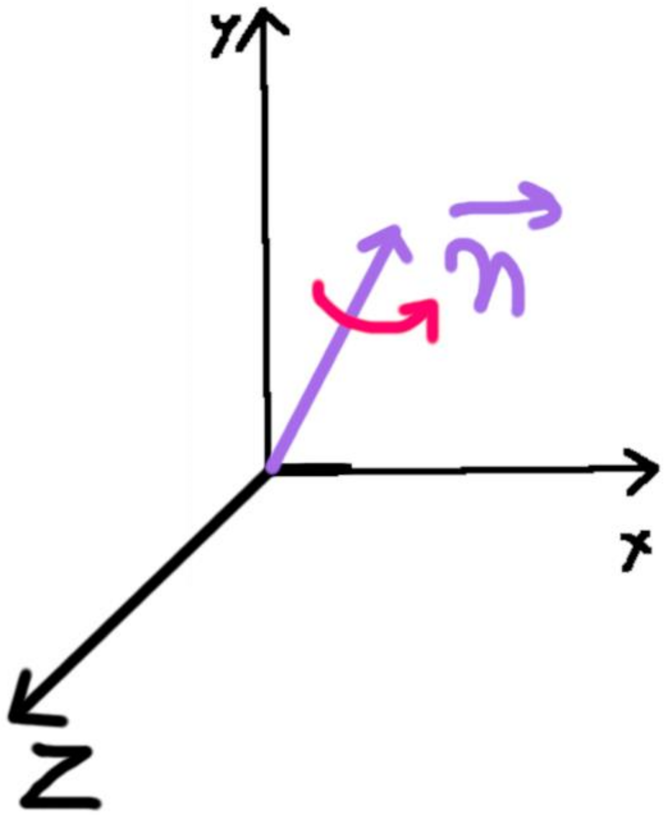
- În cazul 2 puteam să efectuăm rotații fie în sens trignometric, fie în sens opus.

- În cazul 3D, pe lângă acest lucru, trebuie să și stabilim axa în jurul căreia se efectuează rotația.

- Următoarele matrice codifică rotații în jurul fiecărei axe de coordonate:

- $R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 0 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- $R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$



Transformări 3D

Rotație în jurul unei axe aleatoare

- Fie $\vec{n}$ un vector de lungime reprezentând axa de rotație în jurul căreia vom efectua transformarea.
- Atunci:
- $R(\vec{n}, \theta) =$

$$\begin{bmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_z n_y(1-\cos\theta)-n_z\sin\theta & n_x n_z(1-\cos\theta)+n_y\sin\theta & 0 \\ n_x n_y(1-\cos\theta)+n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_y n_z(1-\cos\theta)-n_x\sin\theta & 0 \\ n_x n_z(1-\cos\theta)-n_y\sin\theta & n_y n_z(1-\cos\theta)+n_z\sin\theta & n_z^2(1-\cos\theta)+\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ÎNVĂȚAȚI PE DINAFARĂ FORMULA, O VEȚI AVEA LA EXAMEN ȘI NU VEȚI AVEA VOIE CU MATERIALE CARE SĂ O CONȚINĂ. PUNCT.

Transformări 3D

Scalare

- Fie factorii de scalare s_x, s_y, s_z de-alungul axelor de coordonate.
- Atunci matricea de scalare este:

$$M = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformări 3D

Scalare de-alungul unei axe

- Fie un vector normalizat $\vec{\boldsymbol{n}}$ și un factor de scalare $\boldsymbol{s}$, atunci putem calcula matricea $\boldsymbol{S}(\vec{\boldsymbol{n}}, \boldsymbol{s})$ care scalează un punct de-alungul axei determinate $\vec{\boldsymbol{n}}$ cu un factor $\boldsymbol{k}$.

$$S(\vec{\boldsymbol{n}}, k) = \begin{bmatrix} 1+(k-1)n_x^2 & (k-1)n_x n_y & (k-1)n_x n_z & 0 \\ (k-1)n_x n_z & 1+(k-1)n_y^2 & (k-1)n_y n_z & 0 \\ (k-1)n_x n_z & (k-1)n_y n_z & 1+(k-1)n_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformări 3D

Shearing (skew)

- Operație prin care se adaugă multiplii ai unei coordonate celorlalte două coordonate.

- Exemplu: $x' = x + sz,\; y' = y + tz$

$$H_{xy}(s,t) = \begin{bmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad H_{xz}(s,t) = \begin{bmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & t & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_{yz}(s,t) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ s & 1 & 0 & 0 \\ t & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformări 3D

Translație

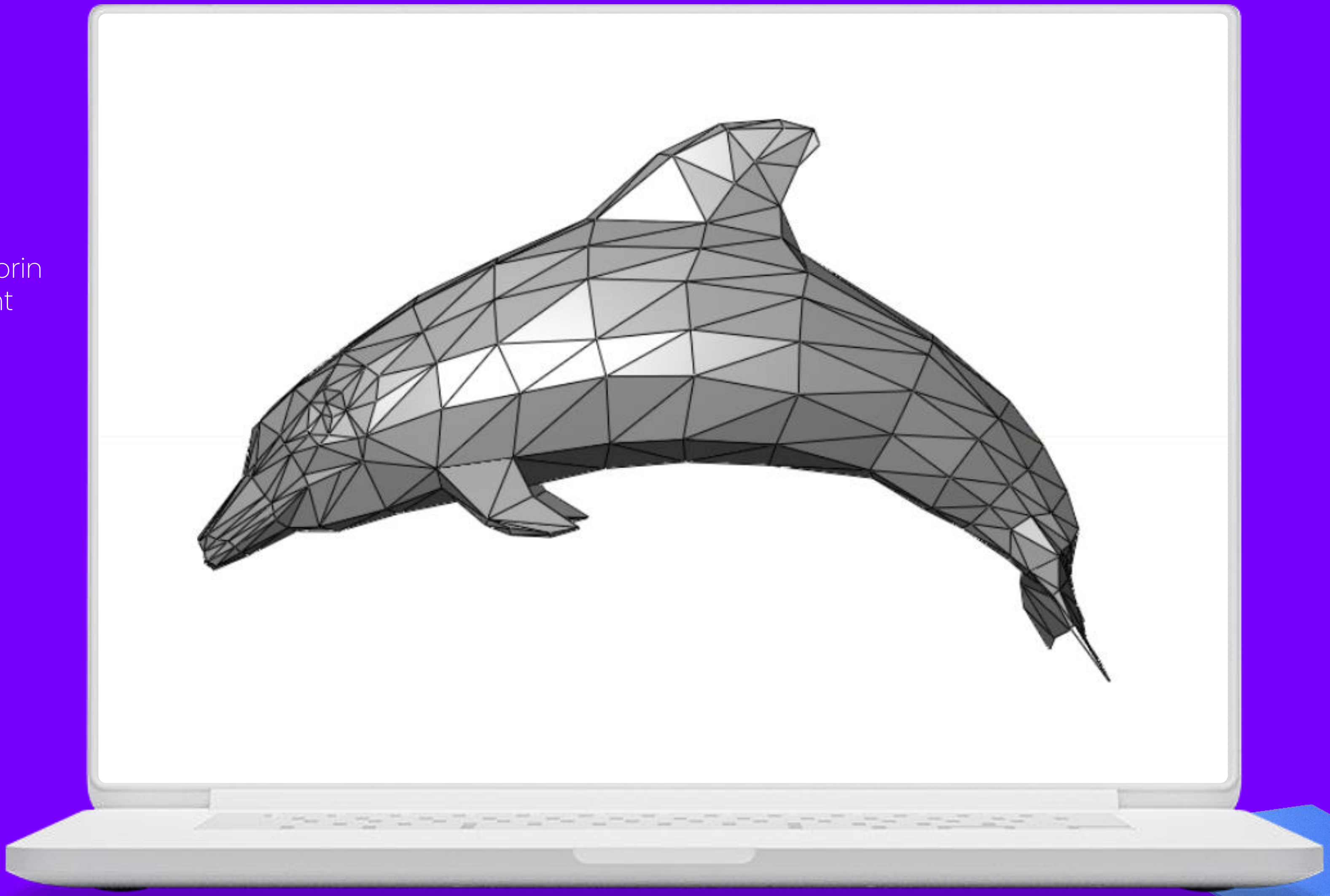
- Analog cu cazul 2D

$$M = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Geometrie

Poligoane

Putem defini obiecte în 2D și în 3D prin definirea poligoanelor din care sunt alcătuite.



Cum definim poligoanele?

Triunghiuri

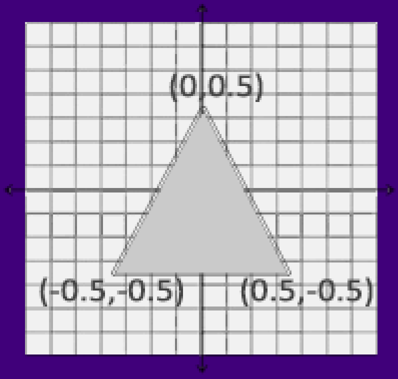
- În jocuri, de regulă poligoanele sunt reprezentate de triunghiuri deoarece placa video are implementat în hardware desenarea acestora (vom continua discuția mai încolo)

- Un triunghi este alcătuit din 3 vârfuri. Aceste vârfuri le vom numi **VERTECȘI.**

Vertecși

- Vertecșii îi putem asemăna cu puncte în spațiul 3D întrucât aceștia au și un punct pe care îl reprezintă.
- Un vertex poate conține mai multe informații decât poziția punctelor.
- Atunci când se crează un model, vertecșii acstuia trebuie să fie încărcați în memoria plăcii video. Game Engine-urile fac asta automat.
- 3 vertecși formează un triunghi
- Exemplu în OpenGL:

```
float vertices[] =
{
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};

unsigned int vbo;
glGenBuffers(1, &vbo);              // creates a vertex buffer object (VBO)
                                    // you can imagine it as a pointer to the
                                    // place in GPU memory where the vertices were copied

glBindBuffer(GL_ARRAY_BUFFER, vbo);  // tell OpenGL to use the new VBO


// actually upload the vertices into GPU memory
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```



Indici

- După ce am copiat vertecșii în memoria plăcii video îi putem folosi pentru a desena obiectele.
- Dar, dar, dacă ne rezumăm la metoda descrisă mai sus, când vom lucra cu modele complexe, vom avea vertecși duplicați în cazul triunghiurilor care împart aceleași vârfuri.
- Pentru a rezolva această problemă, putem încărca o singură dată în memoria plăcii video toți vertecșii geometriei, iar apoi putem specifica poligoanele pe care aceștia îi formează folosind indici.
- 3 indici formează un triunghi

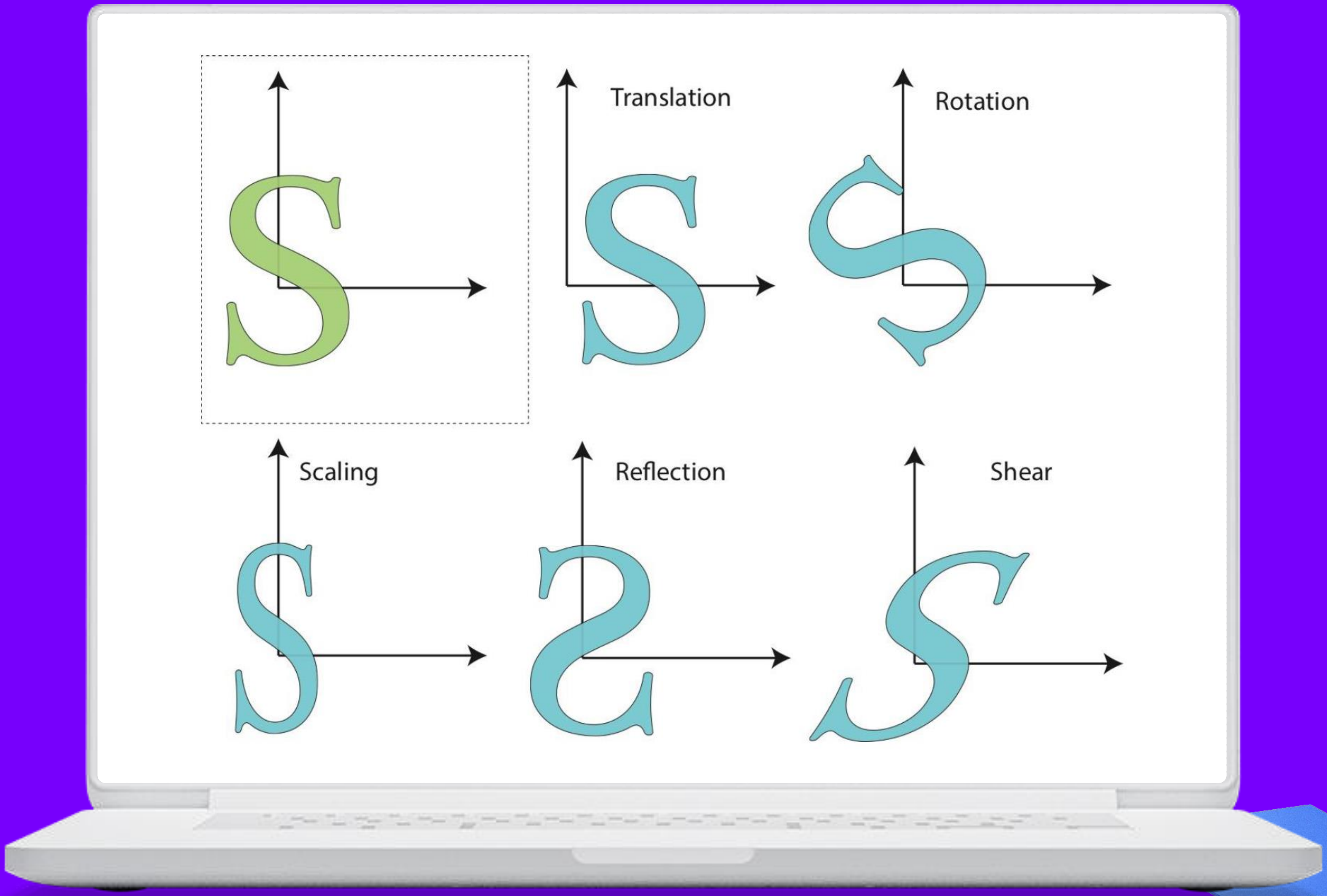
```
float vertices[] =
{
     0.5f,  0.5f, 0.0f,  // top right
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f, -0.5f, 0.0f,  // bottom left
    -0.5f,  0.5f, 0.0f   // top left
};

unsigned int indices[] =
{  // note that we start from 0!
    0, 1, 3,   // first triangle
    1, 2, 3    // second triangle
};
```

```
unsigned int vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

unsigned int ebo;
glGenBuffers(1, &ebo);                          // create a buffer to store the
                                                // indices on the GPU (element
                                                // buffer object or EBO for friends)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);     // bind the EBO

// upload the indices of the vertices that form the triangles
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Transformarea geometriei

Putem aplica transformările prezentate anterior pentru a altera obiectele.

Matricele sunt înmulțite cu poziția fiecărui vertex.

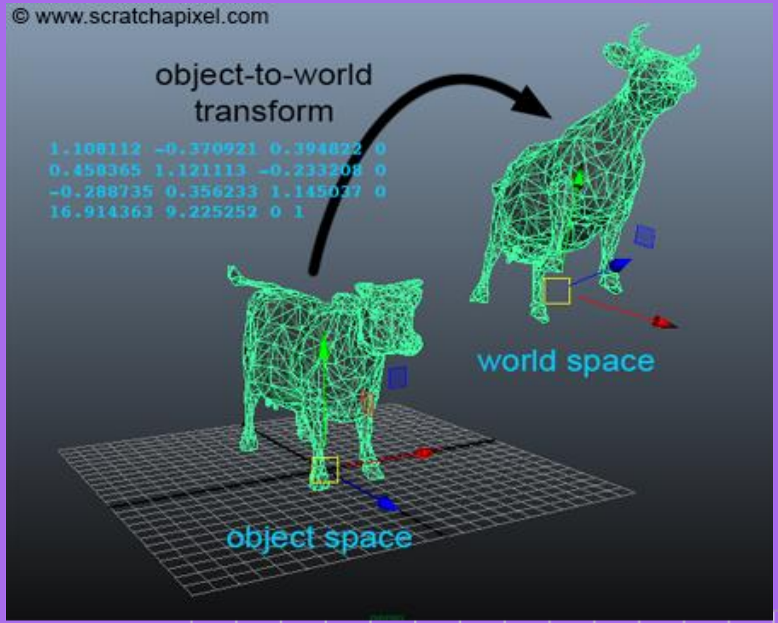


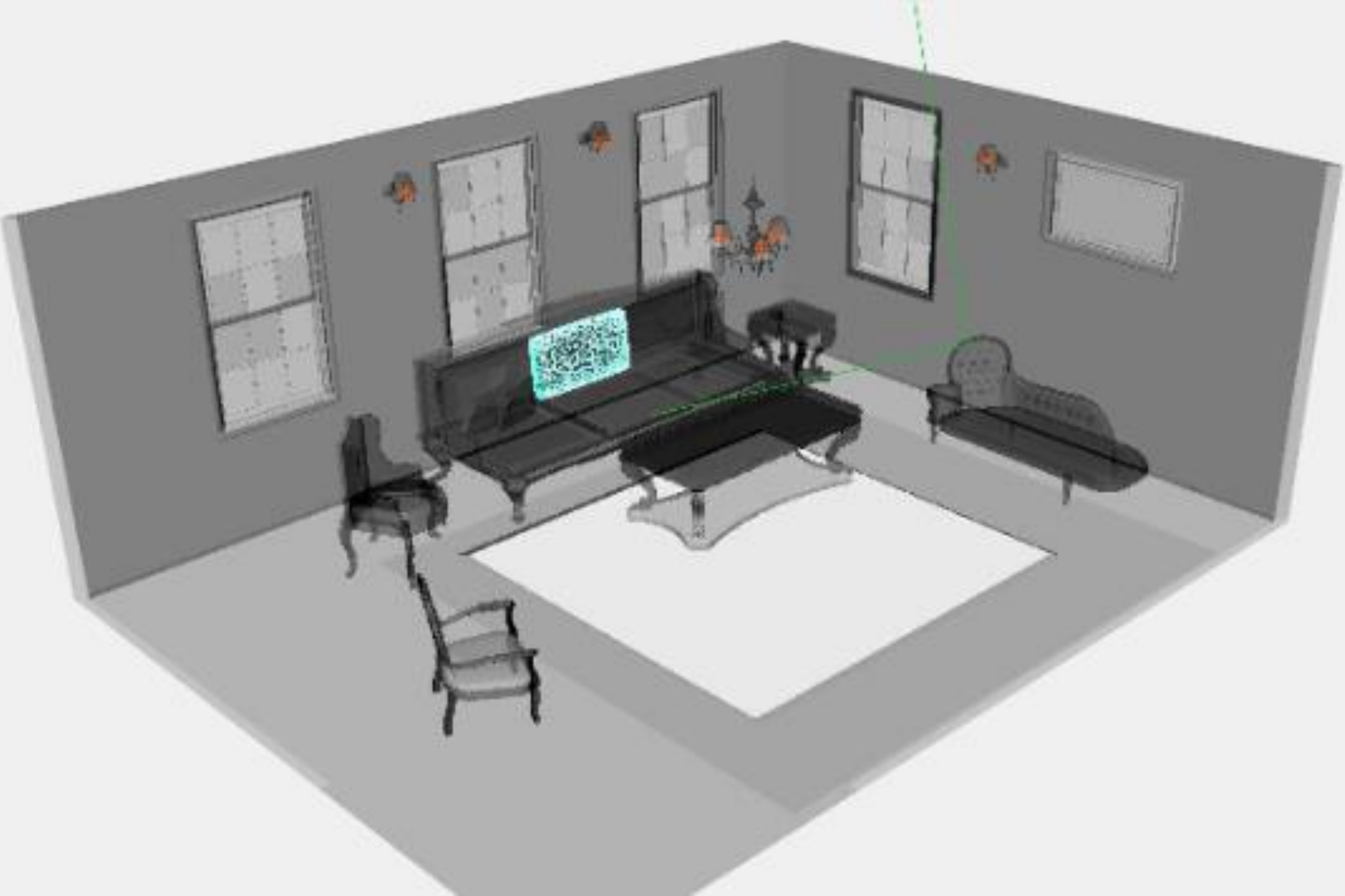
Transformarea geometriei

- Pozițiile vertecșilor sunt definite în **Object Space**, un spațiu în care toate pozițiile sunt relative la poziția obiectului.

- Pentru a poziționa un obiect în scenă se folosește alt spașiu, numit **World Space**.

- Se iau toate transformările care se efectuează asupra obiectului, se calculează matricele acestora, iar apoi se înmulțesc. Rezultatul va fi o matrice numită **World Matrix**. Această matrice va fi înmulțită cu coordonatele fiecărui vertex al geometriei, astfel transformând coodronatele din **Object Space** în **World Space**.

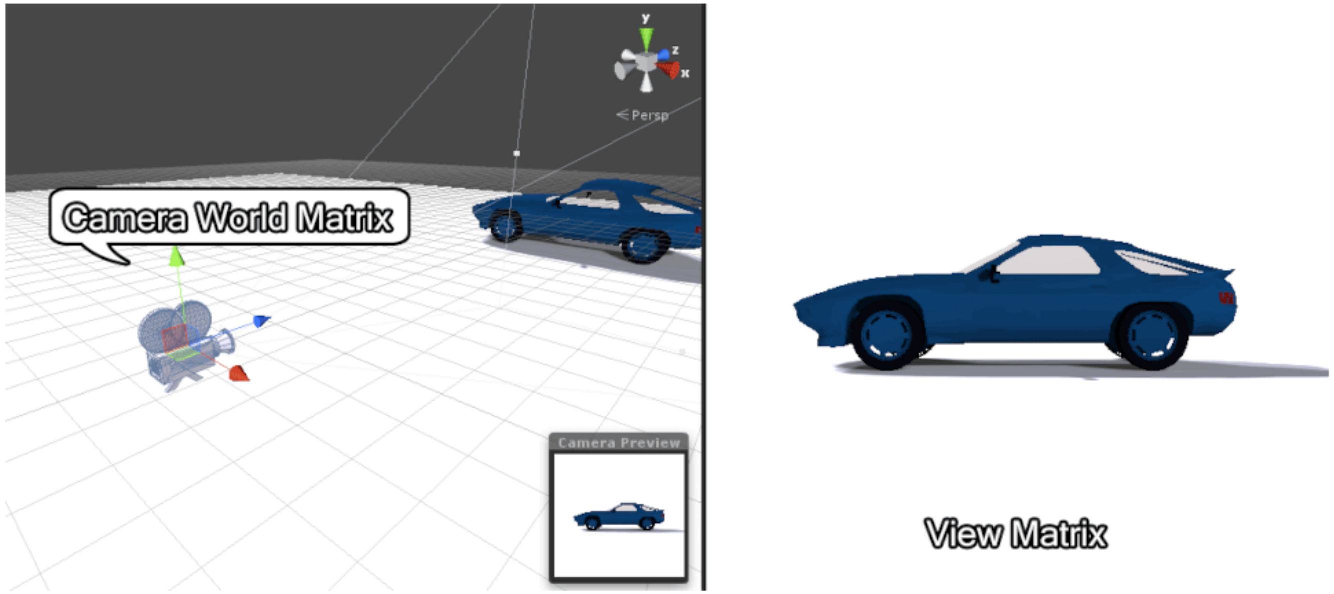




Vizualizare

- Ne putem imagina că în scenă există o cameră care are o poziție și o orientare.
- Putem crea o matrice M_{cam} care să codifice transformarea camerei.
- Cum nu avem o cameră propriu-zisă, trebuie ca asupra geometriei să se aplice o transformare astfel încât să pară că aceasta este văzută din perspectiva camerei. Acest spațiu se numește **View Space**.
- Pentru un punct P din **World Space** în **View Space** este nevoie să aplicăm transformarea $P' = M_{cam}^{-1}P$
- M_{cam}^{-1} se numește **View Matrix**, și există mai multe metode de a calcula o astfel de matrice în funcție de specificul aplicației.
- Dacă avem o matrice ai cărei bază vectorială este ortogonală, iar vectorii sunt unitari, atunci $M^{-1} = M^T$

Vizualizare

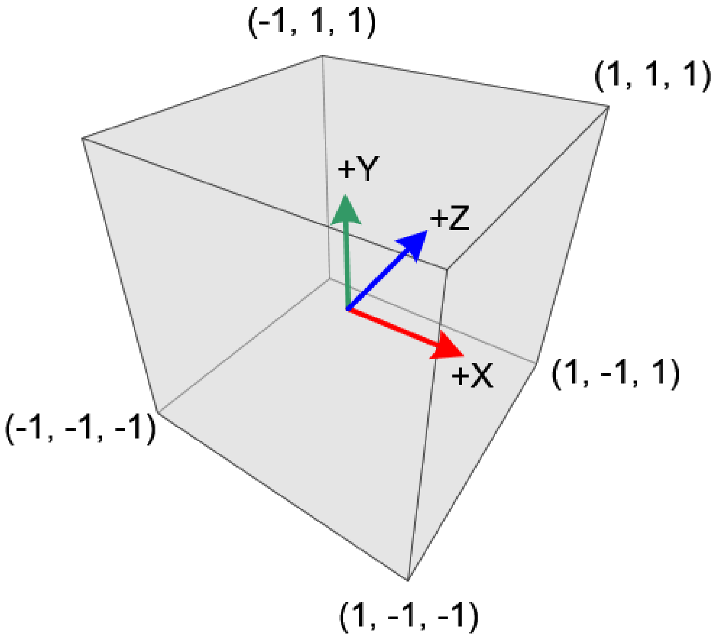


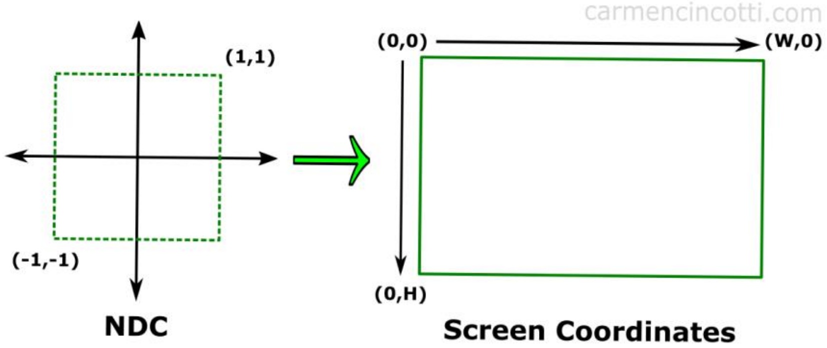
Normalized Display Coordinates (NDC)

- API-urile grafice folosesc coordonate într-un spațiu normalizat (coordonatele au valori între -1 și 1 sau între 0 și 1)

- În OpenGL, NDC-ul poate fi vizualizat ca un cub cu latura de lungime 2 și extremitățile $(-1, -1, -1)$, respectiv $(1, 1, 1)$

- Acest spațiu de coordonate se numește **Clip Space**, și reprezintă spațiul final în care vom dori să aducem coordonatele pentru ca geometria să fie desenată pe ecran.





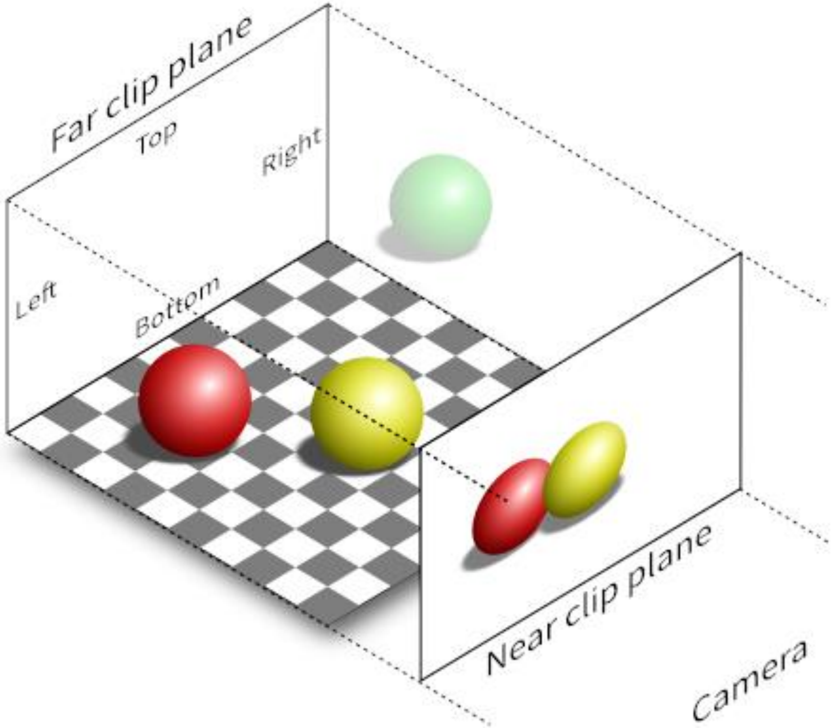
Cum aducem coordonatele din View Space în Clip Space?

Proiecție

- Se crează o matrice de 4×4 care efectuează transformarea.

- Această matrice se numește **Projection Matrix**.

- În funcție de specificul aplicației, aceasta poate fi construită în mai multe moduri.

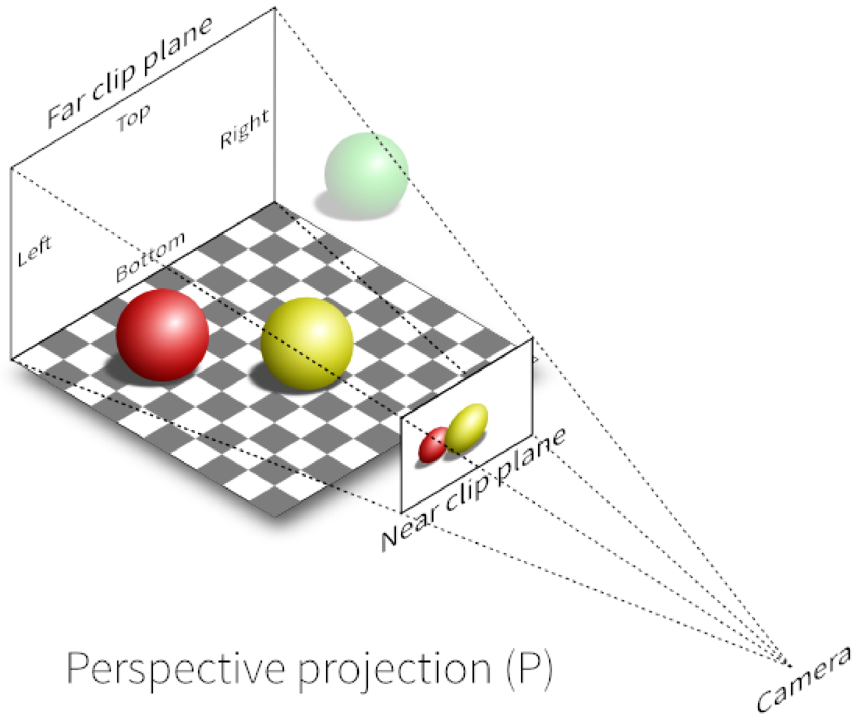


Orthographic projection (O)

Proiecție

Proiecție ortografică

- Proporțiile obiectelor sunt păstrate, indiferent de poziția acestora.
- Liniile paralele rămân paralele.
- $M = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{left+right}{2} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{top+bottom}{2} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$



Proiecție

Proiecție perspectivă

- Proiecție realistă, asemănătoare cu vederea umană
- Se calculează în funcție de FoV (Field of View), aspect ratio și distanța minimă (near plane), respectiv maximă (far plane) pe care o poate vedea camera.
- $M = \begin{bmatrix} \frac{cot\frac{fov_y}{2}}{aspect} & 0 & 0 & 0 \\ 0 & cot\ \frac{fov_y}{2} & 0 & 0 \\ 0 & 0 & \frac{near+far}{mear-far} & \frac{2 \cdot near \cdot far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$

Concluzie

- Geometria este reprezentată de poligoane (triunghiuri)
- Un triunghi este format din 3 vertecși care conțin și informații referitoare la pozițiile vârfurilor.
- Vertecșii sunt încărcați în memoria plăcii video, și se specifică și perechi a câtre 3 indici care formează triunghiurile

Concluzie

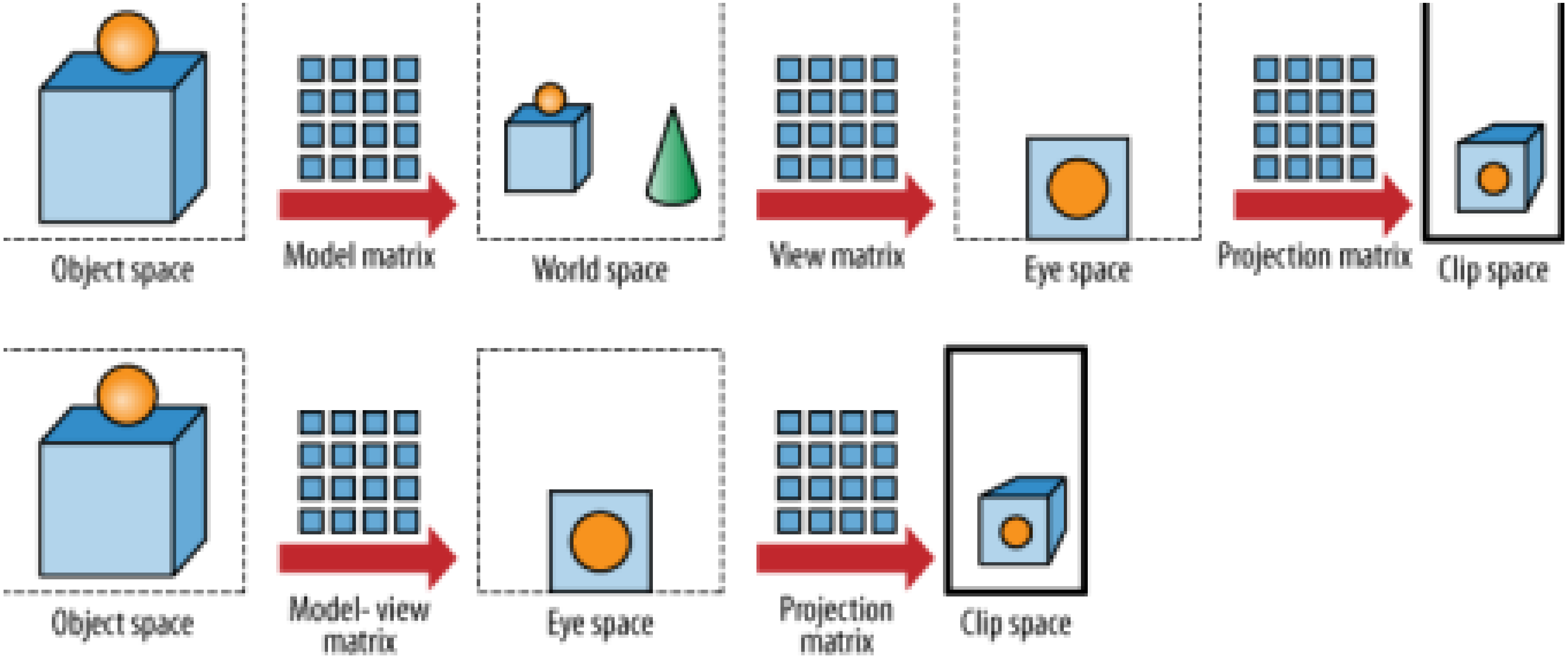
- Pozițiile vârfurilor se află în **Object Space**.

- Pentru a plasa obiectele în scenă se folosește o matrice care transformă coordonatele din **Object Space** în **World Space**. Această matrice se numește **World Matrix**.

- Pentru a transforma o poziție din lume astfel încât aceasta să poată fi văzută de o camera, aceasta este înmulțită cu **View Matrix** (matricea de vizualizare). Acest spațiu se numește **View Space.**

- Pentru a putea afișa pe ecran un poligon, coordonatele acestuia trebuie să fie reprezentate în **Clip Space**. Această transformare se efectuează folosind un **Projection Matrix**

$$P_{clip} = M_{proj} M_{view} M_{world} P_{object}$$



Randare

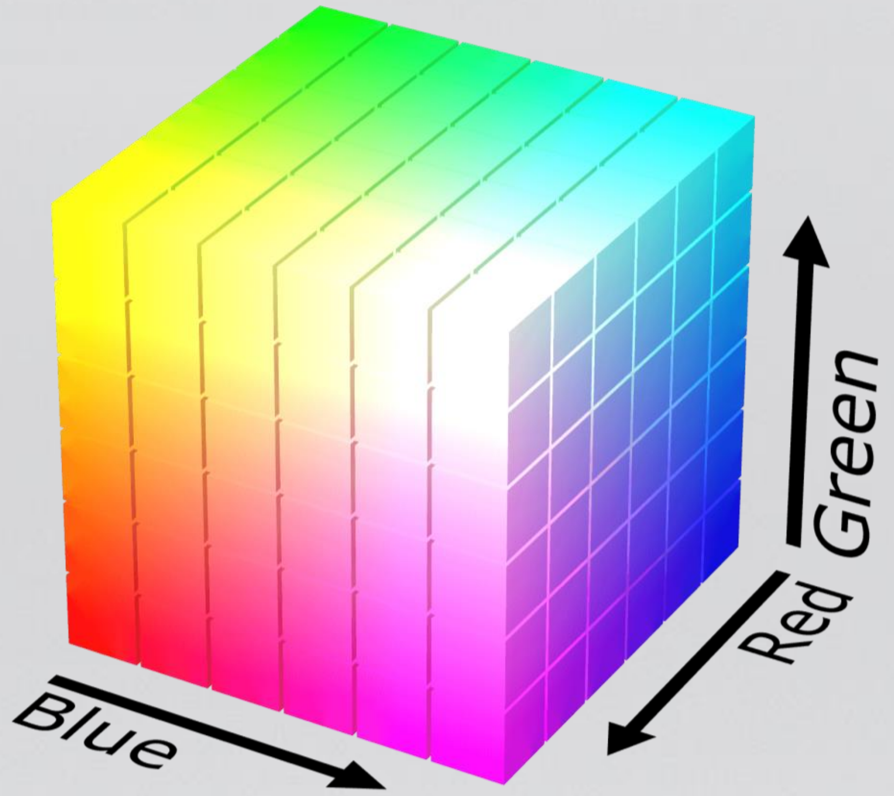
Spațiul culorilor

Culorile sunt obținute combinând internsitățile de pe 3 canale: R (red); G (green); B (blue) – Cubul RGB

Acestea au de regulă valori normalizate (între 0 și 1).

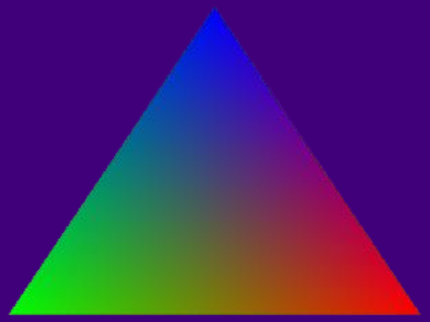
Exemple:

$(0,1,0)$ reprezintă culoarea verde
$(1,1,0)$ reprezintă culoarea galben

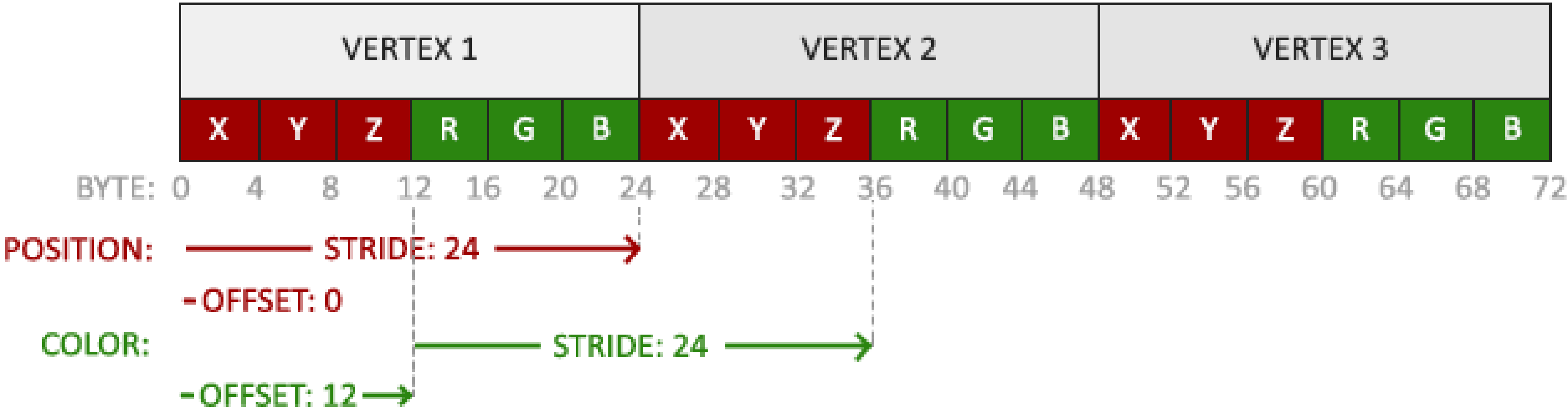


Vertecși

- O informație pe care o putem atașa vertecșilor este culoarea acestora. Astfel, putem avea vertecși care să specifice pozițiile vârfurilor și culorile acestora.

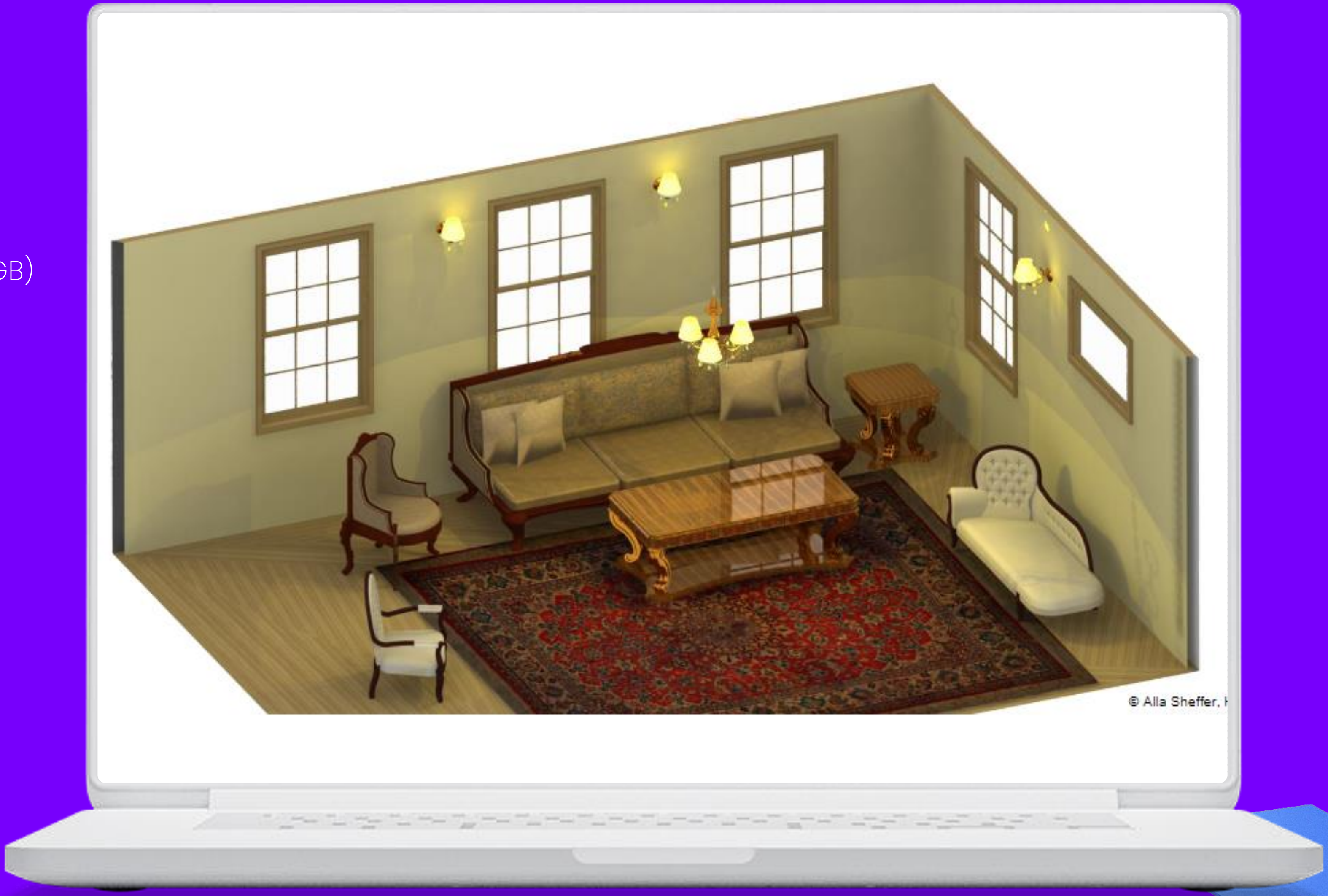


```
float vertices[] =
{
    // positions         // colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,   // bottom left
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // top
};

unsigned int vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float)));
glEnableVertexAttribArray(1);
```


VERTEX 1
VERTEX 2
VERTEX 3
X Y Z R G B X Y Z R G B X Y Z R G B
BYTE: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72
POSITION:
STRIDE: 24
OFFSET: 0
COLOR:
STRIDE: 24
OFFSET: 12

Imagine

O matrice de valori ale culorilor (RGB)

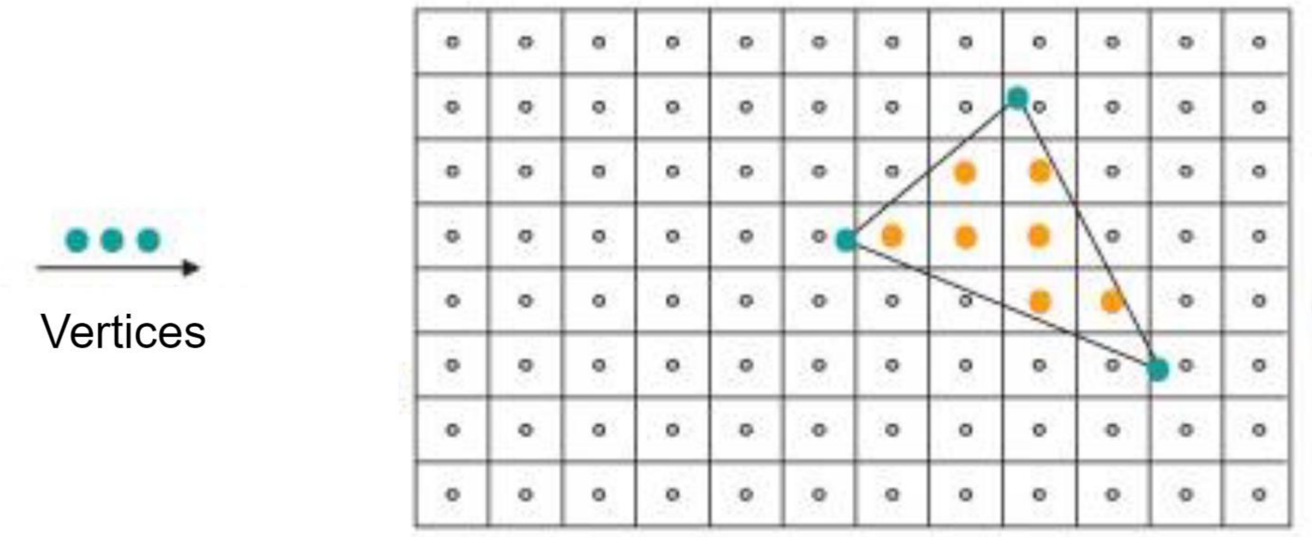


Rasterizare

Pornind de la o primitivă (punct, linie sau triunghi) se determină pixelii acestei primitive.

Rasterizare

- **Se determină pixelii din interiorul triunghiului**
- **Proprietățile vertecșilor sunt interpolate liniar**

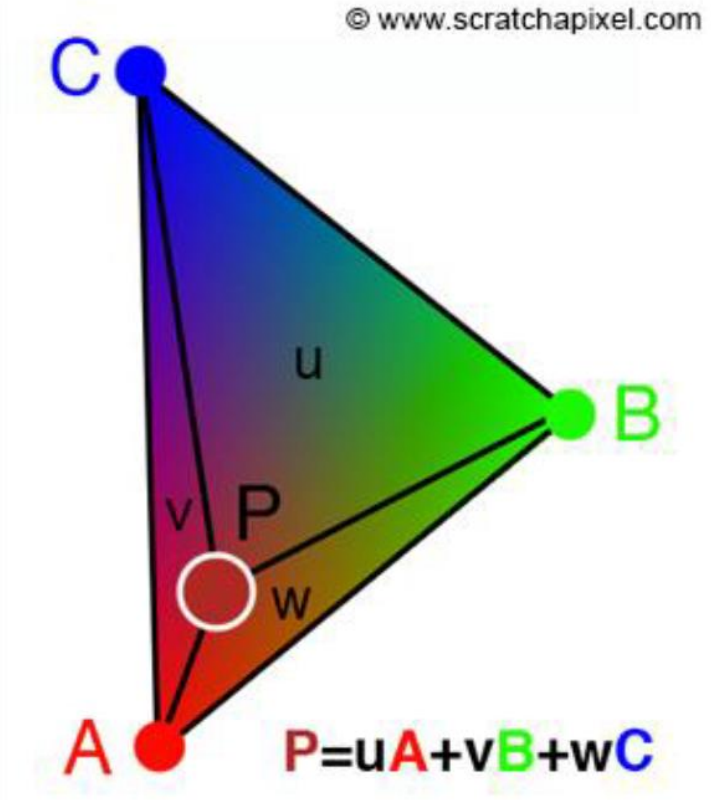


(Pentru fiecare fragment se calculează fie culoarea directă a pixelului, fie proprietăți prin care se poate calcula culoarea: coordinate de textură, normale..)

Interpolare folosind coordonate baricentrice

- **Combinație liniară a proprietăților vertecșilor**
  - ***Ex: culoare, coordonate de textură, normală, tangente.***

- **Ponderile sunt direct proporționale cu ariile triunghiurilor determinate de punctul P și laturile opuse ale triunghiului principal.**



OpenGL Pipeline

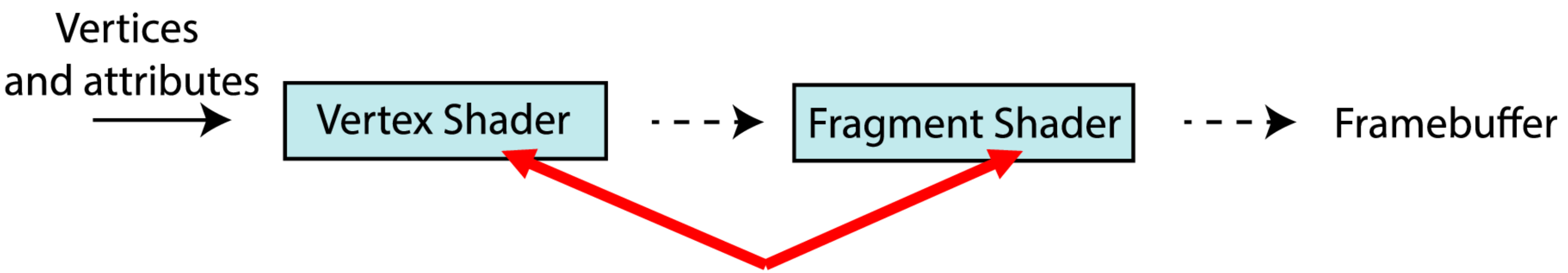
OpenGL Rendering Pipeline (simplified)

1. ***Vertex shader: geometric transformations***
2. ***Fragment shader: pixel-wise color computation***

World Coordinates

Pixel-wise attributes

RGBA image

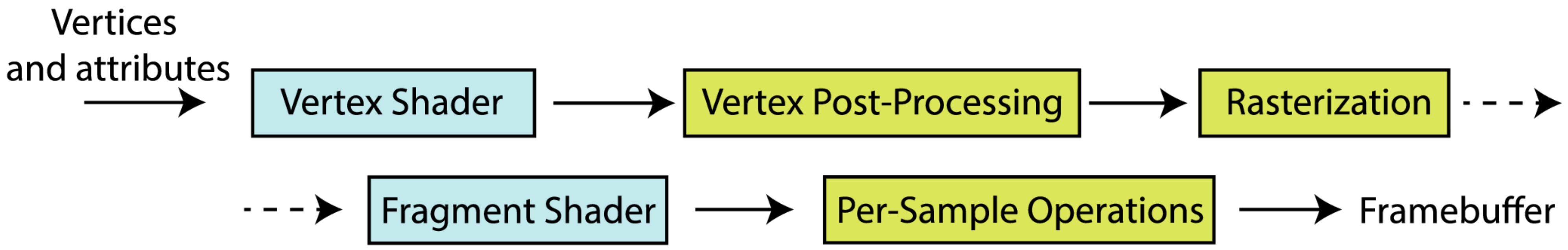


Shader: Programmable functions
to define object appearance locally
(vertex wise or fragment wise)

OpenGL Rendering Pipeline

Input:

- *3D vertex position*
- *Optional vertex attributes: color, texture coordinates,…*



Output:

- ***Frame Buffer :*** *GPU video memory, holds image for display*
- *RGBA pixel color (Red, Green, Blue, Alpha / opacity)*

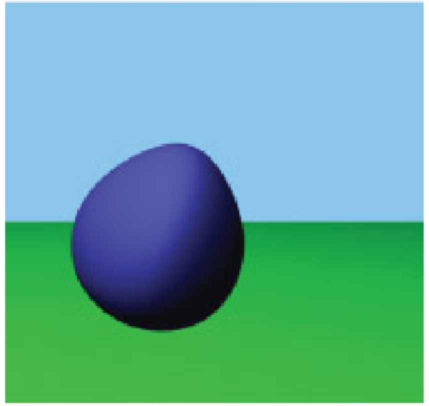
Vertex shader examples

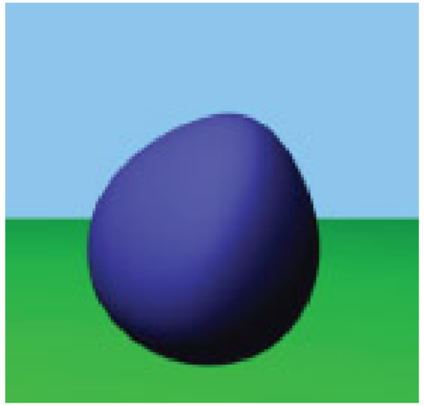
Object motion & transformation

- translation
- rotation
- scaling

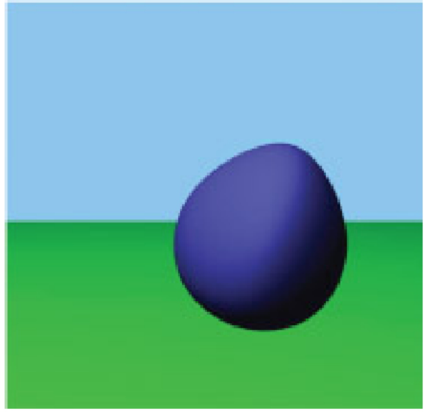
Projection

- Orthographic
  - *simple, without perspective effects*
- Perspective
  - *pinhole projection model*





Scaling



Translation

GLSL Vertex shader

The OpenGL Shading Language (GLSL)

- Syntax similar to the C programming language
- Build-in vector operations
- functionality as the GLM library our assignment template uses

x and y coordinates of a vec2, vec3 or vec4

world -> camera

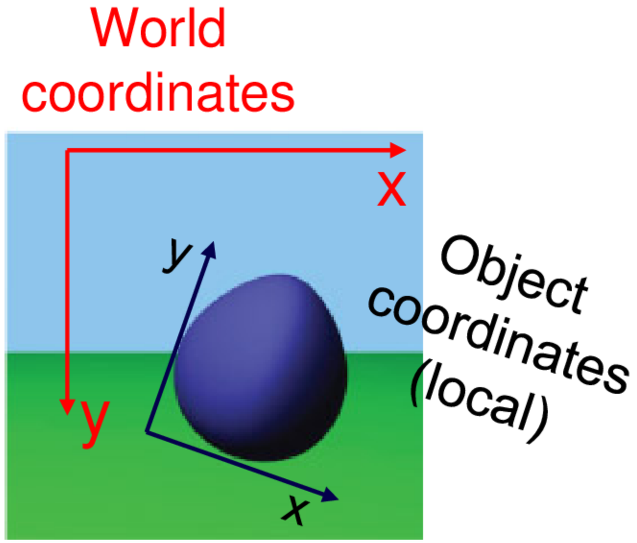
object -> world

```
void main()
{
    // Transforming The Vertex
    vec3 out_pos = projection * transform * vec3(in_pos.xy, 1.0);
    gl_Position = vec4(out_pos.xy, in_pos.z, 1.0);
}
```

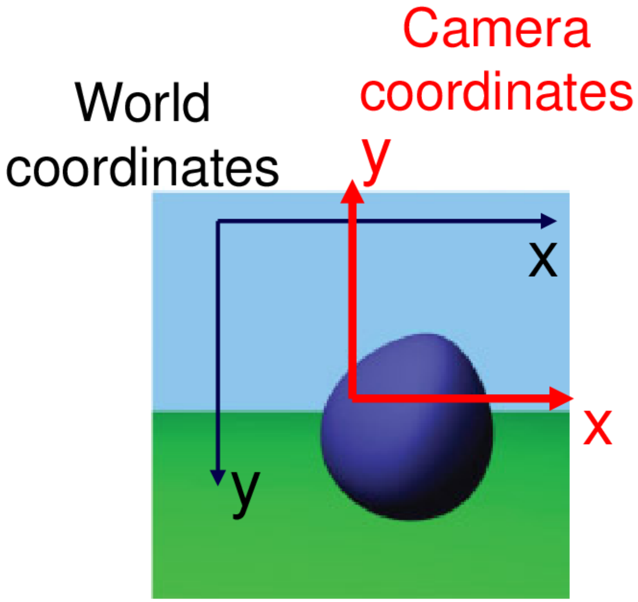
float (32 bit)

vector of 3 (vec3) and 4 (vec4) floats

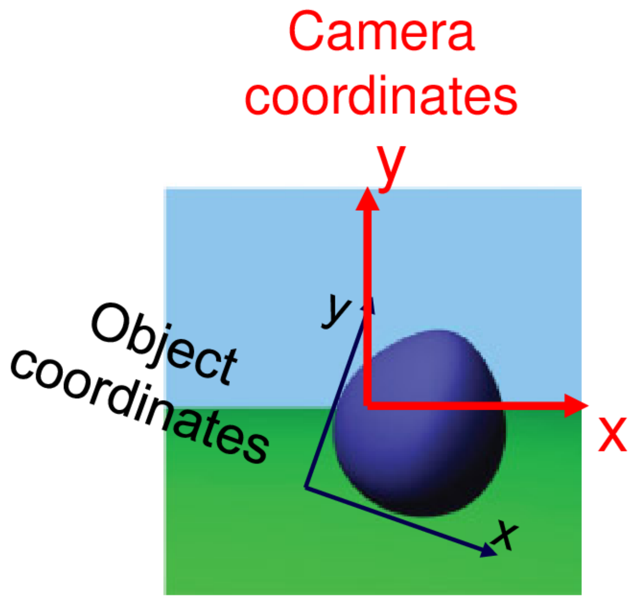
From local object to camera coordinates



object -> world



world -> camera

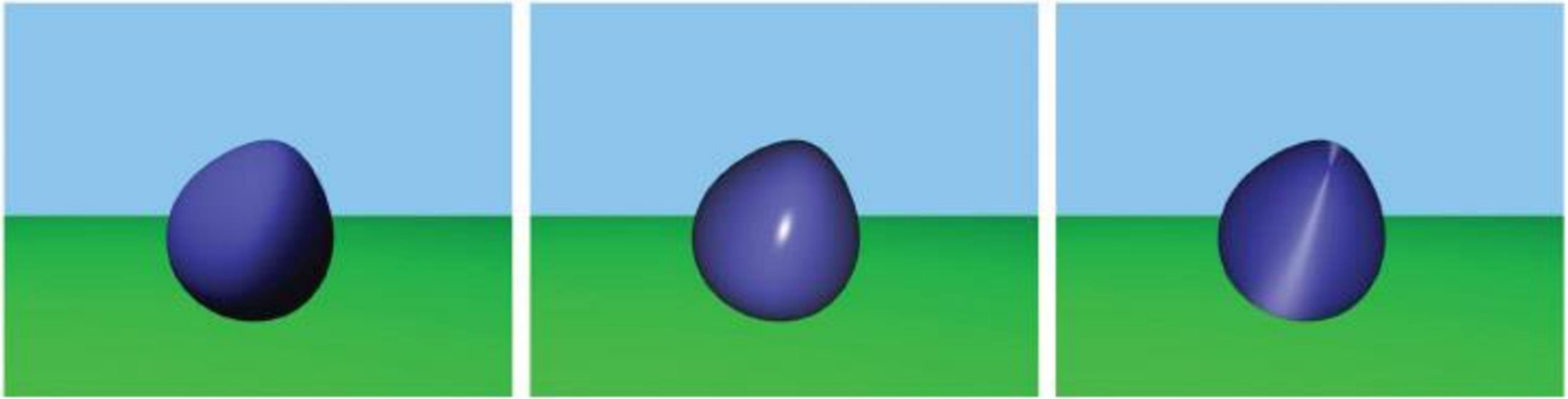


object -> camera

`transform` `projection` `projection * transform`

Fragment shader examples

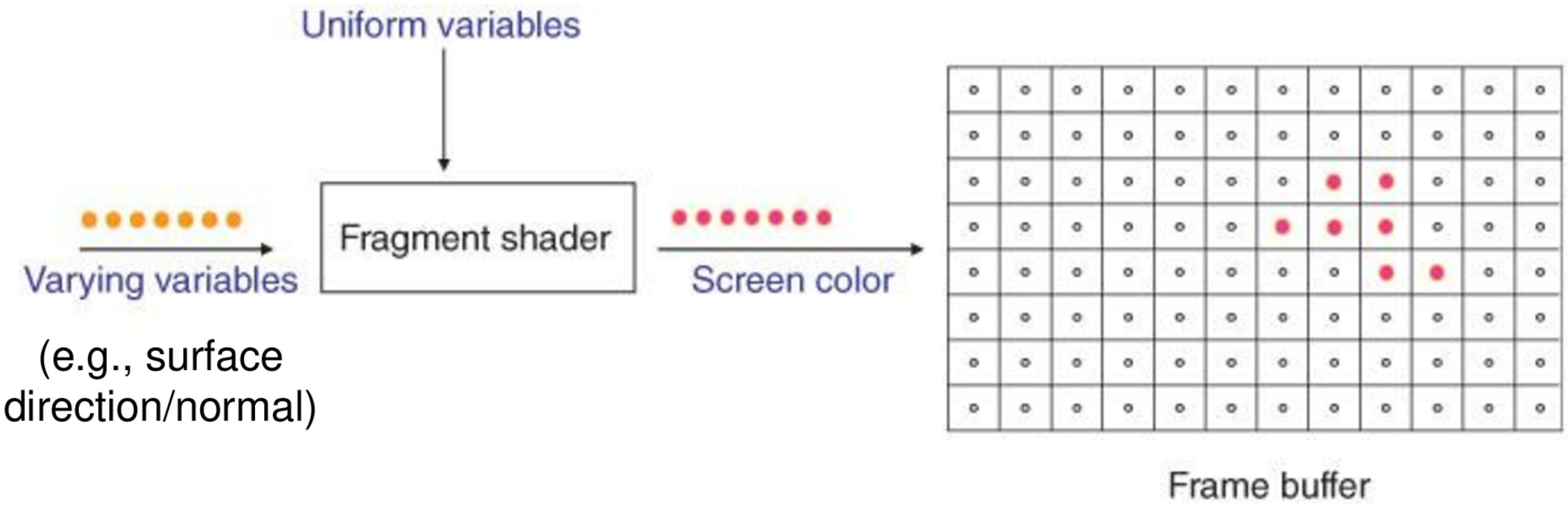
- *simulates materials and lights*
- *can read from textures*



Diffuse **Specular** **Directional**

Fragment shader overview

(e.g., light direction)



(e.g., surface direction/normal)

GLSL fragment shader examples

Minimal:

```
out vec4 out_color;
void main()
{
    // Setting Each Pixel To ???
    out_color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Specify color output

Red, Green, Blue, Alpha

Bibliografie/ Resurse

3dgep Understanding The View Matrix
https://www.3dgep.com/understanding-the-view-matrix/

3D Math Primer for Graphics and Game Development
https://gamemath.com/

Learn OpenGL
https://learnopengl.com/

CPSC 427 VIDEO GAME PROGRAMMING
https://www.cs.ubc.ca/~rhodin/2021_2022_CPSC_427/

FMI Unibuc Grafică pe Calculator