# Számítógépi Grafika Elméleti Kérdések és Válaszok

GEAGT131-B Tárgyhoz

2025. június 11.

## Tartalomjegyzék

# 1. Transzformációs Mátrixok 3D-ben

### Hogyan néz ki az z tengely körüli elforgatás mátrixa a háromdimenziós térben?

Homogén koordinátákban, $\theta$ az elforgatás szöge:

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

### Hogyan néz ki az y tengely körüli elforgatás mátrixa a háromdimenziós térben?

Homogén koordinátákban, $\theta$ az elforgatás szöge:

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

### Hogyan néz ki az x tengely körüli elforgatás mátrixa a háromdimenziós térben?

Homogén koordinátákban, $\theta$ az elforgatás szöge:

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

### Hogyan néz ki a háromdimenziós térben a skálázás transzformációs mátrixa?

Ahol $s_x, s_y, s_z$ a skálázási faktorok az egyes tengelyek mentén:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### Hogyan néz ki a háromdimenziós térben az eltolás transzformációs mátrixa?

Ahol $t_x, t_y, t_z$ az eltolás mértékei az egyes tengelyek mentén:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### Hogyan írható fel egy adott $\phi$ szög esetében síkban az origó körüli elforgatás mátrixa?

2D Descartes koordinátákban:

$$R(\phi) = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix}$$

2D homogén koordinátákban:

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

# 2. Koordináta Rendszerek és Vetítés

### Milyen előnyei miatt használunk homogén koordináta rendszert?

- **Egységes transzformációk:** Az eltolás, forgatás, skálázás és vetítés mind mátrixszorzással reprezentálható. Nem homogén koordinátákban az eltolás összeadás, a többi szorzás.
- **Perspektív vetítés:** Könnyen megvalósítható a homogén osztással.
- **Irányok és pontok megkülönböztetése:** Pontoknál a w komponens 1, irányvektoroknál 0. Ez biztosítja, hogy az irányvektorok ne tolódjanak el.
- **Több transzformáció összefűzése:** Egyszerűen a megfelelő mátrixok összeszorzásával egyetlen transzformációs mátrixba vonhatók össze.

### Hogyan lehet átírni egy Descartes koordinátarendszerbeli pont koordinátáit homogén koordinátarendszerbe?

Egy $(x, y, z)$ Descartes-koordinátájú pont homogén koordinátás megfelelője $(x, y, z, 1)$ vagy általánosabban $(x \cdot w, y \cdot w, z \cdot w, w)$, ahol $w \neq 0$. Leggyakrabban $w = 1$-et választunk. Síkbeli $(x, y)$ pont esetén $(x, y, 1)$.

### Hogyan írható át egy homogén koordinátarendszerbeli pont Descartes koordinátarendszerbe?

Egy $(x', y', z', w)$ homogén koordinátájú pont Descartes-megfelelője $(x'/w, y'/w, z'/w)$, feltéve, hogy $w \neq 0$.

### Miből tudhatjuk, hogy egy homogén koordinátarendszerbeli pontnak nincs Descartes koordinátarendszerbeli megfelelője?

Ha a homogén koordináta $w$ komponense 0. Ebben az esetben a pont egy végtelen távoli pontot vagy irányt jelöl, és nincs közvetlen Descartes-megfelelője (nem konvertálható vissza véges Descartes-ponttá az $x'/w$ stb. osztással).

# 3. Színkeverés

### Melyek az alapszínek additív és szubtraktív színkeverés esetén?

- **Additív színkeverés:** Az alapszínek a Piros (Red), Zöld (Green), Kék (Blue) - RGB. Fények keverésekor használatos, pl. monitorok, projektorok. Az alapszínek különböző intenzitású kombinációjával kapjuk a többi színt. Az alapszínek maximális intenzitású összege fehér fényt ad. Ha egyik alapszín sem világít, feketét kapunk.
- **Szubtraktív színkeverés:** Az alapszínek a Cián (Cyan), Magenta (Magenta), Sárga (Yellow) - CMY. Festékek, pigmentek keverésekor használatos, pl. nyomtatás. A fehér fényből bizonyos hullámhosszakat elnyelnek (szubtrahálnak), a maradékot verik vissza, ezt látjuk színként. Az alapszínek összege elvileg fekete (minden fényt elnyel), de a gyakorlatban inkább sötétbarna, ezért gyakran K (Key/Black) komponenssel egészítik ki (CMYK) a jobb fekete szín és a festéktakarékosság miatt.

# 4. Raszteres Grafika

### Írja le, hogy hogyan lehet egy szakaszt megjeleníteni raszteres grafikus megjelenítésnél!

Egy szakaszt raszteres megjelenítésnél (pl. képernyőn) diszkrét képpontok (pixelek) sorozataként közelítünk. Adott a szakasz két végpontjának koordinátája. Az algoritmus feladata, hogy meghatározza, mely pixeleket kell „bekapcsolni" (megszínezni) ahhoz, hogy a szakasz vizuálisan megjelenjen. Népszerű algoritmusok erre a célra:

- **DDA (Digital Differential Analyzer):** Inkrementális algoritmus, a szakasz meredekségét használja a következő pixel kiválasztásához. Lebegőpontos számításokat igényelhet, ami lassabb lehet.
- **Bresenham-féle vonal algoritmus:** Hatékonyabb, mivel csak egész aritmetikát használ. Egy döntési paraméter segítségével választja ki a két lehetséges következő pixel közül azt, amelyik közelebb esik az ideális vonalhoz.

Mindkét algoritmus a szakasz egyik végpontjából indulva lépésenként halad a másik felé, és minden lépésben kiválaszt egy pixelt.

### Írja le, hogy hogyan lehet egy körvonalat megjeleníteni raszteres grafikus megjelenítésnél!

Hasonlóan a szakaszhoz, a körvonalat is pixelek sorozatával közelítjük. Népszerű algoritmusok:

- **Középponti kör algoritmus (Midpoint Circle Algorithm) / Bresenham-féle kör algoritmus:** Nagyon hatékony, mivel csak egész aritmetikát használ és kihasználja a kör nyolcszoros szimmetriáját. Elég a körív egy oktánsát (45 fokos szeletét) kiszámítani, a többi pont a szimmetria alapján megkapható. Iteratívan, egy döntési paraméter segítségével választja ki a következő pixelt.

Az algoritmus a középpont és a sugár alapján számítja ki a körvonalhoz legközelebb eső pixeleket.

# 5. 3D Modellezés (Wavefront OBJ)

### Milyen módon tárolja a Wavefront OBJ szabványa a modellekhez tartozó adatokat?

A Wavefront OBJ (.obj) egy egyszerű, szöveges ASCII formátum 3D geometria leírására. Nem tárol színeket vagy anyagjellemzőket közvetlenül (ezeket külön .mtl fájlban szokás definiálni), hanem elsősorban a modell alakját írja le. A legfontosabb elemek (sorok prefixumai):

- `v x y z [w]`: Csúcspont (vertex) koordinátái. Az $x, y, z$ a Descartes-koordináták, a $w$ opcionális, homogén koordinátákhoz (alapértelmezetten 1).
- `vt u v [w]`: Textúrakoordináták (texture vertex). Az $u, v$ (és opcionálisan $w$) a textúra leképezéséhez használt koordináták, általában [0,1] tartományban.
- `vn i j k`: Normálvektor (vertex normal) komponensei. Ezek a vektorok a megvilágítás számításához szükségesek.
- `f v1[/vt1[/vn1]] v2[/vt2[/vn2]] v3[/vt3[/vn3]] ...`: Lap (face) definíció. A csúcspontokat indexekkel adja meg. Az indexelés 1-től kezdődik. Minden csúcspont indexe után opcionálisan megadható a hozzá tartozó textúrakoordináta-index és normálvektor-index, perjelekkel elválasztva (pl. `f 1/1/1 2/2/1 3/3/1`). Egy lap több mint három csúcspontból is állhat (poligon), de ezeket a renderelők általában háromszögekre bontják.

- `# comment`: Komment sor.
- `mtllib [fájlnév]`: Külső anyagdefiníciós fájl (.mtl) betöltése.
- `usemtl [anyagnév]`: Az .mtl fájlban definiált anyag használata a következő lapokhoz.

### Miért előnyös, hogy ha a normálvektorok le vannak tárolva a modellfájlban?

A normálvektorok elengedhetetlenek a realisztikus megvilágítás és árnyalás számításához. Ha előre kiszámítva és tárolva vannak a modellfájlban (pl. OBJ fájlban `vn` sorokként):

- **Teljesítmény:** Nem kell minden egyes rendereléskor újra számítani őket a geometria alapján (ami különösen komplex modelleknél időigényes lehet).
- **Sima árnyalás (Smooth Shading):** Lehetővé teszi a csúcspontonkénti normálvektorok használatát (vertex normals). Ezeket gyakran a szomszédos lapok normálvektorainak átlagolásával kapják, ami simább átmenetet biztosít a lapok között, elkerülve a "lapos" (flat shaded) megjelenést.
- **Művészi kontroll:** A modellező szoftverekben a normálvektorok manuálisan is módosíthatók, így a művészek speciális vizuális effekteket érhetnek el (pl. élek keményítése vagy lágyítása anélkül, hogy a geometriát megváltoztatnák).
- **Konzisztencia:** Biztosítja, hogy a modell mindenhol ugyanúgy nézzen ki, függetlenül attól, hogy a renderelő program hogyan számítaná a normálvektorokat.

## 6. OpenGL Alapok

### Milyen grafikus alapelemeket képes megjeleníteni az OpenGL?

Az OpenGL alapvetően egyszerű geometriai primitíveket (geometric primitives) képes megjeleníteni, amelyeket csúcspontok (vertexek) sorozatával definiálunk. Ezek:

- `GL_POINTS`: Egyedi pontok.
- `GL_LINES`: Szakaszok (minden két csúcspont egy szakaszt alkot).
- `GL_LINE_STRIP`: Töröttvonal (az egymást követő csúcsok szakaszt alkotnak).
- `GL_LINE_LOOP`: Zárt töröttvonal (mint a `GL_LINE_STRIP`, de az utolsó csúcsot összeköti az elsővel).
- `GL_TRIANGLES`: Háromszögek (minden három csúcspont egy háromszöget alkot).
- `GL_TRIANGLE_STRIP`: Háromszög-sáv (minden új csúcs az előző két csúccsal alkot egy új háromszöget).
- `GL_TRIANGLE_FAN`: Háromszög-legyező (az első csúcs közös, minden új csúcs az elsővel és az előző csúccsal alkot egy új háromszöget).

Régebbi OpenGL verziók támogattak más primitíveket is (pl. `GL_QUADS`, `GL_POLYGON`), de ezeket a modern OpenGL (core profile) már nem preferálja; helyettük háromszögeket használnak az összetettebb formák leírására.

### Írja le egy lehetséges módszerét egy kúp/gömb/henger megjelenítésének OpenGL-ben!

Ezeket az alakzatokat általában háromszöghálóval (triangle mesh) közelítik, majd ezeket a háromszögeket rajzolják ki OpenGL primitívekkel (pl. `GL_TRIANGLES` vagy `GL_TRIANGLE_STRIP`).

- **Kúp:**
  1. Az alapkört szegmensekre (ívekre) bontjuk. Minden ív végpontja egy csúcs az alapkörön.
  2. Ezeket az alapköri csúcsokat összekötjük a kúp csúcsával (apex), így háromszögeket alkotva a palástón.
  3. Az alapot is háromszögekre lehet bontani, pl. az alapköri csúcsokat egy középső ponttal összekötve.
- **Gömb:**
  1. Paraméteres egyenleteit (pl. gömbi koordináták alapján: $\phi$ és $\theta$ szögek) használva generálunk csúcspontokat a gömbfelületen. Képzeletben "szélességi" és "hosszúsági" körökre osztjuk.
  2. Az így kapott csúcspontokat összekötve kis négyszögeket (amiket két háromszögre bontunk) vagy közvetlenül háromszögeket hozunk létre a gömb felszínén.
  3. Alternatívaként, egy ikozaéder (20 oldalú szabályos test) rekurzív finomításával is közelíthető a gömb (minden háromszöget tovább bontunk kisebb háromszögekre, és az új csúcsokat "kitoljuk" a gömbsugárra).
- **Henger:**
  1. A két alapkört (alsó és felső) szegmensekre bontjuk, hasonlóan a kúphoz.
  2. A palástot az alsó és felső kör megfelelő csúcsait összekötő négyszögekkel (melyek két háromszögre bonthatók) fedjük le.
  3. Az alaplapokat és fedlapokat is háromszögekre bontjuk (pl. egy középső pontból kiindulva, mint a kúp alapját).

Ezekhez aGLUT (OpenGL Utility Toolkit) könyvtár régebben tartalmazott kész függvényeket (pl. `glutSolidSphere`, `glutSolidCone`, `glutSolidCylinder`), de modern OpenGL-ben ezeket általában manuálisan kell implementálni vagy külső geometriai könyvtárakat használni.

## 7. OpenGL Megvilágítás

### A fényeket az OpenGL milyen alapvető összetevőkre bontja?

Az OpenGL (a Phong-féle vagy Blinn-Phong megvilágítási modellek alapján) a fényhatásokat tipikusan három fő összetevőre bontja a felületeken:

1. **Ambient (környezeti) fény:** Ez egyfajta általános, szórt megvilágítás, amely a jelenet minden felületét egyenletesen ér, függetlenül a fényforrások helyzetétől vagy a felület orientációjától. Azt a fényt szimulálja, ami a környezetből minden irányból visszaverődve éri a tárgyakat.
2. **Diffuse (szórt) fény:** Ez a fényforrásból érkező fény azon része, amely egy matt felületről minden irányba egyenletesen verődik vissza. Intenzitása függ a fényforrás irányától a felületi normálishoz képest (Lambert-féle koszinusz törvény).

3. **Specular (tükröződő) fény:** Ez a fényforrásból érkező fény azon része, amely egy fényes felületről egy preferált irányba verődik vissza erősebben, csillanásokat (highlights) okozva. Intenzitása függ a fényforrás irányától, a felületi normálistól és a nézőponttól.

Esetenként egy negyedik komponenst, az **emisszív (kibocsátott) fényt** is figyelembe veszik, ami azt modellezi, hogy a felület maga bocsát ki fényt (pl. egy lámpaernyő). A végső szín általában ezen komponensek összege.

### Röviden mutassa be OpenGL esetében a környezeti (ambient) fényösszetevőt!

A környezeti (ambient) fényösszetevő a megvilágítási modell legegyszerűbb része. Azt a fényt szimulálja, ami nem direkt fényforrásból származik, hanem a környezetből szórtan, minden irányból éri a felületet. Ez biztosítja, hogy az árnyékban lévő részek se legyenek teljesen feketék. Kiszámítása: $I_{ambient} = \text{globalAmbientLightColor} \times \text{materialAmbientColor}$. Ahol:

- globalAmbientLightColor: A jelenet általános környezeti fényének színe és intenzitása.
- materialAmbientColor: A felület anyagának környezeti fényvisszaverő képessége (színe).

Ez az összetevő nem függ sem a fényforrás helyzetétől, sem a felületi normálvektor irányától.

### Röviden mutassa be OpenGL esetében a szórt (diffuse) fényösszetevőt!

A szórt (diffúz) fényösszetevő azt a jelenséget modellezi, amikor a fény egy matt (nem tükröződő) felületről minden irányba egyenletesen verődik vissza. A visszavert fény erőssége attól függ, hogy a fény milyen szögben éri a felületet. Ezt a Lambert-féle koszinusz törvény írja le. Kiszámítása egy fényforrásra: $I_{diffuse} = \text{lightColor} \times \text{materialDiffuseColor} \times \max(0, \mathbf{N} \cdot \mathbf{L})$. Ahol:

- lightColor: A fényforrás színe és intenzitása.
- materialDiffuseColor: A felület anyagának diffúz fényvisszaverő képessége (színe).
- $\mathbf{N}$: A felületi normálvektor (egységvektor).
- $\mathbf{L}$: A pontból a fényforrás felé mutató irányvektor (egységvektor).
- $\mathbf{N} \cdot \mathbf{L}$: A két vektor skaláris szorzata, ami a köztük lévő szög koszinuszával arányos. A $\max(0, ...)$ biztosítja, hogy ha a felület elfordul a fénytől (a szög $> 90°$), ne legyen negatív a fényerő.

### Röviden mutassa be OpenGL esetében a tükröződő (specular) fényösszetevőt!

A tükröződő (spekuláris) fényösszetevő a fényes felületeken megjelenő csillanásokat (highlights) modellezi. Azt a fényt jelenti, ami a fényforrásból a felületről visszaverődve a néző szemébe jut. A visszaverődés iránya a beesési szög tükrözése a felületi normálvektorra (mint egy tükörnél). A csillanás akkor a legerősebb, ha a nézőpont iránya közel esik ehhez a tükröződési irányhoz. Kiszámítása egy fényforrásra (Phong-modell szerint): $I_{specular} = \text{lightColor} \times \text{materialSpecularColor} \times (\max(0, \mathbf{R} \cdot \mathbf{V}))^{\text{shininess}}$. Ahol:

- lightColor: A fényforrás színe és intenzitása.
- materialSpecularColor: A felület anyagának spekuláris fényvisszaverő képessége (színe).
- $\mathbf{R}$: A fényvisszaverődés irányvektora ($\mathbf{R} = 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}$).
- $\mathbf{V}$: A pontból a nézőpont (kamera) felé mutató irányvektor (egységvektor).
- shininess: A "fényesség" (vagy spekuláris kitevő) egy szám, ami a csillanás méretét/élességét szabályozza. Nagyobb érték kisebb, élesebb csillanást eredményez.

### Mi az alapvető különbség a szórt (diffúz) és a tükröződő (spekuláris) fény között?

- **Visszaverődés jellege:**
  - **Szórt (diffúz):** A fény a felületről minden irányba közel egyenletesen verődik vissza. Tipikus matt felületekre jellemző.
  - **Tükröződő (spekuláris):** A fény a felületről egy preferált irányba (a beesési szög tükörképe a normálisra) verődik vissza erősebben. Tipikus fényes, sima felületekre jellemző (csillanások).
- **Nézőpontfüggés:**
  - **Szórt (diffúz):** A felület fényessége (az adott pontban) nem függ a nézőponttól (csak a fényforrás irányától a felületi normálishoz képest).
  - **Tükröződő (spekuláris):** A csillanás erőssége és láthatósága nagymértékben függ a nézőponttól. Csak akkor látszik erősnek, ha a néző iránya közel esik a tükröződés fő irányához.
- **Felületi tulajdonság:**
  - **Szórt (diffúz):** A felület alapszínét adja meg.
  - **Tükröződő (spekuláris):** A felület csillogásának színét adja meg (gyakran fehér vagy a fényforrás színe).

## 8. OpenGL Láthatóság és Renderelési Technikák

### OpenGL esetében mit nevezünk láthatósági problémának?

A láthatósági probléma (vagy rejtett felületek eltávolításának problémája - Hidden Surface Removal, HSR) a 3D számítógépi grafikában annak a feladatnak a neve, hogy egy adott nézőpontból mely felületek, vagy felületrészek láthatók, és melyek vannak takarásban más felületek által. Ha ezt a problémát nem kezeljük, akkor a távolabbi objektumok hibásan "átrajzolódhatnak" a közelebbieken, ami vizuálisan helytelen képet eredményez.

### Mire szolgál a mélységbuffer és hogyan működik?

A mélységbuffer (depth buffer vagy z-buffer) egy elterjedt megoldás a láthatósági probléma kezelésére. **Mire szolgál:** Annak biztosítására, hogy a közelebbi objektumok takarják a távolabbiakat a 2D képen, azaz a helyes mélységi sorrend megjelenítésére. **Hogyan működik:**

1. A mélységbuffer egy extra képernyőméretű puffer, amely minden pixelhez tárol egy mélységértéket (általában a nézőponttól mért távolságot vagy normalizált z-koordinátát).
2. Kezdetben a mélységbuffer minden elemét egy nagyon nagy értékre (a lehető legtávolabbi pont) inicializálják. A színpuffer (frame buffer) pedig a háttérszínre.
3. Amikor egy új fragmentumot (potenciális pixelt) kellene rajzolni egy adott $(x, y)$ pozícióra, kiszámítjuk annak mélységértékét ($z_{new}$).
4. Ezt a $z_{new}$ értéket összehasonlítjuk a mélységbufferben az $(x, y)$ pozíción tárolt $z_{old}$ értékkel.
5. Ha $z_{new} < z_{old}$ (az új fragmentum közelebb van a nézőponthoz), akkor:
   - Az új fragmentum színét beírjuk a színpuffer $(x, y)$ pozíciójára.

- A $z_{new}$ értéket beírjuk a mélységbuffer $(x, y)$ pozíciójára ($z_{old} = z_{new}$).

6. Ha $z_{new} \geq z_{old}$ (az új fragmentum távolabb van vagy ugyanott), akkor az új fragmentum takarásban van, és eldobódik (nem írja felül sem a szín-, sem a mélységpuffert).

Ez a módszer pixelenként működik, és független a poligonok rajzolási sorrendjétől (bár a hatékonyság növelhető, ha a közelebbi objektumokat rajzoljuk előbb).

### Mire szolgál a hátsó lapok eldobása, és hogyan működik a módszer?

A hátsó lapok eldobása (back-face culling) egy optimalizációs technika, amely csökkenti a renderelendő geometria mennyiségét. **Mire szolgál:** Zárt, konvex (vagy általában nem átlátszó) testek esetén azok a lapok, amelyek a nézőponttól elfelé néznek (a "hátoldaluk" látszana), nem lehetnek láthatók. Ezeknek a lapoknak a feldolgozása felesleges, így eldobásukkal gyorsítható a renderelés. **Hogyan működik:**

1. Minden laphoz tartozik egy normálvektor, ami a lap "külső" oldalára merőleges.

2. Kiszámítjuk a lap normálvektorának ($\mathbf{N}$) és a nézőpontból a lap felé mutató vektornak ($\mathbf{V}$) (vagy a kamera nézési irányának) a skaláris szorzatát.

3. Ha $\mathbf{N} \cdot \mathbf{V} > 0$ (OpenGL konvenciók szerint, ahol a kamera a -Z irányba néz, ez azt jelenti, hogy a normálvektor Z komponense a kamera koordináta-rendszerben pozitív), akkor a lap a nézőtől elfelé fordul, tehát "hátsó" lap.

4. A hátsó lapokat nem küldjük tovább a renderelési futószalag további lépéseire (pl. nem kerülnek raszterizálásra).

Fontos a csúcspontok sorrendje (winding order, pl. óramutató járásával ellentétes) a lapok definiálásakor, mert ez határozza meg a normálvektor irányát.

### Miért van szükség dupla bufferelésre? Hogyan működik?

A dupla bufferelés (double buffering) egy technika a vizuális hibák, mint a villogás (flickering) vagy a képszakadozás (tearing) elkerülésére, amelyek akkor jelentkezhetnek, ha egyetlen puffert használnak a kép megjelenítésére és annak frissítésére egyidejűleg. **Mire szolgál:** Folyamatos, zökkenőmentes animáció és képfrissítés biztosítása. **Hogyan működik:**

1. Két puffert (memóriaterületet) használnak a kép tárolására:

   - **Front buffer (elülső puffer):** Ennek a puffernek a tartalma jelenik meg éppen a képernyőn.
   - **Back buffer (hátsó puffer):** Ebbe a pufferbe történik a következő képkocka kirajzolása, rejtve a felhasználó elől. A rajzolási műveletek (pl. törlés, objektumok rajzolása) itt zajlanak.

2. Amikor a hátsó pufferbe a teljes képkocka kirajzolása befejeződött, a két puffer szerepét felcserélik (ezt nevezik buffer swap-nek vagy page flipping-nek).

3. Az eddigi hátsó puffer lesz az új elülső puffer (tartalma megjelenik a képernyőn), az eddigi elülső puffer pedig az új hátsó puffer (ebbe kezdődik a következő képkocka rajzolása).

Ez a csere nagyon gyorsan (gyakran a képernyő vertikális visszafutási szinkronjához igazítva, VSync) történik, így a felhasználó mindig csak a teljesen elkészült képkockákat látja, elkerülve a félkész állapotok megjelenítését.

### Hasonlítsa össze röviden a Gourand és a Phong árnyalási módot!

Mind a Gouraud, mind a Phong árnyalás interpolációs módszer, céljuk a poligonális modellek simább, kevésbé szögletes megjelenítése, mint a flat shading (lapos árnyalás).

- **Gourand árnyalás (Gouraud Shading):**
  - **Működés:** Kiszámítja a megvilágítást (színt) minden egyes csúcspontban a csúcsponti normálvektor alapján. Ezután ezeket a csúcsponti színeket lineárisan interpolálja a poligon (háromszög) belsejében lévő pixelekhez.
  - **Előnyök:** Viszonylag gyors, mivel a költséges megvilágítási számításokat csak a csúcspontokban kell elvégezni.
  - **Hátrányok:** A spekuláris csillanásokat nem mindig kezeli jól. Ha egy csillanás a poligon belsejébe esne, de egyik csúcsra sem, akkor vagy nem jelenik meg, vagy "szétkenődik" a poligonon. Mach-sávok (Mach bands) jelenhetnek meg, ahol az intenzitásváltozás éles.
- **Phong árnyalás (Phong Shading):**
  - **Működés:** Nem a színeket, hanem a csúcsponti normálvektorokat interpolálja lineárisan a poligon belsejében lévő minden egyes pixelhez. Ezután minden pixelre külön-külön elvégzi a teljes megvilágítási számítást az interpolált normálvektor alapján.
  - **Előnyök:** Sokkal jobb minőségű spekuláris csillanásokat eredményez, és általában simább, realisztikusabb megjelenést biztosít, mint a Gouraud árnyalás.
  - **Hátrányok:** Jelentősen számításigényesebb, mivel a megvilágítási számítást pixelenként kell elvégezni.

Összefoglalva: Gouraud gyorsabb, de kevésbé pontos (főleg csillanásoknál); Phong lassabb, de szebb és pontosabb.

### Mit nevezünk festő algoritmusnak? Milyen problémát és hogyan old meg?

**Festő algoritmus (Painter's Algorithm):** Egy korai, egyszerű algoritmus a rejtett felületek eltávolításának problémájára (láthatósági probléma). **Milyen problémát old meg:** Azt a problémát igyekszik megoldani, hogy a 3D jelenet objektumai helyes mélységi sorrendben jelenjenek meg a 2D képen, azaz a közelebbi objektumok takarják a távolabbiakat. **Hogyan oldja meg (elvileg):**

1. A jelenet összes poligonját (vagy objektumát) mélység szerint rendezi, a nézőponttól legtávolabbitól a legközelebbiig.
2. Ebben a "hátulról előre" sorrendben rajzolja ki a poligonokat a képernyőre.

Az elgondolás az, hogy a később (közelebb) rajzolt poligonok felülírják a korábban (távolabb) rajzoltakat, így szimulálva a takarást, ahogy egy festő is a távolabbi részeket festi le először, majd azokat részben vagy egészben lefesti a közelebbi elemekkel. **Problémái:**

- **Ciklikus átfedések:** Nem tudja kezelni az olyan helyzeteket, ahol poligonok kölcsönösen takarják egymást (pl. A takarja B-t, B takarja C-t, és C takarja A-t).
- **Áthatoló poligonok:** Problémás, ha poligonok áthatolnak egymáson.
- **Hatékonyság:** A poligonok mélységi rendezése számításigényes lehet, különösen nagyszámú poligon esetén.

- **Egyértelmű sorrend:** Nem mindig triviális egyértelmű mélységi sorrendet felállítani (pl. egy nagy poligon, ami részben egy másik előtt és mögött is van).

Ezen problémák miatt a festő algoritmust a gyakorlatban felváltották a robusztusabb és hatékonyabb módszerek, mint például a Z-buffer algoritmus.

# 9. Programozás és Függvénykönyvtárak

## Mire szolgálnak a statikus függvénykönyvtárak (static library)?

A statikus függvénykönyvtárak (.lib Windows alatt, .a Linux/macOS alatt) előre lefordított objektumkódokat (függvényeket, adatokat) tartalmazó archív fájlok. **Mire szolgálnak:**

- **Kód újrafelhasználása:** Lehetővé teszik gyakran használt függvények és rutinok összegyűjtését és könnyű beillesztését több különböző programba anélkül, hogy a forráskódot újra és újra le kellene fordítani.
- **Moduláris fejlesztés:** Segítik a szoftverek moduláris felépítését, ahol a különböző funkcionalitások külön könyvtárakba szervezhetők.
- **Linkelés:** A statikus könyvtárak tartalmát a linker a fordítási folyamat végén (linkelési fázis) közvetlenül beépíti a végrehajtható (.exe) fájlba. Csak azok a részek kerülnek be, amelyeket a program ténylegesen használ.

**Jellemzői:**

- A program futtatásához nincs szükség a külső könyvtárfájlra, mivel a kód már a program része.
- Nagyobb lehet a végrehajtható fájl mérete, mivel a könyvtári kódot tartalmazza.
- Ha a könyvtárban hiba van, vagy frissül, minden azt használó programot újra kell linkelni a javított/frissített könyvtárral.

## Mi a különbség a statikus és a dinamikus linkelés között?

A linkelés az a folyamat, amely során a lefordított objektumkódokat (a program saját részei és a használt könyvtárak kódjai) összekapcsolják egyetlen futtatható programmá.

- **Statikus linkelés:**
  - **Mikor történik:** A fordítási időben (pontosabban a linkelési fázisban).
  - **Hogyan működik:** A linker bemásolja a szükséges kódrészleteket a statikus könyvtárakból (.lib, .a) közvetlenül a végrehajtható fájlba.
  - **Előnyök:** A program futtatásához nincs szükség külső könyvtárfájlokra (a könyvtár részei beépülnek), így egyszerűbb a telepítés és kisebb a "DLL hell" esélye.
  - **Hátrányok:** A végrehajtható fájl mérete nagyobb lesz. Ha több program ugyanazt a statikus könyvtárat használja, mindegyik tartalmazni fogja annak egy példányát a memóriában futás közben. A könyvtár frissítése esetén minden programot újra kell linkelni.
- **Dinamikus linkelés:**
  - **Mikor történik:** A program futásidejében.

- **Hogyan működik:** A végrehajtható fájl csak hivatkozásokat tartalmaz a dinamikus könyvtárakban (.dll Windows, .so Linux, .dylib macOS) lévő függvényekre. A tényleges kód a program indításakor vagy szükség esetén töltődik be a memóriába.
- **Előnyök:** Kisebb végrehajtható fájlméret. A könyvtárak megoszthatók a memóriában több futó program között, így csökkentve a teljes memóriahasználatot. A könyvtárak frissíthetők anélkül, hogy a programokat újra kellene fordítani/linkelni (elég a .dll/.so fájlt cserélni).
- **Hátrányok:** A program futtatásához szükség van a megfelelő verziójú dinamikus könyvtárfájlokra a rendszerben ("DLL hell" probléma). Kis mértékű indítási vagy futásidejű többletterhelés a könyvtárak betöltése és a hivatkozások feloldása miatt.

### Milyen problémák megoldásához használjuk az SDL2-őt?

Az SDL2 (Simple DirectMedia Layer 2) egy cross-platform, alacsony szintű multimédiás programozási könyvtár. Elsősorban a következő típusú problémák megoldásához használják:

- **Ablakkezelés:** Grafikus ablakok létrehozása és kezelése különböző operációs rendszereken.
- **Grafikus renderelés:** Hozzáférés biztosítása a hardveresen gyorsított 2D és 3D grafikához (pl. OpenGL, Direct3D, Vulkan, Metal kontextusok létrehozása és kezelése), valamint saját, egyszerű 2D renderelési API biztosítása.
- **Input kezelés:** Billentyűzet, egér, joystick, gamepad és érintőképernyős események egységes kezelése.
- **Hangkezelés:** Hanglejátszás és -felvétel.
- **Időzítés:** Időzítők és a program futásának időbeli vezérlése.
- **Fájlkezelés:** Platformfüggetlen fájl I/O műveletek.
- **Szálkezelés:** Alapvető többszálú programozási primitívek.

Főként játékfejlesztésben, emulátorokban és multimédiás alkalmazásokban népszerű, ahol szükség van a hardverhez való közvetlen, de platformfüggetlen hozzáférésre.

### Egy C programkód esetében célszerűen mi kerül a forrás és a fejléc (header) állományokba?

A C programozásban a kód modularitásának és szervezettségének érdekében a programot általában több fájlra bontják:

- **Fejléc (.h) állomány (Header file):**
  - **Deklarációkat** tartalmaz. Ezek leírják a program interfészeit, de nem tartalmazzák a tényleges implementációt.
  - Tipikus tartalma:
    * Függvény prototípusok (függvény neve, paraméterei, visszatérési típusa).
    * Struktúra (`struct`), unió (textttunion), enumeráció (`enum`) definíciók.
    * Típusdefiníciók (`typedef`).
    * Globális változók deklarációi (`extern` kulcsszóval).
    * Makró definíciók (`#define`).

  - Célja, hogy más forrásfájlok ( `.c`) is „lássák” és használhassák az itt deklarált elemeket anélkül, hogy a teljes implementációt ismerniük kellene, illetve a fordító ellenőrizni tudja a helyes használatot. Általában `#include` direktívával vonják be más `.h` vagy `.c` fájlokba.

- **Forrás (`.c`) állomány (Source file):**
  - **Definíciókat/Implementációkat** tartalmaz. Itt található a függvények tényleges kódja és a globális/statikus változók tényleges létrehozása.
  - Tipikus tartalma:
    * Függvények teljes implementációja.
    * Globális és statikus változók definíciói és inicializálásai.
  - A forrásfájlokat fordítja le a fordító objektumkóddá.

Ez a felosztás segíti a kód újrafelhasználását, csökkenti a fordítási időt (csak a megváltozott .c fájlokat kell újrafordítani), és átláthatóbbá teszi a projekt szerkezetét.

### Miért szerepeltetünk a függvények többségénél első paraméterként egy struktúra mutatót? Mi ennek a megfelelője más nyelvekben?

Ez a technika C nyelvben az objektum-orientált programozás (OOP) bizonyos aspektusainak "szimulálására" szolgál, ahol nincsenek osztályok és metódusok a nyelvbe építve. **Miért használják C-ben:**

- **Állapot enkapszulációja:** A struktúra (textttstruct) tartalmazza az "objektum" adatait (tagváltozóit). A függvények, amelyek ezen az "objektumon" operálnak, első paraméterként megkapják a struktúra egy példányára mutató pointert.
- **Adatok és műveletek összekapcsolása:** Így a függvény "tudja", hogy melyik adatstruktúrán kell dolgoznia. Ez logikailag összekapcsolja az adatokat (a struktúrában) és a rajtuk végezhető műveleteket (a függvények).
- **Több példány kezelése:** Lehetővé teszi, hogy ugyanaz a függvény különböző adatpéldányokon (különböző struktúra-példányokon) működjön, egyszerűen más-más mutató átadásával.

**Megfelelője más nyelvekben:** Ez a minta leginkább az objektum-orientált nyelvekben (pl. C++, Java, Python, C#) használt `this` **(C++, Java, C#) vagy** `self` **(Python) implicit paraméterhez** hasonlít. Amikor egy objektum metódusát hívjuk meg (pl. `objektum.metodus()`), a metódus implicit módon megkap egy mutatót vagy referenciát magára az objektumpéldányra, amelyen meghívták. Ez teszi lehetővé, hogy a metódus hozzáférjen az objektum tagváltozóihoz. C-ben ezt expliciten kell megtenni a struktúra mutatójának átadásával.

### Mi a keretidő, mi a mértékegysége és mire használjuk a programokban?

- **Keretidő (Frame Time):** Az az időtartam, ami egyetlen képkocka (frame) teljes kiszámításához és megjelenítéséhez szükséges a számítógépes grafikában, különösen játékoknál és animációknál.
- **Mértékegysége:** Tipikusan másodperc (s) vagy ezredmásodperc (millisecond, ms).
- **Mire használjuk a programokban:**

– **Animáció és fizika sebességének függetlenítése a hardvertől:** A játékok és animációk sebességét gyakran a keretidőhöz (vagy annak reciprokához, az FPS-hez képest számolt delta időhöz) igazítják. Például egy objektum elmozdulása: $uj\_pozicio = regi\_pozicio + sebesseg * keretido$. Ez biztosítja, hogy az animáció vagy a szimuláció hasonló sebességgel fusson különböző teljesítményű gépeken is. Ha fix idő lépésekkel dolgoznánk, gyorsabb gépeken gyorsabban, lassabbakon lassabban futna a program.
– **Teljesítménymérés és -optimalizálás:** A keretidő mérésével képet kaphatunk a program teljesítményéről. Ha a keretidő túl magas (azaz az FPS alacsony), az azt jelzi, hogy a program túl sok számítást végez egy képkocka alatt, és optimalizálásra lehet szükség.
– **Szinkronizáció:** Bizonyos műveleteket a képkockákhoz lehet igazítani.

A keretidő és az FPS (Frames Per Second - képkocka/másodperc) között szoros kapcsolat van: Keretidő (s) = 1/FPS. Például 60 FPS esetén a keretidő kb. $1/60 \approx 0.01667$ s, azaz 16.67 ms.

# 10. Animáció

### A virtuális színterünkben mi az amit animálni tudunk? (Soroljon fel legalább 5-öt!)

A virtuális színtérben gyakorlatilag bármilyen paramétert animálhatunk, ami a jelenet vizuális megjelenését vagy viselkedését befolyásolja. Legalább 5 példa:

1. **Objektum pozíciója:** Tárgyak mozgatása a térben (eltolás).
2. **Objektum orientációja/forgása:** Tárgyak forgatása a tengelyeik körül.
3. **Objektum mérete/skálázása:** Tárgyak növelése, zsugorítása.
4. **Anyagtulajdonságok:**
   - Szín (pl. diffúz, ambient, specular szín változtatása).
   - Átlátszóság (alpha érték változtatása).
   - Fényesség (shininess érték változtatása).
5. **Textúrakoordináták vagy textúrák:** Mozgó textúrák (pl. vízfelszín), textúrák közötti áttűnés vagy textúraváltás.
6. **Fényforrások tulajdonságai:**
   - Pozíció (mozgó fényforrás).
   - Szín, intenzitás (pl. villogó fény, színváltó fény).
   - Vetülési szög (spotlight esetén).
7. **Kamera paraméterei:**
   - Kamera pozíciója és nézési iránya (nézőpont mozgatása, körbejárás, pásztázás).
   - Látószög (Field of View - FOV) (zoom effektus).
8. **Csontváz-animáció (Skeletal Animation):** Karakterek, élőlények mozgása a belső csontvázuk (bone-ok) mozgatásával és forgatásával, ami deformálja a rá feszített hálót (skinning).

9. **Morfológiai animáció (Morph Target Animation / Blend Shapes):** Egy objektum alakjának folyamatos átmenete több előre definiált alak (célforma) között, pl. arcanimációhoz.
10. **Részecskerendszerek (Particle Systems):** Részecskék pozíciójának, színének, méretének, élettartamának animálása (pl. tűz, füst, robbanás).

# 11. Input Kezelés

### Milyen események tartoznak a billentyűzet/egér kezeléséhez?

- **Billentyűzet események:**
  - **Billentyű lenyomása (Key Down / Key Pressed):** Amikor egy billentyűt lenyom a felhasználó. Az esemény általában tartalmazza a lenyomott billentyű kódját (pl. scan code vagy virtuális billentyűkód) és esetleges módosító billentyűk (Shift, Ctrl, Alt) állapotát.
  - **Billentyű felengedése (Key Up / Key Released):** Amikor egy korábban lenyomott billentyűt felenged a felhasználó.
  - **Szövegbevitel (Text Input / Character Typed):** Magasabb szintű esemény, ami akkor generálódik, amikor a billentyűleütések egy karaktert eredményeznek (figyelembe véve a Shift-et, CapsLock-ot, nyelvi beállításokat, ékezetes karaktereket stb.). Az esemény általában a beírt karaktert tartalmazza Unicode vagy más karakterkódolás szerint.
- **Egér események:**
  - **Egérmozgás (Mouse Motion / Mouse Move):** Amikor a felhasználó mozgatja az egeret. Az esemény általában tartalmazza az egérkurzor új pozícióját (pl. ablakhoz vagy képernyőhöz képest relatív koordináták) és esetleg a relatív elmozdulást az előző pozícióhoz képest.
  - **Egérgomb lenyomása (Mouse Button Down / Mouse Button Pressed):** Amikor egy egérgombot (bal, jobb, középső, stb.) lenyom a felhasználó. Az esemény tartalmazza, hogy melyik gombot nyomták le, és az egérkurzor aktuális pozícióját.
  - **Egérgomb felengedése (Mouse Button Up / Mouse Button Released):** Amikor egy korábban lenyomott egérgombot felenged a felhasználó.
  - **Egérkattintás (Mouse Click):** Gyakran egy lenyomás és egy felengedés kombinációjaként értelmezett esemény. Lehet szimpla vagy dupla kattintás.
  - **Egérgörgő használata (Mouse Wheel Scroll):** Amikor a felhasználó görgeti az egér görgőjét. Az esemény tartalmazza a görgetés irányát és mértékét.
  - **Egérkurzor belépése az ablak területére (Mouse Enter):** Amikor az egérkurzor egy adott ablak vagy vezérlőelem fölé kerül.
  - **Egérkurzor kilépése az ablak területéről (Mouse Leave):** Amikor az egérkurzor elhagyja egy adott ablak vagy vezérlőelem területét.

# 12. Képformátumok

### Soroljon fel legalább 3 olyan képformátumot (rövidítéssel és teljes névvel), amelyik támogatja az átlátszóságot!

1. **PNG** (Portable Network Graphics): Támogatja az alfa-csatornát, ami 8 bites (256 szintű) átlátszóságot tesz lehetővé pixelenként.

2. **GIF** (Graphics Interchange Format): Támogatja az átlátszóságot, de csak indexelt színpaletta esetén, és csak 1 bites (teljesen átlátszó vagy teljesen átlátszatlan) átlátszóságot egyetlen kitüntetett színre.
3. **TIFF** (Tagged Image File Format): Támogathat alfa-csatornát az átlátszósághoz.
4. **WEBP** (WebP): Modern formátum, amely veszteséges és veszteségmentes tömörítést is támogat, mindkettő esetén képes alfa-csatornát kezelni az átlátszósághoz.
5. **SVG** (Scalable Vector Graphics): Vektorgrafikus formátum, amely támogatja az objektumok átlátszóságát.

### Soroljon fel legalább 2-2 tömörített és tömörítetlen képformátumot!

- **Tömörített képformátumok:**
  - **JPEG** (Joint Photographic Experts Group): Tipikusan veszteségesen tömörített, fotókhoz használatos.
  - **PNG** (Portable Network Graphics): Veszteségmentesen tömörített, jól kezeli az éles kontúrokat és az átlátszóságot.
  - **GIF** (Graphics Interchange Format): Veszteségmentesen tömörített (LZW), animációkhoz és egyszerűbb grafikákhoz használatos.
  - **WEBP** (WebP): Támogat veszteséges és veszteségmentes tömörítést is.
- **Tömörítetlen képformátumok (vagy minimálisan tömörített):**
  - **BMP** (Bitmap): Általában tömörítetlen, bár létezik RLE (Run-Length Encoding) tömörített változata is.
  - **PPM/PGM/PBM** (Portable Pixmap/Graymap/Bitmap): Egyszerű, tömörítetlen formátumok a Netpbm csomagból.
  - **RAW** (Nyers képformátumok): Sok digitális fényképezőgép ezt használja, minimális feldolgozással és általában veszteségmentes tömörítéssel vagy tömörítetlenül tárolja a szenzor adatait.
  - **TIFF** (Tagged Image File Format): Lehet tömörítetlen, vagy használhat veszteségmentes (pl. LZW, ZIP) vagy veszteséges (pl. JPEG) tömörítést is.

### Soroljon fel legalább 2-2 veszteséges és veszteségmentes képformátumot!

Ez a kérdés nagyon hasonló az előzőhöz, de a hangsúly a veszteséges/veszteségmentes tulajdonságon van.

- **Veszteséges képformátumok (lossy):** Információvesztéssel járó tömörítést használnak a kisebb fájlméret érdekében. Az eredeti kép nem állítható vissza tökéletesen.
  1. **JPEG** (Joint Photographic Experts Group)
  2. **WEBP** (veszteséges módban)
  3. Néhány **TIFF** variáns (ha JPEG tömörítést használ)
- **Veszteségmentes képformátumok (lossless):** Olyan tömörítést használnak, amely lehetővé teszi az eredeti kép tökéletes visszaállítását.
  1. **PNG** (Portable Network Graphics)
  2. **GIF** (Graphics Interchange Format)

3. **BMP** (RLE tömörítéssel, vagy tömörítetlenül is veszteségmentes)
4. **TIFF** (ha LZW, ZIP tömörítést használ, vagy tömörítetlen)
5. **WEBP** (veszteségmentes módban)

# 13. Fontos Alapfogalmak

### Mit nevezünk felületi normálisnak?

A felületi normális (vagy normálvektor) egy adott pontban a felületre merőleges vektort jelent.

- Sík felületek esetén a normálvektor iránya az egész felületen ugyanaz, és merőleges a síkra.
- Görbült felületek esetén a normálvektor iránya pontról pontra változhat. Az adott pontban a felület érintősíkjára merőleges.

A 3D grafikában a normálvektorok kulcsfontosságúak a megvilágítási számításokhoz (pl. a fényvisszaverődés szögének meghatározásához), a hátsó lapok eldobásához (back-face culling) és más vizuális effektekhez. Általában egységvektorként (1 hosszúságú vektorként) használják őket.

### Mi a szögsebesség mértékegysége? (Adjon meg legalább két alternatívát!)

A szögsebesség az elfordulás szögének időbeli változását írja le. Mértékegységei lehetnek:

1. **Radián per másodperc (rad/s):** Ez a SI mértékegységrendszerbeli hivatalos egység. Gyakran használják fizikában és matematikában. 1 radián az a középponti szög, amelyhez tartozó körív hossza megegyezik a kör sugarával.
2. **Fok per másodperc (°/s):** Gyakran használatos a mindennapi életben és a programozásban, mivel a fokbeosztás intuitívabb lehet (egy teljes kör 360°).
3. **Fordulat per perc (RPM - Revolutions Per Minute):** Főleg motorok, forgó gépek fordulatszámának megadására használják. (1 fordulat = 360° = $2\pi$ radián)
4. **Fordulat per másodperc (RPS - Revolutions Per Second):** Hasonló az RPM-hez, de másodperc alapon.

### Milyen előnyei és hátrányai vannak a raszteres és a vektorgrafikus megjelenítésnek?

- **Raszteres grafika (Bitmap grafika):**
  - **Leírás:** A képet képpontok (pixelek) rácsával ábrázolja, minden pixelnek saját színe van.
  - **Előnyök:**
    - Fotorealisztikus képek, komplex textúrák, finom színátmenetek jól ábrázolhatók.
    - Feldolgozása, megjelenítése (hardveresen) általában gyors.
    - Széles körben elterjedt formátumok (JPEG, PNG, GIF).
  - **Hátrányok:**
    - **Felbontásfüggő:** Nagyításkor pixelesedik, minőségromlás lép fel.
    - **Fájlméret:** Magas felbontás és sok szín esetén nagy fájlméretet eredményezhet.
    - **Szerkeszthetőség:** Az egyes geometriai alakzatok utólagos módosítása nehézkes, mivel csak pixelekkel dolgozunk.

- **Vektorgrafika:**
  - **Leírás:** A képet geometriai primitívek (vonalak, görbék, poligonok stb.) matematikai leírásával ábrázolja.
  - **Előnyök:**
    * **Felbontásfüggetlen:** Tetszőlegesen nagyítható minőségromlás nélkül.
    * **Fájlméret:** Általában kisebb fájlméretet eredményez, különösen vonalas ábráknál, logóknál.
    * **Szerkeszthetőség:** Az egyes objektumok könnyen módosíthatók (méret, szín, pozíció stb.).
    * Ideális logókhoz, illusztrációkhoz, tervrajzokhoz, betűtípusokhoz.
  - **Hátrányok:**
    * Fotorealisztikus képek, komplex textúrák nehezen vagy egyáltalán nem ábrázolhatók vele hatékonyan.
    * Bonyolult, sok objektumból álló vektorgrafikák renderelése lassú lehet.