# Vektor- és mátrixműveletek WebGL-hez

Szécsi László

3D Grafikus Rendszerek

4. előadás

# Mit szeretnénk?

- támogassa a szükséges műveleteket
- legyen könnyen használható

# Eukleidész

# Euklideszi geometria

- Bármely két **pont**on keresztül pontosan egy **egyenes** húzható.
- Bármely **szakasz**ra pontosan egy **egyenes** illeszkedik.
- Bármely közép**pont**tal és **sugár**ral ponsan egy kört lehet rajzolni.
- Bármely két derékszög egyenlő egymással.
- Egy **egyenes**hez egy **pont**on át pontosan egy olyan **egyenes** húzható, aminek az **egyenes**sel nincs közös **pont**ja.

- Ugyanazon dologgal egyenlő dolgok egymással is egyenlők.
- Ugyanazon dolog kétszeresei egyenlők egymással.

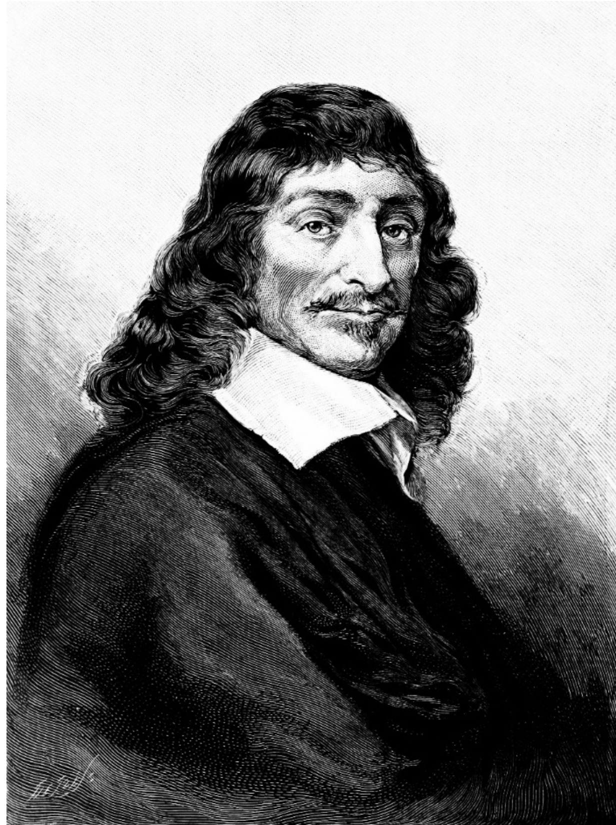
# Az euklideszi geometriában értelmezett fogalmak

- pont, egyenes $\rightarrow$ sík, metszés, illeszkedés
- szakasz, kör $\rightarrow$ távolság, eltolás, vektor
- hossz $\rightarrow$ hossz kétszerese, fele, harmada

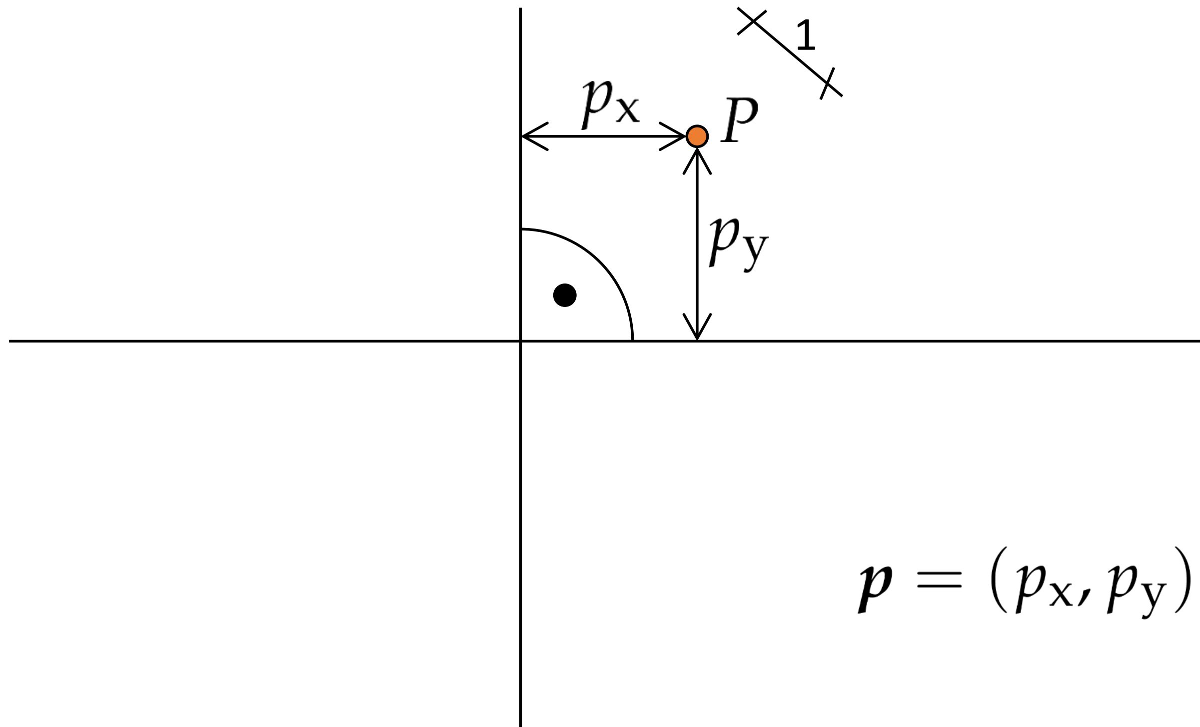
- kerület, terület, felület, térfogat

- súlypont

# Descartes



# Descartes koordináta-rendszer

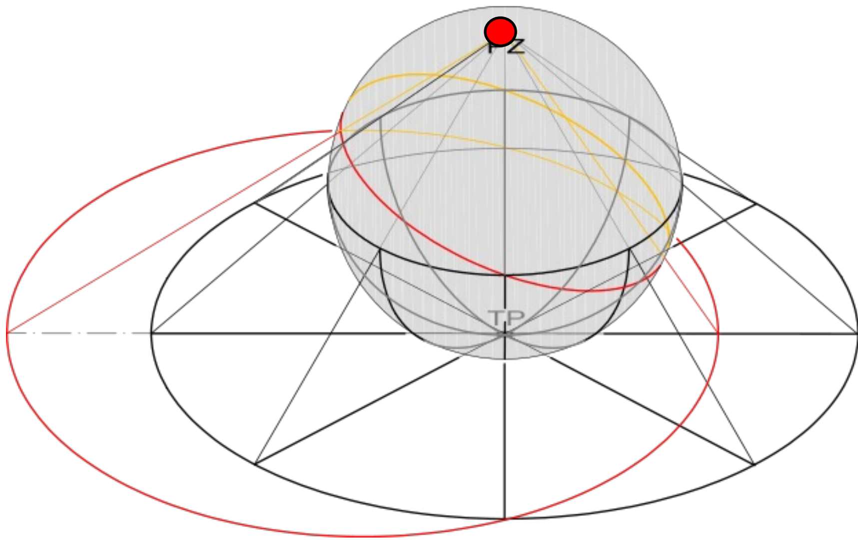


# Analitikus geometria

- pont $\rightarrow$ koordináták
  - Karteziánus koordináta-rendszer
- ponthalmazok (egyenes, szakasz, sík, kör, gömb) $\rightarrow$ egyenletek
- metszéspont $\rightarrow$ egyenletrendszer megoldása
- eltolás, forgatás, tükrözés, nagyítás $\rightarrow$ koordináta-transzformációk

# Bernhard Riemann, Bolyai János

# Projektív geometria




sztereografikus vetítés

perspektíva

# Ferdinand Möbius

# Homogén koordináták

- Descartes + origó középpontú skálázás skálatényezője
  - ha a skálatényező nem nulla, visszaskálázva kaphajuk a Descartes koordinátákat
  - ha a skálatényező nulla, a pont a végtelenben van

$$\boldsymbol{p} = (p_\mathrm{x}, p_\mathrm{y})$$

Descartes

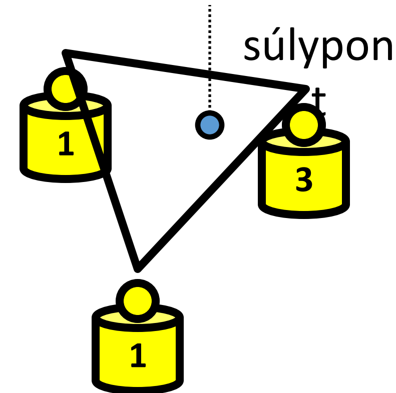
$$\breve{\boldsymbol{p}} = [\breve{p}_\mathrm{x}, \breve{p}_\mathrm{y}, \breve{p}_\mathrm{w}]$$

homogén

$$(p_\mathrm{x}, p_\mathrm{y}) = \left(\frac{\breve{p}_\mathrm{x}}{\breve{p}_\mathrm{w}}, \frac{\breve{p}_\mathrm{y}}{\breve{p}_\mathrm{w}}\right)$$

# Baricentrikus koordináták

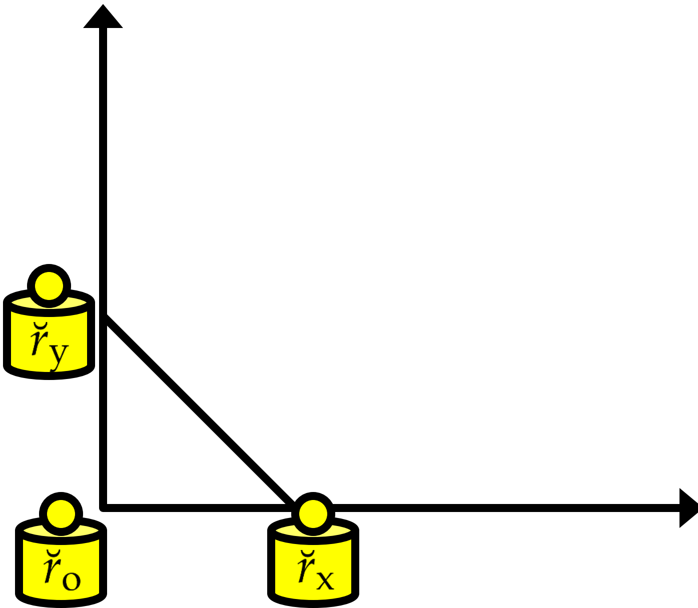
- a, b, c súlyok: pont a síkon
- kitüntetett
  - 3 csúcs
- Miért hasznos?
  - mind pozitív: pont a háromszögön belül
  - a súlyok használhatók a csúcsokhoz rendelt értékek közötti interpolációra
- homogén
  - 2x súly: ugyanaz a pont



# Homogén koordináták

- legyen a 3 pont:

$$(1,0), (0,1), (0,0)$$



# Descartes $\rightarrow$ spec. baricentrikus

- ha a Descartes koordináták: $(r_{\mathrm{x}}, r_{\mathrm{y}})$
- mik a szükséges súlyok? $[\breve{r}_{\mathrm{x}}, \breve{r}_{\mathrm{y}}, \breve{r}_{\mathrm{o}}]$
- írjuk fel a tömegközéppontot:

$$(r_{\mathrm{x}}, r_{\mathrm{y}}) = \frac{\breve{r}_{\mathrm{x}}(1,0) + \breve{r}_{\mathrm{y}}(0,1) + \breve{r}_0(0,0)}{\breve{r}_{\mathrm{x}} + \breve{r}_{\mathrm{y}} + \breve{r}_{\mathrm{o}}}$$

- végtelen sok megoldás van

# Ha súlyösszeg 1

$$(r_\mathrm{x}, r_\mathrm{y}) = \frac{\breve{r}_\mathrm{x}(1,0) + \breve{r}_\mathrm{y}(0,1) + \breve{r}_0(0,0)}{\breve{r}_\mathrm{x} + \breve{r}_\mathrm{y} + \breve{r}_\mathrm{o}} = \frac{(\breve{r}_\mathrm{x} + 0 + 0, 0 + \breve{r}_\mathrm{y} + 0)}{1}$$

$$\begin{aligned} \breve{r}_\mathrm{x} &= r_\mathrm{x} \\ \breve{r}_\mathrm{y} &= r_\mathrm{y} \\ \breve{r}_\mathrm{o} &= 1 - \breve{r}_\mathrm{x} - \breve{r}_\mathrm{y} \end{aligned}$$

# Ha más legyen a súlyösszeg

- legyen a súlyösszeg:

$$\breve{r}_{\mathrm{w}} = \breve{r}_{\mathrm{x}} + \breve{r}_{\mathrm{y}} + \breve{r}_{\mathrm{o}}$$

- ezzel a megoldás:

$$(r_{\mathrm{x}}, r_{\mathrm{y}}) = \frac{\breve{r}_{\mathrm{x}}(1,0) + \breve{r}_{\mathrm{y}}(0,1) + \breve{r}_{0}(0,0)}{\breve{r}_{\mathrm{w}}} = \frac{(\breve{r}_{\mathrm{x}} + 0 + 0, 0 + \breve{r}_{\mathrm{y}} + 0)}{\breve{r}_{\mathrm{w}}}$$

$$\breve{r}_{\mathrm{x}} = r_{\mathrm{x}} \breve{r}_{\mathrm{w}} \qquad \breve{r}_{\mathrm{y}} = r_{\mathrm{y}} \breve{r}_{\mathrm{w}} \qquad \breve{r}_{\mathrm{o}} = \breve{r}_{\mathrm{w}} - \breve{r}_{\mathrm{x}} - \breve{r}_{\mathrm{y}}$$

# Súlyok $\rightarrow$ Descartes

- ha adottak a súlyok $[\breve{r}_{\mathrm{x}}, \breve{r}_{\mathrm{y}}, \breve{r}_{\mathrm{o}}]$
- mi lesz a Descartes kooridináta? $(r_{\mathrm{x}}, r_{\mathrm{y}})$

$$(r_{\mathrm{x}}, r_{\mathrm{y}}) = \left( \frac{\breve{r}_{\mathrm{x}}}{\breve{r}_{\mathrm{w}}}, \frac{\breve{r}_{\mathrm{y}}}{\breve{r}_{\mathrm{w}}} \right)$$

# Homogén koordináták

- sosem érdekes $\breve{r}_{\mathrm{o}}$
- mindig érdekes a súlyösszeg
- homogén koordináták:

$$[\breve{r}_{\mathrm{x}}, \breve{r}_{\mathrm{y}}, \breve{r}_{\mathrm{w}}]$$

- ugyanez a pont 1 összsúllyal: $\left[\frac{\breve{r}_{\mathrm{x}}}{\breve{r}_{\mathrm{w}}}, \frac{\breve{r}_{\mathrm{y}}}{\breve{r}_{\mathrm{w}}}, 1\right]$

- Descartes koordináták: $(r_{\mathrm{x}}, r_{\mathrm{y}}) = \left(\frac{\breve{r}_{\mathrm{x}}}{\breve{r}_{\mathrm{w}}}, \frac{\breve{r}_{\mathrm{y}}}{\breve{r}_{\mathrm{w}}}\right)$

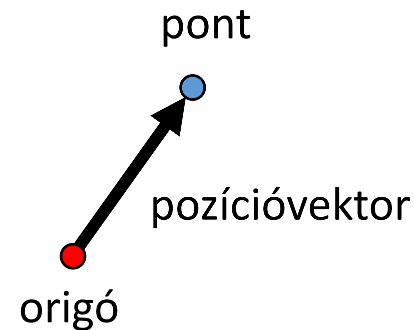
# Vektorok

- geometriai vektor = eltolás
  - $v$ eltolás
  - $|v|$ van hossza
  - $\omega_v$ van iránya
    - ritkán dolgozunk közvetlenül irányokkal
    - $\hat{v}$ adott irányú egységvektor

$$v = \hat{v} \cdot |v|$$

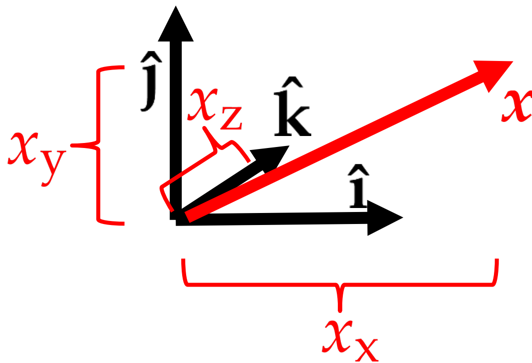
# Vektorok és pontok

- ha van kitüntetett origó
- minden ponthoz rendelhető egy pozícióvektor



# Vektorok 3D Descartes koordináták esetén

a Descartes koordinátarendszer bázisvektorai




a pozíció kifejezhető a bázisvektorok lineárkombinációjaként

$$\boldsymbol{x} = x_x\hat{\mathbf{i}} + x_y\hat{\mathbf{j}} + x_z\hat{\mathbf{k}}$$

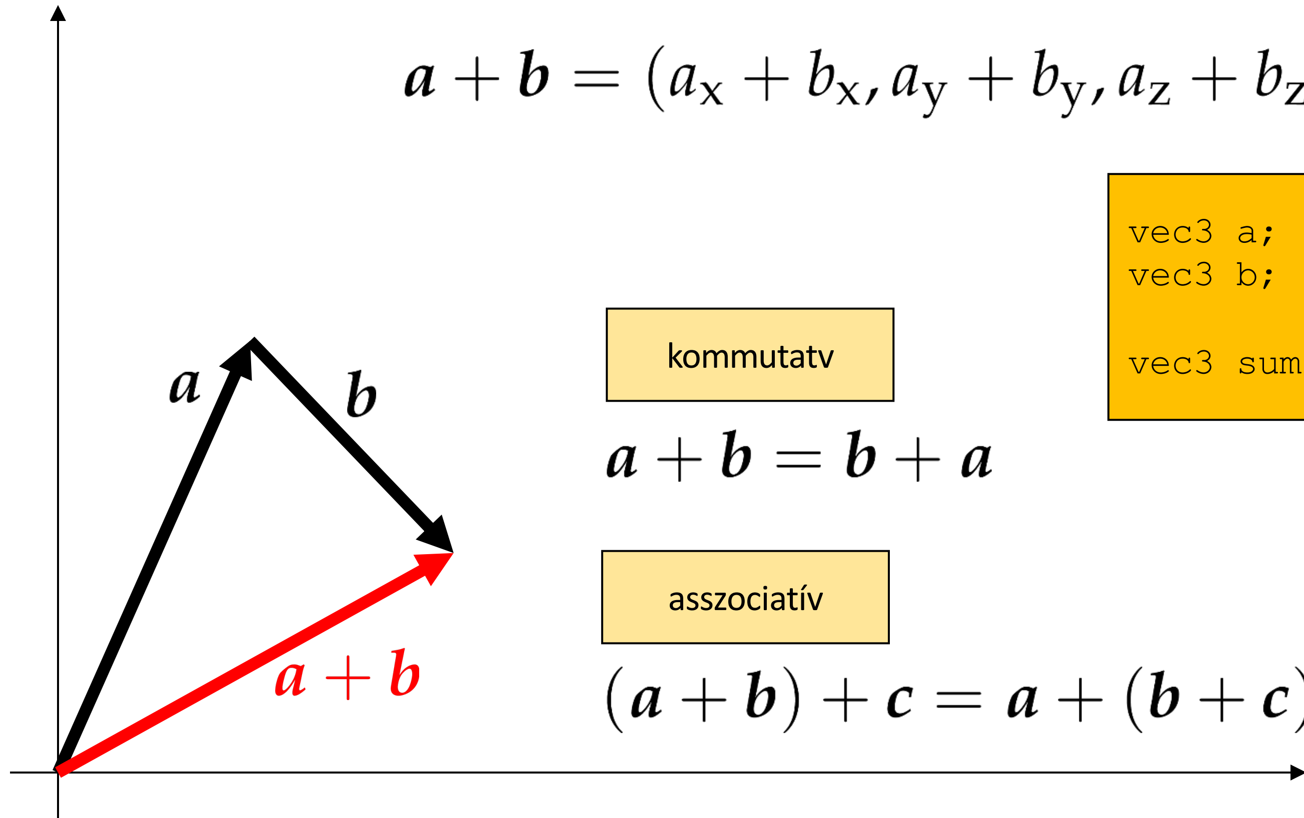
$$\boldsymbol{x} = (x_x, x_y, x_z)$$

sorvektor

$$\boldsymbol{x} = \begin{bmatrix} x_x \\ x_y \\ x_z \end{bmatrix}$$

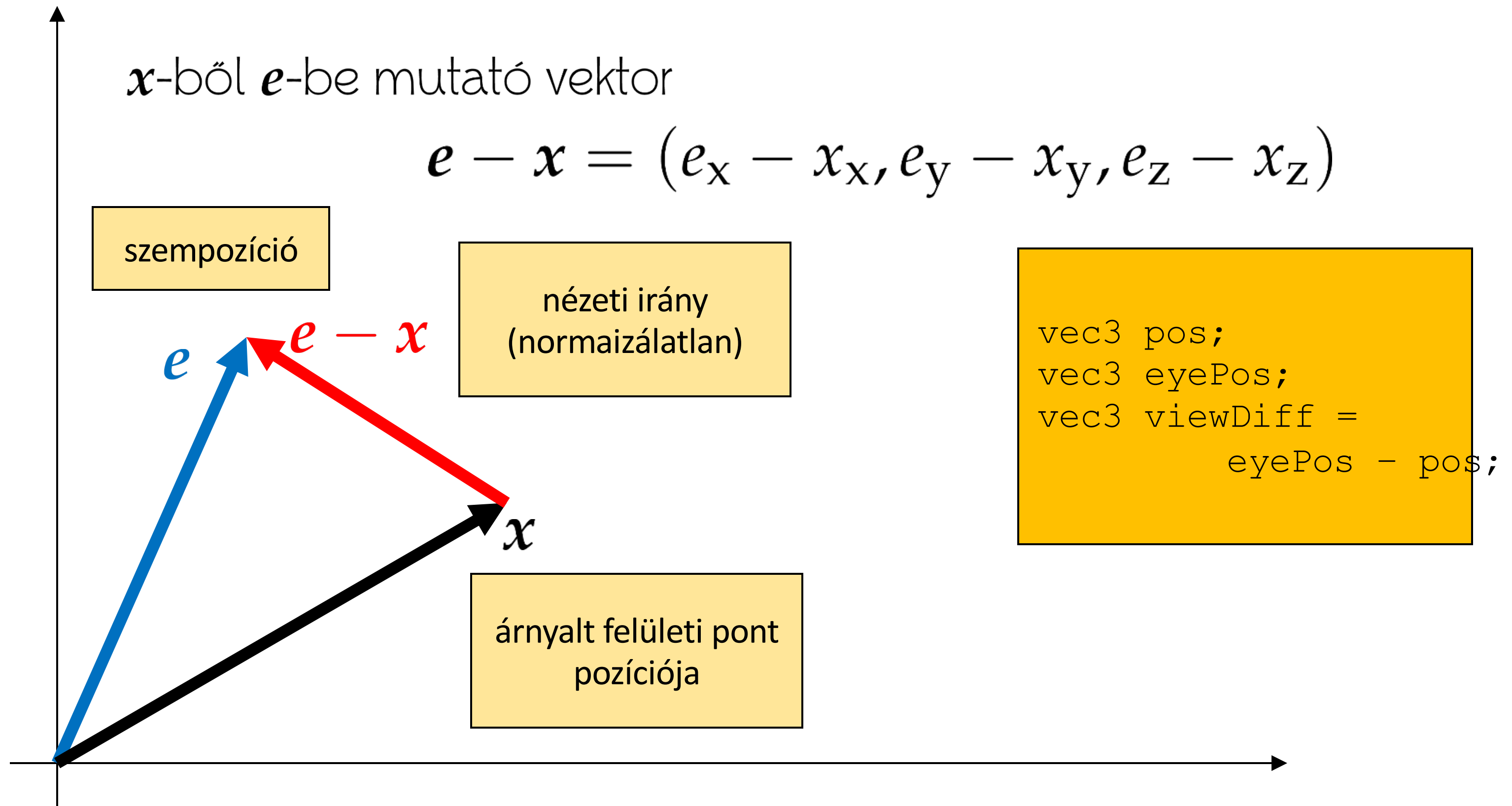
oszlopvektor

# Vektorok összege

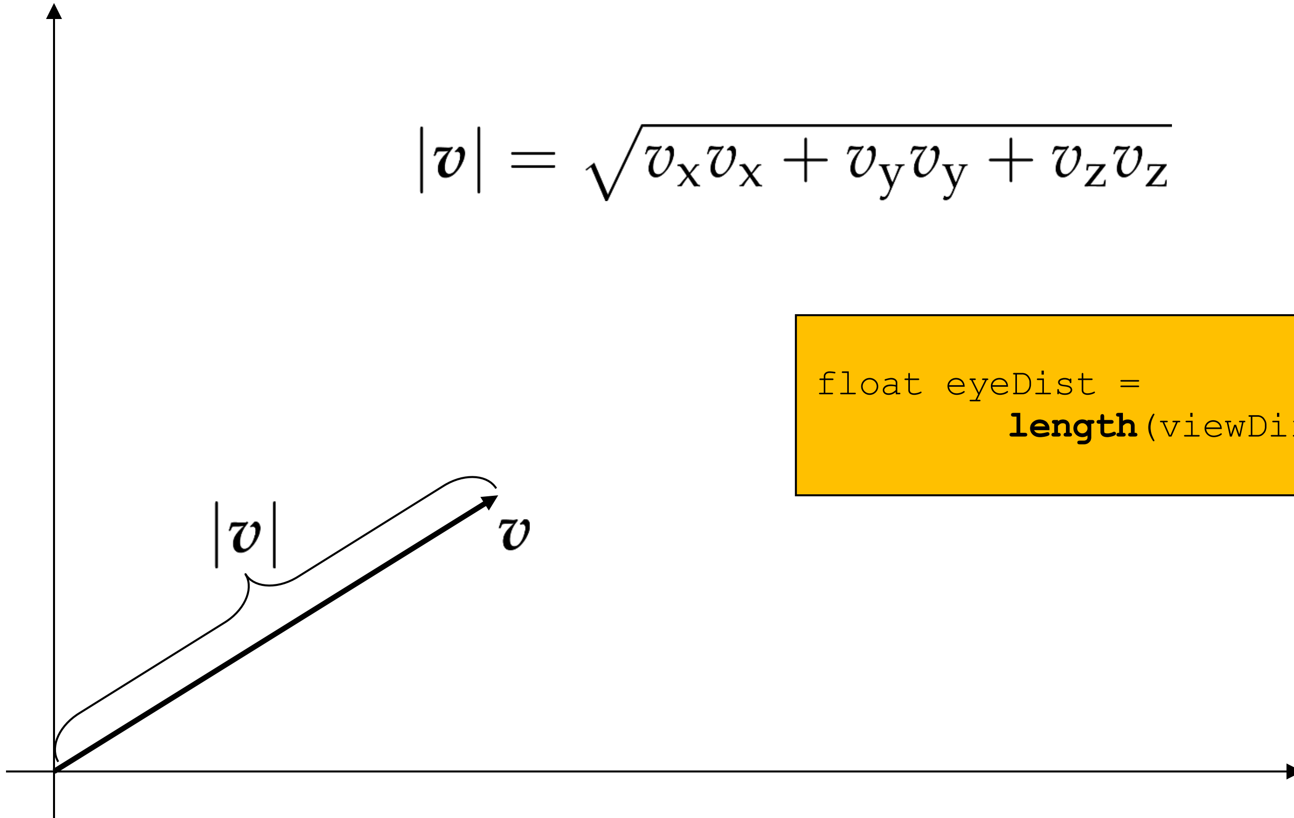


# Vektorösszeadás

# Vektorok különbsége

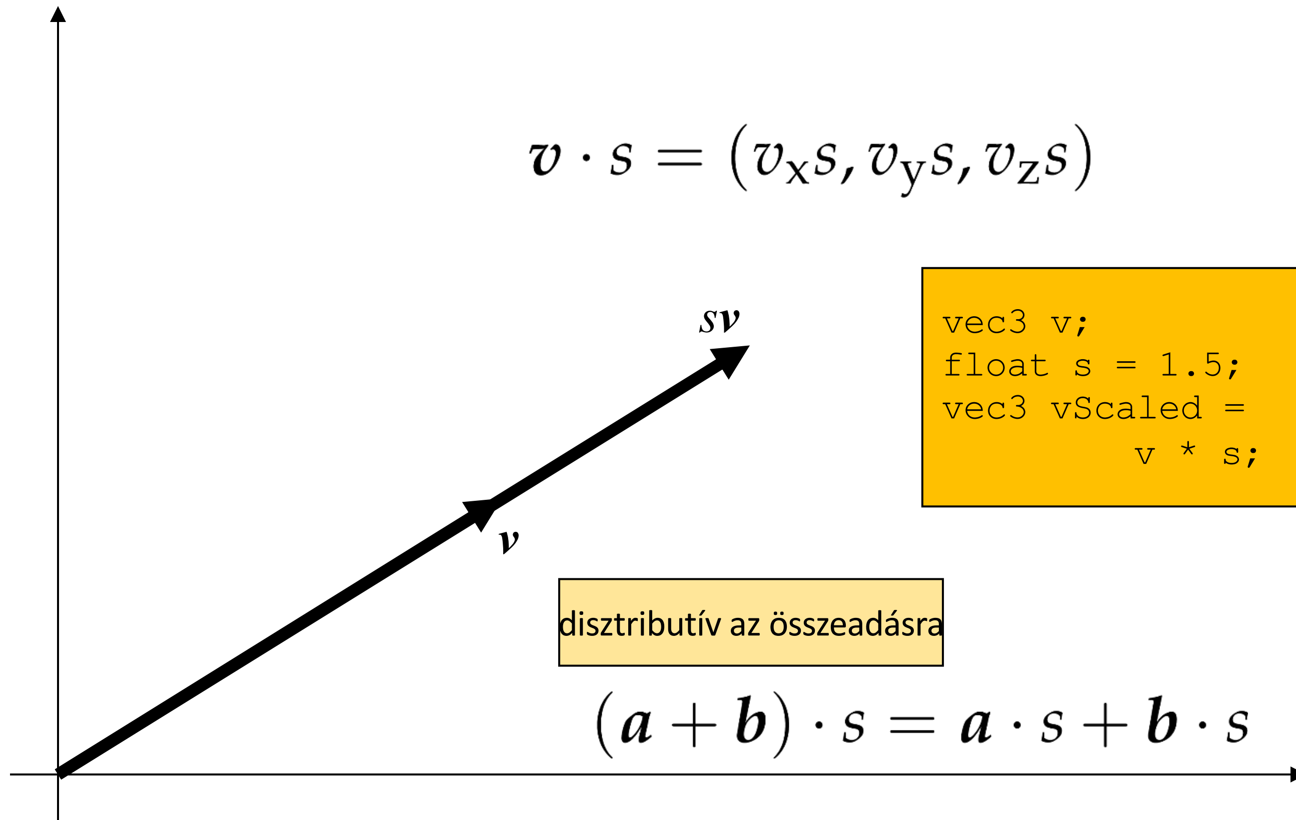


# Vektorhossz

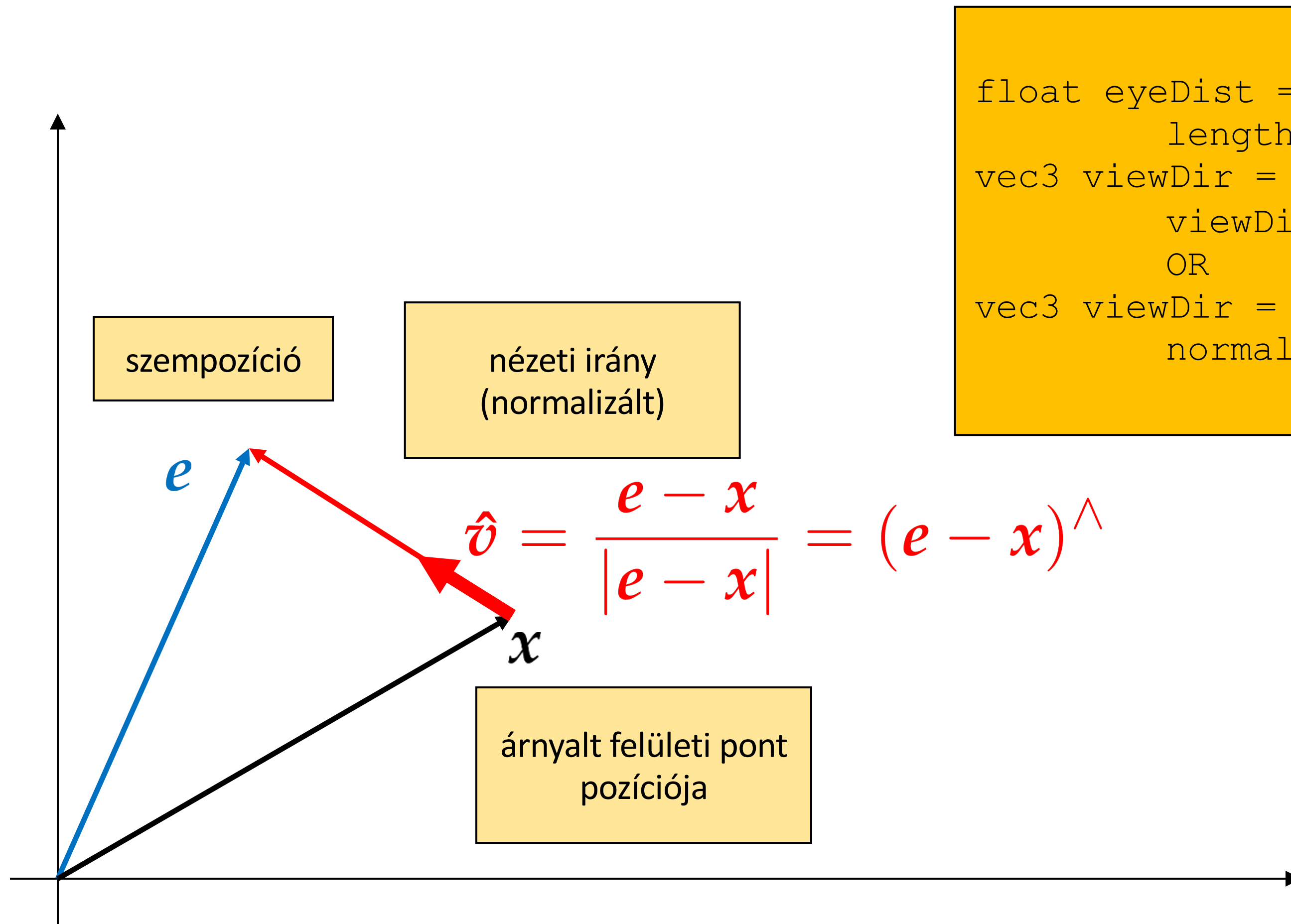



```
float eyeDist =
          length(viewDiff);
```

# Skálázás (vektor szorzása skalárral)

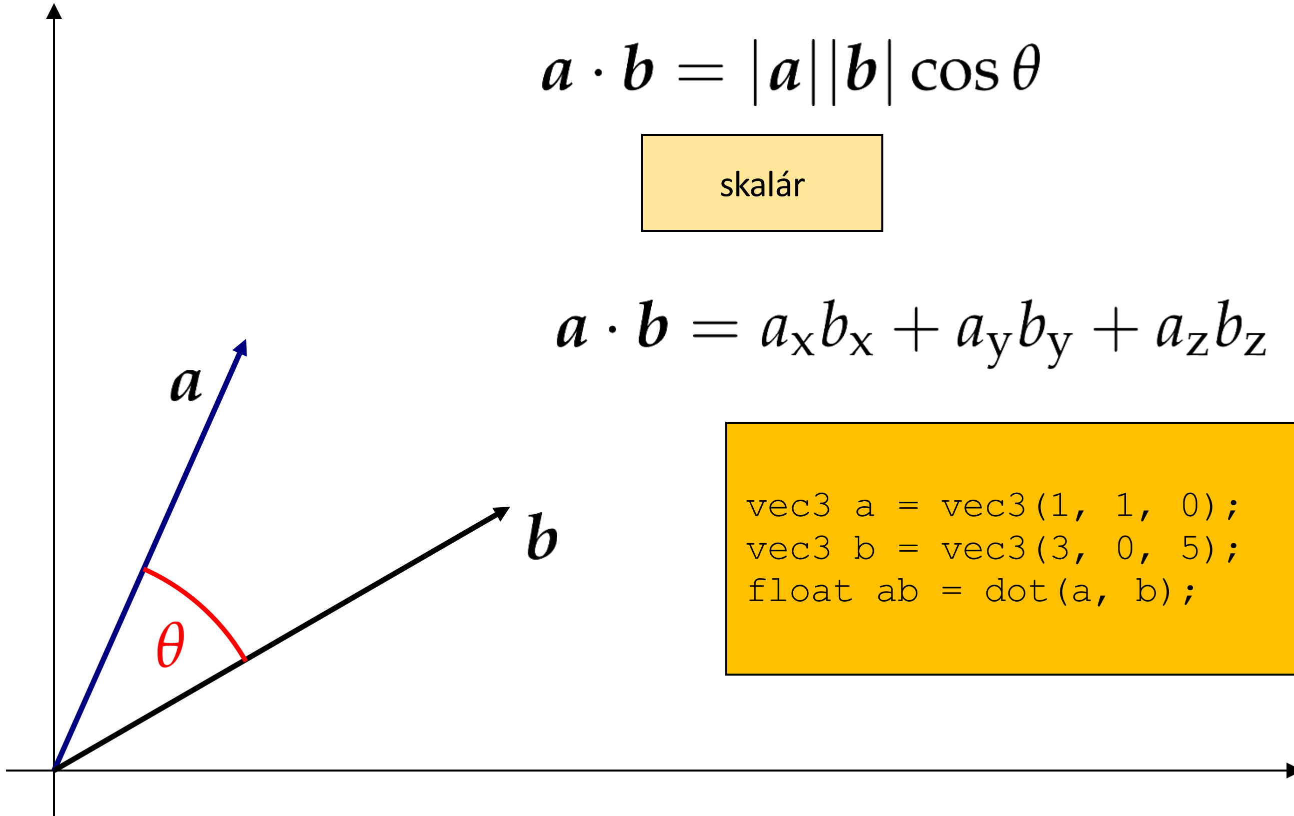


# Normalizálás

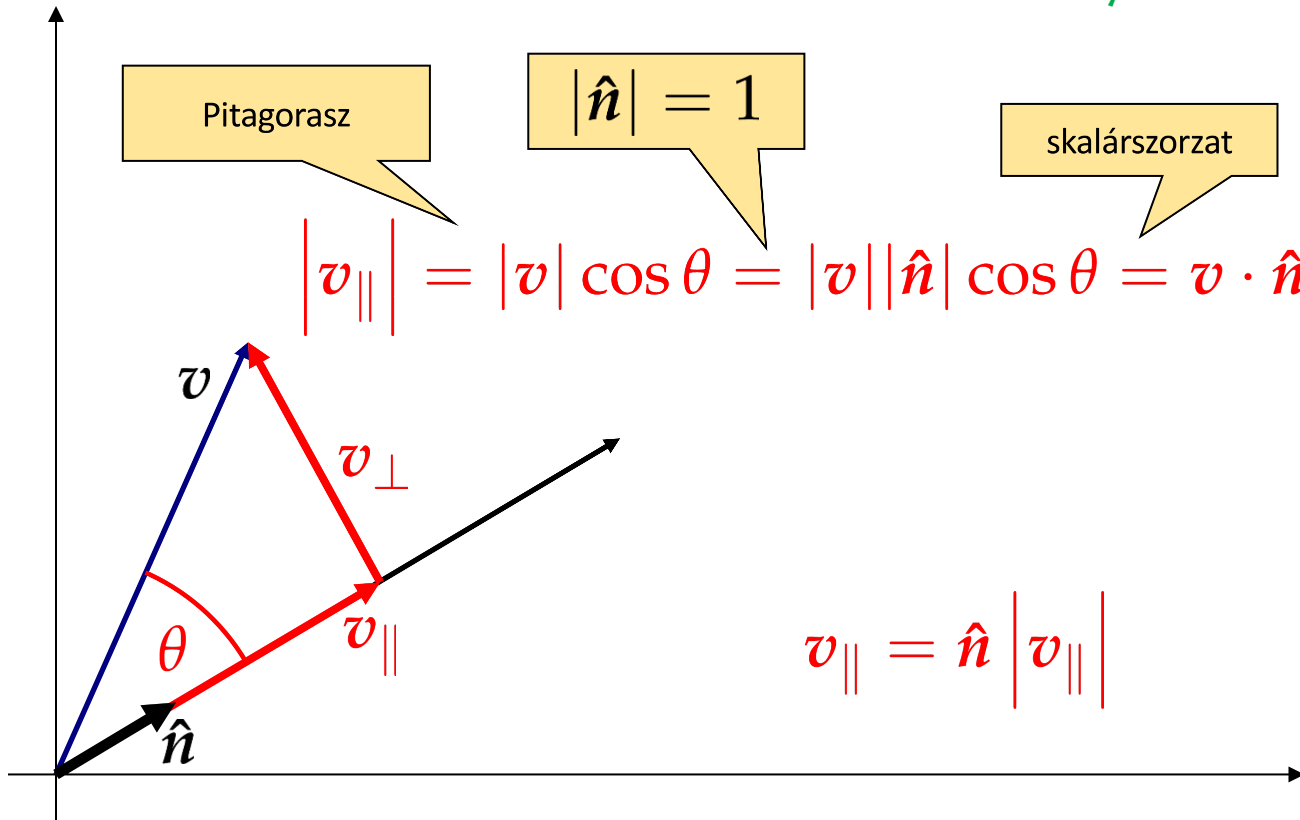



```
float eyeDist =
        length(viewDiff);
vec3 viewDir =
        viewDiff / eyeDist;
        OR
vec3 viewDir =
        normalize(viewDiff);
```

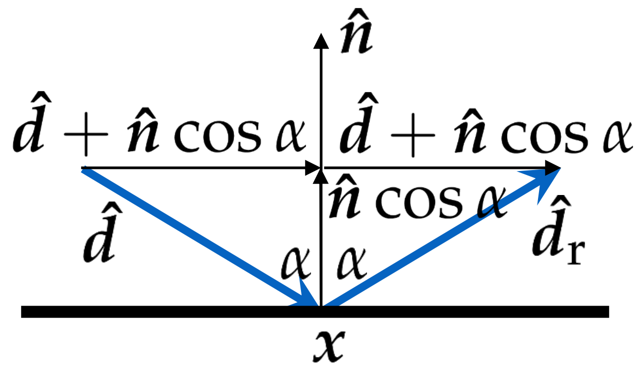
# Skalárszorzat



# Skalárszorzat használata:



# Példa: ideális visszaverődés iránya

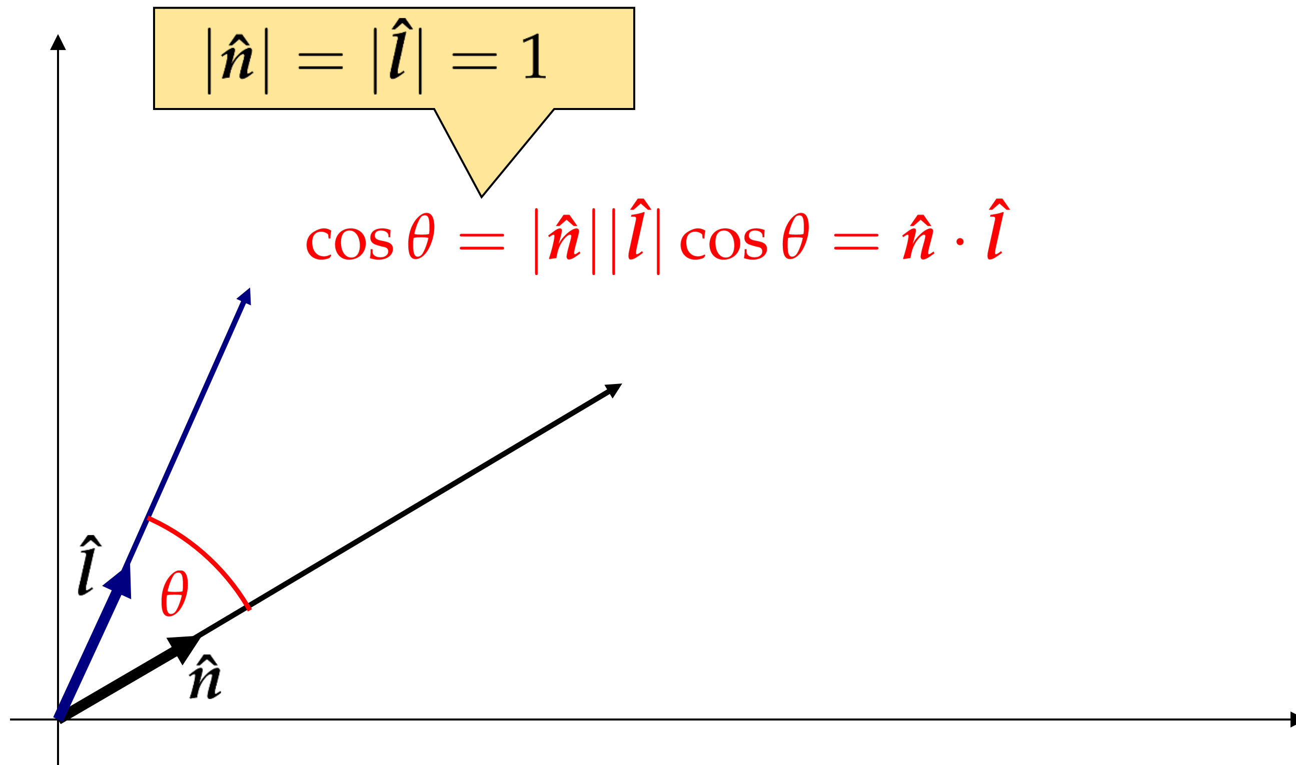



```
vec3 reflect(vec3 inDir,
vec3 normal)
{
  return
   inDir - normal *
   normal.dot(inDir) * 2;
};
```

$$\cos\alpha = -\hat{n}\cdot\hat{d}$$

$$\hat{d}_{\mathrm{r}} = \hat{n}\cos\alpha + \hat{d} + \hat{n}\cos\alpha = \hat{d} - 2\hat{n}\left(\hat{n}\cdot\hat{d}\right)$$

# Skalárszorzat használata: két irány közötti szög cosinusa



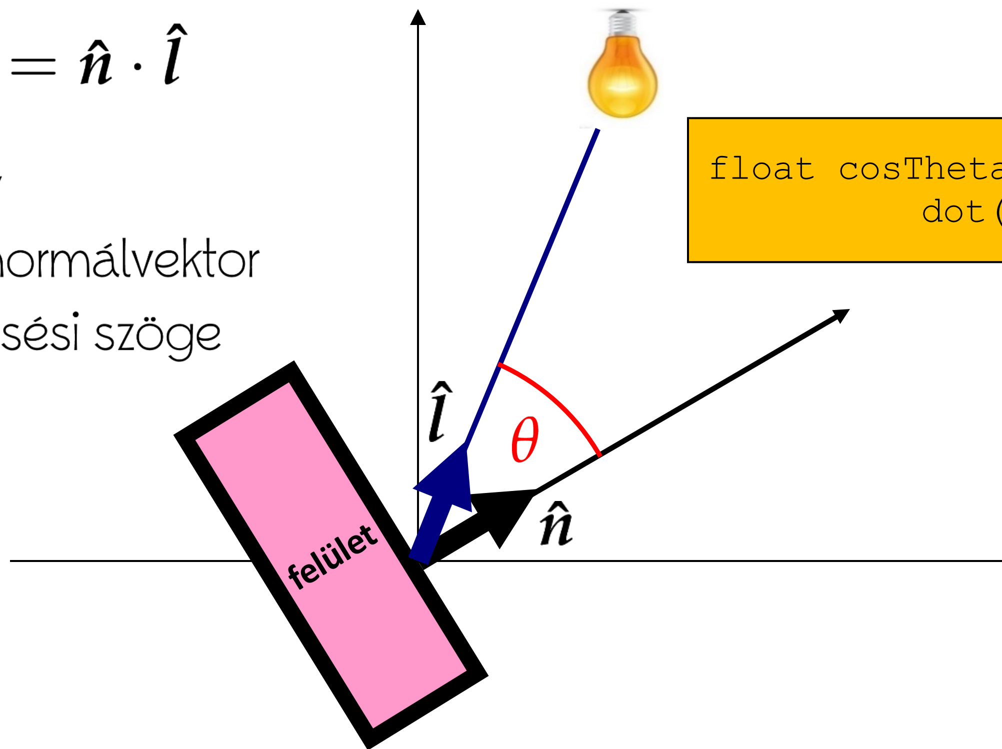
# Példa: fény beesési szögének cosinusa

$$\cos\theta = \hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{l}}$$

$\hat{\boldsymbol{l}}$: fényirány

$\hat{\boldsymbol{n}}$: felületi normálvektor

$\theta$: fény beesési szöge



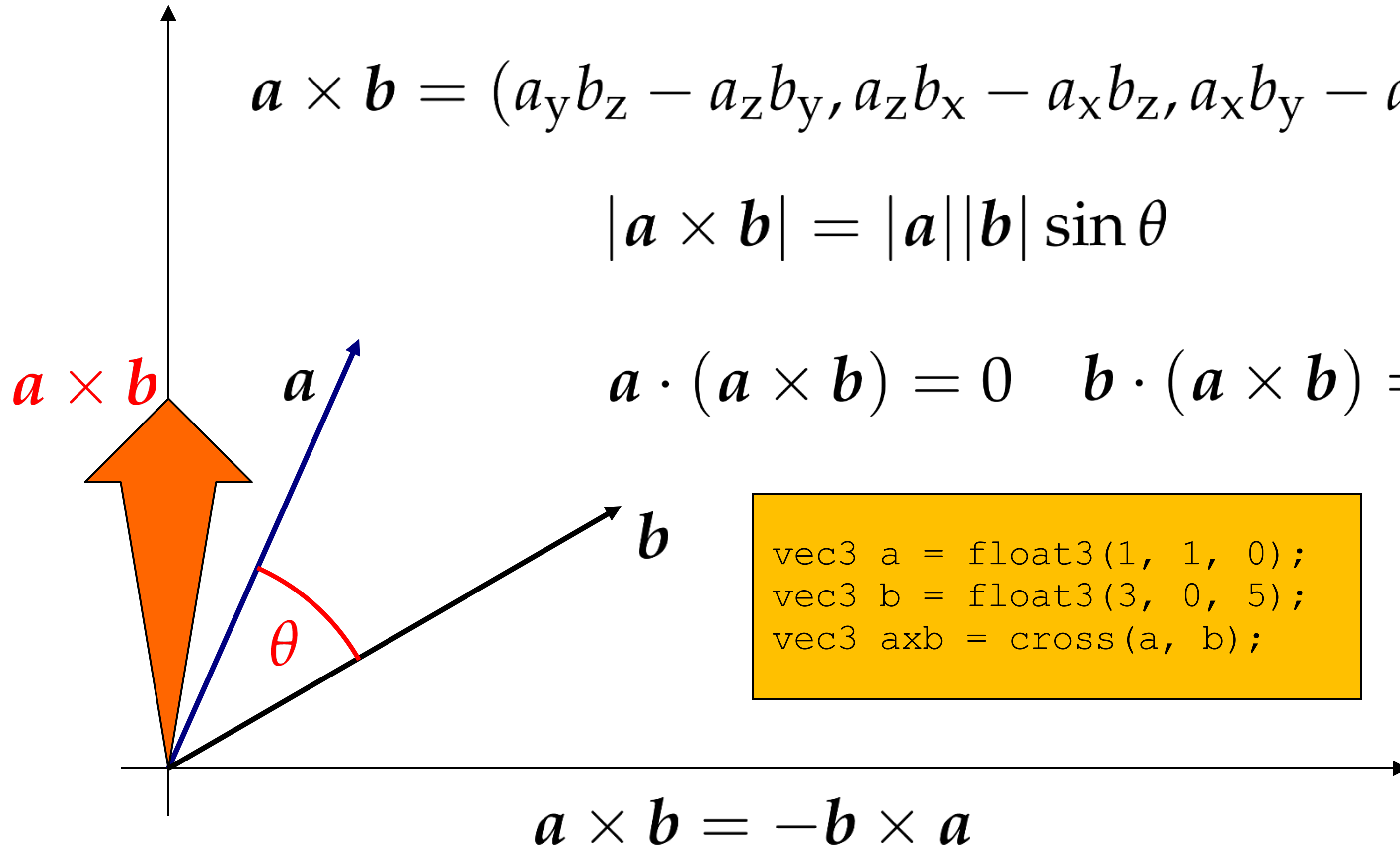

```
float cosTheta =
          dot(lightDir, normal);
```

# Vektor hosszának négyzete

$$\boldsymbol{a} \cdot \boldsymbol{a} = |\boldsymbol{a}||\boldsymbol{a}| \cos 0 = |\boldsymbol{a}|^2$$

```
vec3 a = float3(34, 435, 353);
float a2 = dot(a, a);
```

# Keresztszorzat (vektoriális szorzat)




$$\boldsymbol{a} \times \boldsymbol{b} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

$$|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}| \sin\theta$$

$$\boldsymbol{a} \cdot (\boldsymbol{a} \times \boldsymbol{b}) = 0 \quad \boldsymbol{b} \cdot (\boldsymbol{a} \times \boldsymbol{b}) = 0$$

```
vec3 a = float3(1, 1, 0);
vec3 b = float3(3, 0, 5);
vec3 axb = cross(a, b);
```

$$\boldsymbol{a} \times \boldsymbol{b} = -\boldsymbol{b} \times \boldsymbol{a}$$

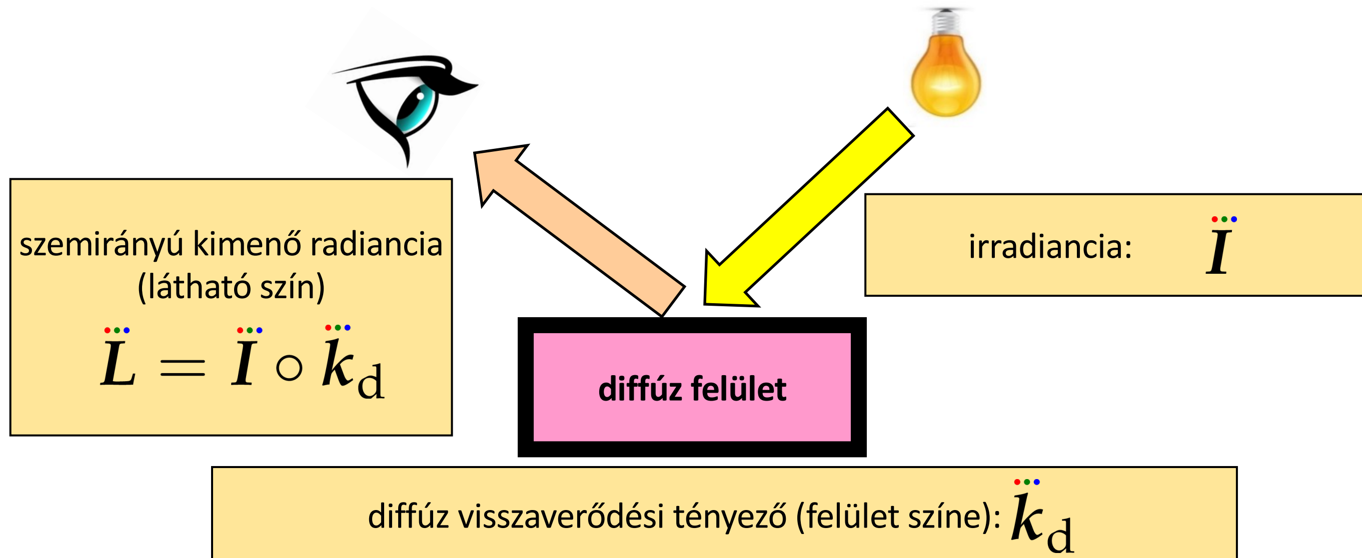
# Keresztszorzat használata

- merőleges keresése adott vektorra
  - keresztszorozzuk egy nem párhuzamos vektorral
- merőleges két vektorra
  - keresztszorzatuk
  - ha az operandusok nem normalizáltak vagy nem derékszögben állnak, az eredmény normalizálizálására szükség lehet
- Példa: parametrikus felület normálvektora
  - két érintővektor (tangens, binormál) keresztszorzata

# Elemenkénti (Hadamard) szorzat

$$\ddot{\boldsymbol{a}} \circ \dddot{\boldsymbol{b}} = (a_r b_r, a_g b_g, a_b b_b)$$

analitikus geometriában nem értelmes, de a grafikában gyakori



# Vektor osztályok C++-ban

- Direct3D/HLSL programozáshoz
- float, int, bool vektorok
- float4x4 mátrix
- teljes HLSL intrinsic függvénykészlet támogatása
- swizzle

# Alap struktúra

```
// float2.h – minden inline
namespace Nest { namespace Math {

class float2
{
public:
	float x;
	float y;
};

}}
```

# Indexelhetőség

```
class float2
{
public:
	union{
		struct {
			float x;
			float y;
		};
		float v[2];
	};
	float operator[](unsigned int i) const
	{
		return v[i];
	}
};
```

# Indexelhetőség 2

```
float operator[](unsigned int i) const
{
      return v[i];
}

float& operator[](unsigned int i)
{
      return v[i];
}
```

# Konstruktorok

```
float2():x(0.0f),y(0.0f){}

float2(float f):x(f),y(f){}

float2(float x, float y):x(x),y(y){}

float2(float x, float y, float z, float w)
      :x(x),y(y){z; w;}

float2(int2 i):x((float)i.x),y((float)i.y){}
```

# Összeadás

```
float2 operator+(const float2& o) const
{
	return float2(x + o.x, y + o.y);
}
```

# Hozzáadás

```
float2& operator+=(const float2& o)
{
	x += o.x;
	y += o.y;
	return *this;
}
```

# Keresztszorzat

```
float3 cross(const float3& o) const
{
	return float3(
		y * o.z - z * o.y,
		z * o.x - x * o.z,
		x * o.y - y * o.x
	);
}
```

# Konstansok

```
//float3.h
static const float3 xUnit;

//constants.cpp
const float3 float3::xUnit(1.0f, 0.0f, 0.0f);
```

# Swizzle

```
union{
	struct {
		float x;
		float y;
	};

	float v[2];

	float2swizzle<2, float2, 1, 0> yx;
	//...
	float3swizzle<2, float3, 0, 1, 0> xyx;
	float3swizzle<2, float3, 0, 1, -2> xy1;
	//...
	float4swizzle<2, float4, 1, 0, 1, 1> yxyy;
```

# Swizzle jobbértékként

```
template<... int s0=0, int s1=0, int s2=0, int s3=0>
class float3swizzle {
      float v[3];

operator float3() const
{
      return float3(
            (s0>=0)?v[s0]:((s0==-1)?0.0f:1.0f),
            (s1>=0)?v[s1]:((s1==-1)?0.0f:1.0f),
            (s2>=0)?v[s2]:((s2==-1)?0.0f:1.0f),
            (s3>=0)?v[s3]:((s3==-1)?0.0f:1.0f));
}
```

# Swizzle balértékként

```
inline floatswizzle& operator =(const T& o)
{
      v[ s0 ] = o[0];
      v[ s1 ] = o[1];
      v[ s2 ] = o[2];
      v[ s3 ] = o[3];
      return *this;
}
```

nyilván nem negatív

# Swizzle balértékként

```
inline floatswizzle& operator +=(const T& o)
{
      v[ s0 ] += o[0];
      v[ s1 ] += o[1];
      v[ s2 ] += o[2];
      v[ s3 ] += o[3];
      return *this;
}
```

# Metódushívás swizzleön

- a vektorosztályok összes const metódusa legyen meg nekik is
  - nem const metódus: csak az értékadás operátorok
- az implementáció létrehoz egy rendes vektorpéldányt, és meghívja rá

# Mátrixok

```
static float4x4 translation(const float3& offset)
{
	float4x4 t = identity;
	t._30 = offset.x;
	t._31 = offset.y;
	t._32 = offset.z;
	return t;
}

inline const float4& operator*=(float4& v, const
float4x4& m)
{
	v = m.transform(v);
	return v;
}
```

# Alternatívák: D3DXVECTOR3

```
D3DXVECTOR3* D3DXVec3Cross(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_    const D3DXVECTOR3 *pV1,
  _In_    const D3DXVECTOR3 *pV2
);
```

# Alternatívák: DirectXMath

```
XMVECTOR XMVector3Cross(
  [in] XMVECTOR V1,
  [in] XMVECTOR V2
);

XMVECTOR XMLoadFloat2(
  [in] const XMFLOAT2 *pSource
);
```

# Vektor objektumok JavaScriptben

- nincs inline: függvényhívás drága!
- csak dinamikus példányosítás: drága!
  - ami olyan szép, mint a C++, az gyors nagyon nem lesz
- nincs operátor overload
  - szép sem lesz soha

# Vektor objektumok Kotlinban

- an operátor overload
  - szép

- de gyorsabb nem lesz, mint a JS
- ha nagyon optimalizálni akarunk, kevésbé tudunk

# Egyszerű vektor konstruktor

```
class Vec2(val x: Float = 0.0f, val y: Float = 0.0f){

  operator inline fun plus(other : Vec2) : Vec2 {
    return Vec2(
      x + other.x,
      y + other.y
      )
  }
  fun toArray(dest : Float32Array){...}
}
```

szokásos objektuminicializálás [OK]

szokásos példányosítás [OK]

number aritmetika lassú

együttműködés a WebGL-lel körülményes, csúnya, lassú

# Típusos tömbbel

```
//a = b + c
val a = b + c
```

```
class Vec2(u: Float = 0.0f, v: Float = 0.0f){
  val storage: Float32Array



  operator inline fun plus(other : Vec2) : Vec2 {
    return Vec2(
      storage.x + other.storage.x,
      storage.y + other.storage.y
      )
  }
}
```

típusos new nagyon lassú

float aritmetika gyors

együttműködés a WebGL-lel triviális

# Koordinátatulajdonságok

```
inline var x : Float
  get() = storage[0]
  set(value) { storage[0] = value }

inline var y : Float
  get() = storage[1]
  set(value) { storage[1] = value }
```

# New kikerülése: művelet értékadással

```
operator fun plusAssign(other : Vec2) {
    storage[0] += other.storage[0]
    storage[1] += other.storage[1]
  }




// a = b + c
a.set(b)
a += c
      //jsperf: 3x gyorsabb, mint val a = b + c
```

# Vektortömbök

- GLSL-ben is van
  - direkt beállításhoz nekünk is kell
  - nem lesznek nagyon nagy tömbök
  - indexelni szeretnénk
- tömbökön végzett műveletek
  - egy függvényhívás
  - sok gyors aritmetika

# Műveletek új tömbök létrehozása nélkül – lusta operátorok

```
interface Gif {
  operator fun invoke(i : Int) : Float
}
class VecArray : Gif {
  override operator fun invoke(i : Int) : Float{
    return storage[i % storage.length]
  }
  infix fun gets(other : Gif){
    for(i in 0 until storage.length) {
      storage[i] = other(i)
    }
  }
  operator fun plus(c : Gif) : Gif {
    return object : Gif {
      override operator fun invoke(i : Int) : Float {
        return this@VecArray(i) + c(i)
      }
    }
  }
```

```
a gets b + c
```

# Kivonás

```
operator fun minusAssign(other : Gif) {
   for(i in 0 until storage.length) {
     storage[i] -= other(i)
   }
 }
```

```
ez is megy:
a -= b + c
```

# Indexelhetőség

```
a[2]
```

```
class Vec2Array() : VecArray() {
  operator fun get(i : Int) : Vec2{
    return Vec2(storage, i*2)
  }
}

class Vec2(backingStorage: Float32Array?, offset: Int = 0) {
  constructor(u: Float = 0.0f, v: Float = 0.0f) : this(null, 0){
    storage[0] = u
    storage[1] = v
  }

val storage: Float32Array =
  backingStorage?.subarray(offset, offset+2)
  ?:Float32Array(2)
```

# Swizzle

```
class Vec4(backingStorage: Float32Array?, offset: Int = 0) {

  var xy : Vec2
    get() = Vec2(storage)
    set(value) { Vec2(storage).set(value) }
  var xyz : Vec3
    get() = Vec3(storage)
    set(value) { Vec3(storage).set(value) }
```

# Mátrixok

- sor- vagy oszlopvektor?
- jobbról vagy balról szorzunk a mátrixszal?
  - ez az előzőből következik
- sor- vagy oszlopfolytonosan tároljuk az elemeket?
  - az előzőektől független
  - WebGL előírásnak meg kell felelni
- jobb- vagy balkezes a koordinátarendszer?
  - az előzőektől teljesen független
  - csak a forgatásmátrix felírásában számít
  - merre van a pozitív forgatás?

# Mátrixok

- **sor**- vagy oszlopvektor?
- **jobbról** vagy balról szorzunk a mátrixszal?
  - ez az előzőből következik
- sor- vagy **oszlopfolytonosan** tároljuk az elemeket?
  - az előzőektől független
  - WebGL előírásnak meg kell felelni
- **jobb**- vagy balkezes a koordinátarendszer?
  - az előzőektől teljesen független
  - csak a forgatásmátrix felírásában számít
  - merre van a pozitív forgatás?

# Mátrixműveletek

- teljesítmény kevésbé függ a paraméterek/kimenet módjától
  - több művelet/függvényhívás
- vektor-mátrix szorzás az a vektoron végrehajtott művelet
  - ugyanúgy, mint a többi
- mátrix-mátrix szorzás többnyire skálázás, elforgatás, eltolásmátrixokkal: erre spec függvények

# vektor-mátrix szorzás

```
val v = Vec4(1.0f, 2.0f, 3.0f, 1.0f)
val m = Mat4().                              // identity
        rotate(PI/2)                         // rotation

v *= m // v <= (-2, 1, 3, 1);
```

# Nincs 2x2, 3x3 mátrix? Sebaj!

```
val v = Vec2(1.0f, 2.0f)
val m = Mat4().
        translate(3.0f, 4.0f)

v.set( v.xy01 * m )
  // v <= (4, 6), the translated vector
```

# Nincs 2x2, 3x3 mátrix? Sebaj!

```
val v = Vec3(0.0f, 1.0f, 0.0f)
val m = Mat4().
        translate(3.0f, 4.0f).
        rotate(PI/2.0f)
v.set( v.xyz0 * m )// v <= (-1, 0, 0), as the
direction is invariant to translation
```

# A konstruktor transzponálja a kapott tömböt

```
val m = Mat4( // row-major!!!!!!
  1.0f, 0.0f, 0.0f, 0.0f,
  0.0f, 1.0f, 0.0f, 0.0f,
  0.0f, 0.0f, 1.0f, 0.0f,
  2.0f, 3.0f, 5.0f, 1.0f
) // a translation matrix
val m2 = Mat4().translate(-2.0f, -3.0f, -5.0f)
  // identity multiplied by
  // translation in the opposite direction

m * = m2 // m <= identity
```