# Raytracing tutorial, avagy sugárkövetés alapfokon

(Írta és rendezte: Farkas Ádám [wolfee]
Nyelvtani segítséget nyújtott: Ludányi Zsófi)

Ebben a tutorialban el fogom magyarázni a sugárkövetés alapjait, egy próbakódon is végiggyalogolunk, és a végén szép képeket fogunk előállítani.

Nem célom ezzel az útmutatóval/cikkel/írással a grafika tárgyat hallgatók helyett megoldani a feladatot – konkrét feladathoz kapcsolódó kódot nem is fogok mutatni –, a célom az, hogy az olvasó megértse a sugárkövetés alapjait, és ebből kiindulva egyéb érdekes programokat fejlesszen. Meg hogy át tudjon jutni a grafika sugárkövetős háziján. ☺

## Síkra vetődött

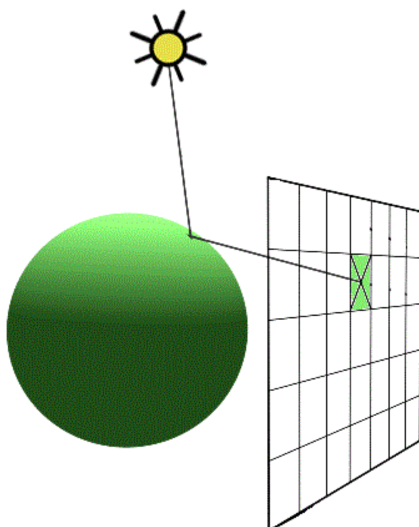
Kezdjük a dolgot egy kis elméleti alapozással, konkrétan azzal, hogy mit is csinálunk sugárkövetéskor! Ahhoz, hogy a valóságot közelíteni tudjuk, először meg kell – nagyjából – értenünk a valóság működését. Nem fogok kusza matematikai fizikai képleteket használni, csak amennyi feltétlenül muszáj.

Tegyük fel, hogy egy szobában vagyunk, ahol van egy darab fényforrás, pár gömbölyű tárgy (pl. labda), és a szoba falai. A fényforrás egységnyi idő alatt végtelen sok fotont emittál (ereszt ki magából), ezek a fénysugarak pedig elindulnak a térben, visszaverődnek a falakról, a gömbökről, miközben energiát veszítenek, és a fotonok egy része a verődések után a szemünkbe jut. Ez alakítja ki a szemünkben azt az érzést, hogy látunk. A szembeérkező fénysugarak energiája pedig a színérzetet alakítja ki. Tehát minél kisebb energiájú fény jut a szemünkbe, annál sötétebbnek látjuk a világot.

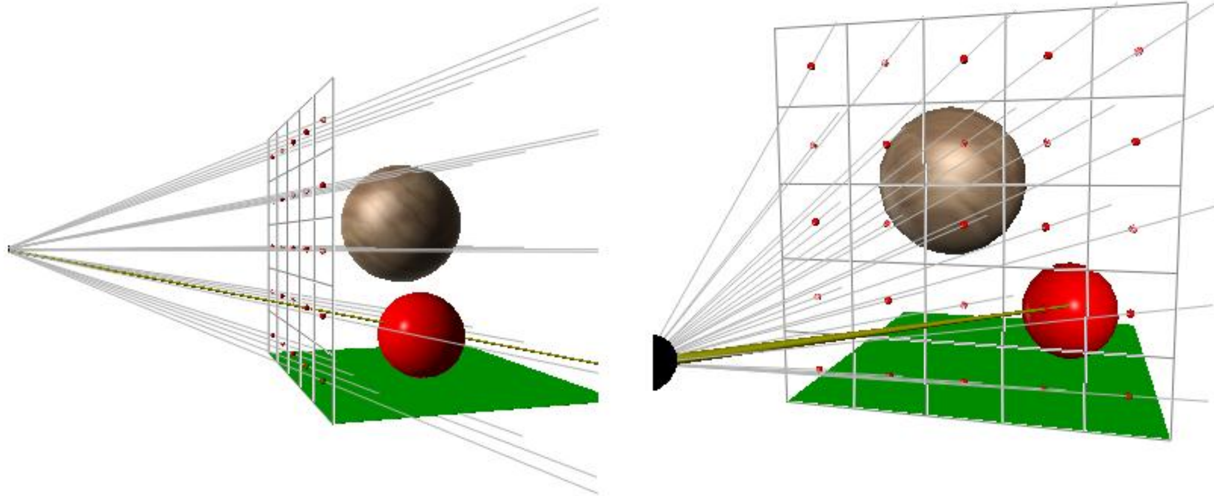
Ebből adódhatna a megoldás nulladik szintű közelítése: vegyünk fel a virtuális terünkben egy fényforrást, és eresszünk ki belőle… mennyi fotont is? A számítógép egy véges erőforrású eszköz. Ilyen szempontból nagyon is véges. Oké, akkor eresszünk ki nagyon-nagyon sokat, majd szép lassan leszámlálódik úgyis. A probléma a megoldással a következő: sokat számolunk fölöslegesen. Ahogy a valóságban, úgy itt is, a kilőtt fénysugarak csak nagyon kis része jutna el a szemünkbe. A többi egyszerűen nem a virtuális szemünkben fog landolni.

A 19. században élt egy Helmholtz nevű német úriember, orvos-fizikus, aki rájött, hogy a fény iránya megfordítható. Tehát ha van egy tükröm , mindegy, hogy a jobb oldalán van a szemem, és a bal oldalán van az égő, vagy fordítva, mindkét esetben ugyanúgy ki tudom égetni a retinámat. Ebből következik, hogy a fény visszaverődése a fény beérkezési szögétől függ. Ez azért nagyszerű hír, mert így mi is meg tudjuk fordítani a sugárkövetésünket. Tehát indítsunk ki a fénysugarakat a szemünkből, és kövessük vissza az útját egészen a fényforrásig!

Na de mennyit? És merre? És hogyan? Ezek a kérdések vetődhetnek fel, ha valaki olyan vadságokat mond, hogy világítsunk a szemünkkel. Ennek megértéséhez kicsit nyúljunk vissza az első megközelítéshez, amely jobban illusztrálja a valóságot. Tehát tegyük fel, hogy van egy fényforrásunk, meg egy szobánk. És mi egy 600×600 pixeles képet akarunk róla csinálni. Ezt el tudjuk érni, ha a szobába belenyomunk egy 600×600-as ernyőt, amelyre felfogjuk a fotonokat. Tulajdonképpen úgy működhetne, mint egy fényképezőgép, ahogy a lenti ábrán is látszik.



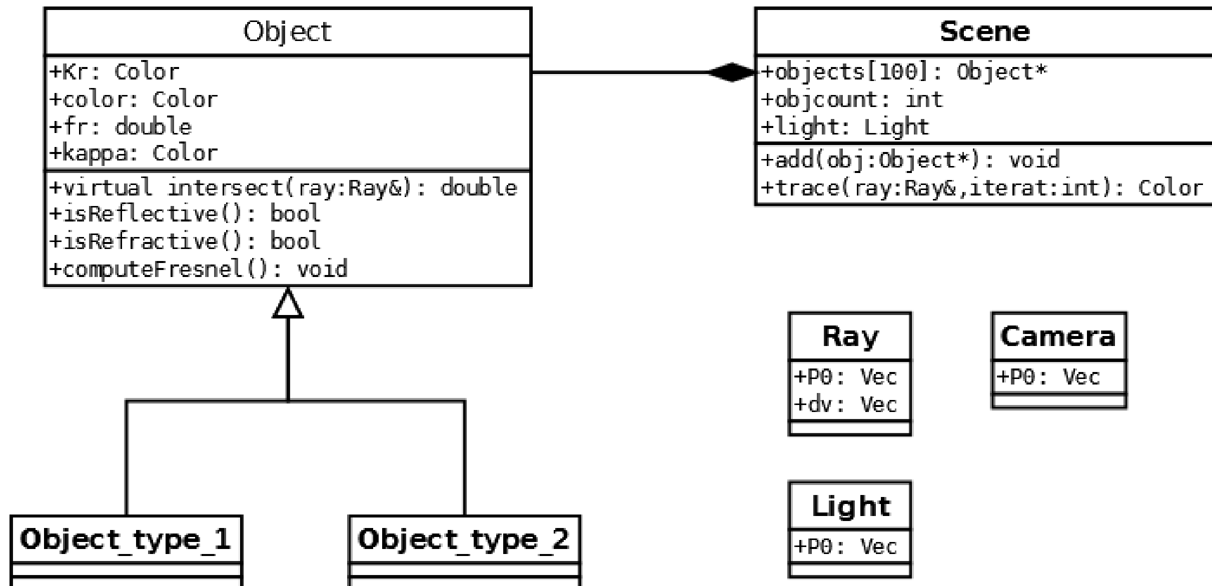
Csináljuk tehát a következőt: vegyünk egy kamerát (virtuális szemünket), egy síkot, amelyre leképezzük a képet (ezt osszuk fel 600×600 pixelre), és a kamerából a sík egyes pontjain keresztül lőjünk sugarakat a színtérbe!



Ha ennek a működését sikerült megérteni, akkor hatalmas lépést tettünk a sugárkövetés megértése felé!

## Nekem Szebi tanította a progkettőt!

Az OOT – ha tetszik, ha nem – jó dolog. Viszont ebben a tutorialban a kevesebb kód érdekében az adatrejtést nem fogjuk kihasználni, majdnem minden publikus lesz. Legyen a programunknak a következő a felépítése:




Feltételezzük – én feltételezem –, hogy létezik egy Color osztály, amelynek van R, G, B double számhármasa, valamint megfelelő műveletei, illetve létezik egy Vec vektor osztály is, x, y, z double hármassal, és megfelelő műveletekkel.

Mit is csinálunk tulajdonképpen? Legyen egy Scene osztályunk, amely tároljon egy tömbben Objectre mutató pointereket. Mivel Object bárhol helyettesíthető a leszármazottaival, ezért megcsinálhatjuk, hogy Object_type_1-et és Object_type_2-t teszünk a helyére.

Jól látható a heterogén kollekció. Aki nem látja, nos, azzal nem tudok mit kezdeni.

Csináljuk meg az osztályokat, és benne függvényeket, üres törzzsel! Segítségként megadom néhány alap dolog implementációját, amiket ujjgyakorlatként bármikor meg kell tudni írni annak, aki elvégezte a progkettőt.

```cpp
#include <math.h>
#include <iostream>

using namespace std;

const double epsilon = 1e-4; // a hiba mértéke, a számítási hibák elkerülése miatt
#define PI 3.1415926536 // elég pontos PI
#define DMAX 5 // a rekurzió maximális mélysége

// Color osztály, néhány alap metódussal.
class Color
{
public:
	double r;
	double g;
	double b;


	Color(double gr = 0.0, double gg = 0.0, double gb = 0.0)
	{
		r = gr;
		g = gg;
		b = gb;
	}

	Color(Color& theOther)
	{
		r = theOther.r;
		g = theOther.g;
		b = theOther.b;
	}

	Color& operator=(Color& theOther)
	{
		r = theOther.r;
		g = theOther.g;
		b = theOther.b;
		return *this;
	}

	Color operator+(Color& theOther)
	{
		Color ret;
		ret.r = r + theOther.r;
		ret.g = g + theOther.g;
		ret.b = b + theOther.b;
		return ret;
	}

	Color operator/(double d)
	{
		Color ret;
		ret.r = r / d;
		ret.g = g / d;
		ret.b = b / d;
		return ret;
	}
};

// Vektor osztály néhány alap metódussal
class Vec
{
public:
	double x, y, z;

	Vec(double x0 = 0, double y0 = 0, double z0 = 0)
	{
		x = x0;
		y = y0;
		z = z0;
	}

	Vec operator+(const Vec &b) const
	{
```

```
        return Vec(x + b.x, y + b.y, z + b.z);
    }

    Vec operator-(const Vec &b) const
    {
        return Vec(x - b.x, y - b.y, z - b.z);
    }

    Vec operator*(double b) const
    {
        return Vec(x * b, y * b, z * b);
    }

    Vec operator/(double b) const
    {
        return Vec(x / b, y / b, z / b);
    }

    Vec mult(const Vec &b) const
    {
        return Vec(x * b.x, y * b.y, z * b.z);
    }

    Vec& norm()
    {
        return *this = *this * (1 / sqrt(x * x + y * y + z * z));
    }

    double length()
    {
        return sqrt(x * x + y * y + z * z);
    }

    double dot(const Vec &b) const
    {
        return x * b.x + y * b.y + z * b.z;
    }

    Vec operator%(Vec &b)
    {
        return Vec(y * b.z - z * b.y, z * b.x - x * b.z, x * b.y - y * b.x);
    }
};

// Sugár osztály
class Ray
{
public:
    Vec P0; // kezdőpont
    Vec dv; // irányvektor
};

// Fény osztály
class Light
{
public:
    Color color; // színe
    Vec P0; // helyzete
};

// Kamera osztály
class Camera
{
public:
    Vec P0; // kamera helyzete
};
```

```
// Általános objektum osztály
class Object
{
public:
        Color color; // szín
        Color Kr; // Fresnel-együttható (számolandó)
        Color fr; // törési tényező
        Color kappa; // Kioltási tényező
        bool isReflective; // tükröző felület?
        bool isRefractive; // törő felület?


        virtual double intersect(Ray& ray) = 0; // metsző függvény. Tisztán virtuális, tehát minden
leszármazottnak meg kell valósítania
        virtual Vec getNormal(Vec& intersect) = 0; // a felületi normálist adott pontban lekérdező
függvény

        void computeFresnel(double costheta) // Fresnel-együtthatót számoló függvény
        {
        }
};

// Színtér objektum
class Scene
{
public:
        Object* objects[100]; // Általános objektumokra mutató pointerek (leszármazottakat fogunk
beletenni)
        int objcount; // tárolt objektumok száma
        Light light; // fény a színtérbe

        // konstruktor
        Scene()
        {
                objcount = 0;
        }

        // színtérhez adás
        void add(Object* object)
        {
                objects[objcount] = object;
                objcount++;
        }

        // a sugárkövetőnk lelke
        Color Trace(Ray& ray, int iterat)
        {
        }
};

int main()
{
        return 0;
}
```

## Az első laszti

Nos, mint mondtam a legelején, gömbökkel fogunk foglalkozni, mivel ennek a testnek viszonylag egyszerű az egyenlete, mégis nagyon jól látszik rajta a tükröződés, fénytörés.

### Egy kis matematika az elejére:

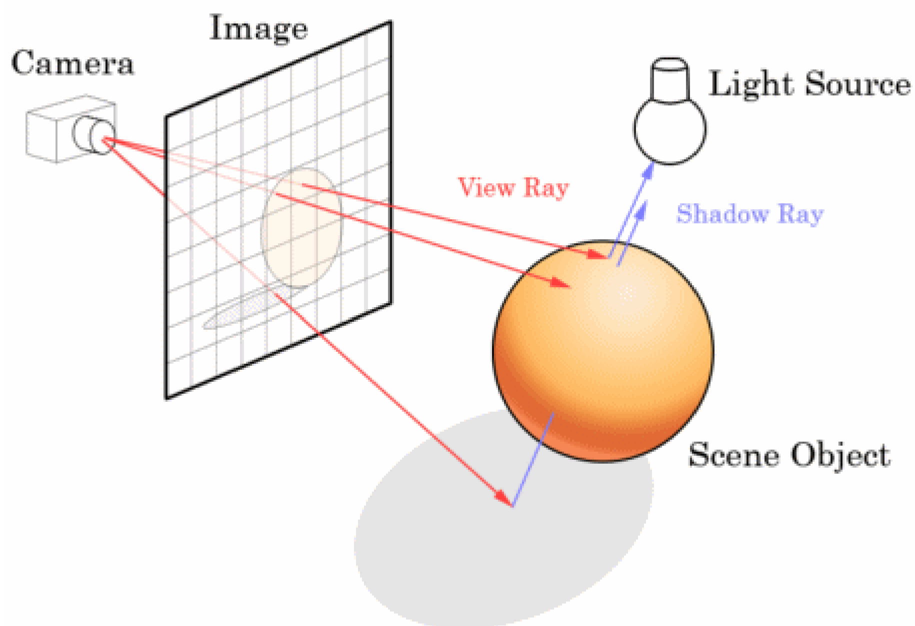
Az OpenGL-t most teljes mértékben kihagyjuk a buliból, csak a kép megjelenítésére fogjuk használni. Tehát a -1,1-es koordinátarendszert el lehet felejteni, mi rendes koordinátarendszerben fogunk dolgozni, ahol a képünk 600×600 egység lesz (ekkora lesz a már korábban megbeszélt ernyőnk).

A gömb a következő alaptulajdonságokkal rendelkezik matematikailag:

Van neki egy középpontja, és egy sugara. Valamint van neki egy tök jó egyenlete, ami így néz ki:

$(x - x0)^2 + (y - y0)^2 + (z - z0)^2 \leq r^2$

Tulajdonképpen mit is akarunk? Beletesszük a gömböt a színtérbe, a kameránkból sugarakat indítunk az ernyőnkön keresztül, és ha eltaláljuk vele a gömböt, akkor kirajzoljuk a gömb színét az ernyő megfelelő pontjára.




A fenti ábrából egyelőre képzeletben hagyjuk el a fényforrást!

A sugarunknak szüksége van egy kezdőpontra és egy irányra. A kezdőpontja (P0) legyen az ernyő megfelelő pontja, az iránya (dv) pedig a kamerából a kezdőpontba mutató vektor. Ekkor az egyenes egy tetszőleges pontja a P1 = P0 + t * dv egyenlettel megadható, ahol „t” egy tetszőleges valós paraméter.

A gömb egyenletéből és a sugárról tudott információk alapján le lehet vezetni, hogy hogyan kell kiszámolni a „t”-t, hogy a sugarunk elmetssze a gömböt. Én most ettől eltekintek, higgyétek el

nekem, hogy amit leírok, az úgy van. Egyébként a $(x - x0)^2 + (y - y0)^2 + (z - z0)^2 - r^2 = P0 + t * dv$ egyenletet kell megoldani t-re. Ha megvan a sugarunkhoz tartozó „t”, akkor ki tudjuk azt is számolni, hogy a térben hol van a metszéspont. Ezt majd a későbbiekben ki fogjuk számolni. De most lépjünk tovább! Az objektumvázlatainkhoz a következőt adjuk hozzá:

```
// Gömb objektum
class Sphere : public Object
{
public:
        Vec origo; // középpont
        double radius; // sugár

        // kontruktor. Egy középpontot és egy sugarat vár paraméterként.
        Sphere(Vec o = 0, double r = 1)
        {
                origo = o;
                radius = r;
        }

        // Metszi-e függvény. ha nem mentszünk, -1 -et adunk vissza.
        double intersect(Ray& ray)
        {
                double dx = ray.dv.x;
                double dy = ray.dv.y;
                double dz = ray.dv.z;
                double x0 = ray.P0.x;
                double y0 = ray.P0.y;
                double z0 = ray.P0.z;
                double cx = origo.x;
                double cy = origo.y;
                double cz = origo.z;
                double R = radius;

                double a = dx * dx + dy * dy + dz * dz;
                double b = 2 * dx * (x0 - cx) + 2 * dy * (y0 - cy) + 2 * dz * (z0 - cz);
                double c = cx * cx + cy * cy + cz * cz + x0 * x0 + y0 * y0 + z0 * z0 - 2 * (cx * x0 +
cy * y0 + cz * z0) - R * R;

                double d = b * b - 4 * a * c;

                if(d < 0)
                {
                        return -1.0;
                }

                double t = ((-1.0 * b - sqrt(d)) / (2.0 * a));

                if(t > epsilon) // ha nem csak számolási hibát vétettünk...
                {
                        return t;
                }
                else
                {
                        return 0.0; // ha a 0 epsilon sugarú környezetében van az érték, az nagy
valószínűséggel számolási hiba
                }
        }

        // adott pontban vett normálist visszaadó függvény
        Vec getNormal(Vec& intersect)
        {
                return (intersect - origo).norm();
        }
};
```

Ez a gömb implementációja jelenleg.

A Trace függvényünket egyelőre a következő módon írjuk meg:

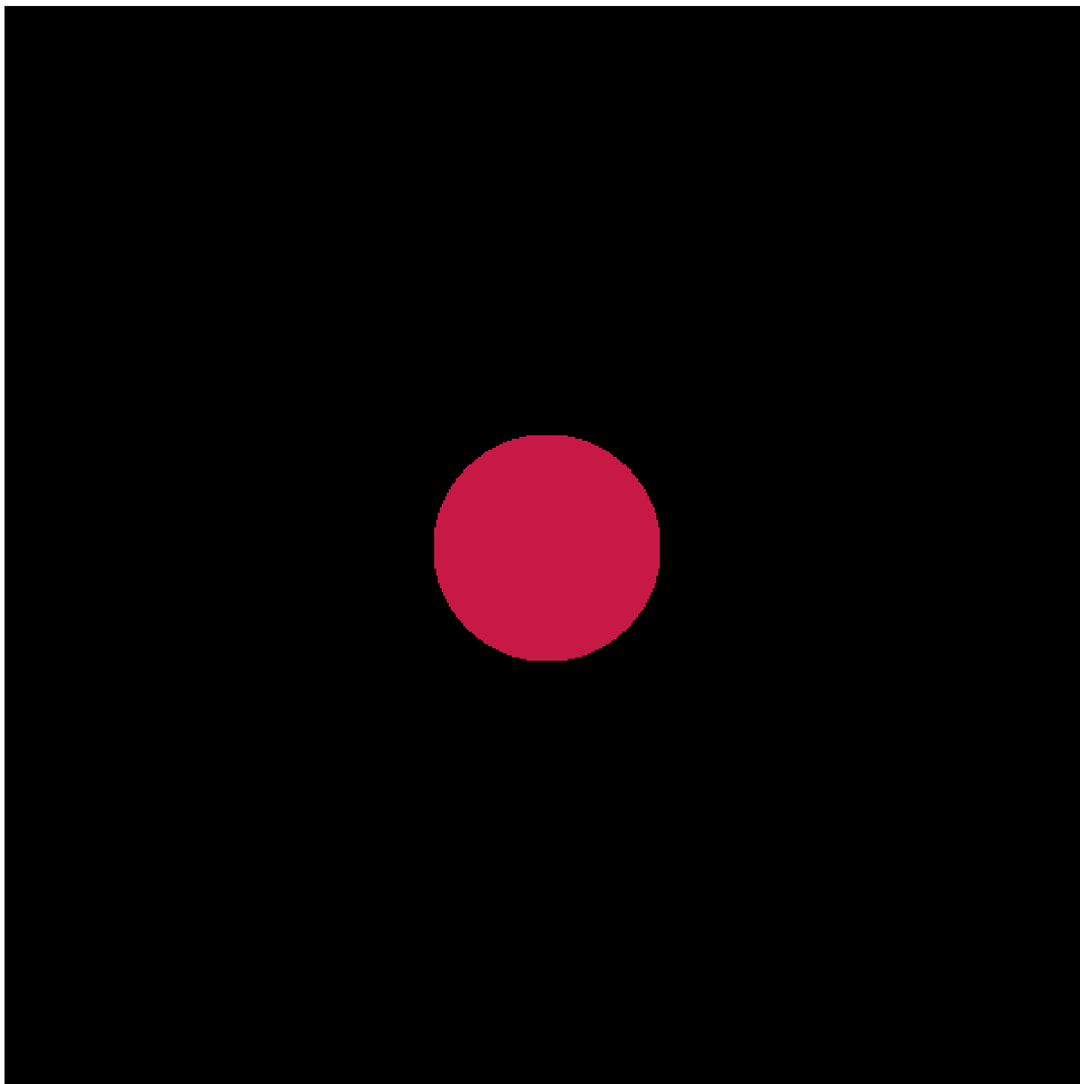
```
Color Trace(Ray& ray, int iterat)
{
	Color color; // a majdani visszatérési szín
	color.r = color.g = color.b = 0; // alapból feketére állítjuk
	double t; // az intersect függvény megoldása
	int index = -1; // objektumok indexeléséhez

	for(int i = 0; i < objcount; i++)
	{
		t = -1.0; // kezdeti értékre állítjuk (-1: nem volt metszés)
		t = objects[i]->intersect(ray); // megpróbáljuk elmetszeni
		if(t >= 0) // ha sikeresen elmetszettük
		{
			index = i; //megjegyezzük az indexet
		}
	}
	if(t >= 0) // ha sikeresen elmetszettük valakit
	{
		color = objects[index]->color; // az elmetszettnek a színét vesszük
	}
	return color; // visszaadjuk a színt
}
```

A main függvényünk pedig a következőképpen nézzen ki:

```
int main()
{
	int width = 600; // kép szélessége
	int height = 600; // kép magassága
	int w2 = width / 2; // szélesség / 2
	int h2 = height / 2; // magasság / 2

	Color** wo; // Color tömb. Ez fogja játszani az ernyő szerepét.
	wo = new Color*[width];
	for(int i = 0; i < width; i++)
	{
		wo[i] = new Color[height];
	}

	Camera cam; // kamera objektum
	cam.P0.x = 0;
	cam.P0.y = 0;
	cam.P0.z = -500;

	Scene scene; // színtér
	scene.add(new Sphere(Vec(0, 0, 300), 100)); // egy új gömb hozzáadás
	scene.objects[0]->color = Color(200, 25, 70); // az új elem színének beállítása

	// sugarak előállítása és a sugárkövetés elindítása
	// végigpásztázzuk az egész ernyőnket, és a kamerából az ernyő diszkrét pontjain át
	// sugarakat lövünk a színtérbe. a sugarak kezdőpontja az ernyő megfelelő pontja, a
	// sugarak irányvektora pedig a kamera és a sugár kezdőpontja közötti vektor
	for(int i = 0; i < height; i++)
	{
		for(int j = 0; j < width; j++)
		{
			Ray ray;
			ray.P0 = Vec((j - w2), (i - h2), 0);
			ray.dv = (ray.P0 - cam.P0);
			ray.dv.norm();
			wo[i][j] = scene.Trace(ray, 0);
		}
	}

	// egy PPM fájlba írom ki az eredményt, amit pl az irfanView programmal lehet megnézni
	FILE* f;
	f = fopen("myray.ppm", "w");
	fprintf(f, "P3\n%d %d\n255\n ", width, height);
```

```
	for(int i = 0; i < height; i++)
	{
		for(int j = 0; j < width; j++)
		{
			fprintf(f, "%d ", min((unsigned int)(wo[i][j].r + 0.5), (unsigned)255));
			fprintf(f, "%d ", min((unsigned int)(wo[i][j].g + 0.5), (unsigned)255));
			fprintf(f, "%d ", min((unsigned int)(wo[i][j].b + 0.5), (unsigned)255));
		}
	}
	fclose(f);

	// rendet rakunk magunk után
	for(int i = 0; i < width; i++)
	{
		delete[] wo[i];
	}
	delete[] wo;

	return 0;
}
```

Ha eddig eljutottunk, akkor nagy valószínűséggel egy lila korongot kell látnunk a képernyőn. Higgyétek el, hogy ez egy sugárkövetett gömb!



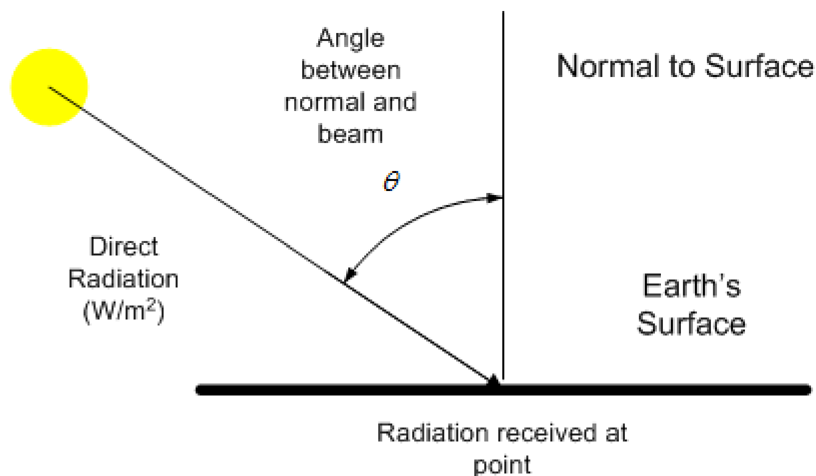
## Szétszórt árnyakban

Az eddig megjelenített napkorong azért – valljuk be – kicsit karcsú. Jó lenne látni, hogy ez tényleg egy gömb. Ezért ebben a fejezetben a diffúz árnyalásról, inglisül diffuse shadinggel fogunk foglalkozni.

Mire is van szükségünk? Kell egy jó árnyalási modell, amitől a körünk gömbölyödni fog. A valóságnak egy viszonylag jó megközelítése a Labmert-törvény, más nevén a diffúz árnyalás. Az árnyalásnak van egy matematikailag korrekt egyenlete, mégpedig a

$L_\lambda = L_\lambda^{in} * k_{d,\lambda} * \cos\theta'$

Ez az egyenlet amennyire fellengzősen hangzik, jelen formájában pont annyira érthetetlen. Tegyük fel, hogy van egy matt felületünk, például az asztal lapja, amely még véletlenül sincsen lelakkozva, lecsokizva, lesörözve stb. Ha fogunk egy elemlámpát/mobilt, és a sötét szobában megvilágítjuk az asztalt, figyelve arra, hogy mindig más szögből érje a fény a lapját, akkor két dolog fog megtörténni: egyrészt szüleink sajnálkozó arcát fogjuk látni, miközben próbáljuk magyarázni, hogy ez egy feladathoz kell, másrészt pedig észrevesszük, hogy az asztallap színe függött a megvilágítás irányától. Ezek után ki lehet találni, hogy a fenti egyenletből mi a cosθ'. Igen, ez a felületi normálishoz mért beérkezési szög koszinusza, amely mindig -1 és 1 közötti szám, de mi csak a 0…1 tartományát vesszük figyelembe. A többi betű jelentése a következő: $L_\lambda$: a kimenő fény intenzitása; $L_\lambda^{in}$: a bejövő fény intenzitása; $k_{d,\lambda}$: az anyag diffúz fénytulajdonsága, azaz a színe. A bejövő fény intenzitása a beérkező fény színének feleltethető meg, a kilépő fényintenzitás pedig a kimeneti színnek. Már csak a beérkezési szögről nem beszéltem: ez a metszési pontban vett felületi normális, és a fénysugárból a beérkezési pontba menő vektor skaláris szorzata.




Ha az elméletet nagyjából sikerült feldolgozni, akkor nézzük meg, hogy ez kódban hogyan fog kinézni!

A Scene osztály konstruktorát a következő módon egészítsük ki (a fény beállításával):

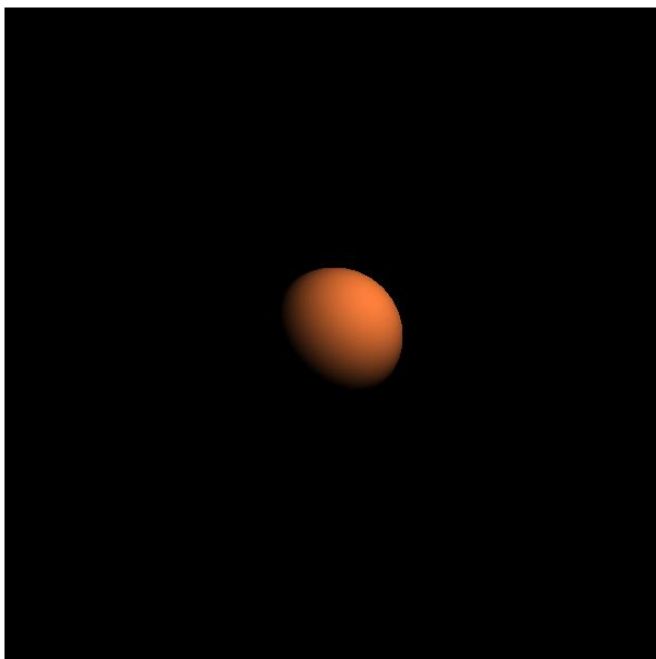
```
Scene()
{
        light.color = Color(255, 128, 60);
        light.P0 = Vec(200, -200, 0);
        objcount = 0;
}
```

A main függvényben ne lila, hanem fehér gömböt hozzunk létre:

```
scene.objects[0]->color = Color(255, 255, 255); // az új elem színének beállítása
```

És végül a trace függvényt a következő módon módosítsuk:

```
Color Trace(Ray& ray, int iterat)
{
	Color color; // a majdani visszatérési szín
	color.r = color.g = color.b = 0; // alapból feketére állítjuk
	double t; // az intersect függvény megoldása
	int index = -1; // objektumok indexeléséhez

	for(int i = 0; i < objcount; i++)
	{
		t = -1.0; // kezdeti értékre állítjuk (-1: nem volt metszés)
		t = objects[i]->intersect(ray); // megpróbáljuk elmetszeni
		if(t >= 0) // ha sikeresen elmetszettük
		{
			index = i; //megjegyezzük az indexet
		}
	}

	if(t >= 0) // ha sikeresen elmetszettük valakit
	{
		Vec intersectPoint; // a metszéspont
		intersectPoint = ((ray.P0) + (ray.dv * t)); // ez lesz a pontos helye
		Vec normal = objects[index]->getNormal(intersectPoint); // a felület normálisa
		Ray iRay; // egy sugár, ami a metszéspontból a fény felé fog majd nézni
		iRay.P0 = intersectPoint + normal * 0.01; // kicsit "arrébb" húzzuk, hogy biztos ne
legyen a gömbünkben a sugár kezdőpontja
		iRay.dv = light.P0 - intersectPoint; // beállítjuk az irányvektorát
		iRay.dv.norm(); // normalizálunk
		double factor = normal.dot(iRay.dv); // ez a cos(theta)
		if(factor < 0) // csak a 0..1 tartományt vesszük figyelembe
		{
			factor = 0;
		}
		color = objects[index]->color; // az elmetszettnek a színét vesszük.

		// és vesszük a Lambert-törvény által meghatározott árnyalást
		// a fény egyes komponenseinek értékét azért kellett 255-tel osztani, mert én
		// a 0...255 értékekkel szeretek dolgozni, viszont a képletbe egy 0...1 értéknek
		// kell kerülnie
		color.r = color.r * (light.color.r / 255.0) * factor;
		color.g = color.g * (light.color.g / 255.0) * factor;
		color.b = color.b * (light.color.b / 255.0) * factor;
	}
	return color; // visszaadjuk a színt.
}
```



# Tyúk vagy tojás?

Bizony a sorrendiség sehol sem mindegy. Tegyünk be a színterünkbe még egy gömböt, lehetőleg az első gömb mögé valahova (z koordinátája legyen nagyobb), és gondoljuk végig, hogy mi fog történni!

A trace függvényünk szépen elkezdi a betétel sorrendjében elmetszegetni az objektumainkat, és a legutoljára megtaláltat fogja nekünk visszaadni. Magyarán jelenleg nem az objektumok Z-sorrendje (mélység szerinti sorrendje) számít, hanem az, hogy mikor tettük bele őket a színtérbe. Ez így nagyon nem jó. A megoldást az fogja jelenteni, ha elmetsszük az összes objektumot a színtérben, és megjegyezzük a legkisebb pozitív „t”-hez tartozó indexet.

Ez az én kódomban a következő módon jelenik meg:

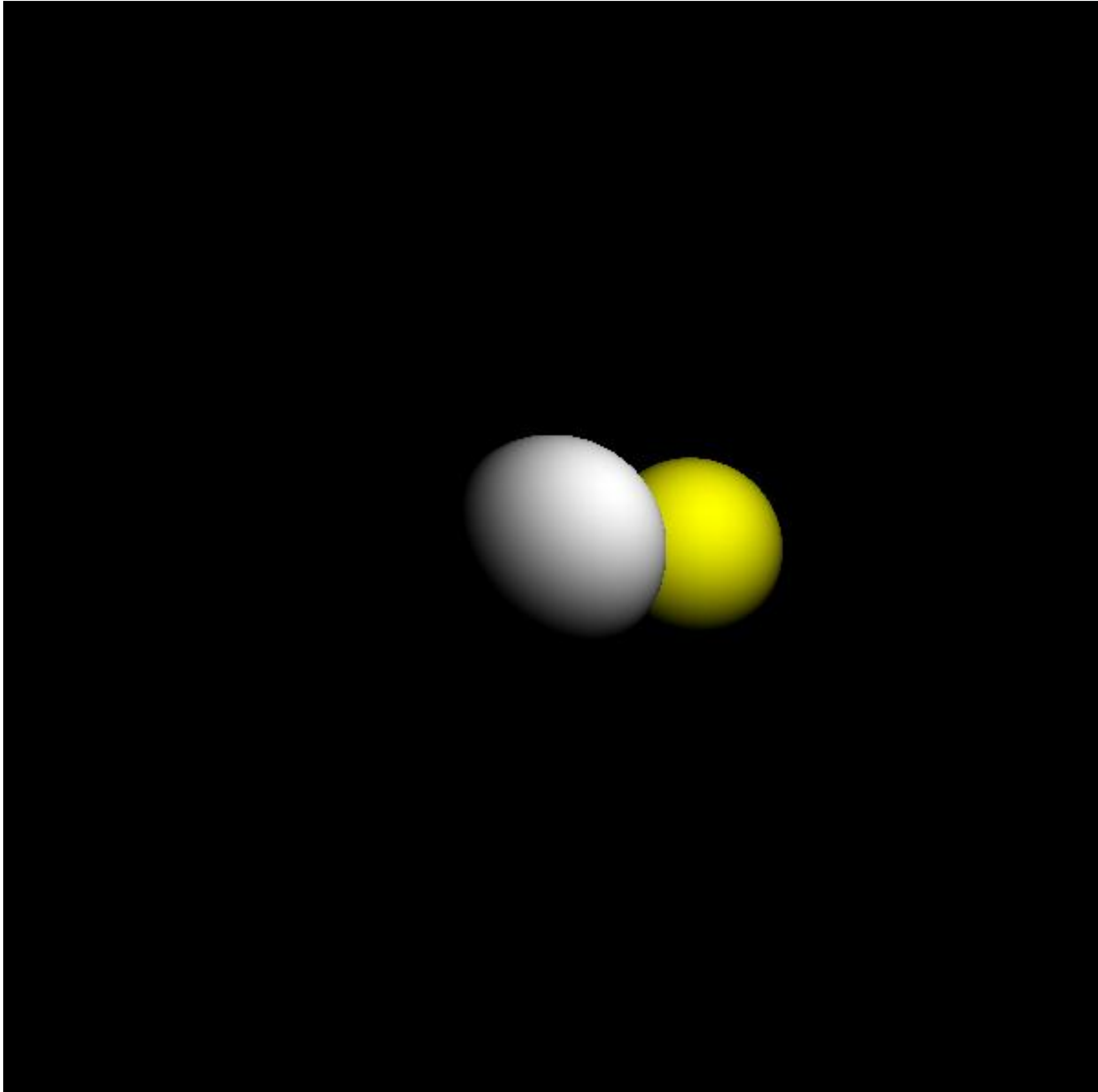
A main-ben adjunk még egy gömböt a színtérhez

```
Scene scene; // színtér
scene.add(new Sphere(Vec(0, 0, 300), 100)); // egy új gömb hozzáadás
scene.objects[0]->color = Color(255, 255, 255); // az új elem színének beállítása

scene.add(new Sphere(Vec(150, 0, 500), 100));
scene.objects[1]->color = Color(255, 255, 0);
```

A trace függvényt pedig a következő szerint módosítottam:

```
Color Trace(Ray& ray, int iterat)
{
        Color color; // a majdani visszatérési szín
        color.r = color.g = color.b = 0; // alapból feketére állítjuk
        double t = -1; // az egyenlet megoldása
        double mint = 99999999999999999999.9999999999999999; // a minimális "t"-t egy jó nagy számra
inicializáljuk
        int minindex = -1; // a mint-hez tartozó index
        for(int i = 0; i < objcount; i++)
        {
                t = objects[i]->intersect(ray); // elmetsszük
                if(t > epsilon) // csak a pozitív megoldások érdekelnek
                {
                        if(t < mint) // az olyanok, amik közelebb vannak, mint az eddig megtalált
legközelebbi
                        {
                                mint = t; // átállítjuk
                                minindex = i; // a minimálisra vonatkozó információkat
                        }
                }
        }

        if(minindex >= 0) // ha sikeresen elmetszettük valakit
        {
                t = mint; // a t legyen a minimálisnak talált
                Vec intersectPoint; // a metszéspont
                intersectPoint = ((ray.P0) + (ray.dv * t)); // ez lesz a pontos helye
                Vec normal = objects[minindex]->getNormal(intersectPoint); // a felület normálisa
                Ray iRay; // egy sugár, ami a metszéspontból a fény felé fog majd nézni
                iRay.P0 = intersectPoint + normal * 0.01; // kicsit "arrébb" húzzuk, hogy biztos ne
legyen a gömbünkben a sugár kezdőpontja
                iRay.dv = light.P0 - intersectPoint; // beállítjuk az irányvektorát
                iRay.dv.norm(); // normalizálunk
                double factor = normal.dot(iRay.dv); // ez a cos(theta)
                if(factor < 0) // csak a 0..1 tartományt vesszük figyelembe
                {
                        factor = 0;
                }
                color = objects[minindex]->color; // az elmetszettnek a színét vesszük.
```

```
            // és vesszük a Lambert-törvény által meghatározott árnyalást
            // a fény egyes komponenseinek értékét azért kellett 255-tel osztani, mert én
            // a 0...255 értékekkel szeretek dolgozni, viszont a képletbe egy 0...1 értéknek
            // kell kerülnie
            color.r = color.r * (light.color.r / 255.0) * factor;
            color.g = color.g * (light.color.g / 255.0) * factor;
            color.b = color.b * (light.color.b / 255.0) * factor;
        }
        return color; // visszaadjuk a színt
}
```



# Görbe tükör

Ha eddig eljutott valaki, akkor innen már nem nehéz, ezt tudom ígérni. Ebben a fejezetben a tükröző felületekkel fogunk foglalkozni, és megértjük, hogy miért is hívják a rekurzív sugárkövetést rekurzívnak.

## Kis kitérő: a sík

Hogy a munkánk látványosabb legyen, építsünk egy sík osztályt, amit majd tudunk talajként használni.

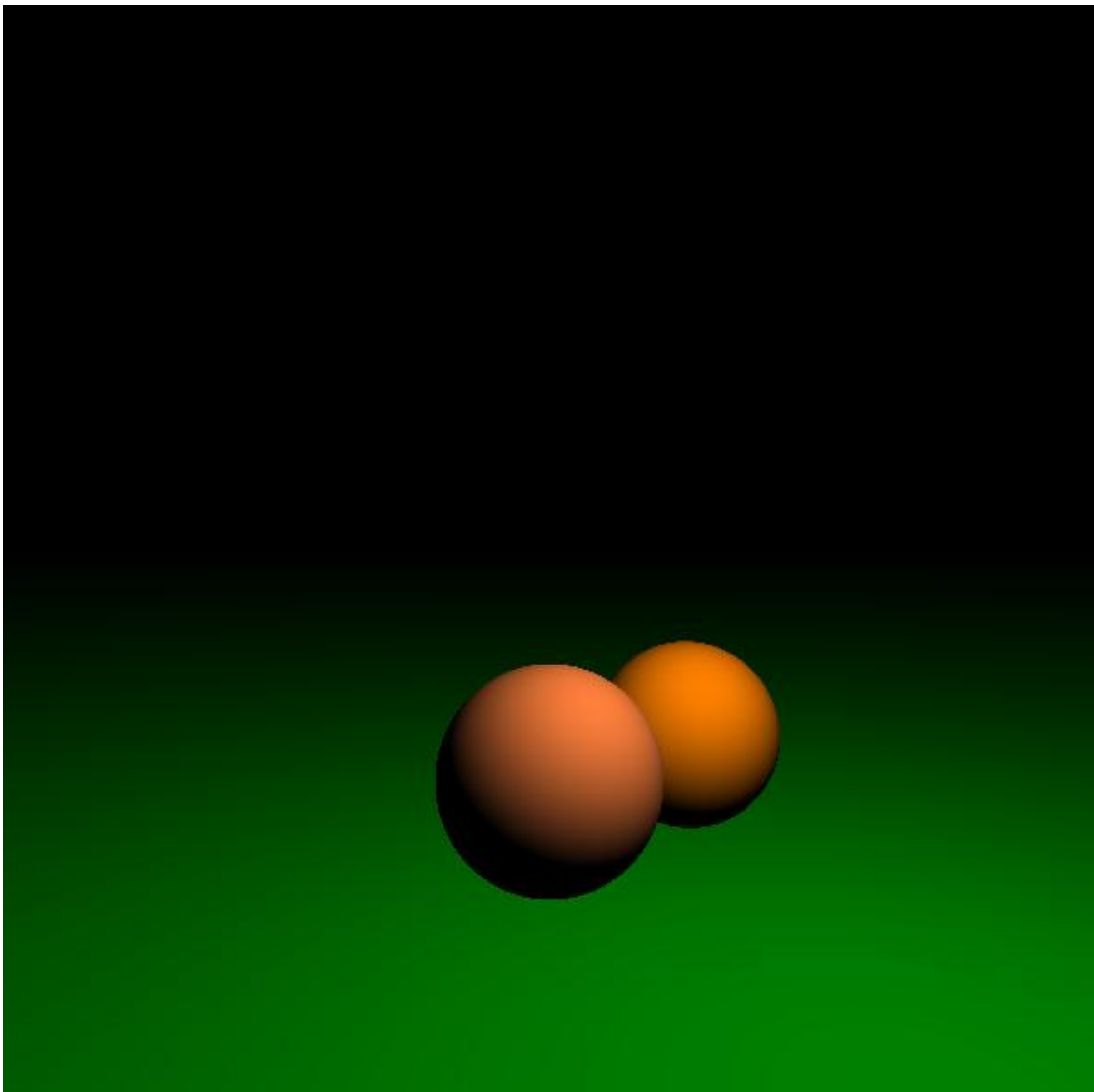
Én egy kicsit előre dolgoztam, szóval az én sík osztályom így néz ki:

```
// Sík osztály
class Plane : public Object
{
public:
	Vec point; // a sík egy pontja
	Vec normal; // a sík normálvektora

	// konstruktor
	Plane(Vec p, Vec n)
	{
		point = p;
		normal = n;
	}

	// metszi-e
	double intersect(Ray& ray)
	{
		double d = normal.dot(ray.dv);
		{
			if(d == 0.0)
			{
				return -1.0;
			}

			double nx = normal.x;
			double ny = normal.y;
			double nz = normal.z;
			double Psx = point.x;
			double Psy = point.y;
			double Psz = point.z;

			double dvx = ray.dv.x;
			double dvy = ray.dv.y;
			double dvz = ray.dv.z;
			double Pex = ray.P0.x;
			double Pey = ray.P0.y;
			double Pez = ray.P0.z;

			double t = -1.0 * ((nx * Pex - nx * Psx + ny * Pey - ny * Psy + nz * Pez - nz
* Psz) / (nx * dvx + ny * dvy + nz * dvz));

			if(t > epsilon) return t;
			if(t > 0) return 0;

			return -1;
		}
	}

	// normálvektor
	Vec getNormal(Vec&)
	{
		return normal;
	}
};
```

A main függvényben adjunk is hozzá egy talajt a színtérhez, olyan helyre, hogy a talajunk a vetítősíkunkat a 600×600-as ablak alján metssze. Ehhez az kel, hogy függőleges irányban 300 egységgel lefele toljuk a síkot. Ezen kívül a gömbjeinket tegyük le a síkra, ne a levegőben lógjanak! Ez kódban így valósul meg:

```
Scene scene; // színtér
scene.add(new Sphere(Vec(0, 200, 300), 100)); // egy új gömb hozzáadás
scene.objects[0]->color = Color(255, 255, 255); // az új elem színének beállítása

scene.add(new Sphere(Vec(150, 200, 500), 100));
scene.objects[1]->color = Color(255, 255, 0);

scene.add(new Plane(Vec(0, 300, 0), Vec(0, -1, 0)));
scene.objects[2]->color = Color(0, 255, 0);
```

Ezek után valami ilyesmi képet kéne kapnunk:



Ezzel vége ennek a kitérőnek.

## Nagy kitérő: Fresnel? Nem, sima nátha.

Ha emlékszünk az Object osztály felépítésére, akkor észrevehetjük, hogy volt benne kappa meg Kr változó, törésmutatót jelentő fr, illetve egy computeFresnel függvény. Ha valaki nem emlékszik rá, akkor olvasson vissza ☺

A lányok tudják, hogy az ideális tükör olyan, ami tökéletesen adja vissza az ő karcsú, hosszú combú, bőrhibától mentes alakjukat. Ilyen tükör sajnos a valóságban nincs. Kicsit komolyabban véve a dolgot: az ideális tükör olyan, ami a fény minden hullámhosszán ugyanazt az intenzitást veri vissza, azaz egyáltalán nincs elnyelése, a kilépő fény színe ugyanaz, mint a belépőé. Ez a valóságban nincs így, minden tükröző anyag elnyel valamennyit. Azt, hogy ezt mennyire teszi, az úgynevezet Fresnel-együtthatóval fejezhetjük ki. Egy anyag adott pontban vett Fresnel-együtthatója függ a fény belépési szögétől, illetve az anyag törésmutatójától. Hogy az életünk cseppet se legyen egyszerű, ezért a fizikusok kitalálták, hogy a törésmutató komplex szám, ezért mi azt az fr + kappa * j alakban fogjuk használni. A Fresnel egyenlettel számolni kifejezetten nehéz, de van egy jó közelítése, amit Lazányi-Schlick képletnek hívnak:

$$Kr(r) \sim [\, (fr(r) - 1)^2 + (kappa(r)^2) + (1 - \cos\theta)^5 * 4fr(r)\, ] \,/\, [(fr(r) + 1)^2 + (kappa(r)^2)\, ]$$

Ebben az egyenletben az r a vörös színkomponenst jelöli. Értelemszerűen ugyanígy kiszámolható a kék és a zöld színtartományra is a Fresnel-együttható.

Az alább megadom a Fresnel-együtthatók számolásának egy lehetséges megvalósítását (Object osztály):

```
void computeFresnel(double costheta) // Fresnel-együtthatót számoló függvény
{
	Kr.r = ((pow((fr.r - 1.0), 2)) + (pow(kappa.r, 2)) + (pow((1.0 - costheta), 5)) * (4 * fr.r)) / ((pow((fr.r + 1.0), 2)) + (pow(kappa.r, 2)));
	Kr.g = ((pow((fr.g - 1.0), 2)) + (pow(kappa.g, 2)) + (pow((1.0 - costheta), 5)) * (4 * fr.g)) / ((pow((fr.g + 1.0), 2)) + (pow(kappa.g, 2)));
	Kr.b = ((pow((fr.b - 1.0), 2)) + (pow(kappa.b, 2)) + (pow((1.0 - costheta), 5)) * (4 * fr.b)) / ((pow((fr.b + 1.0), 2)) + (pow(kappa.b, 2)));
}
```

## A rekurzi jó!

A rekurzív sugárkövetés ötlete a tükröző felületek megjelenésével terjedt el. Az ötlet nagyon egyszerű: vegyük a beérkezési pontot, majd ne az elmetszett test színét adjuk vissza, hanem a beérkezési pontból indítsunk egy új sugarat, és ennek a visszatérési értékét adjuk hozzá az eredeti színhez! Mivel nem akarjuk, hogy a rekurziónk több tükröző felület esetén a végtelenbe tartson, ezért meghatározunk egy iterációs mélységet, amit a kódban DMAX-szal jelölünk. Próbáljuk meg jelen esetben a kódból megérteni, hogy mi is történik a rekurzió és a visszaverődés folyamán:

Először a main függvényben állítsuk jól be a paramétereket

```
Scene scene; // színtér
scene.add(new Sphere(Vec(0, 200, 300), 100)); // egy új gömb hozzáadás
scene.objects[0]->color = Color(255, 255, 255); // az új elem színének beállítása
scene.objects[0]->isReflective = false;

scene.add(new Sphere(Vec(150, 200, 500), 100));
scene.objects[1]->color = Color(255, 255, 0);
scene.objects[1]->isReflective = false;
```

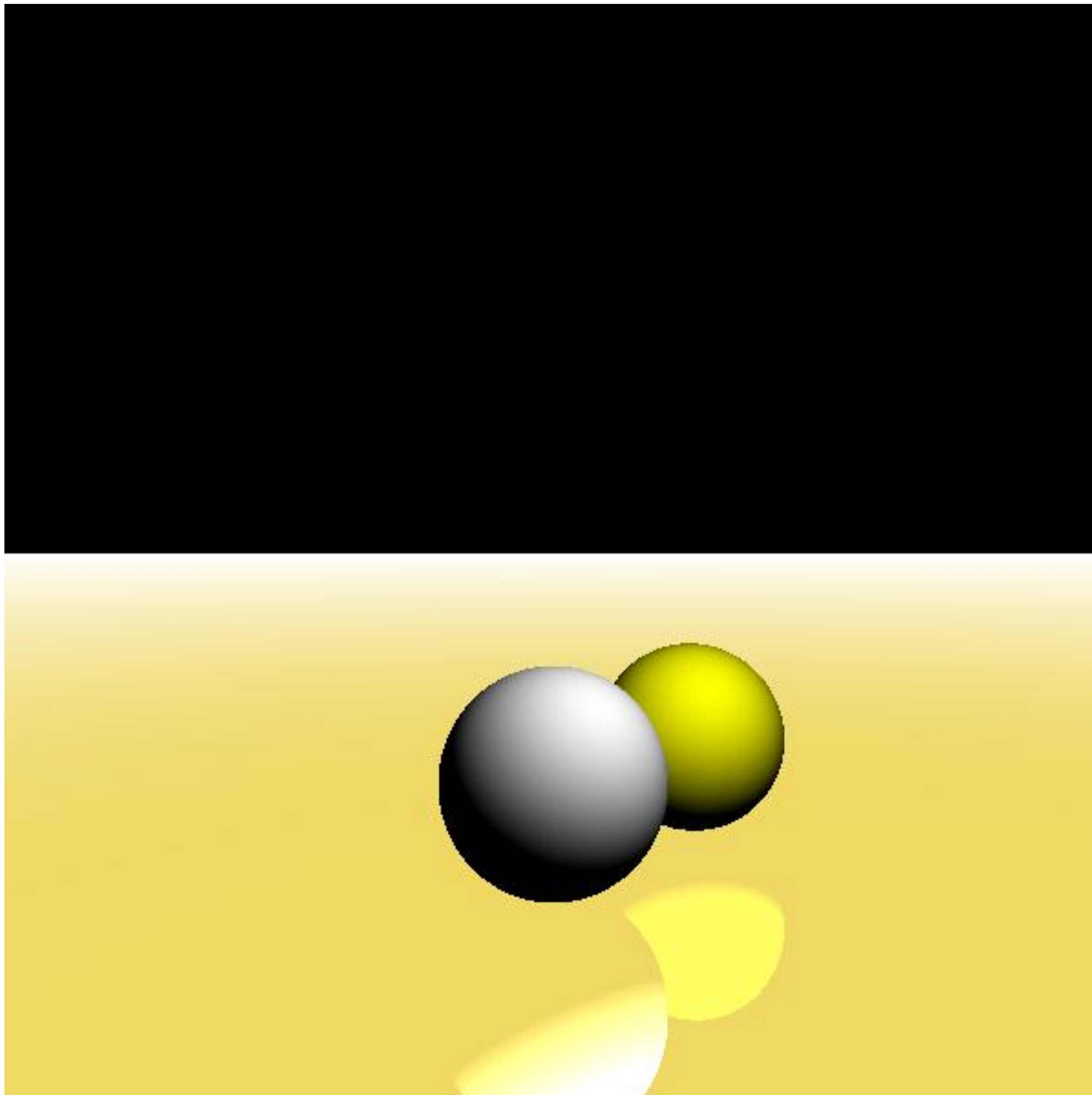
```
scene.add(new Plane(Vec(0, 300, 0), Vec(0, -1, 0)));
scene.objects[2]->color = Color(255, 255, 255); // a szín a visszaverő tulajdonsága miatt mindegy
scene.objects[2]->fr = Color(0.17, 0.35, 1.5); // legyen ez a törési mutatója (aranyra jellemző)
scene.objects[2]->kappa = Color(3.1, 2.7, 1.9); // legyen ez a kappa (aranyra jellemző)
scene.objects[2]->isReflective = true; // legyen tükröző felület
```

Majd pedig a Trace függvényt módosítsuk.

```
Color Trace(Ray& ray, int iterat)
{
        Color color; // a majdani visszatérési szín
        color.r = color.g = color.b = 0; // alapból feketére állítjuk
        if(iterat < DMAX) // ha még nem léptük át az iterációs határt
        {
                iterat++; // megnöveljük eggyel az iteráció számát
                double t = -1; // az egyenlet megoldása
                double mint = 9999999999999999999.99999999999999; // a minimális "t"-t egy jó
nagy számra inicializáljuk
                int minindex = -1; // a mint-hez tartozó index
                for(int i = 0; i < objcount; i++)
                {
                        t = objects[i]->intersect(ray); // elmetsszük
                        if(t > epsilon) // csak a pozitív megoldások érdekelnek
                        {
                                if(t < mint) // az olyanok, amik közelebb vannak, mint az eddig
megtalált legközelebbi
                                {
                                        mint = t; // átállítjuk
                                        minindex = i; // a minimálisra vonatkozó információkat
                                }
                        }
                }

                if(minindex >= 0) // ha sikeresen elmetszettük valakit
                {
                        t = mint; // a t legyen a minimálisnak talált
                        Vec intersectPoint; // a metszéspont
                        intersectPoint = ((ray.P0) + (ray.dv * t)); // ez lesz a pontos helye
                        Vec normal = objects[minindex]->getNormal(intersectPoint); // a felület
normálisa
                        Ray iRay; // egy sugár, ami a metszéspontból a fény felé fog majd nézni
                        iRay.P0 = intersectPoint + normal * 0.01; // kicsit "arrébb" húzzuk, hogy
biztos ne legyen a gömbünkben a sugár kezdőpontja
                        iRay.dv = light.P0 - intersectPoint; // beállítjuk az irányvektorát
                        iRay.dv.norm(); // normalizálunk
                        double factor = normal.dot(iRay.dv); // ez a cos(theta)
                        if(factor < 0) // csak a 0..1 tartományt vesszük figyelembe
                        {
                                factor = 0;
                        }
                        color = objects[minindex]->color; // az elmetszettnek a színét vesszük.

                        // és vesszük a Lambert-törvény által meghatározott árnyalást
                        // a fény egyes komponenseinek értékét azért kellett 255-tel osztani, mert én
                        // a 0...255 értékekkel szeretek dolgozni, viszont a képletbe egy 0...1
értéknek
                        // kell kerülnie
                        if(!objects[minindex]->isReflective) // ha nem visszaverő
                        {
                                color.r = color.r * (light.color.r / 255.0) * factor; // akkor a
diffúz árnyalást használjuk
                                color.g = color.g * (light.color.g / 255.0) * factor;
                                color.b = color.b * (light.color.b / 255.0) * factor;
                        }

                        if(objects[minindex]->isReflective) // ha tükröző
                        {
                                objects[minindex]->computeFresnel(factor); // kiszámoljuk az
adott pontban vett Fresnel-eh.-kat

                        // factor még mindig cos(theta)
                                double costheta2 = -1.0 * ray.dv.dot(normal); // costheta2, a
beérkező sugár és a normális skaláris szorzata
```

```
                        Ray rRay; // visszavert sugarunk
                        rRay.P0 = intersectPoint + normal * epsilon; // kezdeti pontja a
metszés pont egy kicsit eltolva
                        rRay.dv = ray.dv + normal * 2 * costheta2; // az irányát így
számoljuk
                        rRay.dv.norm(); // normalizáljuk

                        Color plusColor = Trace(rRay, iterat); // elindítjuk az új
sugarunkat

                        color = color + plusColor; // majd az eredményt hozzáadjuk az
eddig színhez

                        color.r = color.r * objects[minindex]->Kr.r; // vesszük a szín
Fresnel-eh.-s szorzatát
                        color.g = color.g * objects[minindex]->Kr.g; // ettől lesz
tulajdonképpen színe
                        color.b = color.b * objects[minindex]->Kr.b; // a felületnek
                    }
                }
        }
        return color; // visszaadjuk a színt.
}
```

Valami ilyesminek kéne kijönnie:

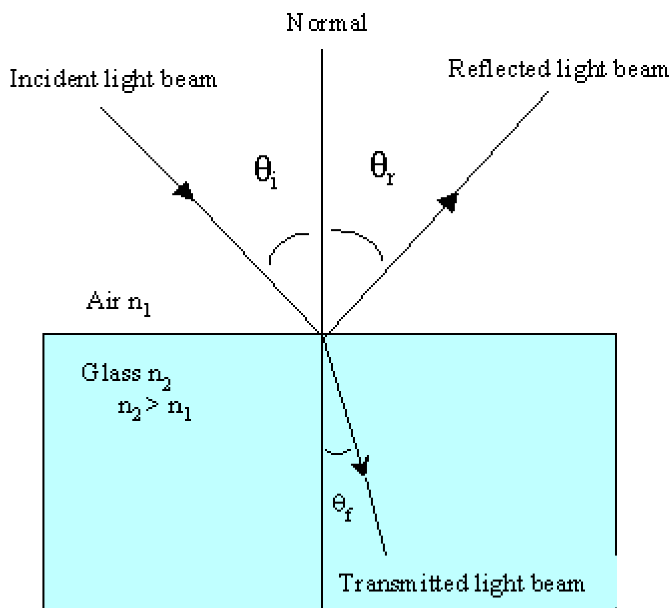


## A távolságot mint üveggolyót megkapod

Eddig eljutottunk oda, hogy már tudunk tükrözni, diffúz felületeket létrehozni, szóval már egy szép képet össze tudnánk állítani. Viszont eszünkbe jut gyerekkorunkból (legalábbis nekem az enyémből) az üveggolyó, és hogy hányszor néztünk keresztül rajta, mert vicces volt. Ezért nézzük át, hogy mi is történik, ha fénytörő felületeket akarunk implementálni!

A matematikai és fizikai levezetést – ha nem haragszotok – megint el fogom kerülni, csupán a leglényegesebb dolgokat vesszük sorra. Nézzük meg, mi is történik a valóságban! A fénytörés akkor jön létre, ha a foton két különböző sűrűségű felület határára ér. Két dolog fordulhat elő: vagy a ritkából a sűrűbb, vagy a sűrűbből a ritkább anyag felé megyünk. Minden közegnek van egy rá jellemző értéke, ezt nevezzük törésmutatónak, és a határátlépés irányától függően (sűrűbből ritkába, vagy fordítva) vesszük ezt a törésmutatót, vagy a reciprokát. A légmentes térnek 1 a törésmutatója, mindent ehhez viszonyítunk. A levegőnek annyira kicsivel több a törésmutatója, hogy általában azt is 1-nek választjuk.

A fénytörés érdekességét tulajdonképpen a kilépési szög adja, azaz hogy a beérkező fény az új közegben mekkora szöget fog bezárni a felületi normálissal. Ezt jól szemlélteti az alábbi ábra:




A Snellius-Descartes törvény értelmében $\sin(\theta_i) / \sin(\theta_f) = fr$, ahol az fr annak az anyagnak a törésmutatója, amelyiket elmetszi a sugarunk. Felmerül a kérdés, hogy akkor most hogyan van ez, a levegő törésmutatója hogyan jön a képbe? A válasz: sehogy. Egy egyszerű példa: két koncentrikus gömbünk van, eltérő sugarakkal, és eltérő sűrűséggel (ebből következően eltérő törésmutatóval). Amikor belépünk a külső gömbbe, akkor értelemszerűen a külső gömb törésmutatójával dolgozunk. Ezután elmetsszük a belső gömböt, akkor annak a törésmutatójával dolgozunk. Elérjük a középpontot, megyünk kifele, és itt jön a vicc: először a belső gömböt metsszük, tehát annak a törésmutatójával fogunk foglalkozni. A példából talán érthető, hogy a levegőnek a törésmutatójával akkor dolgoznánk, ha kilépnénk a levegős közegből a légüres térbe.

Tehát a következőt fogjuk csinálni: elmetsszük a testünket, megnézzük, hogy törő felület-e, megnézzük, hogy a testünk belsejében vagyunk-e vagy nem (merre áll a normálvektor), ha belül vagyunk, akkor átállítjuk a törésmutatót, és a normálvektort, majd megnézzük, hogy be tudunk-e törni a testbe, és ha igen, akkor új sugarat hozunk létre, és rekurzívan lekövetjük.

Azt, hogy a fénysugarunk bejut-e a testbe, vagy túl lapos szögben érkezik-e, és lepattan, a törésmutató határozza meg (egész pontosan azt mutatja meg a törésmutató, hogy mennyi az a „túl lapos szög”).

Ezt egy képletnek a visszatérési értéke adja meg, mégpedig ha a

disc = 1.0 - ((1.0 - cosalpha * cosalpha ) / (n * n))

összefüggésben a disc kisebb, vagy egyenlő, mint nulla, akkor nem sikerült betörnünk az anyagba. A képletben a cosalpha a -1 * bejövő sugár iránya * normális, az n pedig a törésmutató.

Innen már csak a visszavert sugár irányának kiszámítása kell, amire – a teljesség igénye nélkül – megadok itt egy egyenletet:

dv = ray.dv / n + tnormal * (cosalpha / n - sqrt(disc))

Ahol a ray.dv a beérkező fény irányvektora, n a törésmutató, tnormal a megfelelő irányba álló normálvektor, cosaplha pedig a fent meghatározott.

Nézzük meg, hogy ez kódban hogyan néz ki:

A Trace függvényt így egészítjük ki:

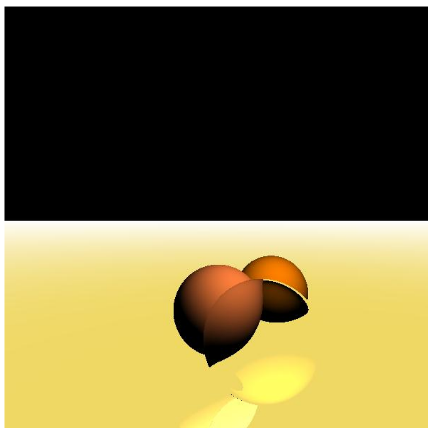
```
Color Trace(Ray& ray, int iterat)
{
	Color color; // a majdani visszatérési szín
	color.r = color.g = color.b = 0; // alapból feketére állítjuk
	if(iterat < DMAX) // ha még nem léptük át az iterációs határt
	{
		iterat++; // megnöveljük eggyel az iteráció számát
		double t = -1; // az egyenlet megoldása
		double mint = 9999999999999999999999.99999999999999999; // a minimális "t"-t
egy jó nagy számra inicializáljuk
		int minindex = -1; // a mint-hez tartozó index
		for(int i = 0; i < objcount; i++)
		{
			t = objects[i]->intersect(ray); // elmetsszük
			if(t > epsilon) // csak a pozitív megoldások érdekelnek
			{
				if(t < mint) // az olyanok, amik közelebb vannak, mint az eddig
megtalált legközelebbi
				{
					mint = t; // átállítjuk
					minindex = i; // a minimálisra vonatkozó információkat
				}
			}
		}

		if(minindex >= 0) // ha sikeresen elmetszettük valakit
		{
			t = mint; // a t legyen a minimálisnak talált
			Vec intersectPoint; // a metszéspont
			intersectPoint = ((ray.P0) + (ray.dv * t)); // ez lesz a pontos helye
			Vec normal = objects[minindex]->getNormal(intersectPoint); // a felület
normálisa
			Ray iRay; // egy sugár, ami a metszéspontból a fény felé fog majd nézni
			iRay.P0 = intersectPoint + normal * 0.01; // kicsit "arrébb" húzzuk,
hogy biztos ne legyen a gömbünkben a sugár kezdőpontja
			iRay.dv = light.P0 - intersectPoint; // beállítjuk az irányvektorát
			iRay.dv.norm(); // normalizálunk
			double factor = normal.dot(iRay.dv); // ez a cos(theta)
			if(factor < 0) // csak a 0...1 tartományt vesszük figyelembe
			{
				factor = 0;
			}
```

```
				color = objects[minindex]->color; // az elmetszettnek a színét vesszük.

				// és vesszük a Lambert-törvény által meghatározott árnyalást
				// a fény egyes komponenseinek értékét azért kellett 255-tel osztani,
mert én
				// a 0...255 értékekkel szeretek dolgozni, viszont a képletbe egy 0...1
értéknek
				// kell kerülnie
				if(!objects[minindex]->isReflective && !objects[minindex]->isRefractive)
// ha nem visszaverő, és nem törő
				{
					color.r = color.r * (light.color.r / 255.0) * factor; // akkor a
diffúz árnyalást használjuk
					color.g = color.g * (light.color.g / 255.0) * factor;
					color.b = color.b * (light.color.b / 255.0) * factor;
				}

				if(objects[minindex]->isReflective) // ha tükröző
				{
					objects[minindex]->computeFresnel(factor); // kiszámoljuk az
adott pontban vett Fresnel eh.-kat

				// factor még mindig cos(theta)
					double costheta2 = -1.0 * ray.dv.dot(normal); // costheta2, a
beérkező sugár és a normális skaláris szorzata
					Ray rRay; // visszavert sugarunk
					rRay.P0 = intersectPoint + normal * epsilon; // kezdeti pontja a
metszés pont egy kicsit eltolva
					rRay.dv = ray.dv + normal * 2 * costheta2; // az irányát így
számoljuk
					rRay.dv.norm(); // normalizáljuk

					Color plusColor = Trace(rRay, iterat); // elindítjuk az új
sugarunkat

					color = color + plusColor; // majd az eredményt hozzáadjuk az
eddig színhez

					color.r = color.r * objects[minindex]->Kr.r; // vesszük a szín
fresnel-eh -s szorzatát
					color.g = color.g * objects[minindex]->Kr.g; // ettől lesz
tulajdonképpen színe
					color.b = color.b * objects[minindex]->Kr.b; // a felületnek
				}

				if(objects[minindex]->isRefractive) // ha törő felülettel van dolgunk
				{
					double n = objects[minindex]->fr.r; // vesszük a törésmutatót

		// a példánkban a törő felület minden hullámhosszon ugyanúgy tör

		// ezért elég csak a vörös komponenst venni

		// értelemszerűen minden fénytartományra külön kiszámolható lenne a törés
					Vec tnormal = normal; // csinálunk egy temporális  normálvektort
					double cosalpha = -1.0 * ray.dv.dot(tnormal); // vesszük a
beesési szög koszinuszának -1-szeresét
					if(cosalpha < 0) // ha ez kisebb, mint 0, akkor a testünk
belsejében vagyunk
					{
						n = 1.0 / n; // vesszük a törésmutató reciprokát
						tnormal = tnormal * -1.0; // és a normálvektort
megfordítjuk
						cosalpha = -1.0 * ray.dv.dot(tnormal); // majd megint
kiszámoljuk cosaplhát
					}

					double disc = 1.0 - ((1.0 - cosalpha * cosalpha ) / (n * n)); //
megnézzük, hogy sikerül-e betörni
```

```
                              if(disc > 0) // ha igen
                              {
                                     Ray fRay; // létrehozunk egy új sugarat
                                     fRay.P0 = intersectPoint + tnormal * epsilon * -1.0; //
az új sugár kezdőpontja ne pontosan

                                     // ott legyen, ahol a beérkezési pont a

                                     // számolási pontatlanságok elkerülése végett
                                     fRay.dv = ray.dv / n + tnormal * (cosalpha / n -
sqrt(disc)); // a tört sugár iránya

                                     fRay.dv.norm(); // normalizáljuk

                                     Color plusColor = Trace(fRay, iterat); // rekurzívan
elindítjuk

                                     color = plusColor;
                              }
                       }
               }
       }
       return color; // visszaadjuk a színt.

}
```

A main függvénybe pedig ezt tegyük pluszba:

```
Scene scene; // színtér
scene.add(new Sphere(Vec(0, 200, 300), 100)); // egy új gömb hozzáadás
scene.objects[0]->color = Color(255, 255, 255); // az új elem színének beállítása
scene.objects[0]->isReflective = false;
scene.objects[0]->isRefractive = false;

scene.add(new Sphere(Vec(150, 200, 500), 100));
scene.objects[1]->color = Color(255, 255, 0);
scene.objects[1]->isReflective = false;
scene.objects[1]->isRefractive = false;

scene.add(new Plain(Vec(0, 300, 0), Vec(0, -1, 0)));
scene.objects[2]->color = Color(255, 255, 255); // a szín a visszaverő tulajdonsága miatt
mindegy
scene.objects[2]->fr = Color(0.17, 0.35, 1.5); // legyen ez a törési mutatója (aranyra
jellemző)
scene.objects[2]->kappa = Color(3.1, 2.7, 1.9); // legyen ez a kappa (aranyra jellemző)
scene.objects[2]->isReflective = true; // legyen tükröző felület
scene.objects[2]->isRefractive = false;

scene.add(new Sphere(Vec(75, 200, 100), 100));
scene.objects[3]->color = Color(0, 0, 0);
scene.objects[3]->fr = Color(1.13, 1.13, 1.13); // alacsony törési index
scene.objects[3]->kappa = Color(1.0, 1.0, 1.0); // egységnyi kappa
scene.objects[3]->isReflective = true; // legyen viszaverő


scene.objects[3]->isRefractive = true; // és törő
```



## Halszemmel a diszkóban

Ugye megállapodtunk abban, hogy a vetítősíkunkból egy 600×600-as ablakot fogunk kivágni, és azon keresztül fogunk benézni a színtérbe. Ha túl közel van a szemünk az ablakhoz, akkor túl nagy szögben fogunk benézni, ha messze vagyunk, akkor túl kis szögben, és meglehetősen érdekesen fog akkor torzulni a tér. Ki lehet próbálni.

A megoldást az jelenti, ha kiszámoljuk, hogy milyen távol álljunk a vászontól. Ehhez egy lehetséges algoritmus a következő:

```
float FOV = 60;
float FOV2 = (float)(FOV / 2.0);

float L = (float)w2 / (float)tan(FOV2 * PI / 180.0);
```

A FOV-val jelöljük, hogy milyen szögben akarunk betekinteni a színtérbe, és az L adja meg, hogy milyen messze kell állnunk a vászontól.

## Lapátoljunk össze

Megtanultuk, hogy mi a sugárkövetés optikai alapja, miért úgy épül fel, ahogy. Megnéztük, hogy hogyan tudunk diffúz felületeket létrehozni, hogyan épülnek fel a tükröző felületek, és mi történik fénytörésnél. Megszüntettük a halszemeffektust, és felépítettünk egy olyan osztályhierarchiát, amellyel könnyen tudunk dolgozni.

Remélem, van pár ember, akinek felkeltettem az érdeklődését, és szép képeket fogtok előállítani sugárkövetéssel.

További sok sikert, és jó programozást!