

Számítógépes grafika

Bevezetés

Szirmay-Kalos László

email: szirmay@iit.bme.hu

<http://cg.iit.bme.hu/>

<http://cg.iit.bme.hu>



3 db kötelező kisházi
1 db önkéntes nagyházi
ZH (pótZH)

Computer Graphics Group
Department of Control Engineering
and Information Technology

1. Előzetes 2. Pótlások 3. Projektek 4. **Önkéntes Nagyha** 5. Előzetes

Számítógépes grafika

Tárgykód:
DK_Szekely-Kovacs Laszlo
Tárgyatago:
VIZUALIS

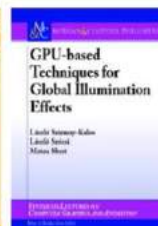
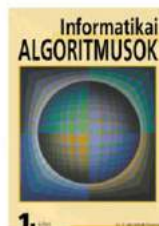
Hetek:
Azoknak is az új tárgyat felvetté jusszunk, akik a korábbi tárgyból alábbiak
rendszereznek, az előzők nem vizuál.

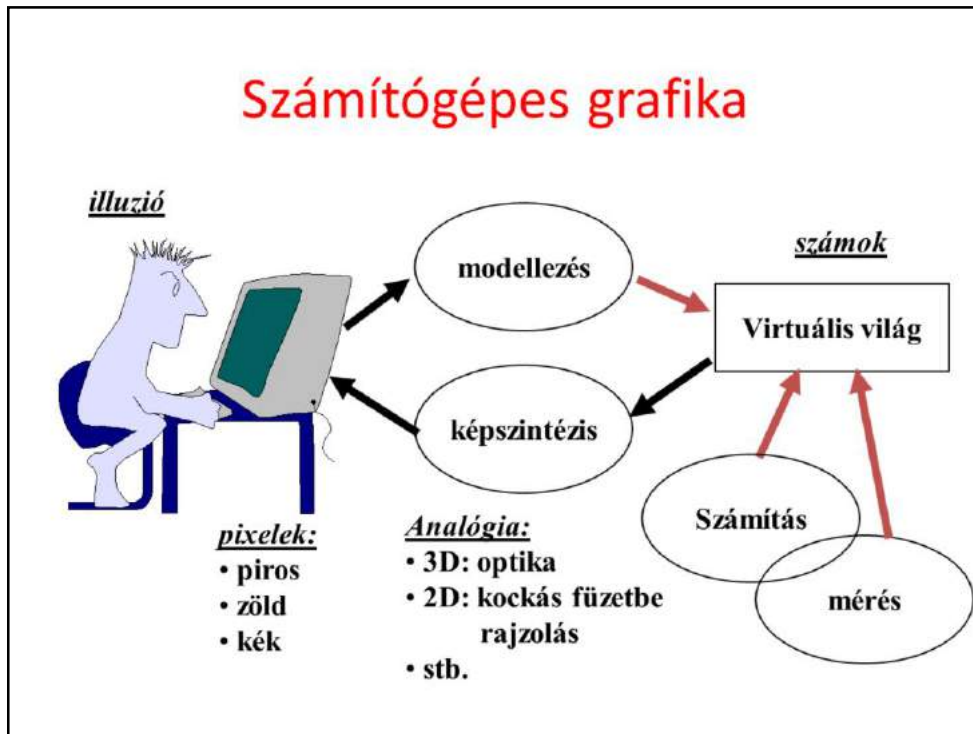
Tematika:
A tárgy a vizuális számítástechnika (számítógépes grafika, pixel, virtuális valóság, jelleltérkép)
alaptul mutatja be. Az elméletet (geometria, optika, analitikus geometria,
radiometria, kinematika) a gyakorlati feladatok segítségével, implementációval
a C++ nyelven, az OpenGL, valamint az a 3D animációs programozás környezetében
mutatjuk meg. A tárgyat haladó hallgatók megismerkedhetnek a 2D és 3D grafikus rendszerek felépítésével, az animáció és
jelleltérkép technikáival, a grafikus kártyák működésével és programozásával. A hallgatókat
elkészíti a haladó feladatok megoldására C++ programozási környezetben és a 3D
animációs szimulációk készítésére (ha jól esik a kártyák, csak a saját programozási
gyártás).

Számítógépes grafika

1. Alapfogalmak
2. Alapfogalmak
3. Alapfogalmak

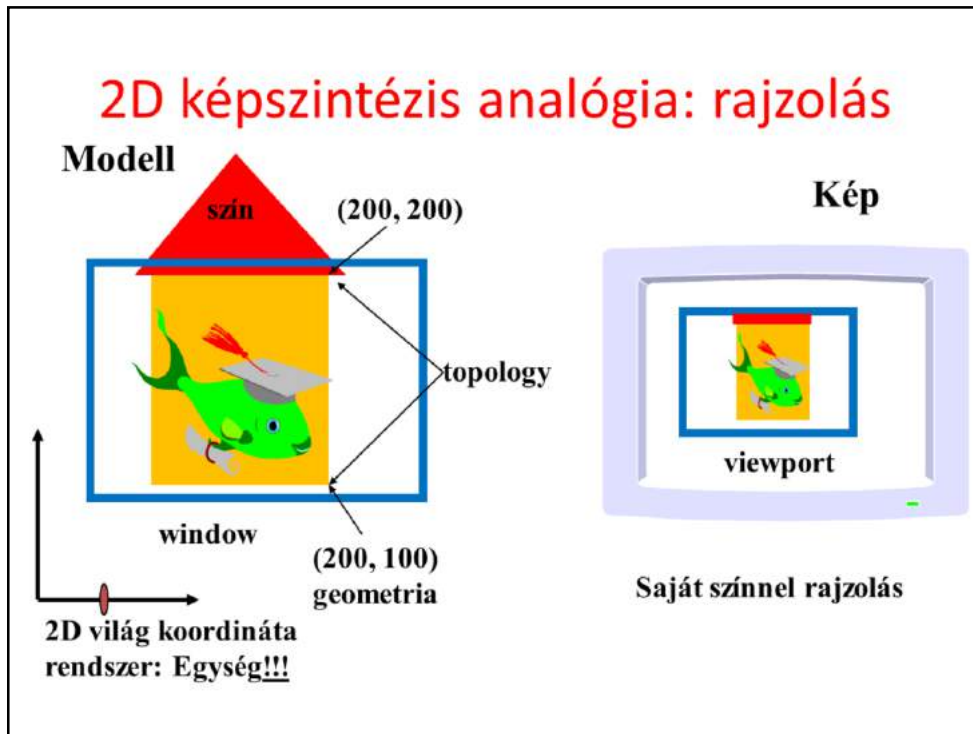
Feladatlapok, leckevezetők





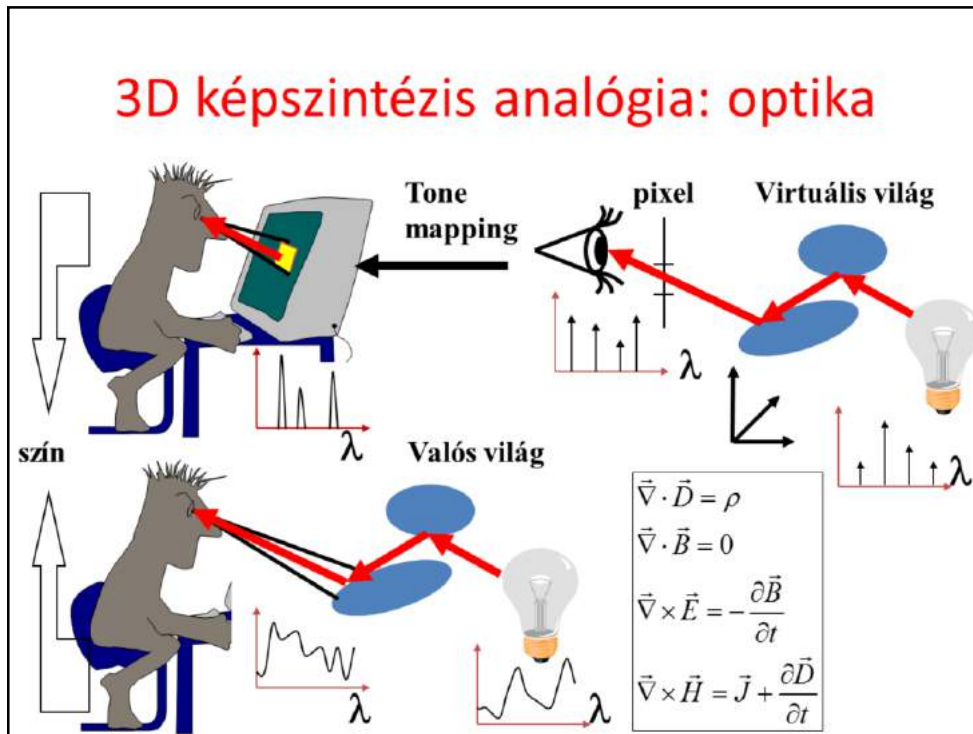
In computer graphics we render virtual worlds by taking a photo of them and presenting their image to the user. The virtual world is stored in the computer memory. The virtual world model can be the result of an interactive modeling process, simulation, measurement, etc.

Rendering can be regarded as an abstract mapping from the virtual world model to the intensity and color values of the computer screen. There are infinite number of possibilities to define this mapping. If we wish to have images that look like real images, we should simulate the image creation process or the real world. For example, we can simulate light transport, i.e. optics, or manual drawing.



Let us look at the details when the virtual world is two dimensional, so objects are in a plane. A convenient reference system is a Cartesian coordinate system with an origin, two axes and also a unit. Using these every point of the plane can be specified by two numbers defining the distance traveled along the two axes and measured with respect to the unit.

With pairs of numbers, points can be defined, which can form primitives by adding topology information. For example, we can say that these three points define a triangle. Primitives are given material properties, which usually include the color.



If we want to create photo like images, we should simulate the light transport and provide the user with the illusion that he watches the real world and not a computer screen.

If we could guarantee that the human eye gets the same photons (i.e. the same number and of the same frequency) from the solid angle subtended by a pixel as the eye got from the real world, then it would not be possible for the user to distinguish between the computer monitor and the real world since the same photons result in similar color impressions. So in computer graphics, we should compute the number and frequency of photons, i.e. the power spectrum of the light that would enter the eye from the solid angle of each pixel. Then the display should be controlled to emit similar photons. Fortunately, we do not have to emit exactly the same spectrum since the human eye is very bad in measuring a spectrum. In fact, the illusion of most of the spectra can be provided by carefully selected red, green and blue intensities. So having calculated the spectrum, we convert it to an equivalent red/green/blue intensity triplet and get the monitor to emit it.

The calculation of the light spectrum requires the solution of the photon transfer or the transfer of electromagnetic waves. The equations describing these phenomena are the Maxwell equations, so in fact, graphics should solve

these fundamental equations to obtain the image.

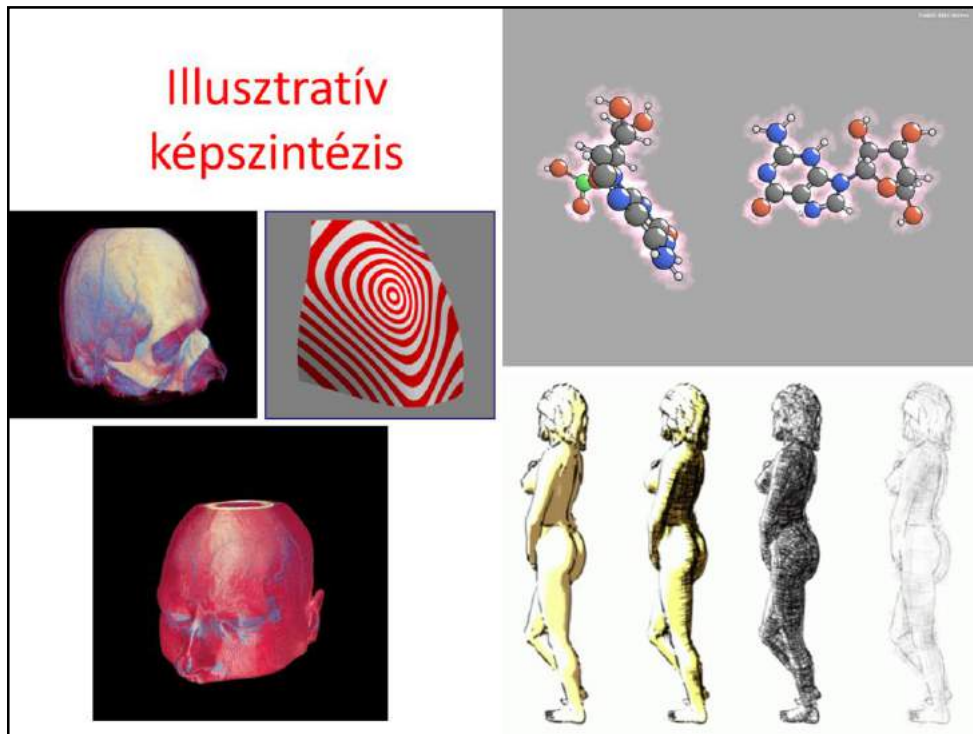
Optika:Fotorealisztikus képszintézis



The results of the simulation of optics laws or Maxwell equations are indeed like real photos.



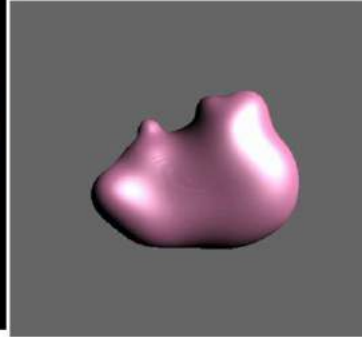
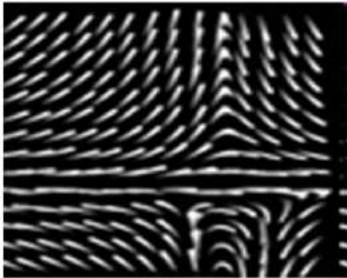
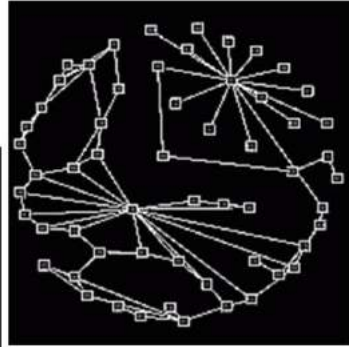
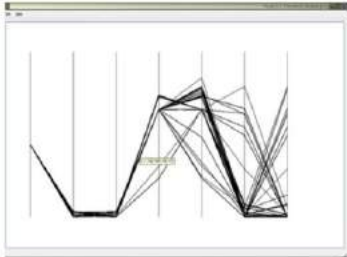
The simulation of optics is useful even if the original data is not directly related to optical parameters. However, we can establish a correspondence between non-optical properties, like density or the extinction of X-ray etc, and hypothetical optical properties like opacity, transparency, color etc. Having established this relationship, we can photograph the scene and present the image which visualizes the data for us.



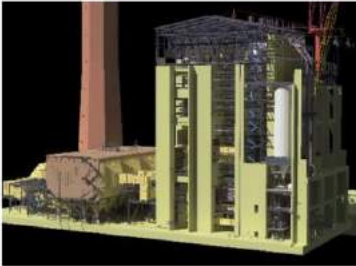
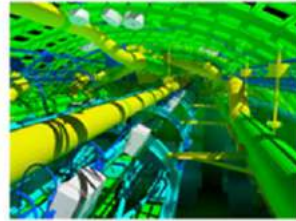
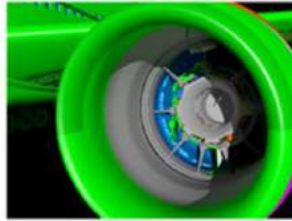
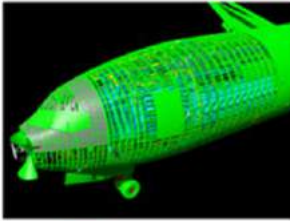
However, in computer graphics it is not obligatory to use only the optics model. Instead, we can simulate the artistic process, like drawing, painting, hatching etc. We can also take other analogies. For example, the lower left image is a flow visualization obtained with the line integral convolution algorithm. This algorithm simulates the process of putting confetti into the flow and photographing it keeping the shutter open for a longer time. The confetties are blurred into the direction of the velocity field of the flow.

Such analogies are essential when there is no direct method for visualizing the data, for example, when it is higher than three dimensional. The right upper image is made with the parallel coordinates method, which displays a seven-dimensional data set. Each dimension is given a vertical line and a point is then a polyline connecting its coordinates on each vertical line.

Adatvizualizáció



Kihívások



- Nagy modell(giga/terabyte)
- Valós idő: pár nsec/pixel

Tematika

- Analitikus geometria ismételés
- Modellezés: görbék, felületek
- Transzformációk, homogén koordináták
- 2D képszintézis, freeglut+OpenGL 3
- 3D képszintézis fizikai alapjai
- Sugárkövetés
- Inkrementális képszintézis, freeglut+OpenGL+GLSL
- GPGPU: CUDA
- Animáció, Játék

Miért ezt a ...-ot tanuljuk, és miért nem ...-ot?

- 3D grafikus rendszerek: JavaScript, WebGL, DirectCompute, OpenCL
- Grafikus játékok fejlesztése: Direct3D/HLSL
- Vizualizáció és képszintézis: RenderMan, Vray, Nuke
- Játékfejlesztés: Blender, GIMP, PhysX, Bullet, Ogre3D, Unity3D
- GPGPU tárgyak: CUDA, OpenCL
- 3D számítógépes geometria és alakzatrekonstrukció:
Blender, ParaView, Sketches
- Képfeldolgozás: OpenCV
- Virtuális és kiterjesztett valóságrendszerek:
OpenCV, OpenGL ES + Android

Analitikus geometria gyorstalpaló

Szirmay-Kalos László

Geometriák

- Geometria
 - Axiómák
 - Alapigazság (tapasztalat)
 - Alapfogalmak implicit definíciója
 - Definíciók és tételek
 - Célok és eszközök
- Fontos geometriák a számítógépes grafikában
 - Euklideszi (sík/tér, metrikus)
 - Projektív
 - Fraktális
 - ...

- Két pont meghatároz egy egyenest.
- Egy egyenesnek van legalább két pontja.
- Ha a egy egyenes, A pedig egy, nem az egyenesen lévő pont, akkor egyetlen olyan egyenes létezik, amely átmegy A -n és nem metszi a -t.

Computer graphics works with shapes. The field of mathematics that describes shapes is the geometry, so geometry is essential in computer graphics.

Geometry, like other fields of formal science, has **axioms** that are based on experience and cannot be argued but are accepted as true statements without arguments. From axioms other true statements, called theorems, can be deduced with logic reasoning.

For example, axioms of the **Euclidean geometry** include the following three postulates. Axioms have two purposes, on the one hand, they are accepted as true statements. On the other hand, axioms implicitly define **basic concepts** like points, lines etc. because they postulate their properties.

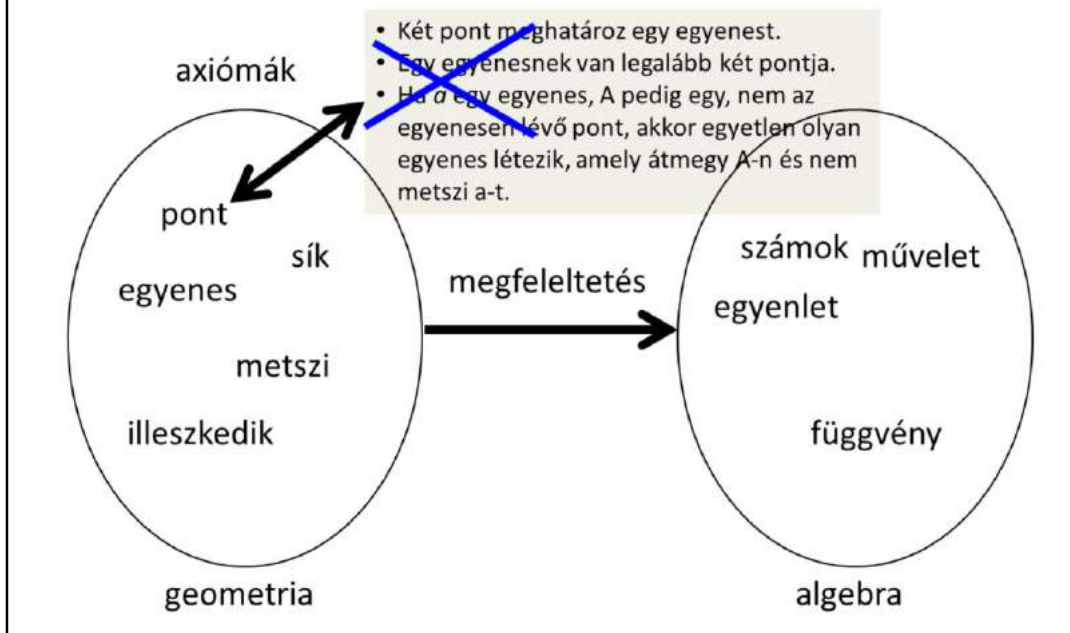
Based on the axioms and the applied tools, there are several different geometries that are different models of the world. Everybody knows the Euclidean geometry of the plane and of the space. We know that it is **metric**, i.e. we can talk of the distance between objects and size is an important concept in it. In Euclidean geometry parallel lines do not intersect, that is, a point at infinity is not part of the Euclidean plane.

However, if we define axioms differently, we can add points at infinity to the plane making all lines, even parallel lines, intersecting. Clearly, this is a different geometry with different axioms and theorems, which is called the **projective geometry**. Projective geometry is not metric since distance cannot be defined in

it. The reason is that the distance from points at infinity is infinite, but infinite is not a number.

In Euclidean geometry size is an important issue, curves are measured by their length, surfaces by their area, and solids by their volume. However, when we try to apply these concepts to natural phenomena, like a snow crystal or a cloud, we fail. We have to realize that natural objects do not have a precise size, so Euclidean geometry is not appropriate for their description. For natural phenomena, we use **fractal geometry**.

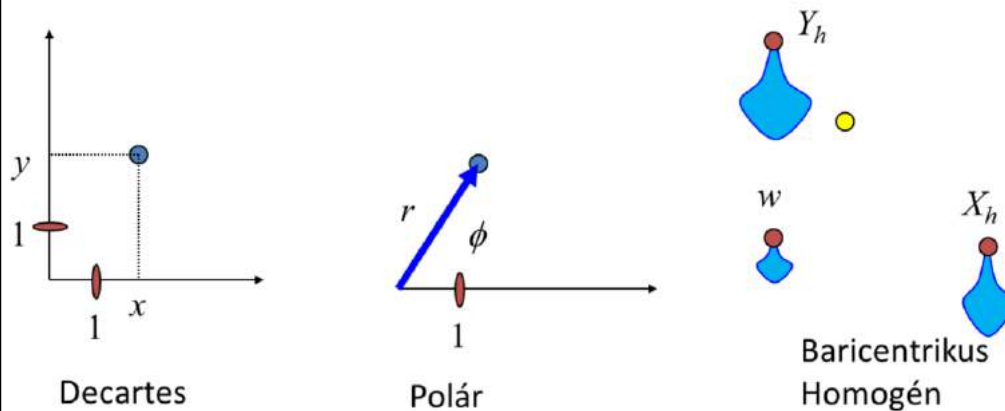
Mindent számmal: analitikus geometria



In computer graphics, we should also take into account that a computer is programmed, which cannot do anything else but calculations with numbers. A computer is definitely not able to understand abstract concepts like point, line etc. So for the application of a computer, geometric concepts must be translated to numbers, calculation and algebra.

A geometry based on algebra, equations and numbers is called analytic geometry or coordinate geometry. To establish an analytic version of a geometry, we have to find correspondences between geometric concepts and concepts of algebra in a way that axioms of the geometry will not contradict to the concepts of algebra. If it is done, we can forget the original axioms and work only with numbers and equations.

Pontok definíciója



Számokkal!

1. Koordinátarendszer (=referencia geometria)
2. Koordináták(=mérés)

The goal is the definition of points with numbers and primitives with equations or functions.

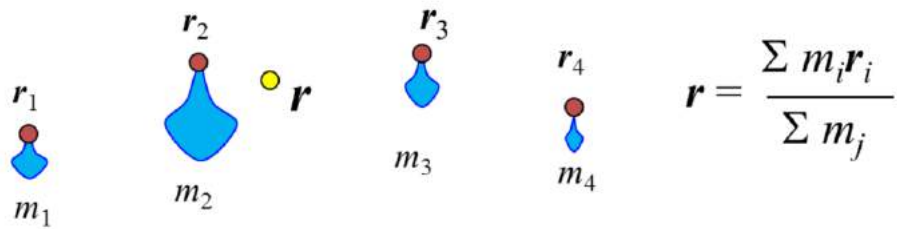
The definition of points with numbers requires a coordinate system and then the measuring of the point with respect to this coordinate system. A Cartesian coordinate system contains two orthogonal lines or axes, and a unit on them, and we measure how far we should walk along them to arrive at the identified point.

A 2D polar coordinate system is a half line, and a point is defined by an angle and a distance. The angle specifies the direction in which we should go from the origin and the distance is interpreted between the origin and the identified point. Note that while we require that all points can be expressed by coordinates, this is not necessarily unambiguous, i.e. in a polar system the origin can be defined with arbitrary angle and with distance zero.

In computer graphics barycentric coordinate systems are also popular. Here, the coordinate system is a set of points (at least 3 in 2D) where we put weights. The resulting mechanical system has a center of mass somewhere, which are identified by the numbers of the weights. Barycentric coordinates are often called homogeneous, due to the property that if we multiply all weights with the same non-zero scalar, then the center of mass is not affected.

However, for such constructions we have already applied many non-trivial concepts like vectors, distance, angles. First, let us start from scratch and revisit these basic building blocks.

Pontok kombinálása



- r az r_1, r_2, \dots, r_n pontok kombinációja
- Súlyok a baricentrikus koordináták
- Ha a súlyok nem negatívak: konvex kombináció
- Konvex kombináció a konvex burkon belül van
- Egyenes (szakasz) = két pont (konvex) kombinációja
- Sík (háromszög) = három pont (konvex) kombinációja

Defining a point as the center of mass of a system where masses placed at finite number of reference points is also called the combination of these points with barycentric coordinates equal to the weights.

Note that we can do this in real life without mathematics and coordinate systems, center mass exists and is real without mathematics and abstraction.

If all weights are non-negative, which has direct physical meaning, then we talk of convex combination since the points that can be defined in this way are in the convex hull of the reference points. By definition, the convex hull is the minimal set of points that is convex and includes the original reference points. For example, when presents are wrapped, the wrapping paper is on the convex hull of the presents.

Using the term combination or convex combination, we can define a line as a combination of two points and a line segment as a convex combination of two points. Similarly, the convex combination of three not collinear points is the triangle, the convex combination of four points not being in the same plane is a tetrahedron.

Vektor

- Vektor = eltolás: \mathbf{v}

- Irány és hossz ($|\mathbf{v}|$)

- Helyvektor

DE vektor \neq pont !!!

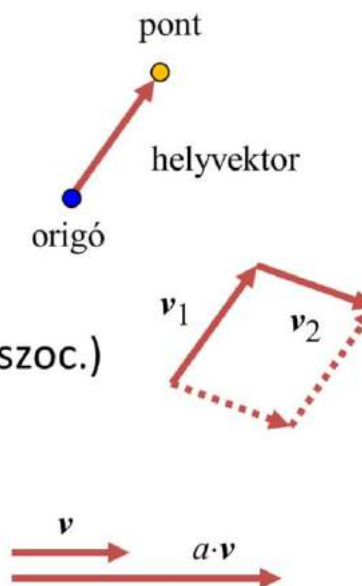
- Vektor összeadás

$$\mathbf{v} = \mathbf{v}_1 + \mathbf{v}_2 \text{ (kommutatív, asszoc.)}$$

$$\mathbf{v}_1 = \mathbf{v} - \mathbf{v}_2 \text{ (van inverz)}$$

- Skálázás (skalárral szorzás)

$$\mathbf{v}_1 = a \cdot \mathbf{v} \text{ (disztributív)}$$



In addition to combining points, we can also translate them. By definition a translation is a **vector**, which has direction and length. The length is denoted by the absolute value of the vector.

If we select a special reference point, called the **origin**, then every point has a unique vector that translates the origin to here, or from the other point of view, every vector unambiguously defines a point that is reached if the origin is translated by this vector. Such vectors are called **position vectors**. The fact that there is a one-to-one correspondence between points and position vectors does not mean that points and vectors would be identical objects (wife and husband are also strongly related and unambiguously identify each other, but are still different objects with specific operations).

Concerning vector operations, we can talk of **addition** that means the execution of the two translations one after the other. The resulting translation is independent of the order, so vector addition is **commutative** (parallelogram rule). If we have more than two vectors, parentheses can be rearranged so it is also **associative**. Vector addition has an inverse, because we can ask which vector completes the translation of \mathbf{v}_2 to get a resulting translation \mathbf{v} .

Vectors can be **multiplied by a scalar**, which scales the length but does not

modify the direction. Scaling is **distributive**, i.e. scaling a sum of two vectors results in the same vector as adding up the two scaled versions.

We have to emphasize that the nice properties of commutativity, associativity, and distributivity are usually not evident and sometimes not even true for vector operations. Be careful!

Skalár (dot, belső) szorzat

- Definíció

$$\mathbf{v1} \cdot \mathbf{v2} = |\mathbf{v1}| \cdot |\mathbf{v2}| \cdot \cos \alpha$$

- Jelentés

Egyik vektor vetülete a másikra * másik hossza

- Tulajdonság

Nem asszociatív!!!

Kommutatív

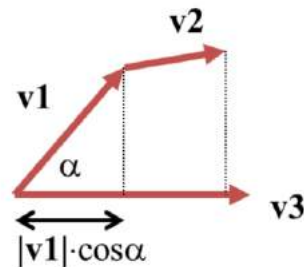
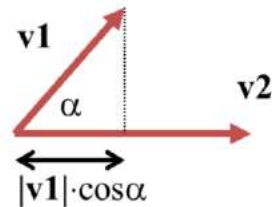
$$\mathbf{v1} \cdot \mathbf{v2} = \mathbf{v2} \cdot \mathbf{v1}$$

Disztributív az összeadással

$$\mathbf{v3} \cdot (\mathbf{v2} + \mathbf{v1}) = \mathbf{v3} \cdot \mathbf{v2} + \mathbf{v3} \cdot \mathbf{v1}$$

$$\mathbf{v} \cdot \mathbf{v} = |\mathbf{v}|^2$$

Két vektor merőleges ha a skalárszorzatuk zérus



Vectors can be multiplied in different ways. The first possibility is the **scalar product** (aka **dot** or inner product) that assigns a scalar to two vectors. By definition, the resulting scalar is equal to the product of the lengths of the two vectors and the cosine of the angle between them.

The geometric meaning of scalar product is the length of projection of one vector onto the other, multiplied by the lengths of the others.

Scalar product is **commutative** (symmetric), which is obvious from the definition.

Scalar product is **distributive with the vector addition**, which can be proven by looking at the geometric interpretation. Projection is obviously distributive (the projection of the sum of two vectors is the same as the sum of the two projections).

Scalar product is **NOT associative**!

There is a direct relationship between dot product and the absolute value. The scalar product of a vector with itself is equal to the square of its length according to the definition since $\cos(0)=1$.

Vektor (kereszt) szorzat

- Definíció

$$|\mathbf{v1} \times \mathbf{v2}| = |\mathbf{v1}| \cdot |\mathbf{v2}| \cdot \sin\alpha$$

Merőleges, jobb kéz szabály

- Jelentés

Terület és merőleges vektor,

(Egyik vektor vetülete a másikra merőleges síkra + 90 fokos elforgatás) * másik hossza

- Tulajdonságok

Nem asszociatív!!!

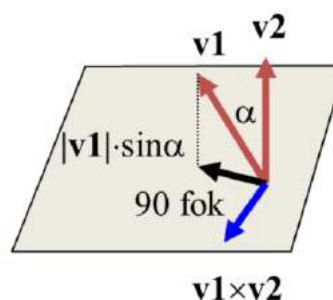
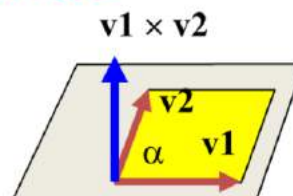
Antiszimmetrikus

$$\mathbf{v1} \times \mathbf{v2} = - \mathbf{v2} \times \mathbf{v1}$$

Disztributív az összeadással

$$\mathbf{v3} \times (\mathbf{v2} + \mathbf{v1}) = \mathbf{v3} \times \mathbf{v2} + \mathbf{v3} \times \mathbf{v1}$$

Két vektor párhuzamos ha vektorszorzatuk zérus.



Vectors can be multiplied with the rules of the **vector (aka cross) product** as well. The result is a vector of length equal to the product of the lengths of the two vectors and the sine of their angle. The resulting vector is perpendicular to both operands and points into the direction of the middle finger of our right hand if our thumb points into the direction of the first operand and our index finger into the direction of the second operand (**right hand rule**).

Cross product can be given two different geometric interpretations. The first is a vector meeting the requirements of the right hand rule and of length equal to the area of the parallelogram of edge vectors of the two operands.

The second geometric interpretation is the projection of the second vector onto the plane perpendicular to the first vector, rotating the projection by 90 degrees around the first vector, and finally scaling the result with the length of the first vector.

Cross product is **NOT commutative** but **anti-symmetric** or alternating, which means that exchanging the two operands the result is multiplied by -1.

Cross product is **distributive with the addition**, which can be proven by considering its second geometric interpretation. Projection onto a plane is

distributive with addition, so are rotation and scaling. Cross product is **NOT associative**.

Descartes koordináta rendszer

- Egyértelmű ($x = \mathbf{v} \cdot \mathbf{i}$, $y = \mathbf{v} \cdot \mathbf{j}$)
- Operációk koordinátákban

Összeadás:

$$\mathbf{v}_1 + \mathbf{v}_2 = (x_1 + x_2)\mathbf{i} + (y_1 + y_2)\mathbf{j}$$

Skalár szorzat:

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = (x_1\mathbf{i} + y_1\mathbf{j}) \cdot (x_2\mathbf{i} + y_2\mathbf{j}) = (x_1x_2 + y_1y_2)$$

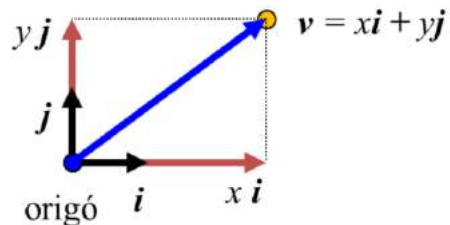
Vektor szorzat:

$$\mathbf{v}_1 \times \mathbf{v}_2 = (x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k}) \times (x_2\mathbf{i} + y_2\mathbf{j} + z_2\mathbf{k}) = (y_1z_2 - y_2z_1)\mathbf{i} + (x_2z_1 - x_1z_2)\mathbf{j} + (x_1y_2 - y_1x_2)\mathbf{k}$$

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}$$

Hossz:

$$|\mathbf{v}| = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \sqrt{x^2 + y^2 + z^2}$$



Having vectors and operations, we are ready to establish a **Cartesian coordinate system**. Let us select one point of the plane and two unit (length) vectors \mathbf{i} and \mathbf{j} that are perpendicular to each other. A vector has unit length if its scalar product with itself is 1 and two vectors are perpendicular if their scalar product is zero since $\cos(90)=0$ (formally: $\mathbf{i} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{j} = 1$ and $\mathbf{i} \cdot \mathbf{j} = 0$).

Now, any position vector \mathbf{v} can be unambiguously given as a linear combination of basis vector \mathbf{i} and \mathbf{j} , i.e. in the form $\mathbf{v} = x\mathbf{i} + y\mathbf{j}$, where x and y are scalars, called the **coordinates**. Having \mathbf{v} , scalar products determine the appropriate coordinates: $x = \mathbf{v} \cdot \mathbf{i}$, $y = \mathbf{v} \cdot \mathbf{j}$. To prove this, let us multiply both sides of $\mathbf{v} = x\mathbf{i} + y\mathbf{j}$ by \mathbf{i} and \mathbf{j} .

As there is a one-to-one correspondence between vectors and coordinate pairs in 2D (and coordinate triplets in 3D), vectors can be represented by coordinates in all operations.

Based on the associative property of vector addition and on distributive property of multiplying a vector by a scalar with addition, we can prove that coordinates of the sum of two vectors are the sums of the respective coordinates of the two vectors.

Similarly, based on the distributive property of dot product with vector addition, we can prove that the dot product equals to the sum of the products of respective coordinates. Here we also exploit that $\mathbf{i} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{j} = 1$ and $\mathbf{i} \cdot \mathbf{j} = 0$.

Finally, based on the distributive property of the cross product with vector addition, we can also express the cross product of two vector with their coordinates. We should also use the cross products of the base vectors $\mathbf{i} \times \mathbf{i} = 0$, $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, etc. The result can be memorized as a determinant where the first row contains the three basis vectors, the second the coordinates of the first operand, the third the coordinates of the second operand.

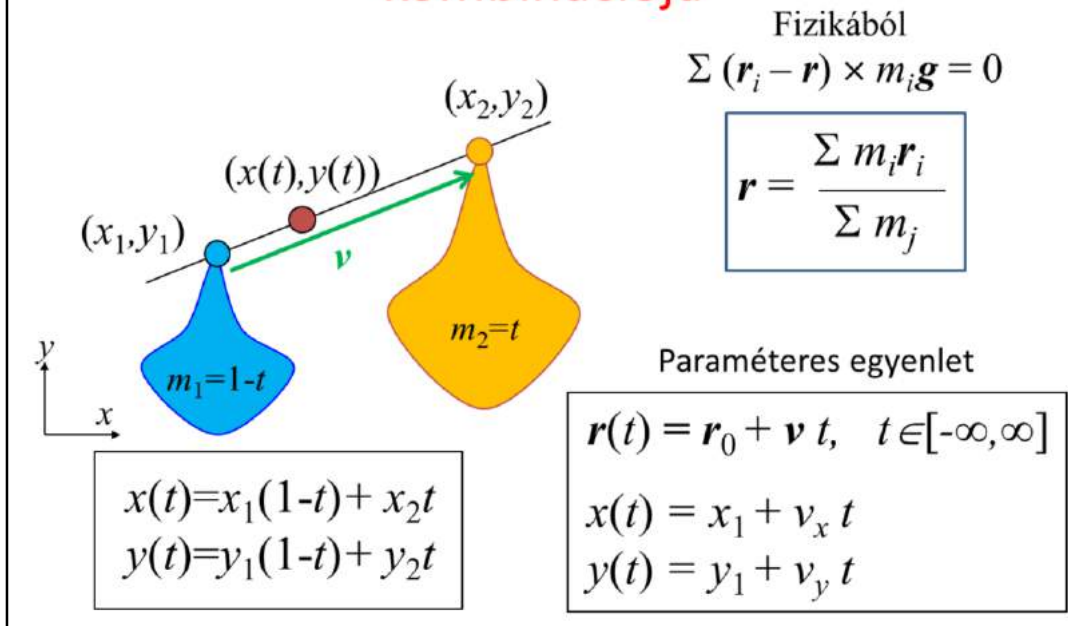
The absolute value of a vector is the square root of the scalar product of the vector with itself. Note that we get the Pythagoras theorem for free.

Vector/Point/Color class

```
struct vec3 {  
    float x, y, z;  
    vec3(float x0, float y0, float z0) { x = x0; y = y0; z = z0; }  
    vec3 operator*(float a) { return vec3(x * a, y * a, z * a); }  
    vec3 operator+(const vec3& v) {  
        return vec3(x + v.x, y + v.y, z + v.z);  
    }  
    vec3 operator-(const vec3& v) {  
        return vec3(x - v.x, y - v.y, z - v.z);  
    }  
    vec3 operator*(const vec3& v) {  
        return vec3(x * v.x, y * v.y, z * v.z);  
    }  
    float Length() { return sqrtf(x * x + y * y + z * z); }  
};  
  
float dot(const vec3& v1, const vec3& v2) {  
    return (v1.x * v2.x + v1.y * v2.y + v1.z * v2.z);  
}  
  
vec3 cross(const vec3& v1, const vec3& v2) {  
    return vec3(v1.y * v2.z - v1.z * v2.y,  
                v1.z * v2.x - v1.x * v2.z,  
                v1.x * v2.y - v1.y * v2.x);  
}
```

The implementation of the theory discussed so far is a single C++ class representing a 3D point or a vector with three Cartesian coordinates. Using operator overloading, the discussed vector (and point) operations are also.

Egyenes (szakasz) mint két pont kombinációja



Having points, we can start defining primitives built of infinitely many points. We have two basic operations on points, combination and finding the vector that translates one point to the other.

If we have a translation vector, we can ask the distance, impose requirements on orthogonality or parallelism.

Combination uses the center of mass analogy, which assigns the center of mass to a set of points by the given formula. The position vectors of individual points are multiplied by the mass placed there and the sum is divided by the total mass.

Let us select two points that will be combined and, for the sake of simplicity, let us assume that the total mass is 1 (we distribute 1 kg mass in the two points). Distributing unit mass has the advantage that we do not have to divide with the total mass since division by 1 can be saved.

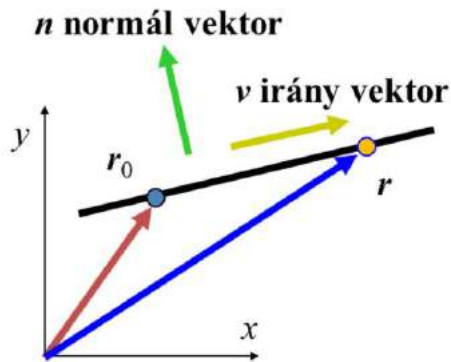
The center of mass will be on a line segment between the two points. Whether it is closer to the first or to the second point depends on t , so by modifying t in $[0,1]$ we can make the center of mass run on the line segment. So, points of the line segment can be expressed by a function of t . Such equation is called parametric equation because we have a free parameter that controls which point

of the primitive is currently selected.

If t can be outside of the unit interval, then a point can also repel the point, thus the center of mass will still be on the line of the two points but outside of the line segment. The equation of the line segment and the line are similar, only the parameter ranges are different. The equation can also be rewritten in another form, where the two points are replaced by one point, called the position vector of the line, and the vector between them, called the direction vector of the line.

2D egyenes

Implicit egyenlet



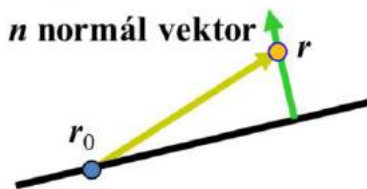
$$\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0) = 0$$

$$n_x(x - x_0) + n_y(y - y_0) = 0$$

$$ax + by + c = 0$$

$$(x, y, 1) \cdot (a, b, c) = 0$$

2D egyenestől mért távolság:



$$\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0) = \text{Vetület } \mathbf{n}\text{-re} \cdot \text{az } \mathbf{n} \text{ hossza}$$

Ha \mathbf{n} egységvektor:

$$\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0) \text{ az előjeles távolság!}$$

Having points, we can start defining primitives built of infinitely many points.

We have two basic operations on points, combination and finding the vector that translates one point to the other. If we have a translation vector, we can ask the distance, impose requirements on orthogonality or parallelism.

Combination uses the center of mass analogy, which assigns the center of mass to a set of points by the given formula. The position vectors of individual points are multiplied by the mass placed there and the sum is divided by the total mass.

Let us select two points that will be combined and, for the sake of simplicity, let us assume that the total mass is 1 (we distribute 1 kg mass in the two points). Distributing unit mass has the advantage that we do not have to divide with the total mass since division by 1 can be saved.

The other way of establishing the equation of the line is based on orthogonality (or, from another point of view, on distance). The difference vectors of any two points on the line are all parallel, so they are all perpendicular to a given vector, called the normal vector of the line. Let one point be a given point, called the position vector of the line, and the other point represent any point (this is called

the running point). Their difference $\mathbf{r}-\mathbf{r}_0$ is perpendicular to normal vector \mathbf{n} if and only if their scalar product is zero. This equation imposes a requirement on running point \mathbf{r} . If \mathbf{r} satisfies this equation, then the point is on the line, otherwise it is not on the line.

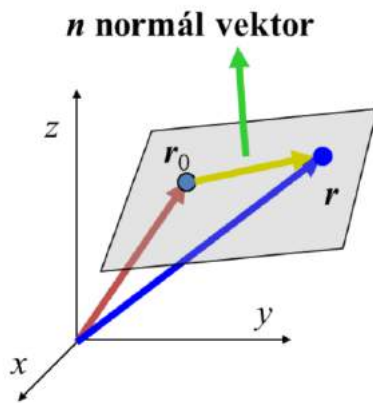
Another interpretation uses the distance. Point \mathbf{r} is on the line if its distance from the line is zero. We know from geometry that the distance should be measured in perpendicular direction, which is

$|\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0)|$ if \mathbf{n} is a unit vector (the difference is projected onto the unit normal vector).

Expressing the line equation with coordinates, we get an implicit linear equation for unknown point coordinates x and y . If a point's x,y coordinates satisfy this equation, the point is on the line.

This implicit equation can also be expressed by the scalar product of two three-element vectors if we use the convention that 2D points have three coordinates where the third coordinate is 1.

Sík



$$\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0) = 0$$

$$n_x(x-x_0) + n_y(y-y_0) + n_z(z-z_0) = 0$$

$$ax + by + cz + d = 0$$

$$(x, y, z, 1) \cdot (a, b, c, d) = 0$$

Ha \mathbf{n} egységvektor:

$\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0)$ a síktól mért távolság!

Kör a síkon

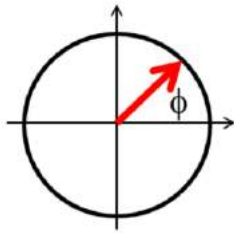
Implicit egyenlet:

Azon $r(x, y)$ pontok mértani helye, amelyek a $c(c_x, c_y)$ középponttól R távolságra vannak:

$$|r - c| = R \quad \leftrightarrow \quad (r - c)^2 = R^2 \quad \leftrightarrow \quad (x - c_x)^2 + (y - c_y)^2 = R^2$$

Paraméteres egyenlet:

A $\sin(\phi)$ és $\cos(\phi)$ definíciója:



$$\begin{aligned} x(\phi) &= c_x + R \cos(\phi) \\ y(\phi) &= c_y + R \sin(\phi) \end{aligned}$$

By definition, a **circle** is a set of points r of distance R (radius) from its center point c . Translating this geometric definition to the language of analytic geometry, we can establish the equation of the circle.

Distance of two points is the absolute value of the vector between them, which must be equal to R . Instead of the distance, we can work with the squared distance since both sides of this equation are positive, so taking the square does not modify the roots. The squared distance is the dot product of the difference vector with itself. Dot product can also be expressed with coordinates, so we can establish an implicit equation of the circle in Cartesian coordinates.

Circle has also a famous parametric equation, which is based on the definition of \cos and \sin : If we rotate a unit vector by ϕ around axis z , the x coordinate of the rotated vector is $\cos(\phi)$ and the y coordinate is $\sin(\phi)$.

Rotated vector of length R can be obtained by scaling by R . If the center is not in the origin but at point c , then we should translate the circle points by c .

Gyakorlatok

- 2D:
 - A parabola implicit egyenete (azon pontok mértani helye, amelyek egyenlő távolságban vannak a p ponttól és az (r_0, n) egyenestől),
 - Az ellipszis (a p_1, p_2 pontoktól mért távolságösszeg = C)
 - Koordinátatengelyekkel párhuzamos tengelyű ellipszis paraméteres egyenlete
- 3D
 - Gömb, henger és paraboloid implicit egyenlete
 - Két kitérő egyenes távolsága: (r_1, v_1) és (r_2, v_2)
 - Azon pontok halmaza, amelyek p_1, p_2 pontoktól mért távolságnégyzet összege = C .

Geometriai modellezés

Szirmay-Kalos László

Görbék: 1D ponthalmazok

A pontok koordinátái kielégítenek egy egyenletet:

<p>– implicit: $f(x, y) = 0$</p> <ul style="list-style-type: none"> • 2D egyenes: $ax + by + c = 0$ • Kör: $(x-x_0)^2 + (y-y_0)^2 - R^2 = 0$ 	<p>$f(\mathbf{r}) = 0$</p> <p>$\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0) = 0$</p> <p>$\mathbf{r} - \mathbf{r}_0 ^2 - R^2 = 0$</p>
<p>– parametrikus: $x = x(t), y = y(t)$</p> <ul style="list-style-type: none"> • 3D egyenes: $x = x_0 + v_x t$ $y = y_0 + v_y t$ $z = z_0 + v_z t$ • Kör: $t \in [0, 1]$ $x(t) = x_0 + R \cos 2\pi t$ $y(t) = y_0 + R \sin 2\pi t$ 	<p>$\mathbf{r} = \mathbf{r}(t)$</p> <p>$\mathbf{r} = \mathbf{r}_0 + \mathbf{v} t, \quad t \in [-\infty, \infty]$</p>
<p>– explicit: $y = F(x)$</p> <ul style="list-style-type: none"> • 2D egyenes: $y = mx + b$ 	

If we want to specify 1D objects, like curves, then we should simultaneously identify (uncountably) infinitely many points. Obviously, defining the points one by one with their Cartesian coordinates is not an option. Instead, we usually specify an equation that has infinitely many roots and these roots are considered as the Cartesian coordinates of points in a set defined by the equation. Assume that we are in 2D when the equation should contain Cartesian coordinates x and y (in 3D there would be a third coordinate as well). The equation can have **implicit** form, which means that x and y are put into an algebraic expression that is made equal to zero. We have a single equation with two unknowns thus, we have the hope of having infinitely many roots, i.e. x, y pairs.

For example, a linear equation containing x and y identifies a line. A circle contains points that are at distance R from the reference point. Expressing this distance with the Pythagoras theorem, we can also develop an equation for the circle.

The equation may also have **parametric form**, where we use a free parameter t that can run in an appropriate interval. Substituting t into two equations defining x and y (or z), we get the Cartesian coordinates of the point corresponding to t .

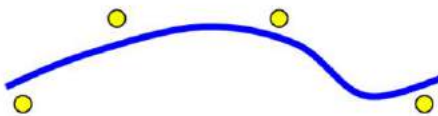
The most obvious, but the least useful equation type is the **explicit form**, where we express y as a function of x . The problem with this representation

is for each x there must be exactly one y . This is usually not the case, think of a circle or a vertical line, for example.

For classic curves, like line, circle, parabola, ellipse, etc. we know their geometric definition, which can be translated to an equation, so the definition means the specification of the free parameters in the equation.

Szabadformájú görbék

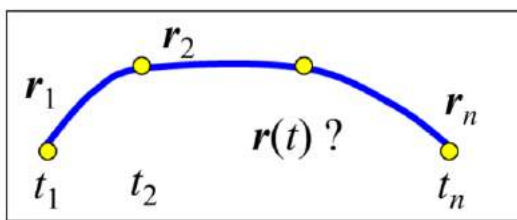
- Definíció vezérlőpontokkal



- Polinom: $x(t) = \sum a_i t^i$, $y(t) = \sum b_i t^i$
- Polinom együtthatók:
 - Interpoláció
 - Approximáció

However, curves we meet usually do not belong to the category of classic curves, so we do not know their equation. These curves are **free form curves**.

As "everything" can be approximated by polynomials, the unknown equations of free form curves are also attacked this way. We approximate their parametric equations with polynomials of parameter t . The problem is that the polynomial coefficients do not have intuitive interpretation, thus we cannot expect the modeler to specify the coefficients directly. Instead, we require the user to specify a finite number of **control points**, and the modeling program automatically computes the polynomial coefficients from the control points. This computation can be an **interpolation** when the resulting curve is expected to go through the control points. Or, the computation can also be an **approximation**, when the resulting curve should just somehow follow the control points, but it does not have to go through each of them. By requiring approximation instead of interpolation, we ease the fitting process so we can impose additional requirements concerning the "quality" of the curve.



Lagrange interpoláció

- Vezérlő pontok: $r_1, r_2, r_3, \dots, r_n$
- Keresd: $r(t) = \sum [a_i, b_i] t^i$ amelyre
 $r(t_1) = r_1, r(t_2) = r_2, \dots, r(t_n) = r_n,$
- Megoldás:

$$r(t) = \sum L_i(t) r_i \quad \rightarrow \quad r(t_k) = \sum L_i(t_k) r_i = r_k$$

$$L_i(t) = \frac{\prod_{j \neq i} (t - t_j)}{\prod_{j \neq i} (t_i - t_j)} \quad L_i(t_k) = \frac{\prod_{j \neq i} (t_k - t_j)}{\prod_{j \neq i} (t_i - t_j)} \quad \begin{cases} 1 & \text{if } i=k \\ 0 & \text{if } i \neq k \end{cases}$$

The first curve is of interpolation type and is known as the **Lagrange interpolation**.

Suppose we specify a sequence of control points r_1, \dots, r_n , and search a parametric function $r(t)$ (one polynomial for each of the x, y or x, y, z coordinates) that goes through them. More precisely, we expect the curve to give control point r_1 for parameter value t_1 , r_2 for t_2 , etc. The interpolation requirement means n constraints, thus the polynomials may have n unknown coefficients to make the number of unknowns equal to the number of equations, and thus obtaining a well defined problem with an unambiguous solution. To find the n unknown polynomial coefficients, we need to solve a linear equation generated by substituting t_1, \dots, t_n into the polynomial and requiring them to be equal to r_1, \dots, r_n , respectively. If we solve it, we obtain the coefficients, which allow the computation of the curve point for arbitrary parameter t .

Instead of solving the linear equation, the solution can be given directly as a combination of the control points with barycentric coordinates $L_i(t)$ that depend on parameter t . The algebraic form of these weight functions, aka **basis functions** or **blending functions** is shown here as the ratio of two products.

To prove that the combination of the control points with these functions satisfies the interpolation constraints, let us examine a basis function L_i when we substitute t_k into it. If $i=k$, the numerator and the denominator of L_i will be similar, so $L_i(t_i) = 1$. However, when $i \neq k$, there will be some j which equals to k , so one factor of the numerator will be $t_k - t_k = 0$, making $L_i(t_k)$ also zero. So L_i is 1 for t_i but is zero for all other discrete parameter values. This means that in sum $L_i(t_k) r_i$, all control points r_i get zero weight except r_k , which gets weight 1, thus $r(t_k) = r_k$.

Note: A point of the Lagrange curve is the combination of control points with weights L_i . According to the definition of combination, the reference points (which are the control points here) should be multiplied with the corresponding weights, the terms should be

added them up, and finally the sum be divided with the total mass. Where is this division? The division can be ignored if the total mass is 1. Is the sum of the weight functions equal to 1 for any t ?? (Yes).

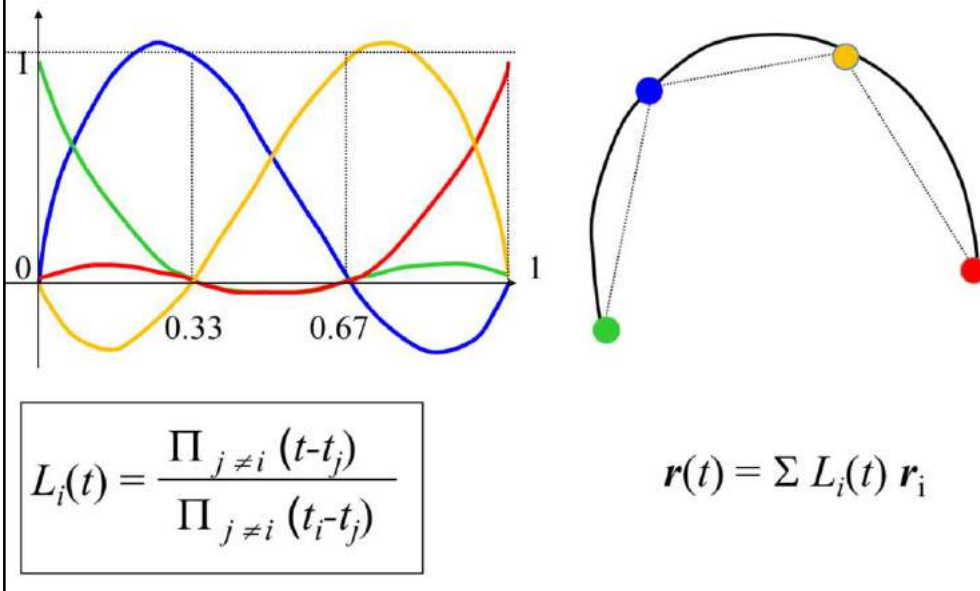
LagrangeCurve

```
class LagrangeCurve {
    vector<vec3> cps;           // control points
    vector<float> ts;           // parameter (knot) values

    float L(int i, float t) {
        float Li = 1.0f;
        for(int j = 0; j < cps.size(); j++)
            if (j != i) Li *= (t - ts[j]) / (ts[i] - ts[j]);
        return Li;
    }
public:
    void AddControlPoint(vec3 cp) {
        float ti = cps.size(); // or something better
        cps.push_back(cp); ts.push_back(ti);
    }

    vec3 r(float t) {
        vec3 rr(0, 0, 0);
        for(int i = 0; i < cps.size(); i++) rr += cps[i] * L(i,t);
        return rr;
    }
};
```

Lagrange interpoláció bázisfüggvényei

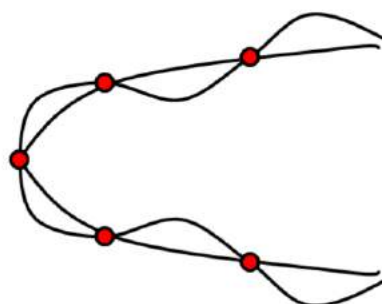
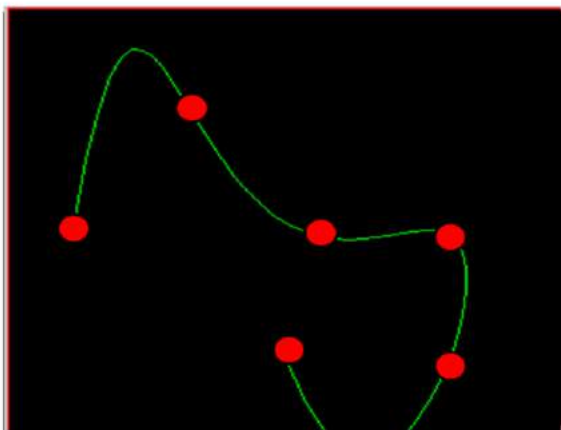


Let us take an example where there are four control points and we expect the curve to interpolate them for $t=0, 0.33, 0.67$, and 1 , respectively. The basis functions are depicted in the Figure. When $t=0$, the weight of the green point is 1 , and the weight of all other points is zero. The curve is then in the green point. When t increases, the weight of the red point gets larger and at $t=0.33$ only the red point has non-zero weight...

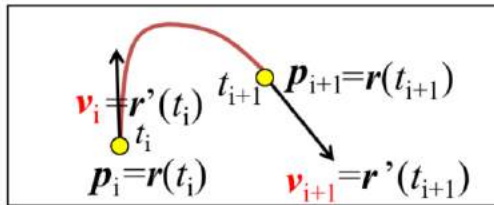
The basis functions **oscillate** between positive and negative values, thus a control point periodically attracts or repels the curve. This is bad since the curve will tend to oscillate.

The other disadvantage of Lagrange interpolation is that it cannot provide **local control**. Local control would mean that the modification of a control point modifies only a smaller part of the curve. However, as all basis functions are non-zero in the whole domain, the complete curve will change.

Lagrange interpoláció problémái



Hermite interpoláció



- $r(t) = a_3(t-t_i)^3 + a_2(t-t_i)^2 + a_1(t-t_i) + a_0$
- $r'(t) = 3a_3(t-t_i)^2 + 2a_2(t-t_i) + a_1$

$$a_0 = p_i$$

$$a_1 = v_i$$

$$a_2 = \frac{3(p_{i+1} - p_i)}{(t_{i+1} - t_i)^2} - \frac{v_{i+1} + 2v_i}{(t_{i+1} - t_i)}$$

$$a_3 = \frac{2(p_i - p_{i+1})}{(t_{i+1} - t_i)^3} + \frac{v_{i+1} - v_i}{(t_{i+1} - t_i)^2}$$

$$r(t_i) = a_0 = p_i$$

$$r(t_{i+1}) = a_3(t_{i+1}-t_i)^3 + a_2(t_{i+1}-t_i)^2 + a_1(t_{i+1}-t_i) + a_0 = p_{i+1}$$

$$r'(t_i) = a_1 = v_i$$

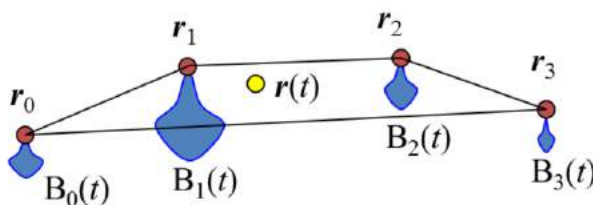
$$r'(t_{i+1}) = 3a_3(t_{i+1}-t_i)^2 + 2a_2(t_{i+1}-t_i) + a_1 = v_{i+1}$$

Hermite (H at the beginning and e at the end are silent because he was a Frenchman) interpolation is a generalization of Lagrange interpolation, where not only the points to be interpolated are given but also the derivatives. Here we discuss only the practically relevant special case, when the curve is defined by two control points and the first derivatives at these control points. We have four constraints, so the polynomial that is unambiguously determined by these constraints if a cubic (of four polynomial coefficients).

The strategy is (always) similar to that of the Lagrange interpolation. We take the polynomial with yet unknown coefficients, substitute the constraints, and get a linear equation for the unknown coefficients. This linear equation is solved.

Bezier approximáció

- Keresd: $r(t) = \sum B_i(t) r_i$
 - $B_i(t)$: ne oszcilláljon
 - Konvex burok tulajdonság
 - $B_i(t) \geq 0$, $\sum B_i(t) = 1$



Lagrange (and Hermite) interpolation tends to oscillate. Let us find a better curve. We still use the center of mass analogy, i.e. the curve will be the composition of control points with weights placed at them. The weights are basis functions $B_i(t)$ and we can ignore division with the total mass if the sum of weights is guaranteed to be equal to 1.

We do not want the oscillation of the Lagrange curve, so we allow only non-negative weights. Composition with non-negative weights is a convex combination, thus all points of the curve, i.e. the complete curve will be in the convex hull of the control points.

Bernstein polinomok

Newton f le binomi lis t tel

$$1^n = (t + (1-t))^n = \sum_{i=0}^n \boxed{\binom{n}{i} t^i (1-t)^{n-i}}$$

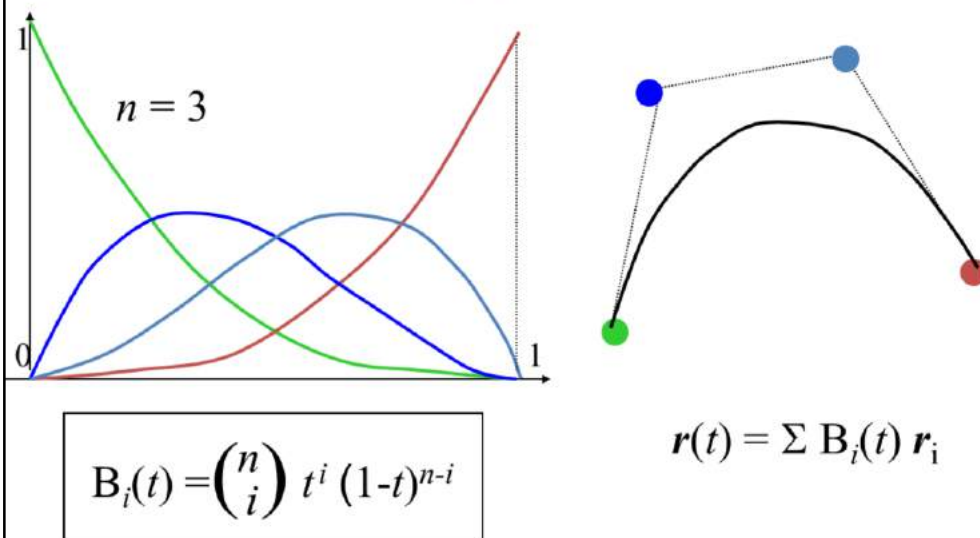
$B_i(t)$

$$B_i(t) \geq 0, \sum B_i(t) = 1: \text{ OK}$$

So, the task is to find a basis function system where each basis function is non-negative in the allowed domain (in $[0,1]$) and their sum is everywhere 1.

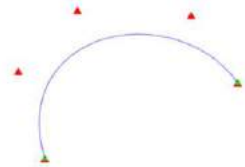
Such basis functions can be constructed by expressing 1 with the Newtonian binomial theorem. The terms are called Bernstein polynomials, which are indeed non-negative if t is in $[0,1]$, and their creation guarantees that their sum is 1.

Bezier approximáció



If $n = 3$ (which is good for 4 control points), the basis functions are $(1-t)^3$, $3*(1-t)^2*t$, $3*(1-t)*t^2$, t^3 . Note that the first basis function is 1 for $t=0$, while all others are zero, so the curve crosses the first control point. Similarly, when $t=1$, the curve is at the last control point. However, other control points are not so lucky, they are usually not interpolated. This is an approximation curve.

BezierCurve



```
class BezierCurve {
    vector<vec3> cps; // control points

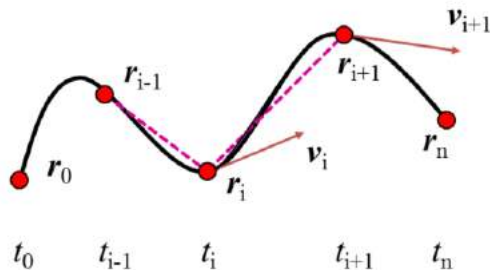
    float B(int i, float t) {
        int n = cps.size()-1; // n deg polynomial = n+1 pts!
        float choose = 1;
        for(int j = 1; j <= i; j++) choose *= (float)(n-j+1)/j;
        return choose * pow(t, i) * pow(1-t, n-i);
    }
public:
    void AddControlPoint(vec3 cp) { cps.push_back(cp); }

    vec3 r(float t) {
        vec3 rr(0, 0);
        for(int i = 0; i < cps.size(); i++) rr += cps[i] * B(i,t);
        return rr;
    }
};
```

Catmull-Rom spline

Minden két vezérlőpont közé egy görbe szegmens

Simaság: a sebesség is legyen közös két egymás utánira



$$v_i = \frac{1}{2} \left(\frac{r_{i+1} - r_i}{t_{i+1} - t_i} + \frac{r_i - r_{i-1}}{t_i - t_{i-1}} \right)$$

Egy görbeszegmens: Hermite interpoláció

Legeslegelső és legutolsó sebesség explicite

Let us define a separate curve segments between every two control points applying Hermite interpolation. Hermite interpolation needs the start and end points (which are available) and the derivatives at these two points (which should be found somehow).

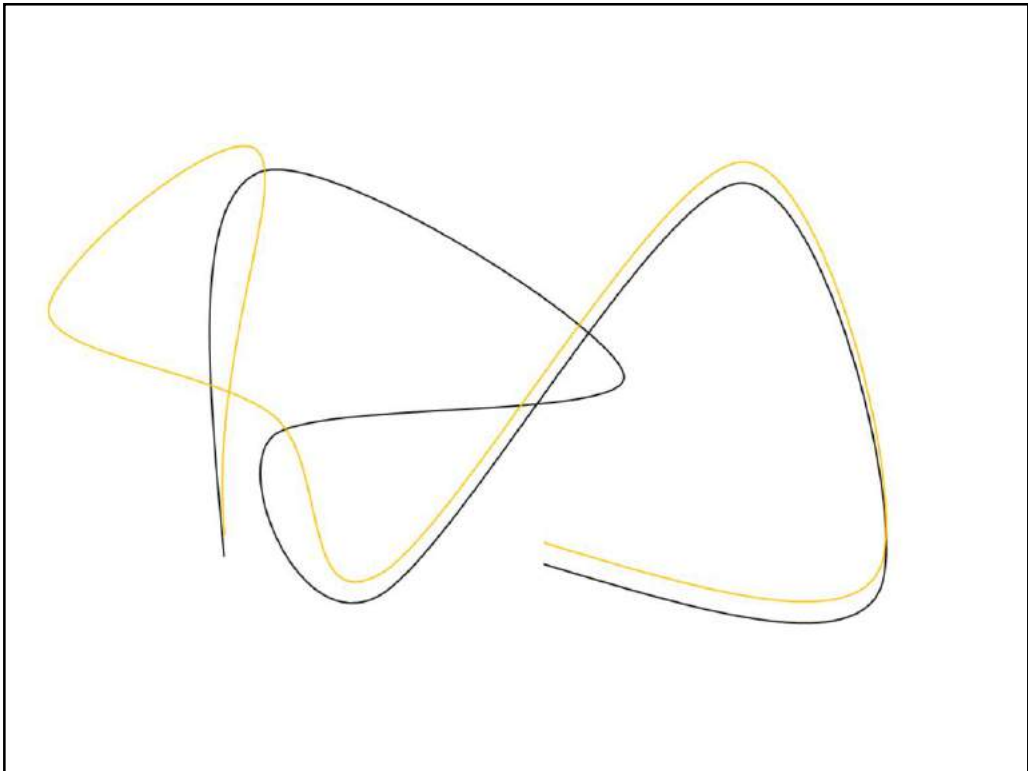
If the speed is uniform and the motion is linear in segment i , then its constant speed equals to $(r_{i+1} - r_i) / (t_{i+1} - t_i)$.

Similarly the constant speed in segment $i+1$ would be $(r_{i+2} - r_{i+1}) / (t_{i+2} - t_{i+1})$. A good approximation is to set the velocity at the control point shared by the two segments to the average of these two velocities. This is the Catmull-Rom spline.

Kochanek and Bartels further generalized this spline and allowed an additional tension parameter that can scale up or down the average velocity. On the other hand, we can use a weighted average when the average of the two constant speeds is obtained.

CatmullRom

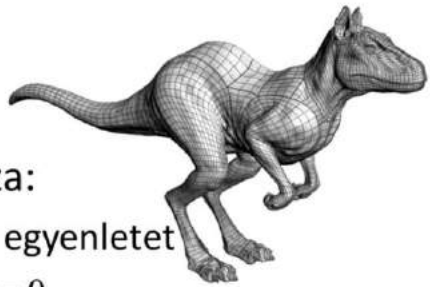
```
class CatmullRom {  
    vector<vec3> cps;           // control points  
    vector<float> ts;          // parameter (knot) values  
  
    vec3 Hermite( vec3 p0, vec3 v0, float t0,  
                  vec3 p1, vec3 v1, float t1,  
                  float t ) {  
  
        ???  
  
    }  
  
public:  
    void AddControlPoint(vec3 cp, float t) { ??? }  
  
    vec3 r(float t) {  
        for(int i = 0; i < cps.size() - 1; i++) {  
            if (ts[i] <= t && t <= ts[i+1]) return Hermite(...);  
        }  
    }  
};
```



The Catmull-Rom spline can be found in PowerPoint and in many drawing packages. It is an interpolating spline with **local control**.

When we move a control point, the average speeds of two linear uniform motions are modified. Thus, the averages of these linear motions are changed at three control points, which can affect four curve segments at most.

Felületek



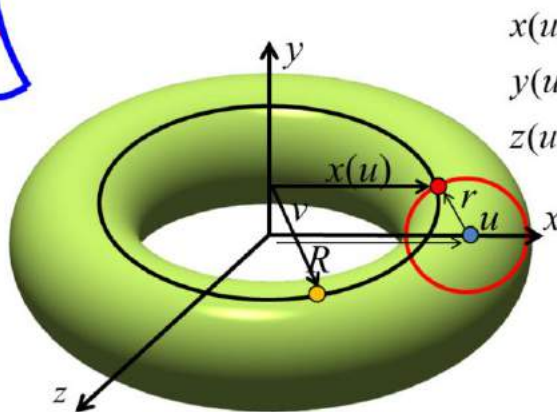
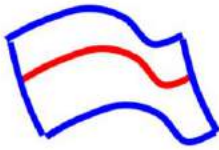
Felület a 3D tér 2D részhalmaza:

- Koordináták kielégítenek egy egyenletet
- **implicit**: $f(x, y, z) = 0$
 - gömb: $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2 = 0$
- **parametrikus**: $x = x(u, v), y = y(u, v), z = z(u, v),$
 $u, v \in [0, 1]$
 - gömb $x = x_0 + R \cos 2\pi u \sin \pi v$
 $y = y_0 + R \sin 2\pi u \sin \pi v$
 $z = z_0 + R \cos \pi v \quad u, v \in [0, 1]$
- **explicit** (magasságmező):
 $z = h(x, y)$

Surfaces are two dimensional subsets of the 3D space. Their definition is very similar to that of curves, but now the parametric equations have two free parameters (parametric equations of curves map a line segment onto the curve, parametric equations of surfaces map a square onto the surface).

$$\mathbf{r}(u, v) = \mathbf{r}_v(u)$$

Tórusz



$$x(u) = R + r \cos(u)$$

$$y(u) = r \sin(u)$$

$$z(u) = 0$$

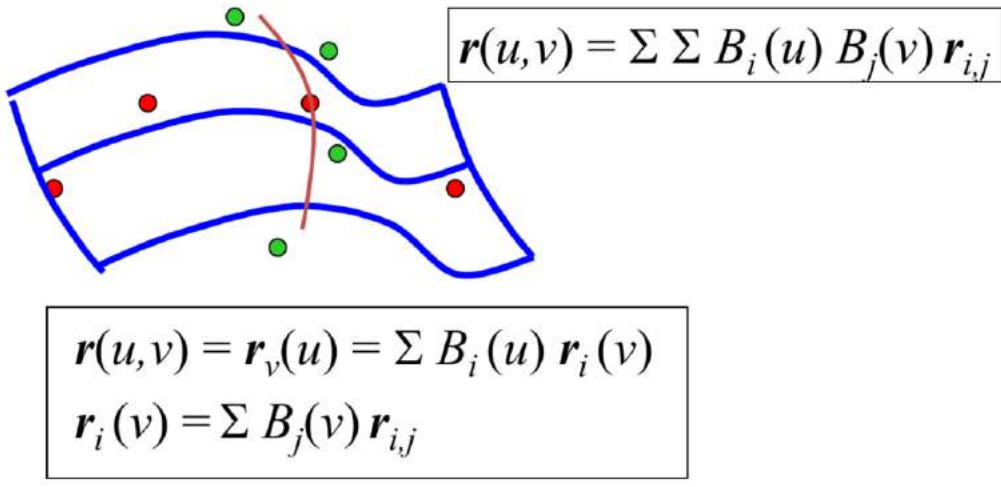
$$x(u, v) = x(u) \cos(v) = (R + r \cos(u)) \cos(v)$$

$$y(u, v) = y(u) = r \sin(u)$$

$$z(u, v) = x(u) \sin(v) = (R + r \cos(u)) \sin(v)$$

Szabadformájú felület

Definíció vezérlőpontokkal

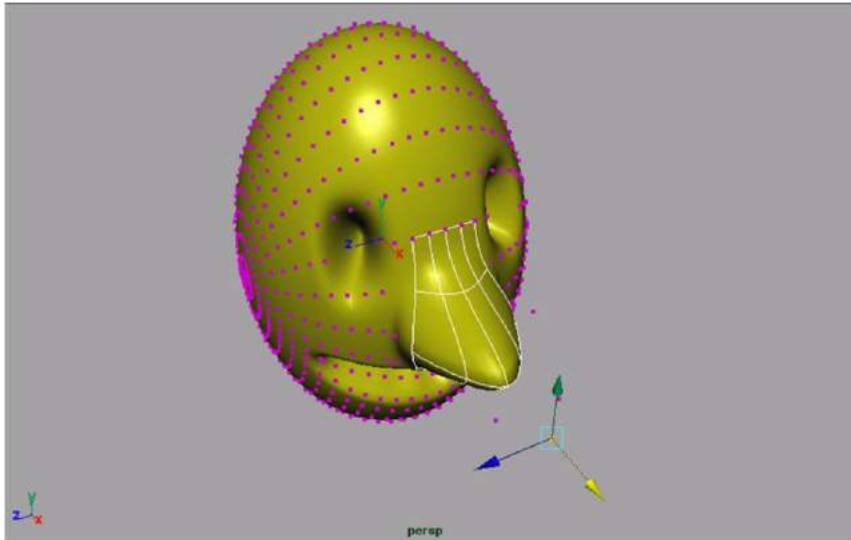


The definition of curves traced back the problem to the specification of a few control points. We use the same approach here.

First, we trace back the definition of surfaces to curves. Let us fix one of the free variables of the surface, which results in a one-variable parametric form, a curve. This curve is on the surface and is called **isoparametric curve**. A curve can be well defined by control points. Now let us fix the isoparametric value differently, which leads to another isoparametric curve that can be defined with different control points. As the isoparametric value changes, the control points of the corresponding isoparametric curve also change. These changes are also curves, so we can express the path of the control point by blending other control points.

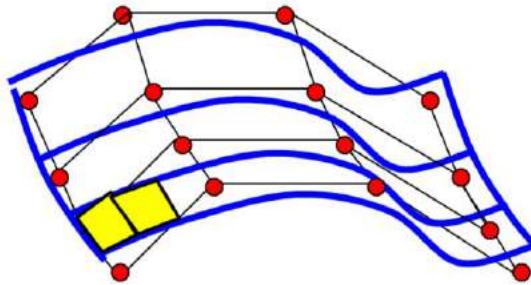
Substituting this into the equation of the isoparametric curve, we obtain the equation of the surface, which is a combination of control points forming a **control cage** or **control polyhedron**. The blending or weighting function of control point $\mathbf{r}_{i,j}$ is the product of basis functions B_i parameterized with u , and B_j parameterized with v .

Vezérlőpontok módosítása



Surface definition is basically the modification of control points that attract the surface if weights are non-negative.

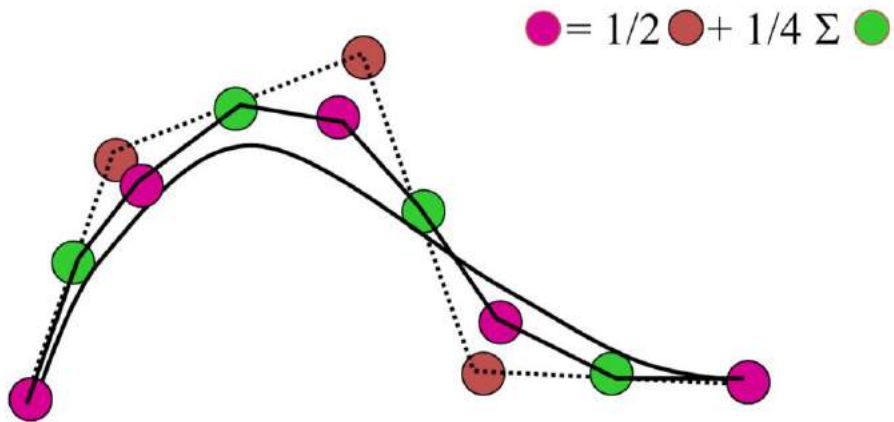
Poligonháló finomítása



So far we used the following strategy. We started with the control cage or control mesh. Using the center of mass analogy, a continuous and smooth parametric surface is developed. However, when we render this smooth surface, we should decompose it to small triangles since the GPU can handle only triangles and not smooth surfaces. So the very beginning of this process is a rough mesh and the very end is a fine mesh.

Can we get rid of the complicated mathematics of blending, splines, smooth interpolation etc. and obtain the fine mesh directly from the rough mesh?

Subdivision görbék



Subdivision curves or surfaces are based exactly on this idea.

Let us consider a curve defined by a few control points. The polyline connecting the control points is a rough approximation of the desired smooth curve. This rough polyline is refined by subdividing it by inserting a point at the middle of each line segment and then moving the original vertices to the weighted average of their original location and the two middle points.

The new polyline looks smoother. If we are not satisfied, we can repeat the process recursively.

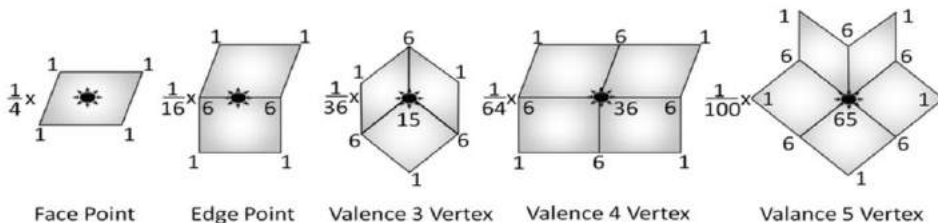
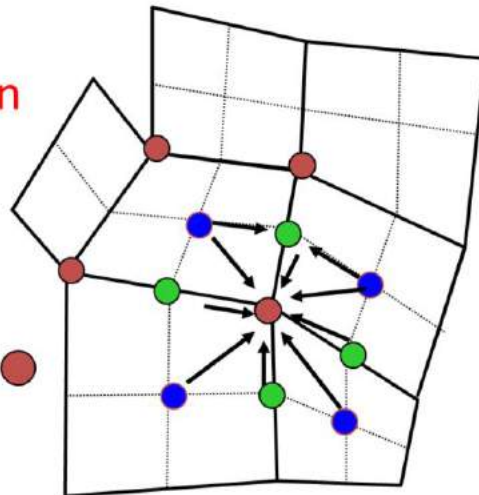
Catmull-Clark subdivision felület

$$\bullet = 1/4 \Sigma \bullet$$

$$i = 1/2 \Sigma \bullet$$

$$\bullet = 1/v^2 \Sigma \bullet + 2/v^2 \Sigma i + (v-3)/v \bullet$$

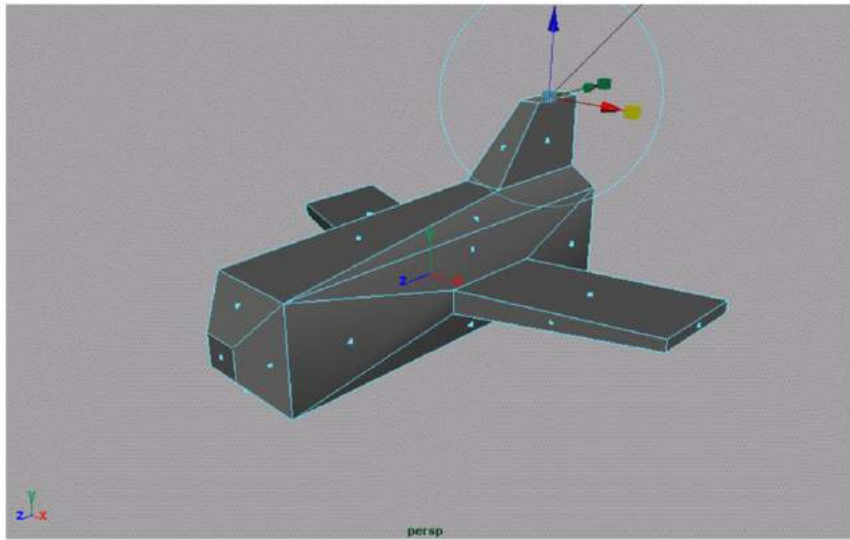
$$\bullet = 1/4 \Sigma \bullet + 1/2 \Sigma i$$



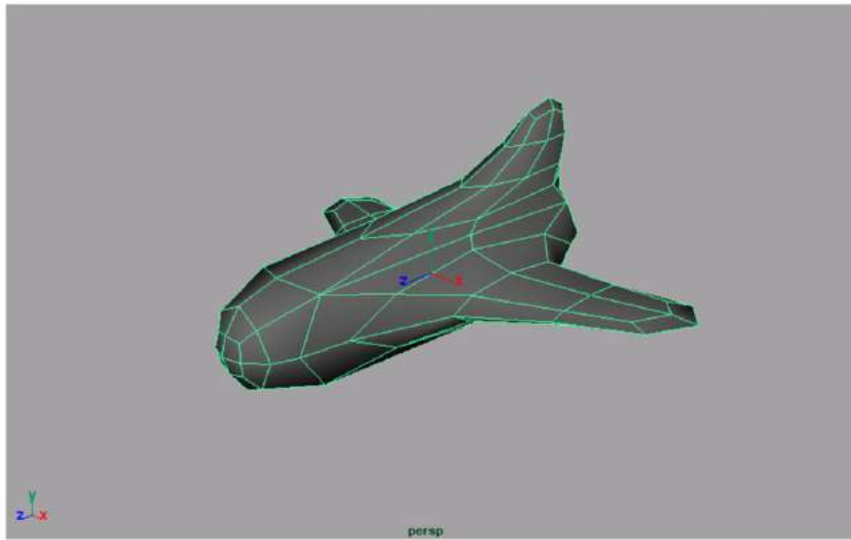
The idea can be extended to surfaces. Here only one type of subdivision surfaces is introduced, which is called the Catmull-Clark surface. We assume that the original mesh is built of quadrilaterals. Although the algorithm can work with other meshes as well, after the first subdivision step, the mesh will always be a **quadrilateral mesh**.

The subdivision starts by the computation of **face center** and **edge center** points, which double the resolution but do not alter the shape yet. Then we first move the original vertices to the weighted average of the surrounding face centers, edge centers and of the point itself. The averaging scheme also depends on how many faces share this point, which is called the **valence** of this vertex. Having moved the original vertices, we find the final location of the edge centers as well.

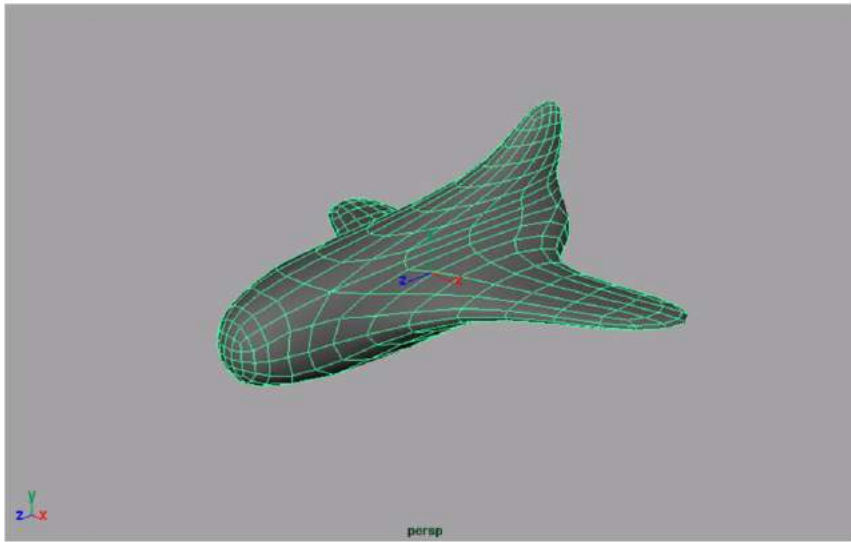
Durva polygon modell



Subdivision 1

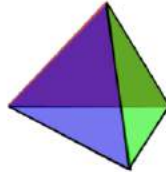


Subdivision 2



B-rep

- Test = határoló felületek



- Topológiai érvényesség (Euler tétel ha egy darabból áll és nincsenek benne lyukak) :

$$\text{csúcsok} + \text{lapok} = \text{élek} + 2$$

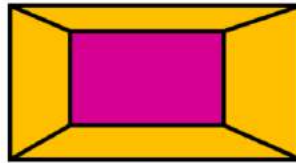
Boundary representation defines a solid by the boundary surfaces or faces. Specifying the boundaries independently would not work since it would be possible to create something where the boundaries do not enclose a 3D solid or the enclosing is not watertight. We should specify edges, faces and vertices simultaneously to always guarantee that the object is topologically valid.

A famous equation that can be used to check topological validity is the **Euler theorem**. It can be applied for 3D objects that are isomorphic to a sphere (they turn to a sphere when pumped up). Objects with holes or consisting of multiple independent pieces do not belong this category (they are isomorphic to a torus or more than one sphere). The Euler equation can be generalized to cover these cases as well, when it is called **Euler-Poincare equation**.

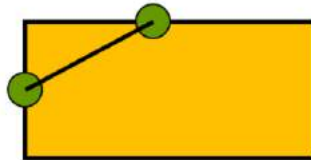
With the Euler's theorem, when we have an object, counting the vertices, faces and edges allows the determination whether or not this object is valid. However, when it turns out that it is invalid, it is usually too late. What we need is elementary operations that keep the validity of the Euler equation provided it was valid before the application of the operation. Such elementary operations are called Euler operations.

Euler operátorok

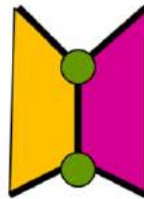
- Lap kihúzás



- Lap felvágás



- Él törlés



- Csúcs szétvágás



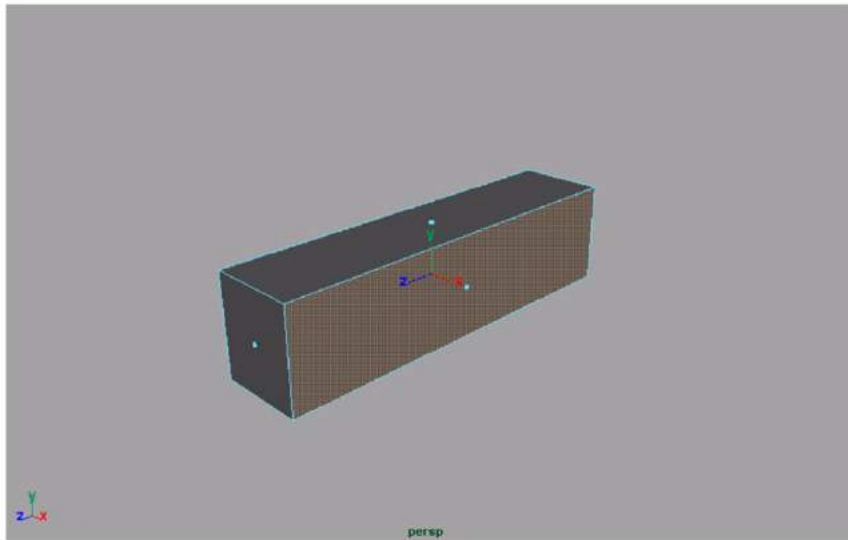
A few examples of Euler operators are shown here. Face extrude extrudes a face by automatically filling the holes between the original and extruded face with new faces and edges. Counting the numbers of new faces (4), edges (8) and vertices (8), we can prove that it is indeed an Euler operator.

Face split requires the user to select two edges of a face and to specify two points on them. These new points are connected by a new edge.

Altogether, this operation introduces 2 vertices, 3 edges (one new and two that are obtained as the subdivision of the original edges with the new points) and a new face (the new edge subdivides the face into two).

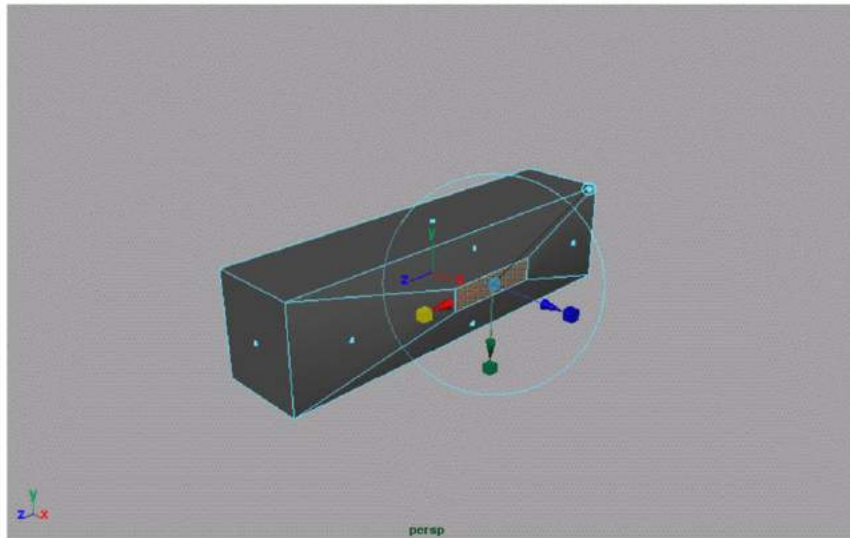
Edge collapse removes an edge with one of its end points. Vertex split is the inverse of this operation.

Kezdet: érvényes téglatest

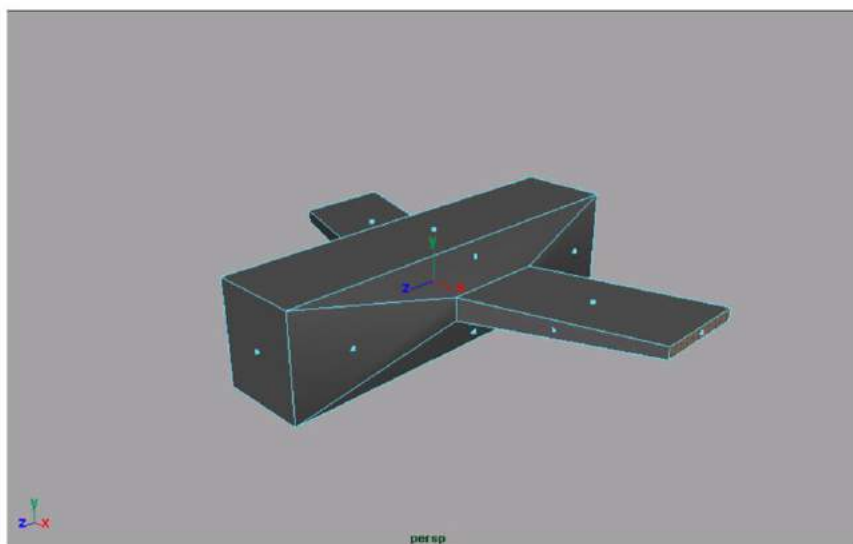


To create a space ship, we can start with a topologically valid object, e.g. a transformed cube, then we execute a sequence of face extrusions. As face extrude is an Euler operator, the result will automatically be a topologically valid object.

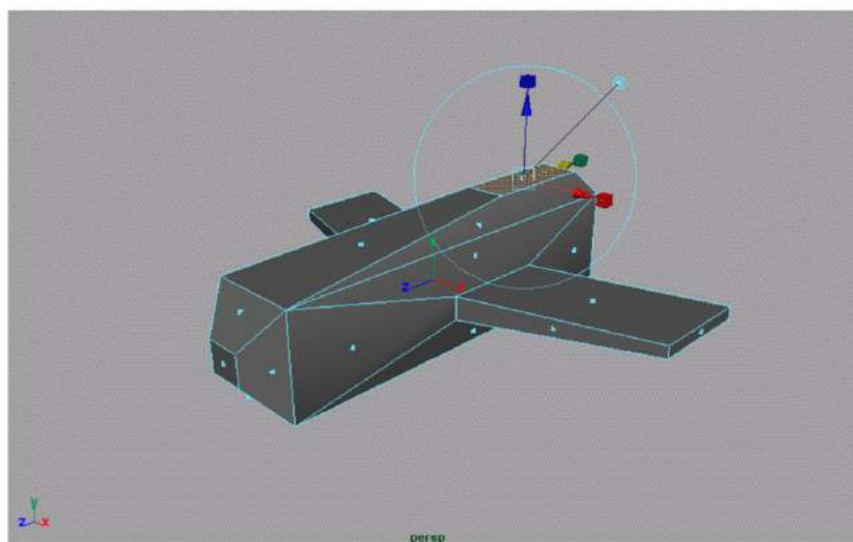
Lap kihúzás



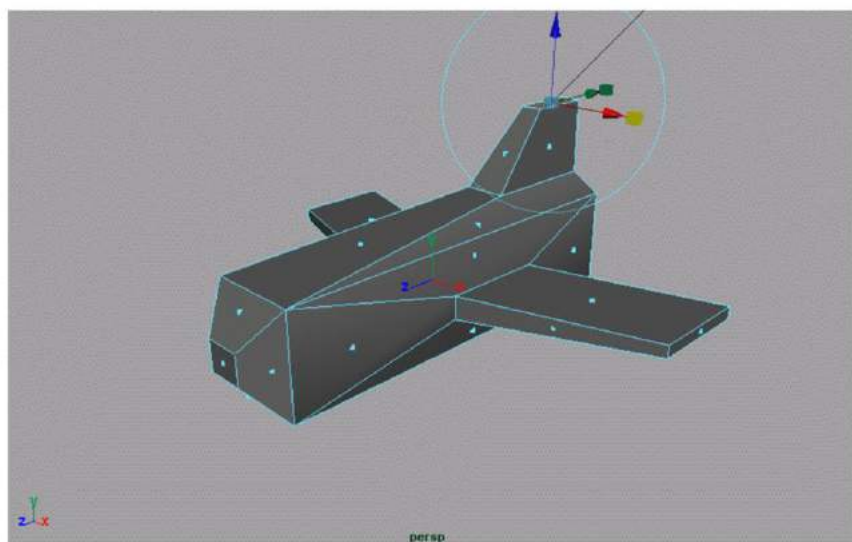
Lap kihúzás



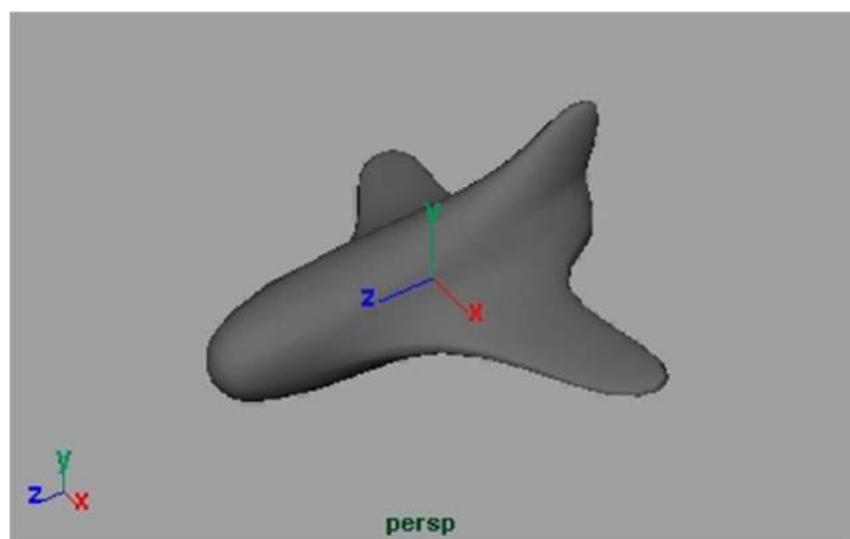
Lap kihúzás



Lap kihúzás

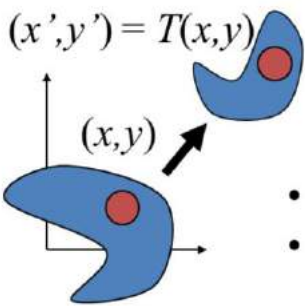


Subdivision simítás



Transzformációk

Szirmay-Kalos László

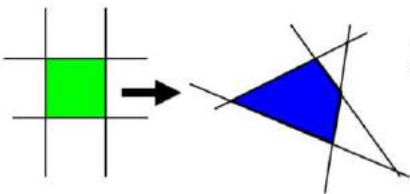


$(x', y') = T(x, y)$

(x, y)

Transzformációk

- Tönkre tehetik az egyenletet
- Korlátozzuk a transzformációkat és az alakzatokat úgy, hogy invariáns legyen
 - Pont, egyenes (szakasz), sík (poligon)
- **Affin transzformációk**
 - Párhuzamos egyenes tartó
 - Descartes koordinátákban lineáris



Homogén lineáris transzformációk

Egyenest egyenesbe
Homogén koordinátákban lineáris

Geometric transformations assign a point to a point, so it is a point valued function of points.

Geometric transformation may destroy the equation and the type of an object. Even simple scaling turns a sphere into an ellipsoid, so the equation, program, representation will change. To avoid this, we limit the allowed transformations and object types to those which guarantee that the object type is preserved. Linear elements, like points, line segments, and polygons may approximate any 0,1 or 2 dimensional object.

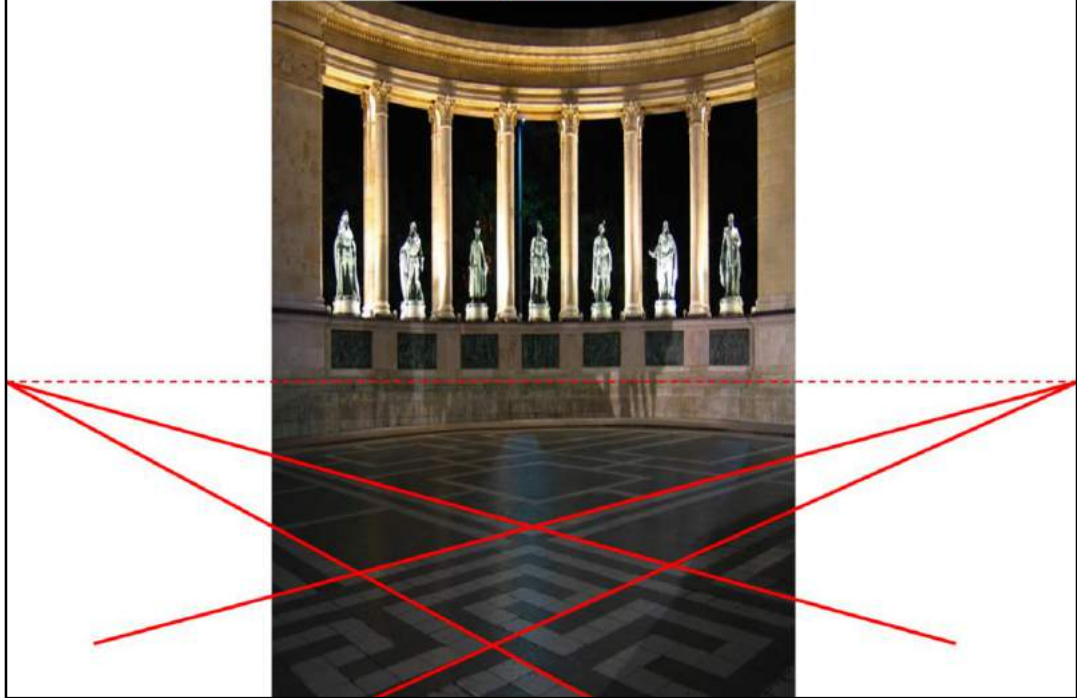
Affine transformations that can be expressed as linear functions of the Cartesian coordinates map lines to lines and also preserve parallel lines.

This theorem can be proved by realizing that a line can have a linear equation and with linear equation only lines can be described. So, if a linear equation of a line is combined with the linear function of the transformation, we get a linear equation, which thus must be a line. If this transformation could make parallel lines intersection or intersedted lines parallel, then this transformation would create a point out of nothing or would make a point disappear. A linear function is not able to do that.

Affine transformations are not the widest set of transformations preserving lines and polygons. The widest set is homogeneous linear transformations (homogeneous coordinates are multiplied by a matrix), which includes central projection as well.

To find this wider set of transformations, we should understand that no transformation of the Euclidean plane can make two parallel lines intersecting, since that would create a point from nothing. The problem is the Euclidean geometry itself and its property that parallel lines do not intersect. To consistently discuss how lines can be transformed to lines without keeping the parallelism, we should step out of the Euclidean geometry. The proper geometry is the projective geometry.

Perspektíva



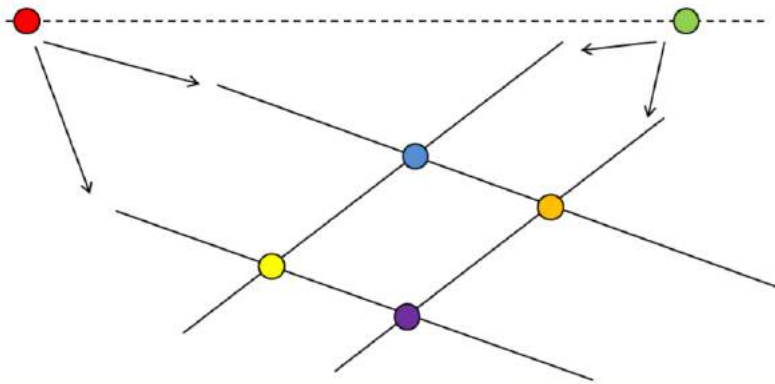
We can see the transformations of parallel lines to intersecting ones in every moment of our life. The phenomenon is called perspective.

Grafika vizsgák javítása



Perspective of the table.

Euklideszi → Projektív sík



- Két pont meghatároz egy egyenest.
- Egy egyenesnek van legalább két pontja.
- Ha a egy egyenes, A pedig egy, nem az egyenesen lévő pont, akkor egyetlen olyan egyenes létezik, amely átmeny A -n és nem metszi a -t.

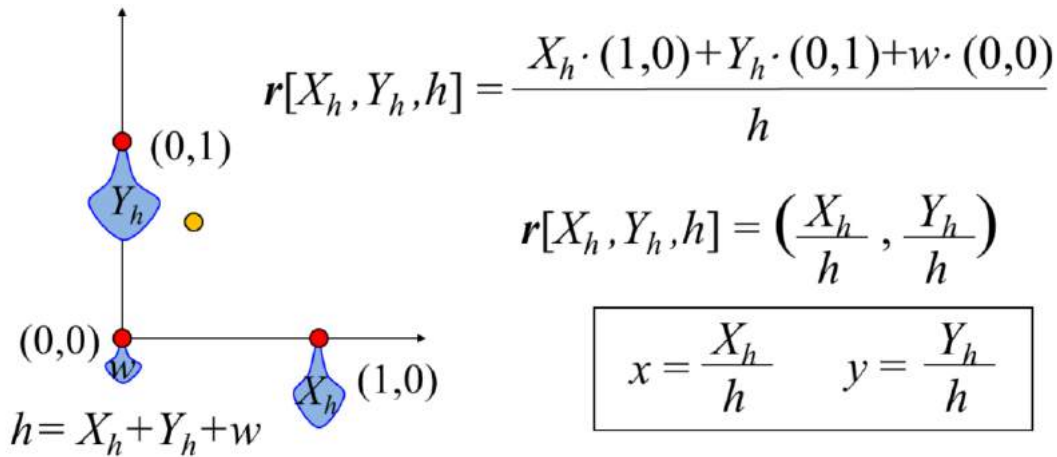


- Két pont meghatároz egy egyenest.
- Egy egyenesnek van legalább két pontja.
- **Két egyenes mindig egy pontban metszi egymást.**

To establish projective geometry, the axioms need to change. The parallel axiom of the Euclidean geometry is deleted, and instead of this we postulate that „two lines intersect each other in exactly one point”. As a result, the Euclidean plane must be extended with ideal points. Each line is given one ideal point, assigning the same ideal point to two lines if and only if they are parallel. Ideal points will be on a line.

Homogén koordináták (2D)

$$(x, y) \rightarrow [x, y, 1] \sim [x \cdot h, y \cdot h, h] = [X_h, Y_h, h]$$

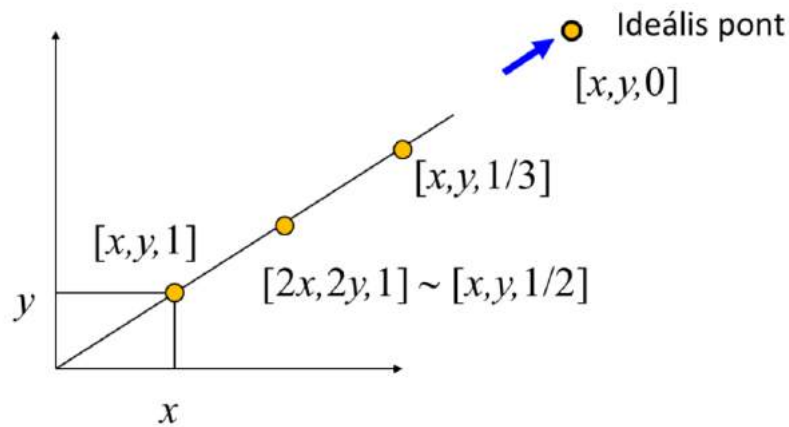


Homogeneous coordinates are defined by extending the Cartesian coordinates by an additional coordinate that is equal to 1, and multiplying all coordinates by an arbitrary non-zero scalar h . An intuitive interpretation of homogeneous coordinates in 2D is the following: we put weight X_h in point (1,0), weight Y_h in point (0,1), and $w=h-X_h-Y_h$ in the origin. Value h is the total mass distributed. Three numbers X_h, Y_h, h identify a point in 2D which is the center of mass of this mechanical coordinate system.

Based on the construction, it is obvious that any point of the Euclidean space, which can be given by Cartesian coordinates, can also be represented by homogeneous coordinates with non zero h . It is also true that any homogeneous coordinate triple where h is not zero, can also be given by Cartesian coordinates, which can be obtained by dividing the first two coordinates by the third.

So, if h is not zero, homogeneous coordinates can represent the same set of points as Cartesian coordinates.

Homogén koordináták ideális pontokhoz: $h=0$



Euklideszi sík+ ideális pontok = projektív sík

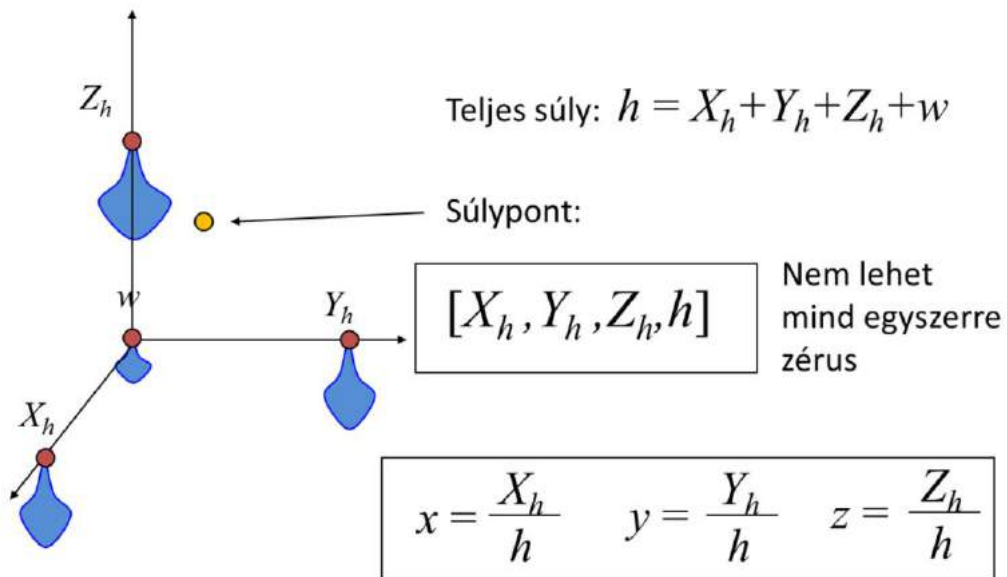
Homogeneous coordinates $[Xh, Yh, h]$ can also be interpreted in the following way: (Xh, Yh) specify the direction of the point, and h is a scaling of the distance.

Let us consider a point of Cartesian coordinates x, y , which can be given in homogeneous coordinates as $[x, y, 1]$.

Now, let us consider another point that is in the same direction, but twice as far as (x, y) . This farther point is $(2x, 2y)$ in Cartesian coordinates, $[2x, 2y, 1]$ in homogeneous coordinates, or $[x, y, 1/2]$ in homogeneous coordinates. Similarly, the point that is also in the same direction but is f times farther away is $[x, y, 1/f]$. So the interpretation of a homogeneous triplet is that the first two coordinates are Cartesian ones and show the direction, and the third coordinate is an inverse scaling of the distance. When f is infinity, so $1/f$ is zero, then we get $[x, y, 0]$, which is at the direction of (x, y) , but at infinity.

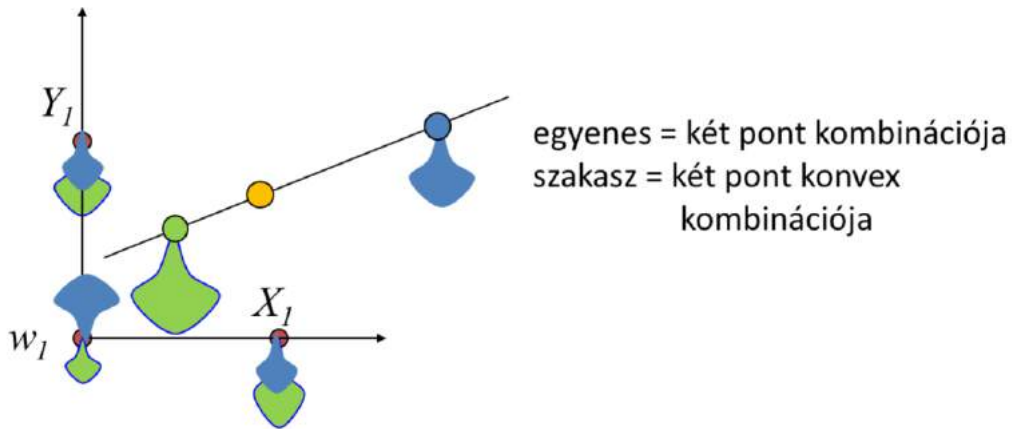
With homogeneous coordinates we can express **ideal points**, i.e. points at infinity that are the intersections of parallel lines. Note that in Euclidean geometry parallel lines do not intersect. So, when we work with homogeneous coordinates instead of Cartesian ones, we describe the projective plane that contains the ideal points as well, and not the Euclidean plane.

Homogén koordináták (3D)



3D points can also be represented with homogeneous coordinates, i.e. the 3D Cartesian space can also be extended to 3D projective space. The center of mass analogy puts weight X_h at reference point $(1, 0, 0)$, weight Y_h at $(0, 1, 0)$, weight Z_h at $(0, 0, 1)$, and finally $w = h - X_h - Y_h - Z_h$ at the origin. Using the definition of the center of mass, from a quadruple of homogeneous coordinates, the corresponding Cartesian coordinate triplet can be obtained by homogeneous division (of course, only if h is not zero).

Egyenes a projektív térben



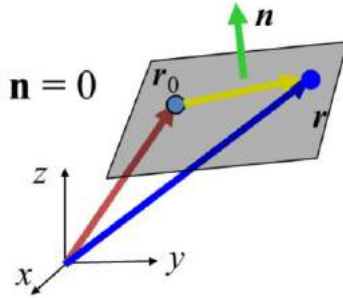
$$[X(t), Y(t), Z(t), h(t)] = [X_1, Y_1, Z_1, h_1] \cdot t + [X_2, Y_2, Z_2, h_2] \cdot (1-t)$$

We shall transform not only points but lines and planes as well, so we need the equations of lines and planes in homogeneous coordinates. We use the center of mass analogy. A point is specified by placing X_1, Y_1, Z_1, h_1 weights at the ends of the basis vectors and the origin respectively, and another point is specified with X_2, \dots weights. Both mechanical systems can be replaced by equivalent systems storing all weights in the center of mass. So when the two systems are combined, the final center of mass will be along a line between the two centers of masses. If we increase the weights of the first mechanical system proportionally scaling all weights, the location of the center of mass of the first system does not change, but it has larger total mass. So the center of mass of the combined system moves towards the first system along the line of the two centers of masses.

Thus, using this combination, we can obtain points on the line defined by the two centers of masses. If scaling is not negative, then we obtain the convex combination of the two points, which is a line segment. With allowing negative scaling, the total line can be specified.

Sík

$$(\mathbf{r} - \mathbf{r}_0) \cdot \mathbf{n} = 0$$



Euklideszi tér, Descartes koordináták:

$$n_x x + n_y y + n_z z + d = 0$$

Euklideszi tér, homogén koordináták:

$$n_x X_h/h + n_y Y_h/h + n_z Z_h/h + d = 0 \quad h \neq 0$$

Projektív tér:

$$n_x X_h + n_y Y_h + n_z Z_h + d h = 0$$

~~$h \neq 0$~~

$$[X_h, Y_h, Z_h, h] \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = 0$$

In Euclidean geometry, using Cartesian coordinates, the plane is a linear equation of the coordinates. To find the plane in projective space, the points at infinity are added to this plane. First, Cartesian coordinates are replaced by homogeneous ones, assuming that h is not zero (it is forbidden to divide by zero). Then, both sides are multiplied with h . In this new equation we do not divide by h , so we can ignore the "h is not zero" requirement. This corresponds to adding ideal points to the plane.

The projective plane is thus a homogeneous linear equation of homogeneous coordinates. We can also express it as a dot product of two 4D vectors, one describes the point, the other the parameters of the plane.

Homogén lineáris transzformációk

- **Homogén koordinátavektor szorzása mátrixszal**

- 2D transzformáció 3x3 mátrix

$$[X_h', Y_h', h'] = [X_h, Y_h, h] \cdot \mathbf{T}_{3 \times 3}$$

- 3D transzformáció 4x4 mátrix

$$[X_h', Y_h', Z_h', h'] = [X_h, Y_h, Z_h, h] \cdot \mathbf{T}_{4 \times 4}$$

- Transzformációk konkatenációja: Asszociatív

$$\begin{aligned} [X_h', Y_h', Z_h', h'] &= (...([X_h, Y_h, Z_h, h] \cdot \mathbf{T}_1) \cdot \mathbf{T}_2) \dots \mathbf{T}_n = \\ &= [X_h, Y_h, Z_h, h] \cdot (\mathbf{T}_1 \cdot \mathbf{T}_2 \dots \mathbf{T}_n) = \\ &= [X_h, Y_h, Z_h, h] \cdot \mathbf{T} \end{aligned}$$

Homogeneous linear transformations are the multiplications of the vector of homogeneous coordinates by a matrix. The vector can be a row vector when it is on the left side of the matrix. On the other hand, the vector can also be a column vector, and stands on the right side. The two approaches are similar, just the matrix should be transposed accordingly. We shall prefer the case when the vector is a row vector, because it is more intuitive when multiple transformations are executed on after the other.

A 2D point is described by 3 homogeneous coordinates, thus the transformation matrix is of 3x3 size.

For 3D points, the matrix has 4x4 elements.

In practice we execute not only a single transformation, but a sequence of transformations. This can be imagined as transforming the point with T1, then the result by T2, etc. However, as matrix multiplication is associative, i.e. parentheses can be regrouped, we obtain the same result if we multiply the point with the product of concatenation of the transformation matrices. Any sequence of transformations can be expressed as a single matrix multiplication. If we consider points as row vectors, then the order of transformation matrices will correspond to the order of their execution.

Affin transzformációk

- Ha az utolsó oszlop $[0,0,1]^T$ vagy $[0,0,0,1]^T$
- Descartes koordinátákra lineáris
- Párhuzamos egyenestartó

$$x' = a_{11}x + a_{21}y + a_{31}$$

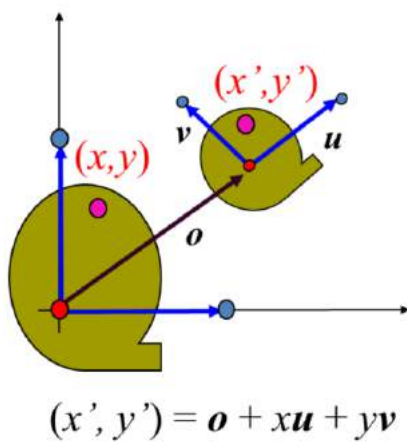
$$y' = a_{12}x + a_{22}y + a_{32}$$

$$[x', y', 1] = [x, y, 1] \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{bmatrix}$$

If the last column of the matrix is 0,0,1 in 2D and 0,0,0,1 in 3D, then the transformation is affine, i.e. it maps lines to lines and preserves parallel lines. From another point of view, the new Cartesian coordinates are linear functions of the original Cartesian coordinates.

Such transformation matrices do not modify the last homogeneous coordinate h.

Affin transzformáció mátrix sorai



$$\begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ o_x & o_y & 1 \end{bmatrix}$$

$$[0, 0, 1] \quad [o_x \quad o_y \quad 1]$$

$$[1, 0, 1] \quad [o_x + u_x \quad o_y + u_y \quad 1]$$

$$[0, 1, 1] \quad [o_x + v_x \quad o_y + v_y \quad 1]$$

In case of affine transformations, the third column is $[0, 0, 1]$ and the row vectors of the remaining part of the matrix have important meaning. They describe what happens with basis vector i , j , and the origin if the transformation is executed.

Homogén lineáris transzformációk tulajdonságai

- Pontot pontba, egyenest egyenesbe, konvex kombinációkat konvex kombinációkba képeznek le

Példa: egyenest egyenesbe

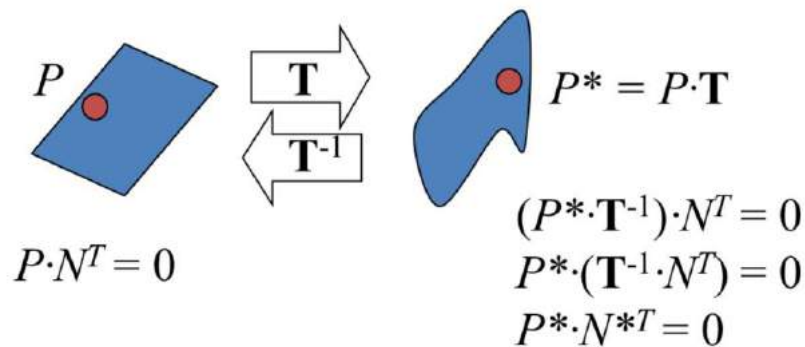
$$[X(t), Y(t), Z(t), h(t)] = [X_1, Y_1, Z_1, h_1] \cdot t + [X_2, Y_2, Z_2, h_2] \cdot (1-t)$$

$$P(t) = P_1 \cdot t + P_2 \cdot (1-t) \quad // \cdot \mathbf{T}$$

$$P^*(t) = P(t) \cdot \mathbf{T} = (P_1 \cdot \mathbf{T}) \cdot t + (P_2 \cdot \mathbf{T}) \cdot (1-t)$$

Homogeneous linear transformations are matrix multiplications of 4 element vectors in 3D and 3 element vectors in 2D. Such linear operations preserve linear computations, so a line is transformed to a line or to a point if the line degenerates, which never happens if \mathbf{T} is invertible.

Invertálható homogén lineáris transzformációk: síkot síkba



$N^* = N \cdot (T^{-1})^T$

Inverse transpose

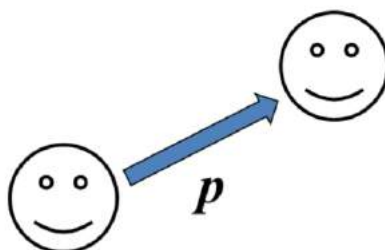
Invertible homogeneous transformations map planes to planes. If the transformation is not invertible, it may happen that the resulting plane degenerates to a line or to a point.

A plane is a collection of points P that satisfy the plane equation. Multiplying every point P by matrix T , we get a collection of points P^* . To find an equation for P^* , we transform P^* back to get P since we know that P satisfies the original equation.

As matrix multiplication is associative, we express a similar equation for the transformed points as well, so they are also on a plane. We can even determine the parameters of the plane (e.g. normal vector). If the parameters are a column vector, the parameters of the original plane must be left-multiplied with the inverse of the transformation matrix.

Eltolás

$$(x', y', z') = (x + p_x, y + p_y, z + p_z)$$

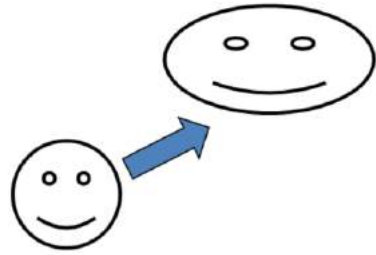


$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix}$$

The first elementary transformation considered is the 3D translation. This transformation computes the sum of the Cartesian coordinates of the point and of the translation vector p . This operation can be represented by a homogeneous transformation matrix, where the diagonal elements are 1, the last row contains the translation vector and all other elements are zero.

Skálázás

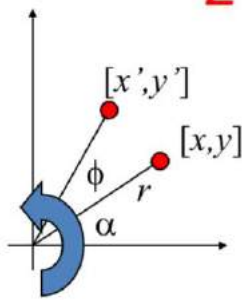
$$x' = S_x \cdot x, \quad y' = S_y \cdot y, \quad z' = S_z \cdot z$$



$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The second transformation is scaling along the coordinate axes. This scales x coordinates by S_x , y coordinates by S_y and z coordinates by S_z . Scaling is a diagonal homogeneous linear transformation, including the scaling factors and 1 in the diagonal.

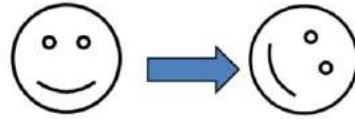
Z tengely körüli forgatás



$$z' = z$$

$$x = r \cos \alpha$$

$$y = r \sin \alpha$$



$$x' = r \cos(\alpha + \phi) = \boxed{r \cos \alpha} \cos \phi - \boxed{r \sin \alpha} \sin \phi$$

$$y' = r \sin(\alpha + \phi) = r \cos \alpha \sin \phi + r \sin \alpha \cos \phi$$

$$x' = r \cos(\alpha + \phi) = x \cos \phi - y \sin \phi$$

$$y' = r \sin(\alpha + \phi) = x \sin \phi + y \cos \phi$$

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} \cos \phi & \sin \phi & 0 & 0 \\ -\sin \phi & \cos \phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation, for example around axis z , is a congruence transformation, thus it surely belongs to the category of homogenous linear transformations.

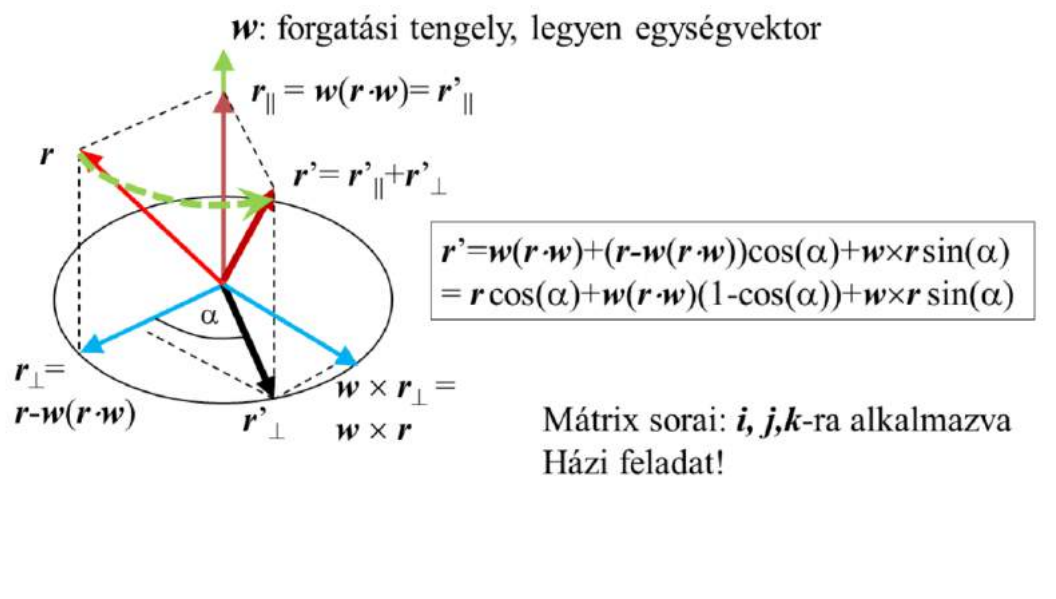
If we rotate around axis z , coordinate z is left unchanged and x, y are modified. Let us express x, y with polar coordinates r, α . Rotation does not modify r , but the polar angle is increased by the rotation angle ϕ .

Using trigonometric identities, we can express the transformed point's x', y' coordinates, which indeed can be realized by a matrix multiplication.

Z-tengely körüli forgatás homogén lineáris transzformációs mátrixa



Általános \mathbf{w} tengely körüli forgatás Rodrigues formula



Let us find the matrix rotating around a line crossing the origin and of direction \mathbf{w} . For the sake of notational simplicity, \mathbf{w} is assumed to have unit length.

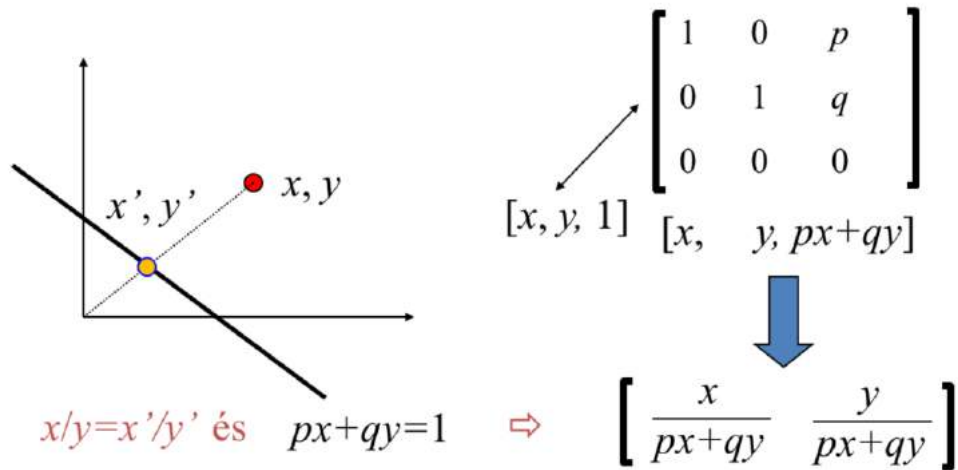
Let us decompose the vector to be rotated, \mathbf{r} , into a component that is parallel with \mathbf{w} , $\mathbf{r}_{\parallel} = \mathbf{w}(\mathbf{r} \cdot \mathbf{w})$, and a vector that is perpendicular to it, $\mathbf{r}_{\perp} =$

$\mathbf{r} - \mathbf{w}(\mathbf{r} \cdot \mathbf{w})$. The parallel vector is not changed by the rotation. The perpendicular component remains in the plane that is perpendicular to \mathbf{w} .

The rotated perpendicular vector is expressed as a linear combination of \mathbf{r}_{\perp} , and a vector that is in the same plane and is perpendicular to \mathbf{r}_{\perp} . This vector is $\mathbf{w} \times \mathbf{r}_{\perp} = \mathbf{w} \times \mathbf{r}$. If \mathbf{r}_{\perp} is rotated by angle alpha, then it will be $\mathbf{r}'_{\perp} \cos(\alpha) + \mathbf{w} \times \mathbf{r}_{\perp} \sin(\alpha)$.

Making the substitutions, we get the Rodrigues formula. How do we get a matrix? We should evaluate this formula on $(1, 0, 0)$ and get the first 3 components of the first row vector of the matrix. The other two rows are obtained similarly.

Középpontos vetítés (2D)

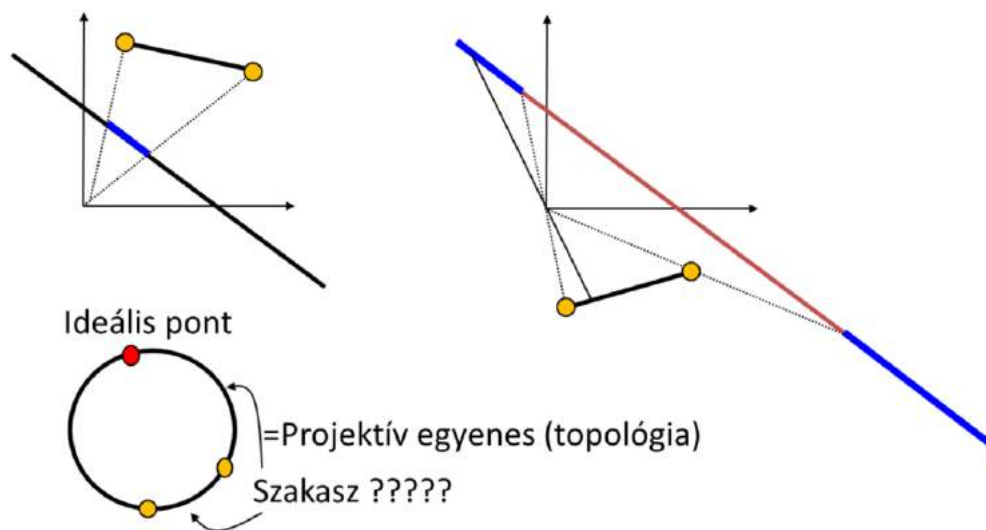


So far, we discussed affine transformations by first introducing them and then developing a matrix for each of them. Let us now reverse the direction of this process and consider a 3x3 matrix, i.e. a transformation in the 2D plane and let us determine what this transformation does. To make it more exciting, the third column is not 0, 0, 1, so it is probably a non-affine transformation. Executing the vector-matrix multiplication, we can obtain the transformation of point (x,y) in homogeneous and also in Cartesian coordinates. Note that the new Cartesian coordinates are non-linear functions of the original Cartesian coordinates, so this transformation is not affine.

What does this transformations? It is a central projection onto a line of equation $px + qy = 1$ assuming the origin as the center of the projection.

With homogeneous linear transformations we can express even non affine transformations but can still be sure that this transformation maps lines to lines, line segments to line segments, etc.

Átfordulási probléma

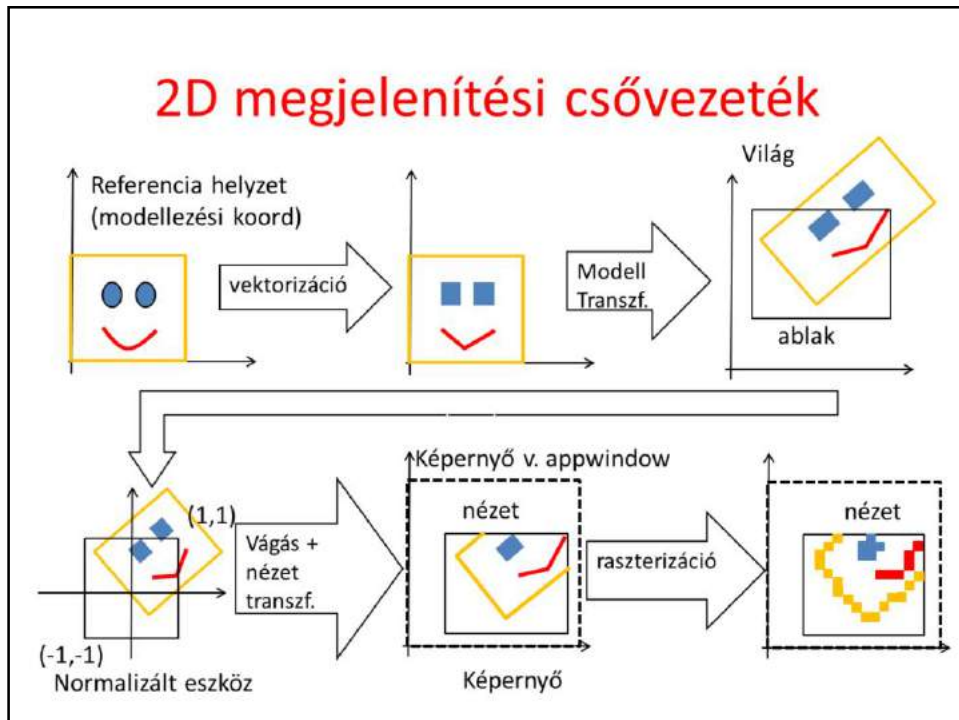


Let us execute this transformation for line segments. We are happy because it is enough to transform the two endpoints and the transformed pair of points can be connected by a line segment according to the properties of homogeneous linear transformations. For the first example, this is indeed true. However, for the second example, the transformation is seemingly not a line segment but its complement on the line, i.e. two half lines.

This is just a virtual contradiction. These two half lines also form a line segment in projection plane. The ideal point at the "end" of the line glues the two ends together. The conclusion is that we should be careful since two points on a line can define two line segments that complement each other, similarly as two points on a circle can define two complementing arcs (a line in projective plane is topologically equivalent to a circle, we can go around it).

2D képszintézis

Szirmay-Kalos László

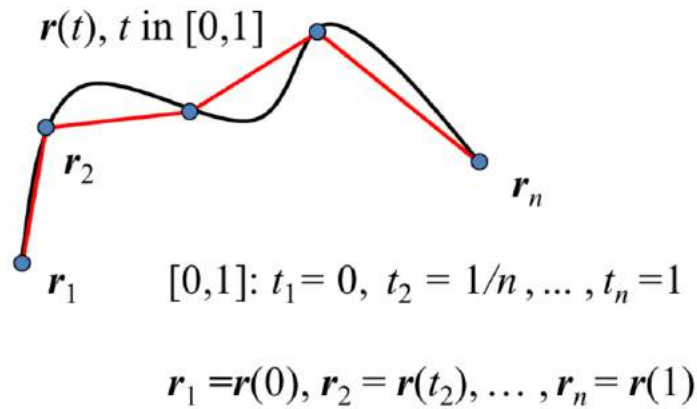


2D rendering is a sequence, called pipeline, of computation steps. We start with the objects defined in their reference state, which can include points, parametric or implicit curves, 2D regions with curve boundaries. As we shall transform these objects, we they are vectorized, so curves are approximated by polylines and regions by polygons. The rendering pipeline thus processes only point, line (polyline) and triangle (polygon) primitives.

Modeling transformation places the object in world coordinates. This typically involves scaling, rotation and translation to set the size, orientation and the position of the object. In world, objects meet each other and also the 2D camera, which is the window AABB (axis aligned bounding box or rectangle). We wish to see the content of the window in the picture on the screen, called viewport. Thus, screen projection transforms the world in a way that the window rectangle is mapped onto the viewport rectangle. This can be done in a single step, or in two steps when first the window is transformed to a square of corners $(-1, -1)$ and $(1, 1)$ and then from here to the physical screen. Clipping removes those objects parts that are outside of the camera window, or alternatively outside of the viewport in screen, or outside of the square of corners $(-1, -1)$ and $(1, 1)$ in normalized device space. The advantage of

normalized device space becomes obvious now. Clipping here is independent of the resolution and of the window, so can be easily implemented in a fix hardware. Having transformed primitives onto the screen, where the unit is the pixel, they are rasterized. Algorithms find those sets of pixels which can provide the illusion of a line segment or a polygon.

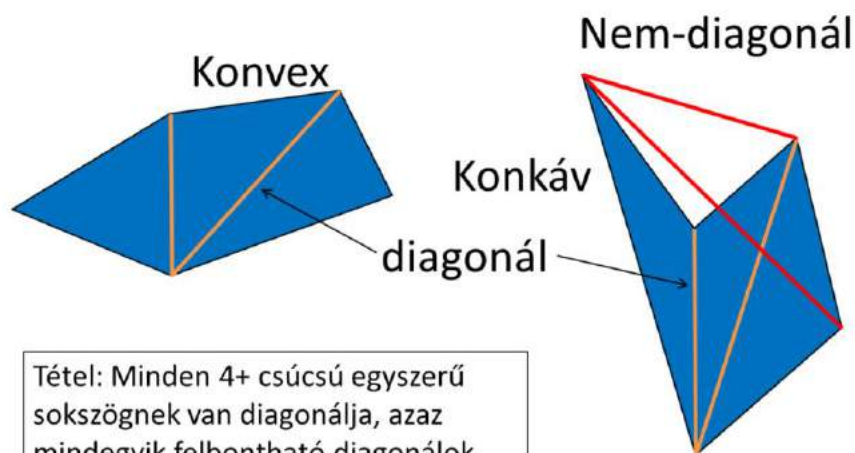
Vektorizáció



Vectorization is trivial for parametric curves. The parametric range is decomposed and increasing sample values are substituted into the equation of the curve, resulting in a sequence of points on the curve. Introducing a line segment between each subsequent pair of points, the curve is approximated by line segments.

If the curve is closed, using the same strategy, a polygon approximation of the region can be found.

Poligon háromszögekre bontása




Tétel: Minden 4+ csúcsú egyszerű sokszögnek van diagonálja, azaz mindegyik felbontható diagonálok mentén.

Polylines are often further decomposed to line segments and polygons to triangles. Such a decomposition has the advantage that the resulting data element has constant size (a line segment has 2 vertices a triangle has 3), and processing algorithms will be uniform and independent of the size of the data element.


Polylines can be easily decomposed to line segments. However, polygons are not so simple to decompose to triangles unless the polygon is convex. A polygon is broken down to smaller polygons and eventually to triangles by cutting them along diagonals. A diagonal is a line segment connecting two non-neighboring vertices, that is fully contained by the polygon. If the polygon is convex, then any line segment connecting two non-neighboring vertices is fully contained by the polygon (this is the definition of convexity), thus all of them are diagonals. This is not the case for concave polygons, when line segments connecting vertices can intersect edges or can fully be outside of the polygon.

The good news is that all polygons, even concave ones, have diagonals, so they can be broken to triangles by diagonals (prove it). An even better news is

that any polygon of at least 4 vertices has special diagonals, that allow exactly one triangle to be cut.

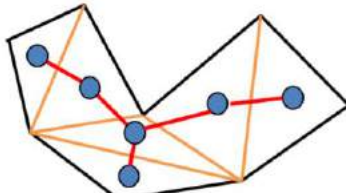


Fül



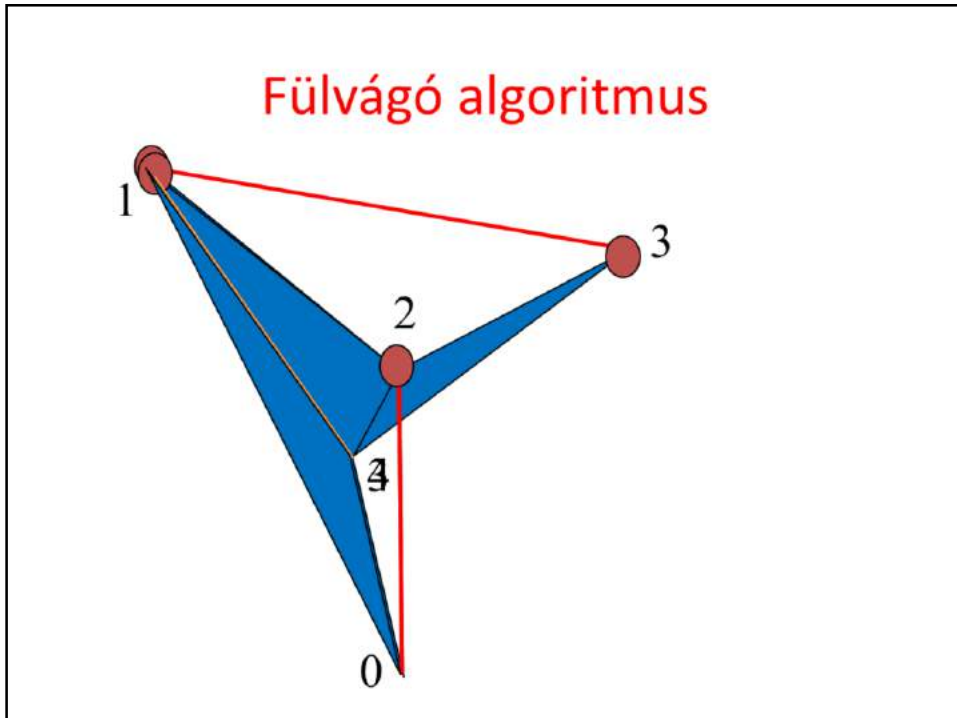
- p_i fül, ha $p_{i-1} - p_{i+1}$ diagonál
- Fül levágható!
- **Fülvágás:**
keress fület és nyissz!

Két fül tétel: Minden legalább 4 csúcsú egyszerű sokszögnek van legalább 2 fül.



A vertex is an ear if the line segment between its previous and next vertices is a diagonal. According to the two ears theorem, every polygon of at least 4 vertices has at least two ears. So triangle decomposition should just search for ears and cut them until a single triangle remains.

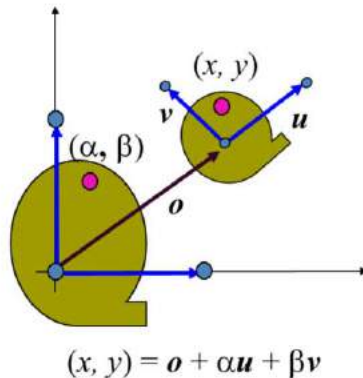
The proof of the two ears theorem is based on the recognition that any polygon can be broken down to triangles by diagonals. Let us start with one possible decomposition, and consider triangles as nodes of a graph, and add edges to this graph where two triangles share a diagonal. This graph is a tree since it is connected (the polygon is a single piece) and cutting every edge, the graph falls apart, i.e. there is no circle in it. By induction, it is easy to prove that every tree of at least 2 nodes has at least two leaves, which correspond to two ears.



For every step, we check whether or not a vertex is an ear. The line segment of its previous and next vertices is tested whether it is a diagonal. This is done by checking whether the line segment intersects any other edge (if it does, it is not a diagonal). If there is no intersection, we should determine whether the line segment is fully outside. Selecting an arbitrary inner point, e.g. the middle, we check whether this point is inside the polygon. By definition, a point is inside if traveling from this point to infinity, the polygon boundary is intersected odd number of times.

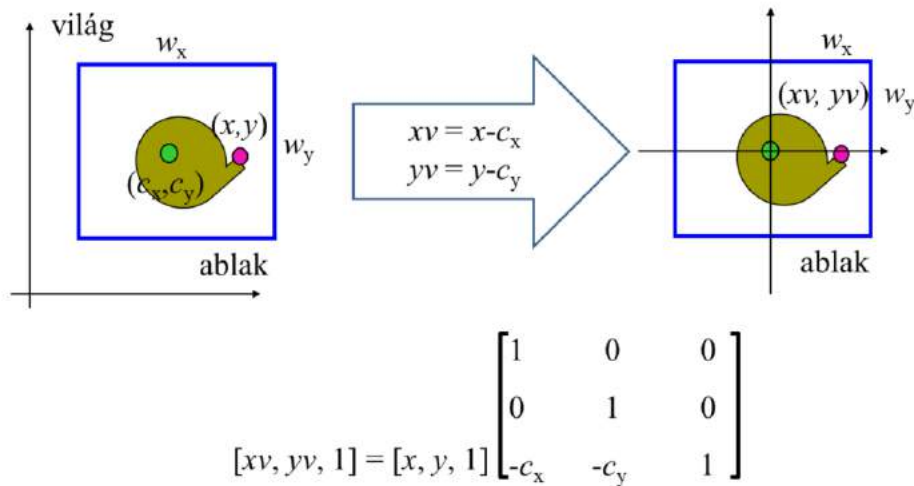
Modellezési transzformáció

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \phi & \sin \phi & 0 \\ -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ p_x & p_y & 1 \end{bmatrix}$$



The first relevant step of rendering is placing the reference state primitives in world, typically scaling, rotating and finally translating its vertices. Recall that it is enough to execute these transformations to vertices, because points, lines and polygons are preserved by homogeneous linear transformations. These are affine transformations, and the resulting modeling transformation matrix will also be an affine transformation. If the third column is 0,0,1, then other matrix elements have an intuitive meaning, they specify what happens with basis vector i , basis vector j , and the origin itself.

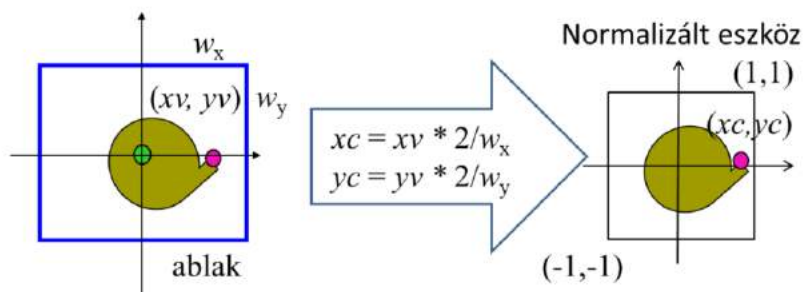
View transzformáció: kameraablak közepe az origóba



Screen projection maps the window rectangle, which is the camera in 2D, onto the viewport rectangle, which can be imagined as the photograph. This simple projection is usually executed in two steps, first transforming the window onto a normalized square, and then transforming the square to the viewport.

Transforming the window to a origin centered square of corners $(-1,-1)$ and $(1,1)$ is a sequence of two transformations: a translation that moves the center of the camera window to the origin; a scaling that modifies the window width and height to 2. These are affine transformations that can also be given as a matrix.

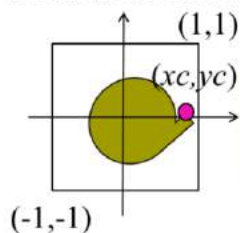
Projekció: kameraablak a (-1, -1)-(1, 1) négyzetbe



$$[xc, yc, 1] = [xv, yv, 1] \begin{bmatrix} 2/w_x & 0 & 0 \\ 0 & 2/w_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

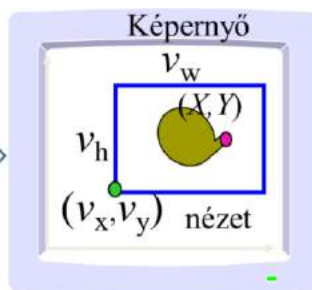
Viewport transzformáció: Normalizáltból képernyő koordinátákba

Normalizált eszköz



$$X = v_w(x_c + 1)/2 + v_x$$
$$Y = v_h(y_c + 1)/2 + v_y$$

Képernyő

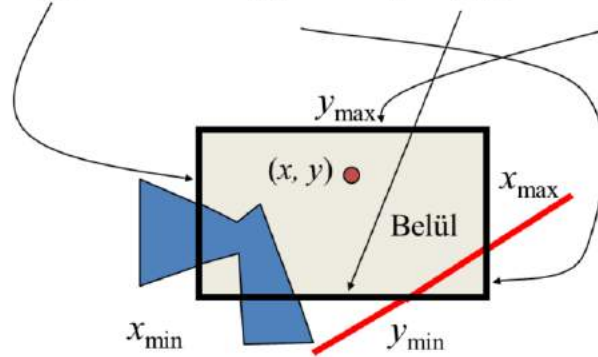


```
glViewport(vx, vy, vw, vh);
```

Vágás

Pont vágás:

$$x > x_{\min} = -1 \quad x < x_{\max} = +1 \quad y > y_{\min} = -1 \quad y < y_{\max} = +1$$



Kívül

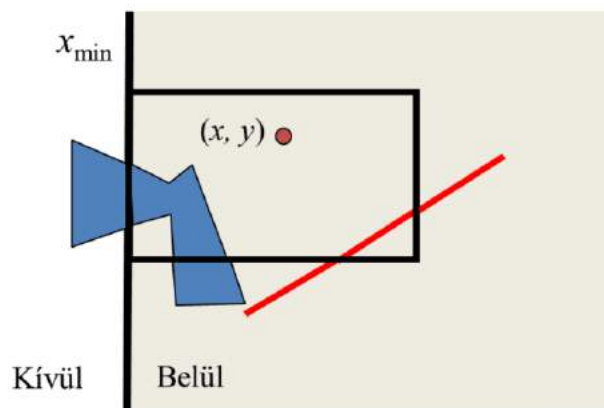
Clipping is executed usually in normalized device space where x, y must be between -1 and 1. To be general, we denote the limits by x_{\min} , x_{\max} ...

A point is preserved by clipping if it satisfies

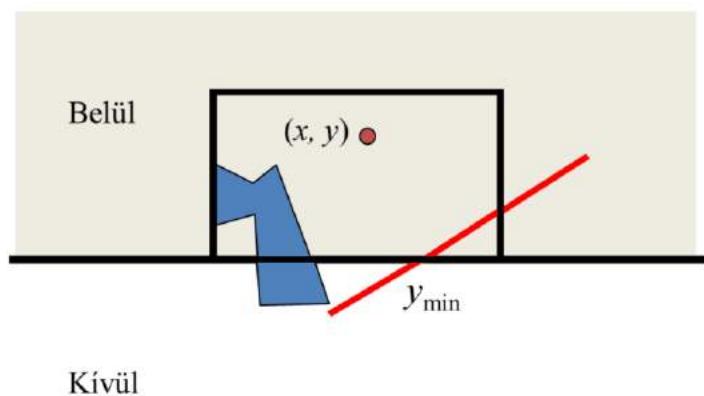
half planes since the clipping rectangle is the intersection of the half planes.

This concept is very useful when line segments or polygons are clipped since testing whether or not the two endpoints of line segment or vertices of a polygon are outside the clipping rectangle cannot help to decide whether there is an inner part of the primitive.

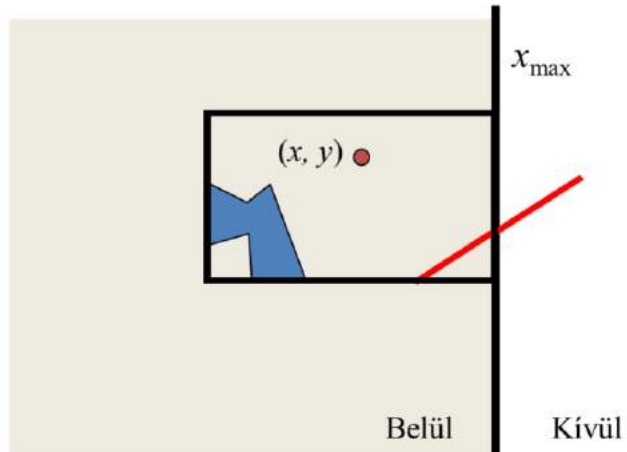
Vágás



Vágás

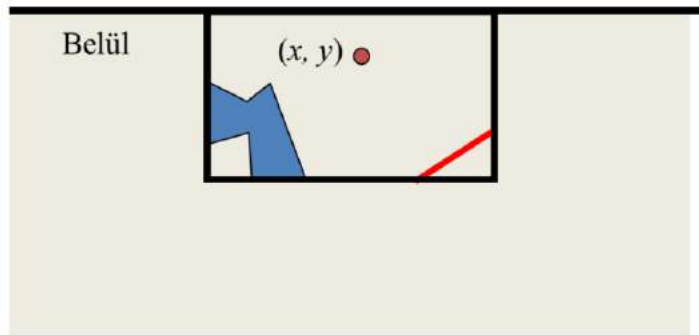


Vágás

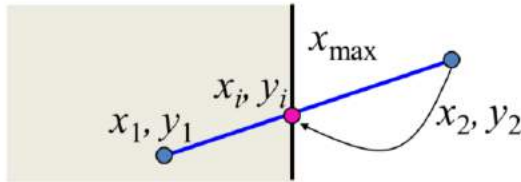


Vágás

Kívül



Szakasz vágás



$$x(t) = x_1 + (x_2 - x_1)t, \quad y(t) = y_1 + (y_2 - y_1)t$$

$$x = x_{\max}$$

$$\text{Metszés: } x_{\max} = x_1 + (x_2 - x_1)t \Rightarrow t = (x_{\max} - x_1) / (x_2 - x_1)$$

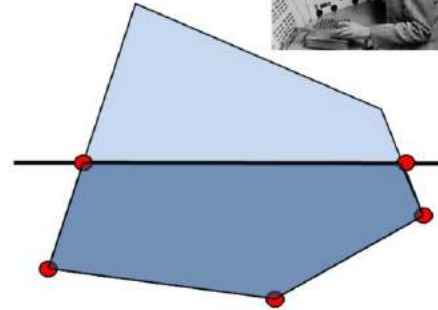
$$x_i = x_{\max} \quad y_i = y_1 + (y_2 - y_1) (x_{\max} - x_1) / (x_2 - x_1)$$

Let us consider a line segment and clipping on a single half plane. If both endpoints are inside, then the complete line segment is inside **since the inner region, the half plane, is convex**. If both endpoints are outside, then the line segment is completely outside, **since the outer region is also convex**. If one endpoint is inside while the other is outside, then the intersection of the line segment and the clipping line is calculated, and the outer point is replaced by the intersection.

Sutherland-Hodgeman poligonvágás



```
PolygonClip(p[n] ⇨ q[m])  
  m = 0;  
  for( i=0; i < n; i++) {  
    if (p[i] belső) {  
      q[m++] = p[i];  
      if (p[i+1] külső)  
        q[m++] = Intersect(p[i], p[i+1], vágóegyenes);  
    } else {  
      if (p[i+1] belső)  
        q[m++] = Intersect(p[i], p[i+1], vágóegyenes);  
    }  
  }  
}
```



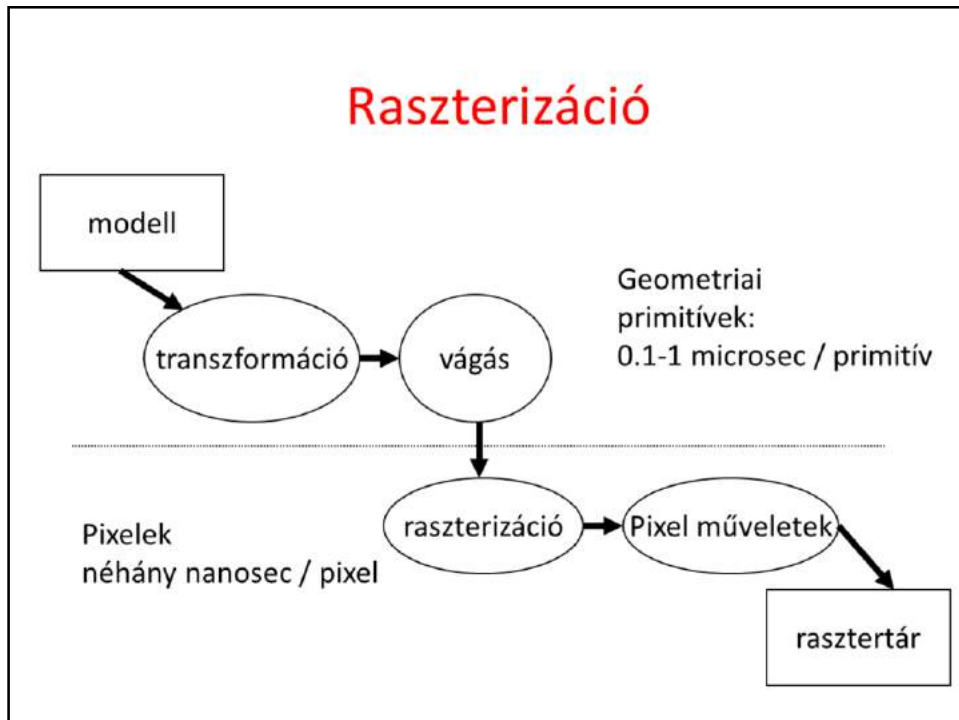
Első pontot még egyszer
a tömb végére

Polygon clipping is traced back to line clipping. We consider the edges of the polygon one-by-one. If both endpoints are in, the edge will also be part of the clipped polygon. If both of them are out, the edge is ignored. If one is in and the other is out, the inner part of the segment is computed and added as an edge of the clipped polygon.

The input of this implementation is an array of vertices p and number of points n . The output is an array of vertices q and number of vertices m .

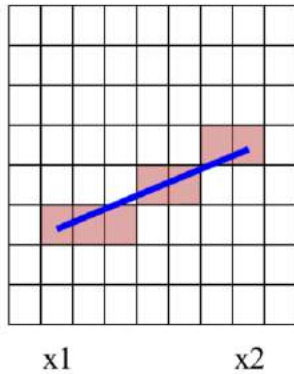
Usually, we can assume that the i th edge has endpoints $p[i]$ and $p[i+1]$.

However, the last edge is an exception since its endpoints are $p[n-1]$ and $p[0]$.



Before starting the discussion of rasterization it is worth looking at the pipeline and realizing that rasterization uses a different data element, the pixel, while phases discussed so far work with geometric primitives. A primitive may be converted to many pixels, thus the performance requirements become crucial at this stage. In order to maintain real-time frame rates, the process should output a new pixel in every few nanoseconds. It means that only those algorithms are acceptable that can deliver such performance.

Szakasz rajzolás



Egyenes egyenlete:

$$y = mx + b$$

Egyeneshúzás

```
for( x = x1; x <= x2; x++) {  
    Y = m*x + b;  
    y = Round( Y );  
    write( x, y );  
}
```

Line drawing should provide the illusion of a line segment by coloring a few pixels. A line is thin and connected, so pixels should touch each other, should not cover unnecessary wide area and should be close to the geometric line. If the slope of the line is moderate, i.e. x is the faster growing coordinate, then it means that in every column exactly one pixel should be drawn (connected but thin), that one where the pixel center is closest to the geometric line. The line drawing algorithm iterates on the columns, and in a single column it finds the coordinate of the geometric line and finally obtains the closest pixel, which is drawn.

This works, but a floating point multiplication, addition and a rounding operation is needed in a single cycle, which is too much for a few nanoseconds. So we modify this algorithm preserving its functionality but getting rid of the complicated operations.

Inkrementális elv

- Egyenlet: $Y(X) = mX + b = Y(X-1) + m$

```
DDADrawLine(int x1, int y1, int x2, int y2) {  
    float m = (y2 - y1)/(x2 - x1);  
    float y = y1;  
    for(int x = x1; x <= x2; x++) {  
        int Y = round(y);  
        WRITE(x, Y, color);  
        y = y+m;  
    }  
}
```

The algorithm transformation is based on the incremental concept, which realizes that a linear function (the explicit equation of the line) is evaluated for an incremented X coordinate. So when X is taken, we already have the Y coordinate for X-1. The fact is that it is easier to compute Y(X) from its previous value than from X. The increment is m, the slope of the line, thus a single addition is enough to evaluate the line equation. This single addition can be made faster if we used fixed point number representation and not floating point format. As these numbers are non integers (m is less than 1), the fixed point representation should use fractional bits as well. It means that an integer stores the Tth power of 2 multiple of the non-integer value. Such values can be added as two integers.

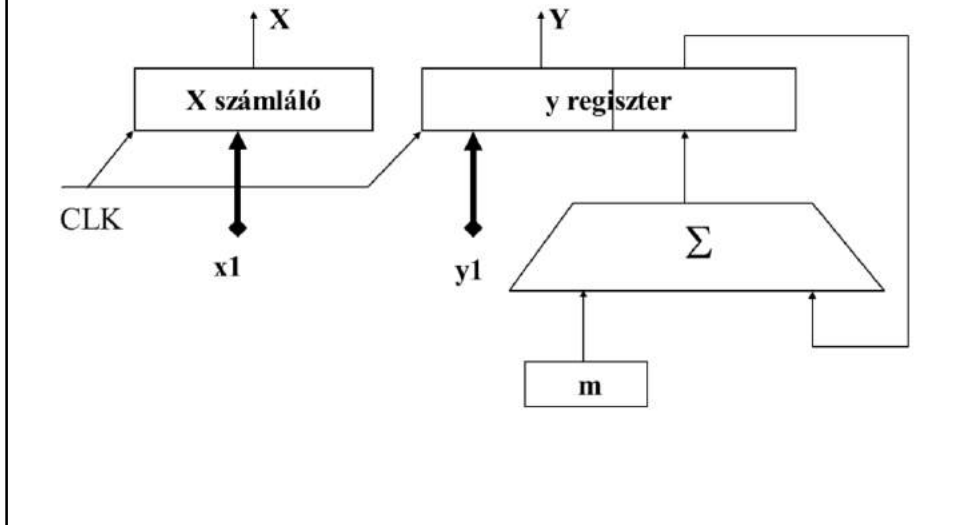
The number of fractional bits can be determined from the requirement that even the longest iteration must be correct. If the number of fractional bits is T, the error caused by the finite fractional part is 2^{-T} in a single addition. If errors are accumulated, the total error in the worst case is $N 2^{-T}$ where N is the number of additions. N is the linear resolution of the screen, e.g. 1024. In screen space the unit is the pixel, so the line will be correctly drawn if the total error is less than 1. It means that T=10, for example, satisfies all requirements.

The line drawing algorithm based on the incremental concepts is as follows. First the slope of the line is computed. The y value is set according to the end point. This y stores the precise location of the line for a given x, so it is non integer. In a for cycle, the closest integer is found, the pixel is written, and – according to the incremental concept – the new y values for the next column is obtained by a single addition.

Rounding can be replaced by simple truncation if 0.5 is added to the y value.

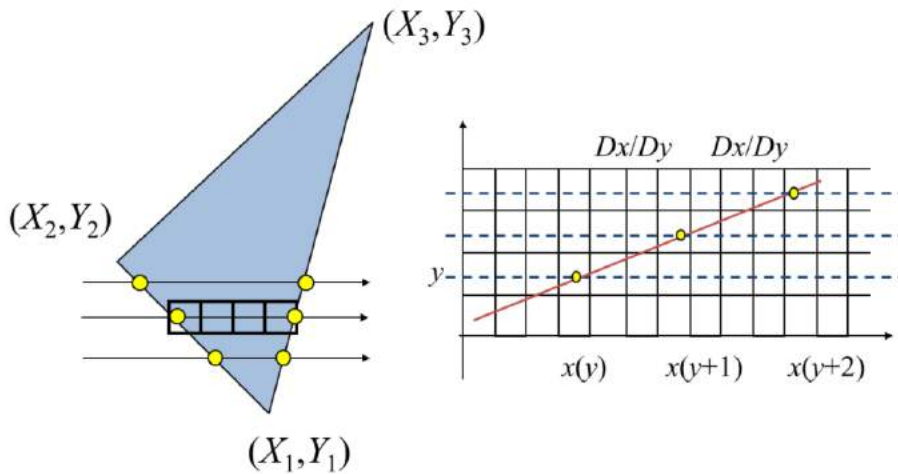
If fixed point representation is used, we shift m and y by T number of bits and rounding ignores the low T bits.

DDA szakaszrajzoló hardver



This algorithm can be implemented in hardware with a simple counter that generates increasing x values for every clock cycle. For y we use a register that stores both its fractional and integer parts. The y coordinate is incremented by m for every clock cycle.

Háromszög kitöltés



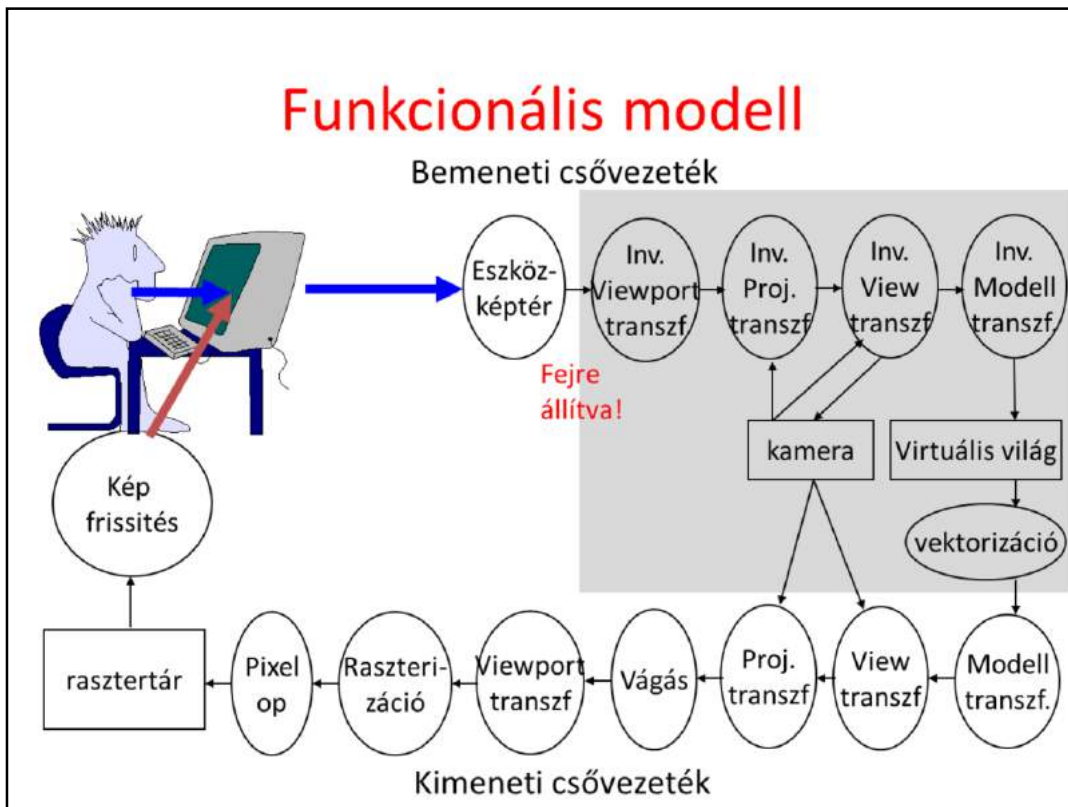
For triangle rasterization, we need to find those pixels that are inside the triangle and color them. The search is done along horizontal lines of constant y coordinate. These lines are called scan lines and rasterization as scan conversion. For a single scan line, the triangle edges are intersected with the scan line and pixels are drawn between the minimum and maximum x coordinates.

The incremental principle can also be applied to determine scan-line and edge intersections. Note that while y is incremented by 1, the x coordinate of the intersection grows with the inverse slope of the line, which is constant for the whole edge, and thus should be computed only once.

Again, we have an algorithm that uses just increments and integer additions.

Grafikus hardver/szoftver alapok

Szirmay-Kalos László

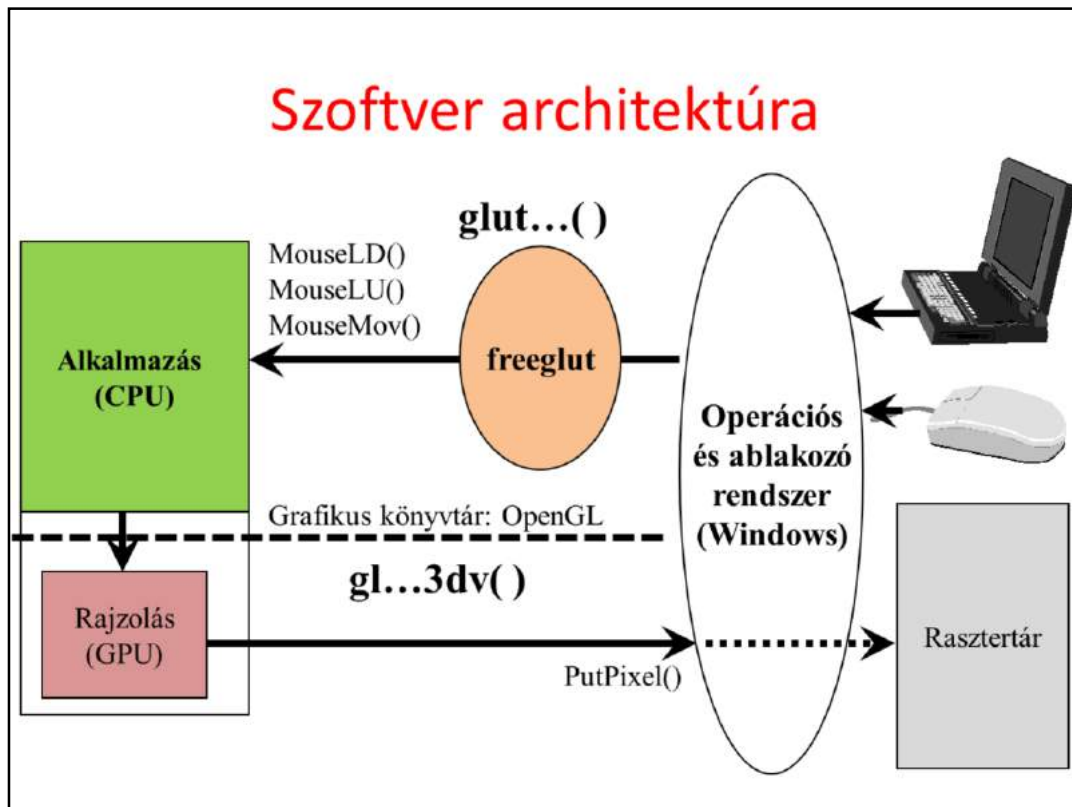


From system point of view, a graphics application handles the user input, changes the internal state, called the virtual world by modeling or animating it, and then immediately renders the updated model presenting the image to the user. This way, the user immerses into the virtual world, i.e. he feels that he is promptly informed about its current state.

The process from the input to the virtual world is called the input pipeline. Similarly, the process mapping the virtual world to the screen is the output pipeline.

The complete system is a (control) loop with two important points, the virtual world and the user. In the output pipeline, the virtual world is vectorized first since only lines and polygons can be transformed with homogeneous linear transformations. Then modeling, view and projection transformations are executed moving the current object to normalized device space. Here clipping is done, then the object is transformed to the screen, where it is rasterized. Before being written in the frame buffer, pixels can undergo pixel operations, needed, for example, to handle transparent colors. The frame buffer is read periodically to refresh the screen. The user can see the screen and interact with the content by moving the cursor with input devices and starting actions like pressing a button. Such actions generate events taking also the screen space position with them.

Screen space is the whole screen in full-screen mode or only the application window. The unit is the pixel. Note that screen space is different for the operating system and for opengl. For MsWindows and XWindow, axis y points downward while in opengl y points upward. Thus, y must be flipped, i.e. subtracted from the vertical resolution. The input pixel coordinate goes from the screen to modeling space, thus inverse transformations are applied in the reverse order. With input devices not only the virtual world can be modified but also the camera can be controlled.



Our graphics application runs under the control of an operating system together with other applications. The operating system handles shared resources like input devices and the frame buffer as well, so a pixel data in the frame buffer can be changed only via the operating system. Modifying pixels one by one from the application would be too slow, so a new hardware element, called the GPU, shows up that is responsible for many time consuming steps of rendering. The GPU is also a shared device that can be accessed via the operating system. Such accesses are calls to a library for the application program. We shall control the GPU through a C graphics library called OpenGL.

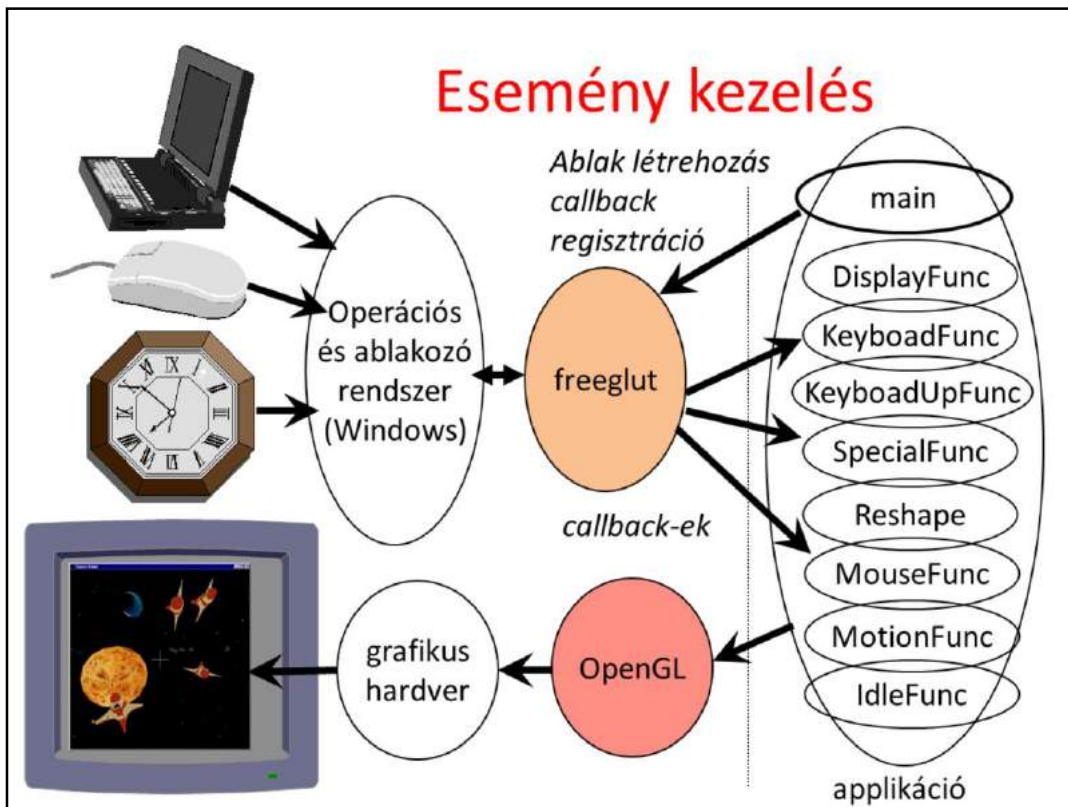
On the other hand, to catch input events handled by the operating system, we need another library. We shall utilize the freeGLUT for this purpose, due to its simplicity and the portability (it runs over MsWindows, Xwindow, etc.)

The operating system separates the hardware from the application. The operating system is responsible for application window management and also letting the application give commands to the GPU via OpenGL, not to mention the re-programming of the GPU with shader programs.

OpenGL is collection of C functions of names starting with gl. The second part of the name shows what the function does, and the final part allows to initiate the same action with different parameter numbers and types (note that there is not

function overloading in C).

To get an access to the GPU via OpenGL, the application should negotiate this with the operating system, for which operating system dependent libraries, like the wgl for MsWindows and glX for Xwindow are available. Using them is difficult, and more importantly, it makes our application not portable. So, to hide operating system dependent features, we use GLUT, which translates generic commands to the operating system on which it runs. It is simple and our application will be portable. GLUT function names start with glut.



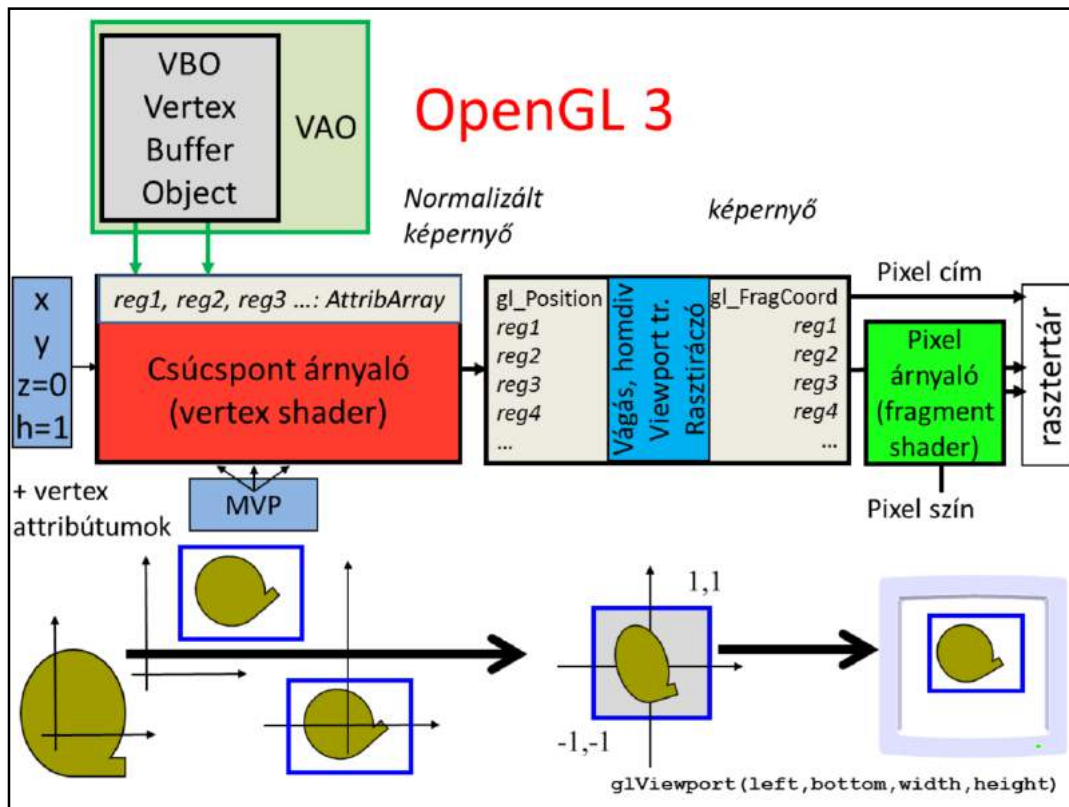
The graphics output is implemented by OpenGL. The application window management and the input are the responsibilities of GLUT. Our application consists of a main function and a set of event handlers (we use event driven programming paradigm in interactive systems). In main, our application program interacts with GLUT and specifies the properties of the application window (e.g. initial resolution and what a single pixel should store), and also the event handlers.

An event handler is a C function that should be programmed by us. This function is connected to a specific event of GLUT, and having established this connection we expect GLUT to call our function when the specific event occurs. A partial list of possible events are:

- Display event that occurs when the application window becomes invalid and thus GLUT asks the application to redraw the window to restore its content.
- Keyboard event occurs when the user presses a key having ASCII code.
- Special event is like Keyboard event but is triggered by a key press having no ASCII code (e.g. arrows and function keys).
- Reshape handler is called when the dimensions of the application window are changed by the user.
- Mouse event means the pressing or releasing the button of the mouse.

- Idle event indicates the time elapsed and our virtual world model should be updated to reflect the new time.

Event handler registration is optional with the exception of the Display event. If we do not register a handler function, nothing special happens when this event occurs.



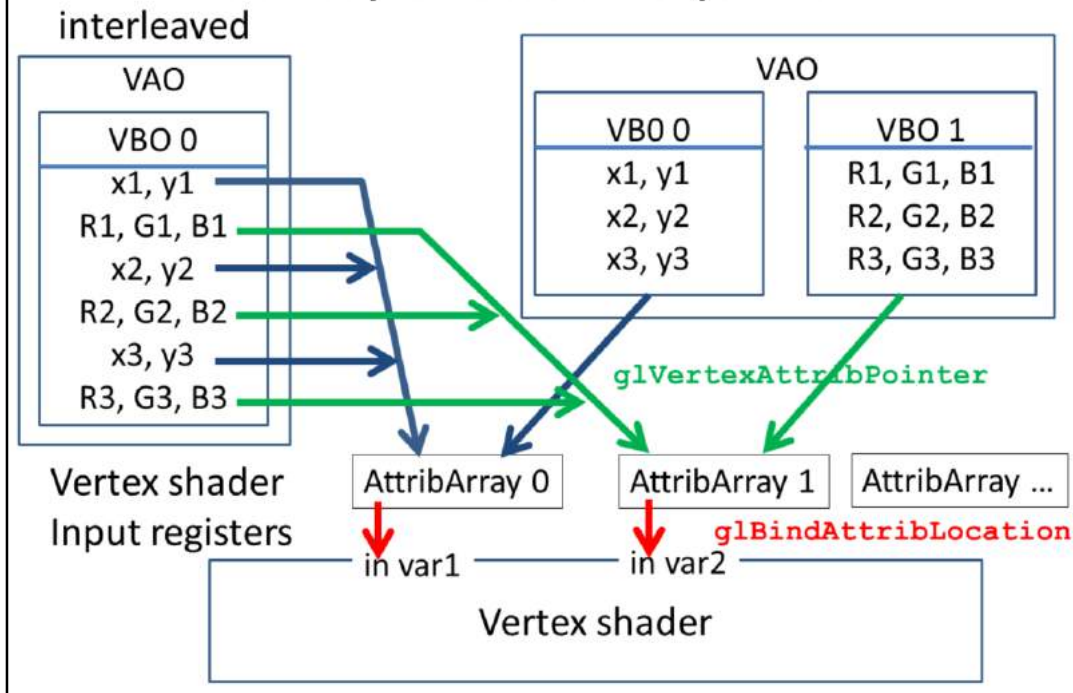
Primitives (e.g. a line or a polygon) go down the pipeline, each having multiple vertices associated with their homogeneous coordinates and possible attributes (e.g. vertex color). Primitives must be transformed to normalized device space for clipping, which requires the transformation of its vertices with the modeling, viewing and projection transformation matrices. Clipping is done, so is homogeneous division if the fourth homogeneous coordinate is not 1. Then the primitive is transformed to screen space taking into account the viewport position and size. The primitive is rasterized in screen space.

For performance reasons, OpenGL 3 retained mode requires the application to prepare the complete data of the vertices and attributes of a single object rather than passing them one by one. These data are to be stored in arrays on the GPU, called Vertex Buffer Object (VBO). An object can have multiple VBOs, for example, we can put coordinates in a single array and vertex colors in another. Different VBOs are encapsulated into a Vertex Array Object (VAO) that also stores information about how the data should be fetched from the VBOs and sent to the input registers of the vertex shader. A single input register can store four 32 bit long words (4 floats called **vec4**, or four integers) and is called Vertex Attrib Array.

The responsibility of the Vertex Shader is to transform the object to normalized

device space. If the concatenation of model, view and projection matrices is given to the Vertex Shader, it is just a single matrix-vector multiplication. The output of the Vertex Shader goes to output registers including `gl_Position` storing the vertex position in normalized device space and other registers storing vertex attributes. Clipping, homogeneous division, viewport transformation and rasterization are performed by the fixed function hardware of the GPU, so these steps cannot be programmed. The output of the rasterization step is the sequence of pixels with pixel coordinates and interpolated vertex attributes. Pixel coordinates select the pixel that is modified in the frame buffer. From other vertex attributes and global variables, the pixel color should be computed by another programmable unit called the fragment shader.

Csúcspont adatfolyamok



Let us zoom out the connection of the vertex buffer objects, vertex shader input registers called **AttribArrays**, and vertex shader input variables. The object is described in arrays called **VBOs**. For example, coordinates can be stored in one array, colors in another (this strategy is called **Structure Of Arrays**, or **SOA** for short). To allow the Vertex Shader to process a vertex, its input registers must be filled with the data of that particular vertex, one vertex at a time. Function **glVertexAttribPointer** tells the GPU how to interpret the data in VBOs, from where the data of a single vertex can be fetched, and in which **AttribArray** a data element should be copied. For example, coordinates can be copied to **AttribArray0** while colors to **AttribArray1** (a single register can store 4 floats).

When the Vertex Shader runs, it can fetch its input registers. It would not be too elegant if we had to refer to the name of the input register, e.g. **AttribArray 0**, so it is possible to assign variable names to it with **glBindAttribLocation**. For example, **AttribArray0** can be the "vertexPosition".

Note that this was only one possibility of data organization. For example, it is also perfectly reasonable to put all data in a single array where coordinates and attributes of a single vertex are not separated (this strategy is the **Array Of Structures**, or **AOS**). In this case **glVertexAttribPointer** should tell the GPU where an attribute starts in the array and what the step size (stride) is.

Az első OpenGL programom

```
#include <GL/glew.h>
#include <GL/freeglut.h>

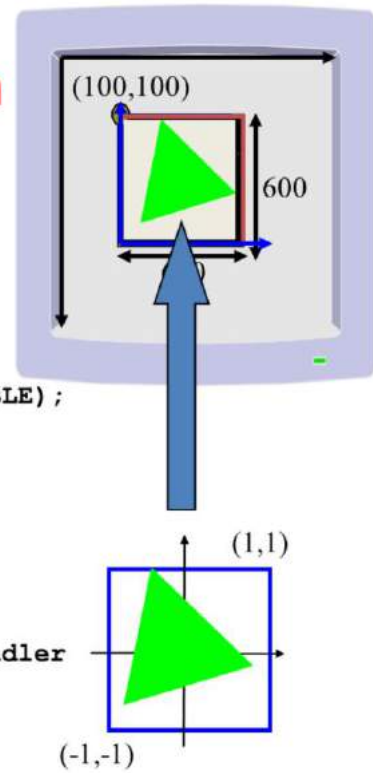
int main(int argc, char * argv[]) {
    glutInit(&argc, argv); // init glut
    glutInitContextVersion(3, 0);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);

    glutCreateWindow("Hi Graphics");
    glewExperimental = true; // magic
    glewInit(); // init glew

    glViewport(0, 0, 600, 600);
    onInitialization();

    glutDisplayFunc(onDisplay); //event handler

    glutMainLoop();
    return 1;
}
```



In the main function of a graphics application, we set up the application window with the help of GLUT telling the initial position, size, what data should be stored in a pixel, and also what functions should be called when different events happen. At the end, the message loop is started, which runs in circles, checks whether any event occurred for which we have registered an event handler, and if this is the case, it calls the respective event handler.

The main function can also be used to initialize data in OpenGL (on the GPU), especially those which are needed from the beginning of the program execution and which do not change during the application. We need shader programs from the beginning, so this is a typical place to compile and link shader programs and upload them to the GPU.

Let us start with the main function. Two header files are needed.

GLEW is the OpenGL Extension Wrangler library that finds out what extensions are supported by the current GPU in run time. GLUT is a windowing utility toolkit to set up the application window and to manage events.

In the main functions, first the application window is set up with glut calls:

- glutInit initializes glut and allows use to communicate with the GPU via

OpenGL.

- `glutInitContextVersion` sets the required OpenGL version. In this case, we want opengl 3.0.
- `glutInitWindowSize` specifies the initial resolution of the application window.
- `glutInitWindowPosition` specifies where it is initially placed relative to the upper left corner of the screen.
- `glutInitDisplayMode` tells glut what to store in a single pixel. In the current case, we store 8 bit (default) R,G,B, and A (opacity) values, in two copies to support double buffering.
- `glutCreateWindow` creates the window, which shows up.

The Extension Wrangler is initialized

- `glewExperimental = true`: GLEW obtains information on the supported extensions from the graphics driver, so if it is not updated, then it might not report all features the GPU can deliver. Setting `glewExperimental` to true gets GLEW to try the extension even if it is not listed by the driver.
- `glewInit` makes the initialization

From here, we can initialize OpenGL.

- `glViewport` sets the render target, i.e. the photograph inside the application window
- `onInitialization` is our custom initialization function discussed on the next slide.

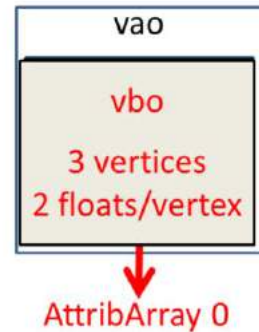
The remaining functions register event handlers and start the message loop. For the time being, only the `onDisplay` is relevant, which is called whenever the application window becomes invalid. We use this function to render the virtual world, which consists of a single green triangle, directly given in normalized device space.

onInitialization()

```
unsigned int shaderProgram;
unsigned int vao; // virtual world on the GPU

void onInitialization() {
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao); // make it active

    unsigned int vbo; // vertex buffer object
    glGenBuffers(1, &vbo); // Generate 1 buffer
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    // Geometry with 24 bytes (6 floats or 3 x 2 coordinates)
    static float vertices[] = {-0.8,-0.8, -0.6,1.0, 0.8,-0.2};
    glBufferData(GL_ARRAY_BUFFER, // Copy to GPU target
                 sizeof(vertices), // # bytes
                 vertices,         // address
                 GL_STATIC_DRAW); // we do not change later
    glEnableVertexAttribArray(0); //VertexAttribArray 0
    glVertexAttribPointer(0, // vbo ->VertexAttribArray 0
                          2, GL_FLOAT, GL_FALSE, // two floats/attrib, not fixed-point
                          0, NULL);              // stride, offset: tightly packed
}
```



In `onInitialization` those opengl data are initialized that are typically constant during the application, so they do not have to be set in every drawing. In our program, this includes the constant geometry (the triangle), and the GPU shader programs. The `shaderProgram` and the `vao` are set here but also used in the `onDisplay`, therefore they are global variables.

First we allocate one vertex array object and its id is `vao`. With Binding, this is made active, which means that all subsequent operations belong to this `vao` until another `vao` is bound or the current one is unbound with binding 0.

In the second step, one vertex buffer object is allocated, which will be part of the active `vao` (we have just one, which is active). This vertex buffer object is made active, so all subsequent operations are related to this until another is bound.

Array "vertices" stores the geometry of our triangle, and is obviously in the CPU memory. It contains 6 floats, i.e. 24 bytes. With **`glBufferData`** the 24 bytes are copied to the GPU. With the last parameter of `glBufferData` we can specify which type of GPU memory should be used (the GPU has different types of memory with different write and read speeds and capacity, so the driver may decide where to copy this 24 bytes based on our preference). We say that the 24 bytes will not be modified but it would be great if it could be fetched fast (constant memory

would be an ideal choice). So far we said nothing about the organization and the meaning of the data, it is simply 24 bytes, the GPU does not know that it defines 3 vertices, each with 2 Cartesian coordinates, which are in float format.

glVertexAttribPointer() defines the interpretation of the data and also that the data associated with a single vertex goes to the input register (AttribArray) number 0. It specifies that a single vertex have two floats, i.e. 8 bytes. If it was non floating point value, it would also be possible to put the binary point to the most significant bit, but we set this parameter to `GL_FALSE`.

The last two parameters tell the GPU how many bytes should be stepped from one vertex to the other (if it is 0, it means that the step size is equal to the data size, 2 floats in this case), and where the first element is (at the beginning of the array, so the pointer offset is zero). Stride and offset are essential if interleaved vbos are used.

The output of the fixed function part is the sequence of pixels (called fragments) belonging to the current primitive and also the variables that are output by the vertex shader, having interpolated for the current pixel. The pixel address is in register `gl_FragCoord`, which cannot be modified, but from the other registers and uniform variables, the color of this fragment can be obtained by the fragment shader processor. It has one uniform input called the color, which will determine the output color stored in variable `outColor`.

The very beginning of the pipeline, vertex coordinate variable `vp` is connected to the vertex shader input register (`VertexAttrib`) number 0 as told by **`glBindAttribLocation`**. The output of the fragment shader goes to the frame buffer as requested by **`glBindFragDataLocation`**.

Finally, the shader program is linked, copied to the shader processors to be executed by them.

<pre>#version 130 precision highp float; uniform mat4 MVP; in vec2 vp; void main() { gl_Position = vec4(vp.x, vp.y, 0, 1) * MVP; }</pre>	<pre>#version 130 precision highp float; uniform vec3 color; out vec4 outColor; void main() { outColor = vec4(color, 1); }</pre>
---	---

```
void onDisplay( ) {
    glClearColor(0, 0, 0, 0); // background color
    glClear(GL_COLOR_BUFFER_BIT); // clear frame buffer

    // Set vertexColor to (0, 1, 0) = green
    int location = glGetUniformLocation(shaderProgram, "color");
    glUniform3f(location, 0.0f, 1.0f, 0.0f); // 3 floats

    float MVPtransf[4][4] = { 1, 0, 0, 0, // MVP matrix,
                              0, 1, 0, 0, // row-major!
                              0, 0, 1, 0,
                              0, 0, 0, 1 };

    location = glGetUniformLocation(shaderProgram, "MVP");
    glUniformMatrix4fv(location, 1, GL_TRUE, &MVPtransf[0][0]);

    glBindVertexArray(vao); // Draw call
    glDrawArrays(GL_TRIANGLES, 0 /*startIdx*/, 3 /*# Elements*/);
    glutSwapBuffers( ); // exchange buffers for double buffering
}
```

We have registered a single event handler (onDisplay) that reacts to the event occurring when the application window gets invalid (DisplayFunc).

In this function, the virtual world (consisting of the single green triangle) is rendered.

glClearColor sets the color with which the pixels of the application window is cleared. This color is black. The actual clearing is done by glClear.

GL_COLOR_BUFFER_BIT stands for the frame buffer storing color values.

Drawing consists of setting the values of uniform variables of shaders and then forcing the geometry through the pipeline, which is called the draw call. Finally, the buffer used for drawing so far is swapped with the buffer the user could see so far by **glutSwapBuffers**, so the result will be visible to the user.

The fragment shader has a single uniform variable called color and of type vec3, which can be set with

function **glUniform3f(location, 0.0f, 1.0f, 0.0f)** to value (0,1,0)=green. Note that "3f" at the end of the function name indicates that this function takes 3 float parameters. Parameter location is the serial number of this uniform variable, which can be found with **glGetUniformLocation(shaderProgram, "color")** which returns the serial number of uniform variable called "color" in the shader programs.

The vertex shader has uniform variable MVP of type mat4, i.e. it is a 4x4 matrix. First, its serial number must be obtained, then its value can be set with **glUniformMatrix4fv**. Here fv means that matrix elements are floats (f) and instead of passing the 16 floats by value, the address of the CPU array is given (v) from which the values can be copied (pass by address). The second parameter of **glUniformMatrix4fv** says that 1 matrix is passed, the third parameter that this is a row-major matrix and should be kept this way.

This issue can cause a lot of confusion:

- In C or C++ two-dimensional matrices are of row-major.
- In GLSL two-dimensional matrices are of column-major.

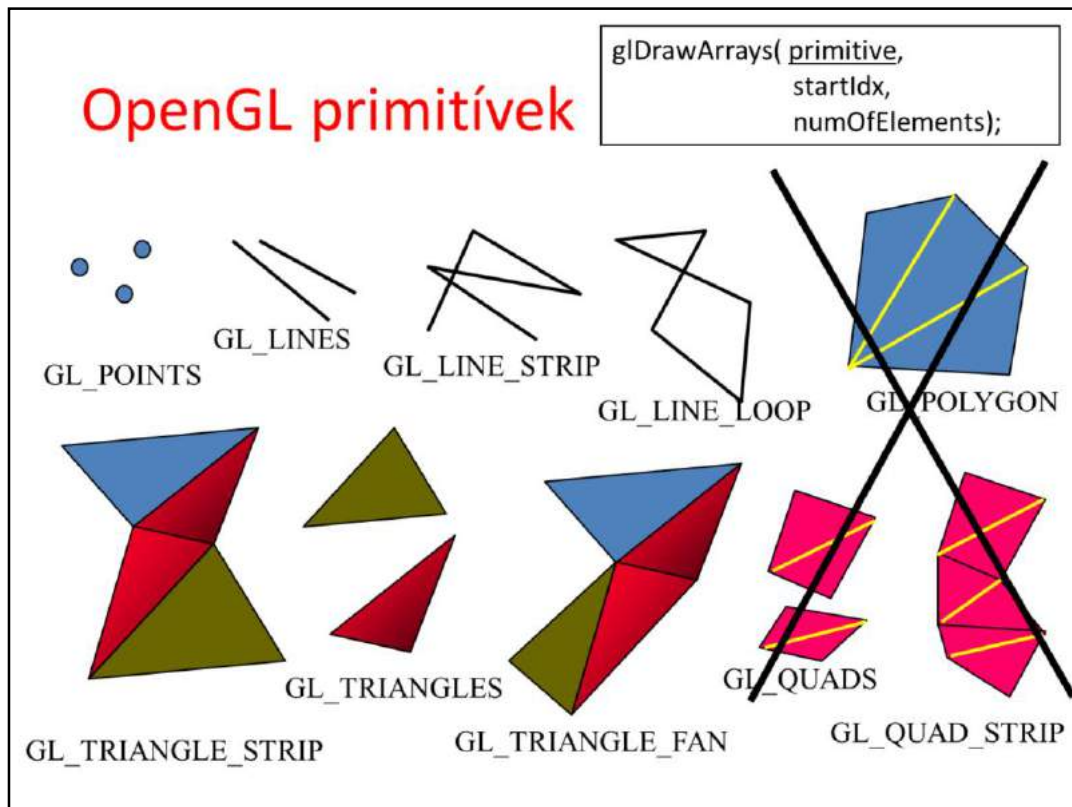
So if we use them without caution, we might apply the transpose of the matrix and not what we wanted. There are many solutions for this problem:

1. Use an own 2D matrix class in C++ that follows the column-major indexing scheme, and consider vectors of points as row vectors both on the CPU and on the GPU.
2. Use an own 2D matrix class in C++ that follows the column-major indexing scheme, and consider vectors of points as column vectors both on the CPU and on the GPU. The matrices will be transposed with respect to the previous solution.
3. Use the standard 2D matrix indexing on the CPU (row-major) and the standard 2D matrix indexing on the GPU (column-major), but consider points as row vector in the CPU program (and therefore put on the left side of the transformation matrix) and column vector in the GPU program (and put on the right side of the matrix).
4. Use the standard 2D matrix indexing on the CPU (row-major) but transpose the matrix when passed to the GPU, and consider points as row vector both in the CPU program

and in the GPU program.

We use option 4, and setting the third parameter of **glUniformMatrix4fv** to TRUE enables just the required transpose.

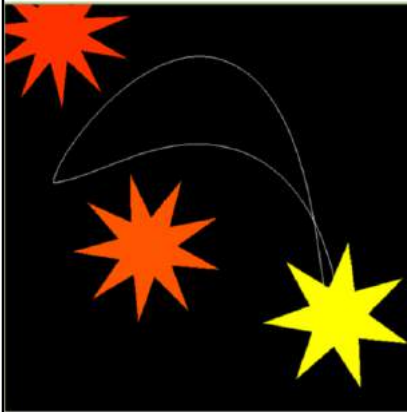
Vertex Array Objects are our virtual world objects already uploaded to the GPU. With **glBindVertexArray(vao)** we can select one object for subsequent operations (drawing) and finally **glDrawArrays** gets the current VAO to feed the pipeline, i.e. this object is rendered. We may not send all vertices of this object, so with startIdx and number of elements a subset can be selected. Setting startIdx to 0 and sending all 3 points, our whole triangle is rendered. The first parameter of **glDrawArrays** tells the GPU the topology of the primitive, that is, what the vertices define. In our case, triangles, and as we have only 3 vertices, a single triangle.



This is the set of possible primitive types. Basically points, line segments and triangles, but in sophisticated options sharing vertices is also possible.


1. házi

Twinkle, twinkle little star



A 2D virtuális világ három, legalább 7-ágú forgó és pulzáló csillagot tartalmaz, amelyek színe különböző. A legfényesebb csillag az egérklikkek által kijelölt zárt -0.8-as tenziójú Catmull-Rom spline-t követi, amelynek csomóértékei a gomblenyomáskori idők. A legutolsó és legelső kontrolpont között 0.5 sec telik el, majd a csillag periódikusan újra bejárja a görbét. A görbe mindenhol folytonosan deriválható, a legelső pontban is. A másik két csillagot a legfényesebb csillag a Newton féle gravitációs erővel vonzza, azaz mozgatja. A gravitációs konstanst úgy kell megválasztani, hogy a mozgás élvezhető legyen. SPACE lenyomására a virtuális kameránkat a fényes csillaghoz kapcsolhatjuk, egyébként a kamera statikus. Bónusz: Doppler effektus

Házi keret könyvtárakkal



Department of Control Engineering
and Information Technology

1. Munkatársak 2. Publikációk 3. Projektek 4. Oktatott tárgyak 5. Előrejelző

Grafikus alap hardver és szoftver

Számítógépes grafika

1. Számítógépes grafika
2. Számítógépes vizualizáció
3. Jólételevételezés
4. Grafikus ábrák felépítése
5. 3D grafikus rendszerek
6. GPU architektúrák
7. GPU programozás és párhuzamos rendszerek
8. Párhuzamos programozás laboratórium
9. Vizualizáció és kódolás
10. Technológiai Platformok 1.
11. Technológiai Platformok 2.
12. GPU általános célú programozás (GPGPU)
13. 3D - geometria modellezés, alakzatkonstrukció, nyomatás
14. Önálló laboratórium

Számítógépes grafika

1. Alapfogalmak
2. Analitikus geometria
3. Geometriai modellezés
4. Geometriai transzformációk
5. 2D képfeldolgozás


Számítógépes grafika

1. Számítógépes grafika
2. Számítógépes vizualizáció
3. Jólételevételezés
4. Grafikus ábrák felépítése
5. 3D grafikus rendszerek
6. GPU architektúrák
7. GPU programozás és párhuzamos rendszerek
8. Párhuzamos programozás laboratórium
9. Vizualizáció és kódolás
10. Technológiai Platformok 1.
11. Technológiai Platformok 2.
12. GPU általános célú programozás (GPGPU)
13. 3D - geometria modellezés, alakzatkonstrukció, nyomatás
14. Önálló laboratórium

Grafikus rendszerek, 2D kimeneti és bemeneti csövezetek, megjelenítők.
Grafikus könyvtárak, eseménykezelés, eszközfüggetlenség, OpenGL 3 és freeglut, Vertex array object, Vertex buffer object, Vertex/fragment shaders, Uniform variables

Példaprogram glw és freeglut könyvtárakkal

Képek:

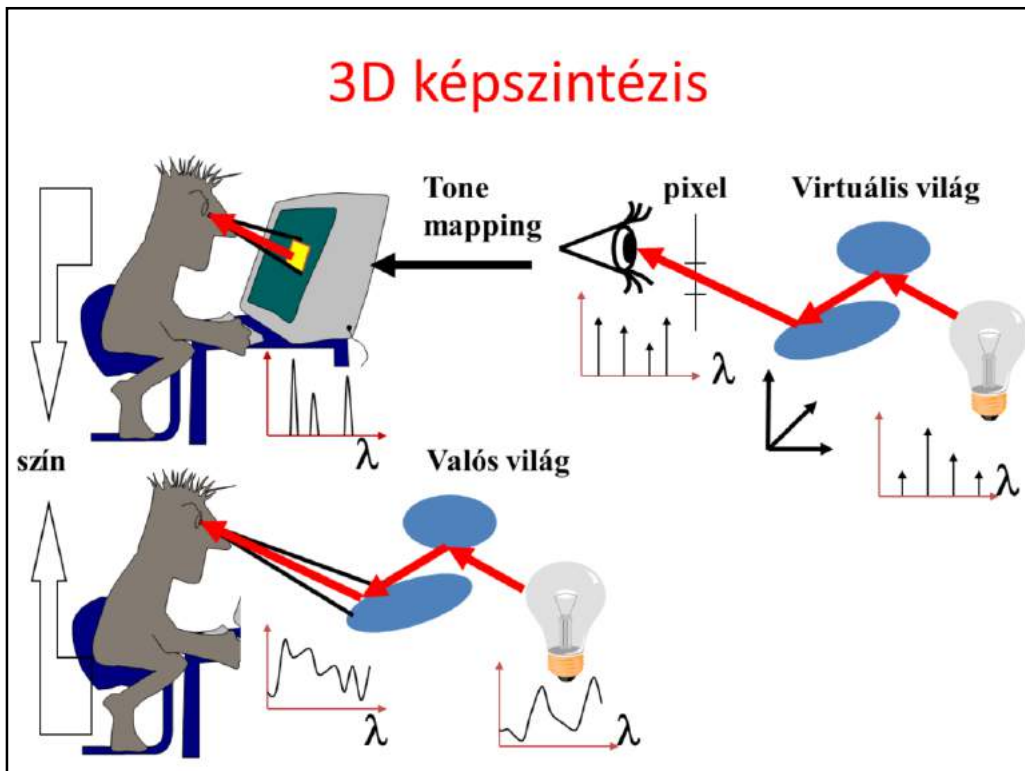


Előadás-anyagok
1. Grafikus hardver - HW
1.7.15 előadás

*Science is either physics or
stamp collecting.*
Rutherford

3D képszintézis fizikai alapmodellje

Szirmay-Kalos László



In order to compute the image, the power arriving at the eye from the solid angle of each pixel needs to be determined on different wavelengths.

We establish a virtual world model in the computer memory, where the user is represented by a single eye position and the display by a window rectangle. Then we compute the power going through the pixel toward the eye on different wavelengths, which results in a power spectrum.

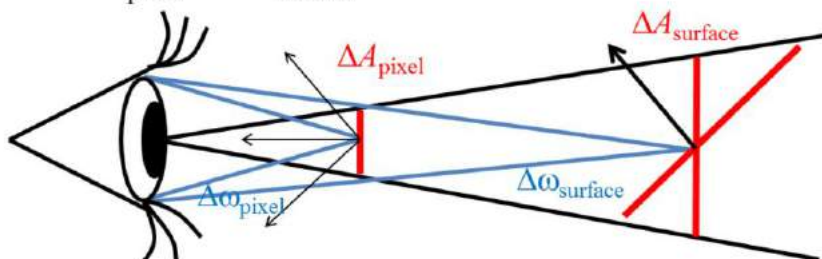
If we can get the display to emit the same photons (i.e. the same number and of the same frequency), then the illusion of watching the virtual world can be created. As the human eye can be cheated with red, green, and blue colors, it is enough if the display emits light on these wavelengths. The last step of rendering is the conversion of the calculated spectrum to displayable red, green and blue intensities, which is called tone mapping. If we compute the light transfer only on these wavelengths, then this step can be omitted and the resulting spectrum can be used directly to control the monitor.

One crucial question is what exactly should be computed that describes the strength of the light intensity and when the pixel is controlled accordingly, provides the same color perception as the surface. Note that the pixel is at a different distance than the visible surface. The orientations of the display

surface and of the visible surface are also different. The total emitted power would definitely be not good since it would mean less photons for the eye for farther sources.

Sugársűrűség (Radiancia): $L(\mathbf{x}, \mathbf{V})$

Wanted: $\Delta\Phi_{\text{pixel}} = \Delta\Phi_{\text{surface}}$



Geometria: $\Delta A_{\text{pixel}} \cos\theta_{\text{pixel}} \Delta\omega_{\text{pixel}} = \Delta A_{\text{surface}} \cos\theta_{\text{surface}} \Delta\omega_{\text{surface}}$

Radiancia = Egységnyi vetített terület által egységnyi

térszögbe sugárzott teljesítmény

[W/sr/m²]

$$L(\mathbf{x}, \mathbf{V}) = \frac{\Delta\Phi}{\Delta A \cos\theta \Delta\omega}$$

We should work with power density instead of the power, that is computed with respect to the solid angle in which the light is emitted and with respect to the size of the projected surface. The density computed as the power divided by the projected surface and the solid angle of emission is called the radiance.

An important theorem states that if two surfaces have the same radiance, then they look identical no matter whether they are at a different distance or have different orientation. The proof is based on that if in a solid angle the eye would gather the same number of photons, i.e. energy, then it would not be able to distinguish the source surfaces. Let us compute this power for two surfaces that are seen in the same solid angle and have the same radiance.

If the surface is closer, then its real area is smaller, but the solid angle in which the pupil of the eye can be reached from this surface is larger. Both the solid angle and the surface changes with the square of the distance and the two factors compensate each other. If the surface is not perpendicular to the viewing direction, then the surface seen in a given solid angle is larger, but the cosine factor will be proportionally smaller, so again we see no difference.

So, the conclusion is that we should compute the radiance of a surface and set

the pixel of the display to have the same radiance. Then the two surfaces will be identical for the eye.

The fact that surfaces having the same radiance but at different distances look similar can also be interpreted as that the radiance does not change along a ray.

Fény-felület kölcsönhatás

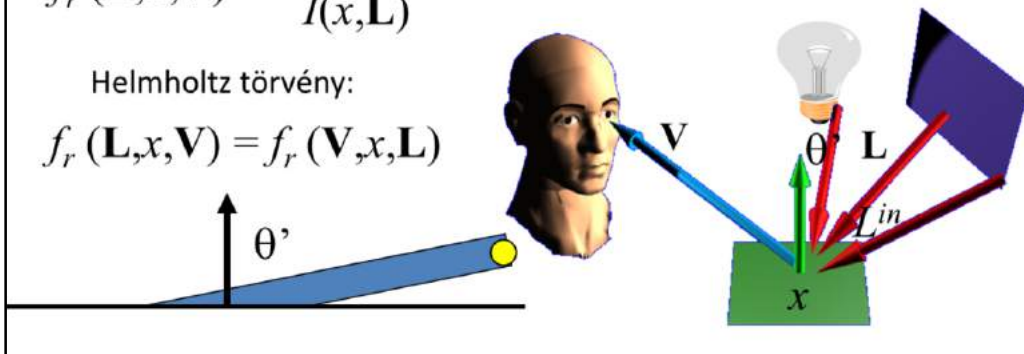
Radiancia = Irradiancia · BRDF

$$L(x, \mathbf{V}) = L^{in}(x, \mathbf{L}) \cdot \cos\theta' \cdot f_r(\mathbf{L}, x, \mathbf{V})$$

$$f_r(\mathbf{L}, x, \mathbf{V}) \stackrel{\text{def}}{=} \frac{L(x, \mathbf{V})}{I(x, \mathbf{L})}$$

Helmholtz törvény:

$$f_r(\mathbf{L}, x, \mathbf{V}) = f_r(\mathbf{V}, x, \mathbf{L})$$



The reflected radiance of a surface depends on the irradiance and the likelihood of the reflection. The irradiance is the incident radiance and a geometric factor that expresses that the illumination is weaker if the light arrives from a non-perpendicular direction since a unit cross section light beam illuminates a larger surface on which the photons are distributed. This cosine term is also called the geometric term and term expresses that a non perpendicular illumination is spread over a larger surface. The likelihood of reflection is expressed by the Bi-directional Reflectance Distribution Function. In real life, BRDFs are symmetric.

Fénynél a hullámhosszok külön kezelhetők



- Relativisztikus tömeg kicsi: $m = E/c^2 = hf/c^2$
- A foton energia (hullámhossz) nem változik rugalmas ütközésnél
- Elnyelődési valószínűség energiafüggő

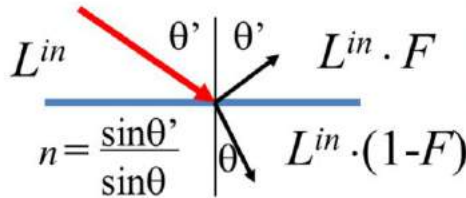
In computer graphics we consider photons in the visible wavelength range, roughly from 300 to 700 nanometer wavelengths. A photon has zero rest mass, otherwise it would not be able to fly with the speed of light. However, it has non-zero energy and impulse. The energy is proportional to frequency f of the light as stated by Einstein who invented this law when examining the photonelectric effect. He got his Nobel prize for this and not for the theory of relativity. Using the equivalence of the energy and mass, which was also published by Einstein as a short paper in 1905, we can assign a relativistic weight to the photon as the Planck constant h multiplied by the frequency and divided by the square of the speed of light.

If f is small, then this relativistic mass is small. When photons meet a material, photons collide or scatter by the electrons or less probably with the core of atoms. For photons belonging to the visible spectrum, the relativistic mass of the photon is much smaller than the mass of the electron, thus a photon bounces off the electron like a ball bounces off from a rigid wall or a billiard ball bounces off from the edge of the table. If the collision is elastic, then the photon energy is preserved and the electron does not change its energy level.

If the collision is inelastic, then the energy of the photons is absorbed by the electron, this is the photoelectric effect, and the number of photons gets smaller. The probability of inelastic scattering, i.e. the albedo associated with a collision is energy dependent.

Summarizing when photons meet electrons, their number may get smaller but their energy level and consequently their frequency remain the same. This is the reason that in computer graphics wavelengths or frequencies are handled independently.

Sima felület: Fresnel egyenlet



$$n = \frac{\sin \theta'}{\sin \theta}$$



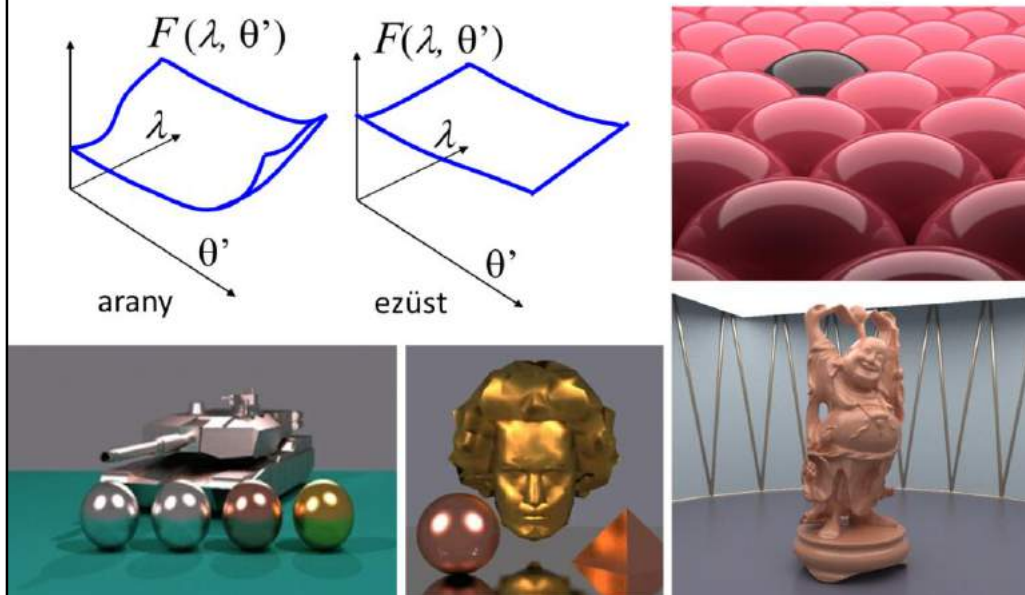
„Sima” = 1 pixelben látható
felület síknak tekinthető

$$F = \frac{1}{2} \left| \frac{\cos \theta - (n + kj) \cos \theta'}{\cos \theta + (n + kj) \cos \theta'} \right|^2 + \frac{1}{2} \left| \frac{\cos \theta' - (n + kj) \cos \theta}{\cos \theta' + (n + kj) \cos \theta} \right|^2$$

$$F \approx F0 + (1 - F0) \cdot (1 - \cos \theta')^5, \quad F0 = \frac{(n - 1)^2 + k^2}{(n + 1)^2 + k^2}$$

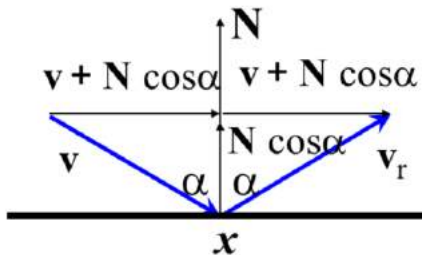
The simplest arrangement for the light transfer is a single plane that separates the space into two half spaces of different materials. According to the laws of geometric optics, the illumination ray is broken to a reflection ray meeting the reflection law and a refraction ray obeying the Snell's law of refraction. Here n is the index of refraction, which expresses the ratios of speeds of light outside and inside the material. The Fresnel equations define the amount of reflected energy (i.e. the probability that a photon is reflected). The Fresnel function can be calculated from index of refraction n , extinction k , incident angle θ' and refraction angle θ . The extinction is negligible for non-metals. We also show a simplified Fresnel term.

Fresnel függvény



The Fresnel function depends on the wavelength and on the incident angle. When we see an object, we can observe surfaces of many different orientations, so we perceive the Fresnel function as a whole.

Tükörirány



$$\cos \alpha = - (v \cdot N)$$

$$v_r = v + 2 N \cos \alpha$$

```
vec3 reflect(vec3 inDir, vec3 normal)
{
    return inDir - normal * dot(normal, inDir) * 2.0f;
};
```

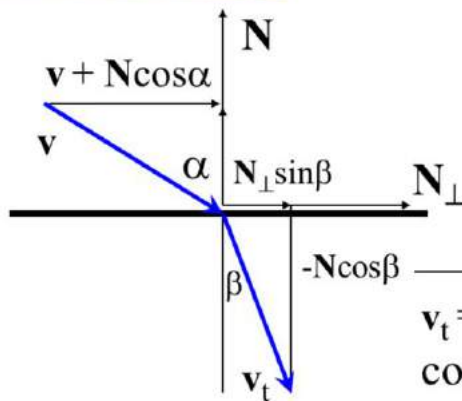
To render smooth surfaces, we should compute the ideal reflection direction. Assume that incident direction v and surface normal N are unit length vectors. Incident direction v is decomposed to a component parallel to the normal and a component that is perpendicular to it. Then, the reflection direction is built up from these two components.



Törési irány

Snellius-
Descartes

$$n = \frac{\sin \alpha}{\sin \beta}$$



$$N_{\perp} = \frac{v + N \cos \alpha}{\sin \alpha}$$

$$v_t = N_{\perp} \sin \beta - N \cos \beta$$

$$v_t = v/n + N(\cos \alpha/n - \cos \beta)$$

$$\cos \beta = \sqrt{1 - \sin^2 \beta} = \sqrt{1 - \sin^2 \alpha / n^2}$$

$$v_t = v/n + N(\cos \alpha/n - \sqrt{1 - (1 - \cos^2 \alpha)/n^2})$$

The refraction direction calculation is also similar. The refraction direction v_t is expressed as a combination of the normal vector and a vector that is perpendicular to the normal, N_{\perp} . These vectors should be combined with weights $\cos(\beta)$ and $\sin(\beta)$ where β is the refraction angle.

N_{\perp} is expressed from $v + N \cos(\alpha)$ by dividing it with its length $\sin(\alpha)$.

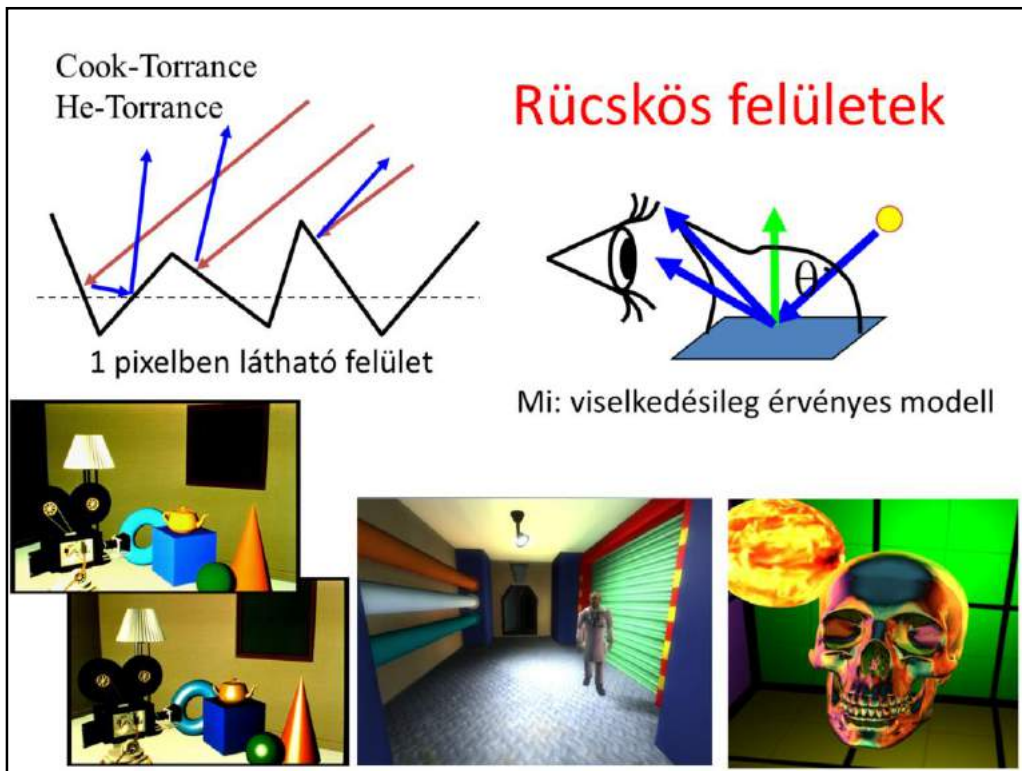
Then $\sin(\beta)/\sin(\alpha)$ is replaced by the reciprocal of the index of refraction.

SmoothMaterial class

```
class SmoothMaterial {
    vec3  F0; // F0
    float n;  // n
public:
    vec3 reflect(vec3 inDir, vec3 normal) {
        return inDir - normal * dot(normal, inDir) * 2.0f;
    }
    vec3 refract(vec3 inDir, vec3 normal) {
        float ior = n;
        float cosa = -dot(normal, inDir);
        if (cosa < 0) { cosa = -cosa; normal = -normal; ior = 1/n; }
        float disc = 1 - (1 - cosa * cosa)/ior/ior;
        if (disc < 0) return reflect(inDir, normal);
        return inDir/ior + normal * (cosa/ior - sqrt(disc));
    }
    vec3 Fresnel(vec3 inDir, vec3 normal) {
        float cosa = fabs(dot(normal, inDir));
        return F0 + (vec3(1, 1, 1) - F0) * pow(1-cosa, 5);
    }
};
```

Putting these together, we can implement a material class representing ideally smooth surfaces. By “smooth” surface we mean a surface that can be assumed to be planar if we consider just a region that is visible in a pixel.

The material properties are expressed by the Fresnel function at perpendicular illumination, F_0 , which is wavelength dependent since the index of refraction and the extinction are wavelength dependent. In addition to the Fresnel, we also need the index of refraction for the calculation of the refraction direction. Here, wavelength dependence is usually ignored, so we do not simulate dispersion.



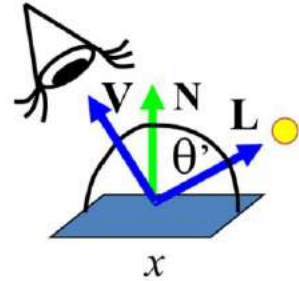
Surfaces are usually not smooth, so they reflect light not just in the ideal reflection direction but practically in all possible directions. Physically, we can imagine these rough surfaces as a random collection of ideal mirror microfacets that reflect light according to their random orientation.

As we see not a single microfacet in a pixel, but a large collection of them, we perceive the average radiance reflected by this collection.

Photons may have a single scattering on these microfaces when the average is maximum around the ideal reflection direction of the mean surface. On the other hand, photons may get scattered multiple times, when they “forget” their original direction, so the reflection lobe will be roughly uniform.

Instead of following a probabilistic reasoning, we handle these rough surfaces as a black-box, i.e. empirical model. That is, we describe the behavior of the surface based on everyday experience without any structural analysis. By experience, we say that a rough surface reflects light into all directions, but more light is reflected into the neighborhood of the ideal reflection direction.

Diffúz visszaverődés



- Radiancia = Irradiancia · BRDF
- A nézeti iránytól független
- A BRDF a nézeti iránytól független
- Helmholtz: a BRDF megvilágítási iránytól is független
- A BRDF irányfüggetlen:

$$f_r(\mathbf{L}, x, \mathbf{V}) = k_d(x, \lambda)$$

- Diffúz visszaverődés = nagyon rücskös
 - sokszoros fény-anyag kölcsönhatás
 - színes!

Our first model is for very rough surfaces where all photons get reflected multiple times. Such materials (snow, sand, wall, chalk, cloth etc) have a matte look, they look the same from all viewing directions. Thus, the radiance, which equals to the incident radiance times the BRDF times the geometry term, is independent of the viewing direction. Incident radiance and the geometry term are already independent of the viewing direction, thus the BRDF must also be independent of the viewing direction. According to Helmholtz reciprocity, if the BRDF is independent of the viewing direction, it must be independent of the illumination direction as well, so the BRDF is direction independent.

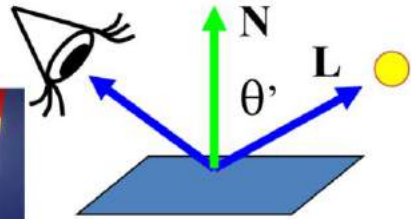
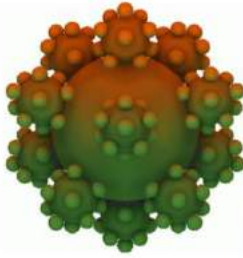
Diffuse surfaces correspond to very rough surfaces where a photon collides many times. The Fresnel depends on the wavelength, which is strong for metals and weak for non-metals. Even if a single reflection changes the spectrum just a little, multiple reflections amplify this effect, so the final reflected light will have a modified spectrum. Diffuse reflection is primarily responsible for the “own color” of the surface.

Lambert törvény

- Pont/irány fényforrásra válasz
 - BRDF irányfüggetlen,
DE a sugársűrűség függ a megvilágítási iránytól

$$L^{ref} = L^{in} k_d \cos^+\theta'$$

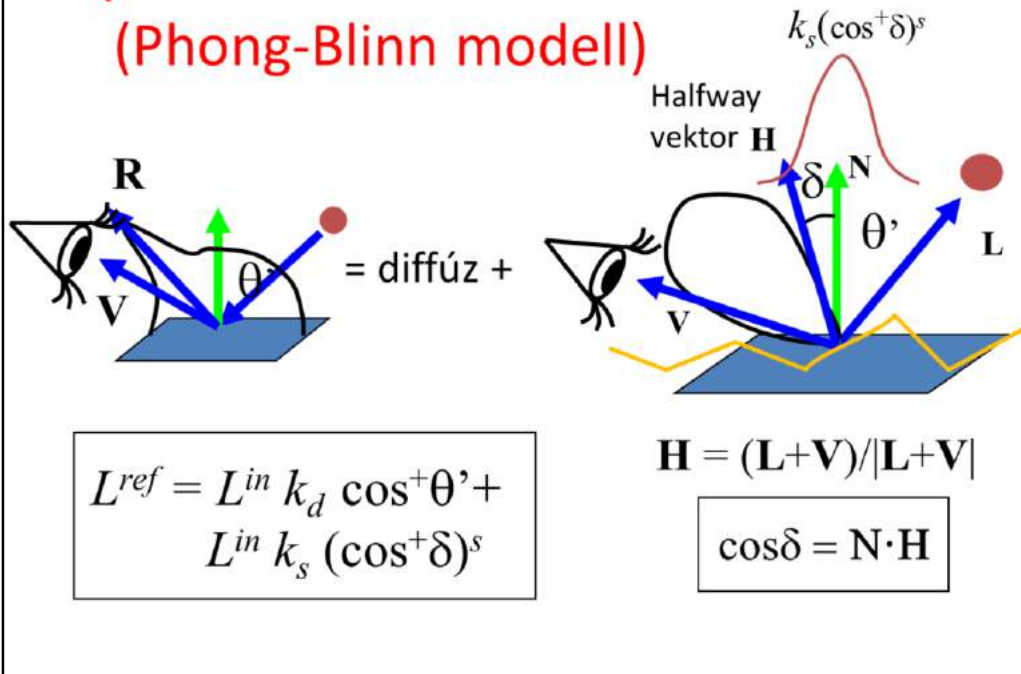
$$\cos\theta' = \mathbf{N} \cdot \mathbf{L}$$



The reflected radiance is the incident radiance times the BRDF, which is constant now, and the geometry term. So for diffuse surfaces, the reflected radiance is proportional to the cosine of the orientation angle. This cosine can be computed as a dot product of the unit surface normal and the unit illumination direction.

If the cosine is negative, i.e. the angle between the surface normal and the illumination direction is greater than 90 degrees, then the illumination is blocked by the object whose surface is considered. In such cases, the negative value is replaced by zero.

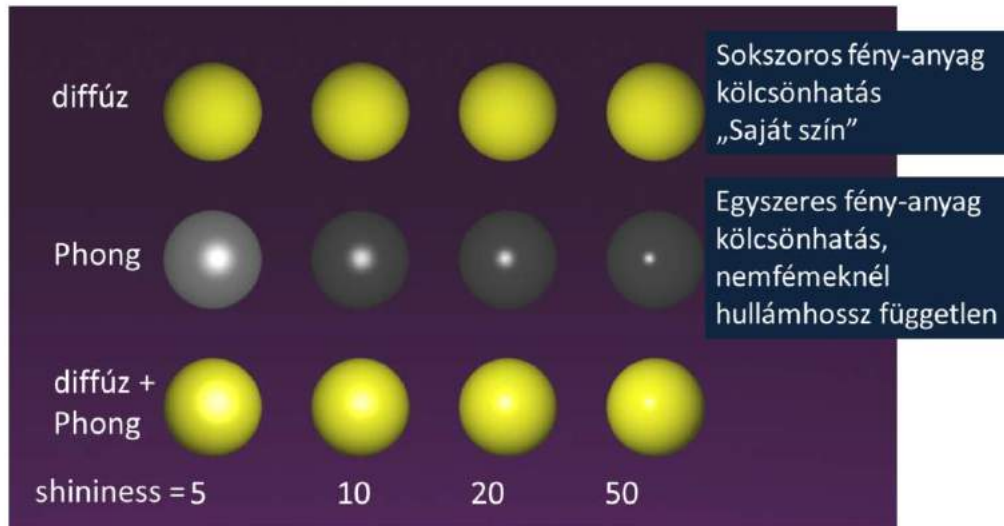
Spekuláris visszaverődés (Phong-Blinn modell)



Shiny, glossy or specular surfaces also reflect the light in all possible directions, but they look differently from different viewing directions. We can observe the blurred reflection of the light sources, thus they reflect more light close to the ideal reflection direction.

We model such surfaces as a combination of diffuse reflection where the radiance is constant and a specular reflection where the radiance is great around the ideal reflection direction. According to the microfacet model, diffuse reflection is caused by multiple light microfacet interaction while specular reflection is the result of a single light microfacet interaction. In order to model the specular reflection lobe, we need a function that is maximum at the reflection direction and decreases in a controllable way if the viewing direction gets farther from the reflection direction. Phong and Blinn proposed the $\cos^{\text{shininess}}(\delta)$ function where δ is the angle between the macroscopic surface normal and the microfacet normal. The shininess exponent defines how shiny the surface is.

Diffúz+Phong anyagok



Diffuse reflection simulates multiple light-surface interaction and is colored. Specular reflection is the model of the single light-surface interaction and it is proportional to the Fresnel function. For non metals, the wavelength dependence of the Fresnel is moderate, so for non metals the specular reflection is said to be "white".

RoughMaterial class

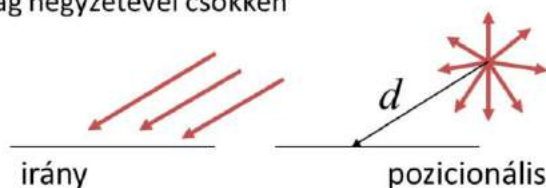
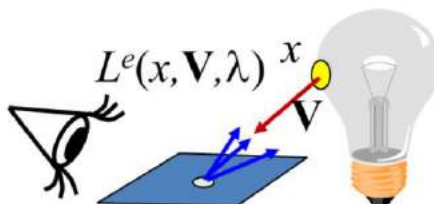
```
class RoughMaterial {
    vec3 kd, ks;
    float shininess;
public:
    vec3 shade( vec3 normal, vec3 viewDir, vec3 lightDir,
                vec3 inRad)
    {
        vec3 reflRad(0, 0, 0);
        float cosTheta = dot(normal, lightDir);
        if(cosTheta < 0) return reflRad;
        reflRad = inRad * kd * cosTheta;
        vec3 halfway = (viewDir + lightDir).normalize();
        float cosDelta = dot(normal, halfway);
        if(cosDelta < 0) return reflRad;
        return reflRad + inRad * ks * pow(cosDelta, shininess);
    }
};
```

Fényforrások

- Geometria+sugársűrűség:

- Absztrakt fényforrások:

- Írány fényforrások: egyetlen irányba sugároz, a fénysugarak párhuzamosak, az intenzitás független a pozíciótól
- Pozicionális fényforrás: egyetlen pontból sugároz, az intenzitás a távolság négyzetével csökken



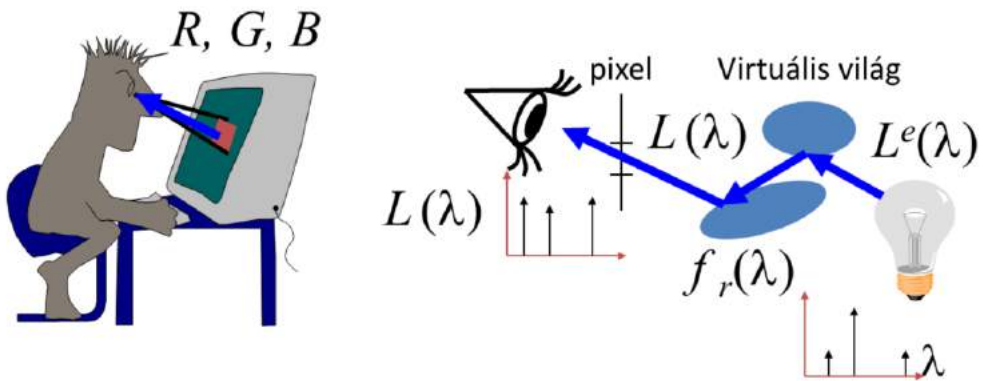
Real light sources are defined by their emission radiance, L^e . When the reflected radiance of a point is considered, the contribution of all those light source points should be added which are visible from the point of interest. This means integration. Thus, we often prefer abstract light source models, that can illuminate a surface just from a single direction, which saves integration.

In case of directional light sources, the radiance is constant everywhere, so is the illumination direction. In other words, the illumination rays are parallel. The Sun is an example for directional light source if we are on the Earth.

For point light sources, the illumination direction points from the location of the source to the illuminated point. The radiance decreases with the square of the distance.

If we ignore the dependence of the radiance on the distance, directional light sources can be considered as point sources being at infinity.

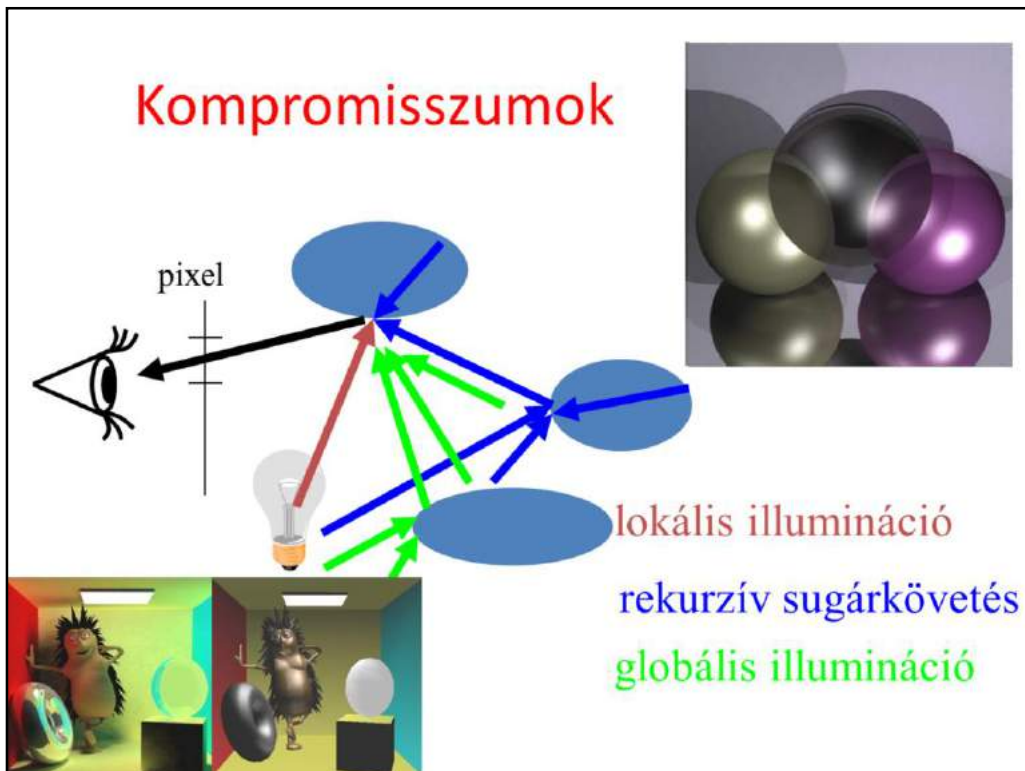
Képszintézis



- Pixelben látható felület meghatározása
- A látható pont szem irányú sugársűrűsége
- R, G, B konverzió

Rendering requires the determination of the surface that is visible through a pixel, then the computation of the radiance of this surface in the direction of the eye.

The radiance computed at least on the wavelengths of red, green, and blue, and the results will be written into the frame buffer. In this calculation, light sources are defined by their radiance or power, rough surfaces with their BRDFs, smooth surfaces with their Fresnel functions.



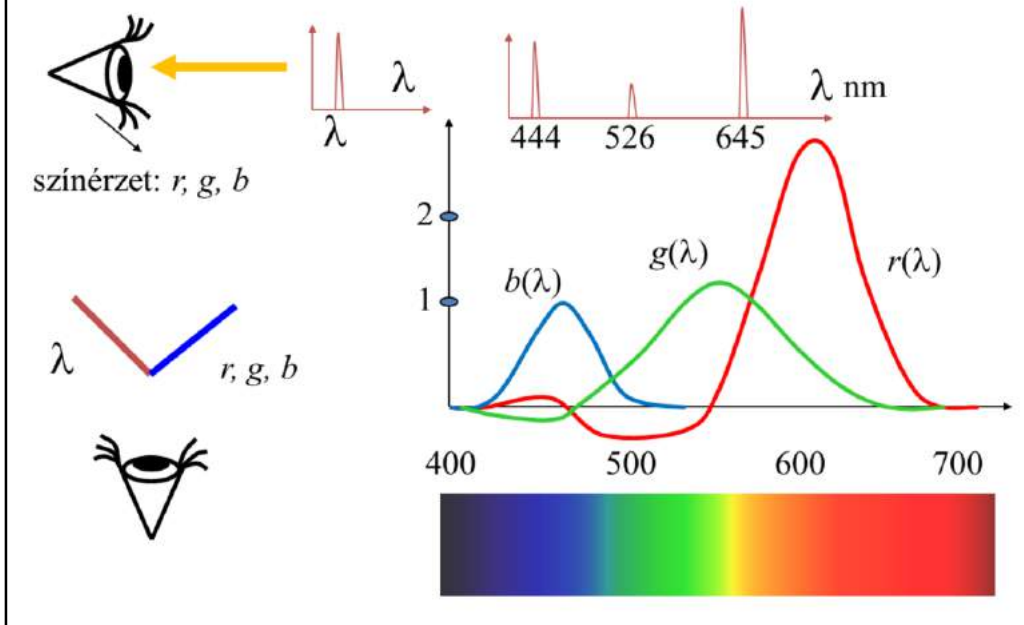
There are different tradeoffs between accuracy of the light transport computation and the speed of the computation.

In the local illumination setting, when the radiance of a surface is calculated, we consider only the direct contribution of the light sources and ignore all indirect illumination.

In recursive ray tracing, indirect illumination is computed only for smooth surfaces, in the ideal reflection and refraction directions.

In the global illumination model, indirect illumination is taken into account for rough surfaces as well. In engineering applications we need global illumination solutions since only these provide predictable results. However, in games and real time systems, local illumination or recursive ray tracing will also be acceptable.

Színérzékelés: monokromatikus fény



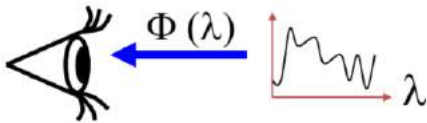
Light is an electromagnetic wave, color is just an illusion created by the human eye and the brain. As the eye is a poor spectrometer, we can cheat it with a different spectrum, the eye cannot tell the difference. This fact is exploited by displays, which can emit light just around three wavelengths. So the task is to convert the computed spectrum to the intensities of the three lamps associated with a pixel. To solve this, we should understand how the illusion of color is created. As the illusion is deep in our brain, we can use only subjective comparative experiments to find out what color means.

In our experiment, we have two white sheets, the first is illuminated by a unit power monochromatic light beam of wavelength λ , the other is by three lamps of controllable intensities and of wavelengths, say, 444, 526, 645 nanometers, which could be seen as red, green and blue (we could choose other reference wavelengths as well, they just have to be far enough; this particular selection is justified by the fact that there exists materials that emit light on these wavelengths). A human observer sits in front of the two white sheets and his task is to control the intensities of the three lamps in order to eliminate any perceived difference between the two sheets. If it happens, the monochromatic light and the controlled three component light provide the same color and are called metamers. If the same experiment is repeated in many discrete wavelengths, three color matching functions can be obtained.

Note that the red and the green matching function have negative parts as well, which means, for example, that the 500 nm light can be matched only if some red is added to it.

In the second experiment we can try to match two, three, etc. component light beams and beams of non-unit intensity. We will come to the conclusion that the corresponding r,g,b values of polychromatic light are the sums of the r,g,b, primaries of the monochromatic components, and also that if the intensity of the beam is not unit, then the r,g,b intensities should also be multiplied by the same factor. This means that colors are linear objects.

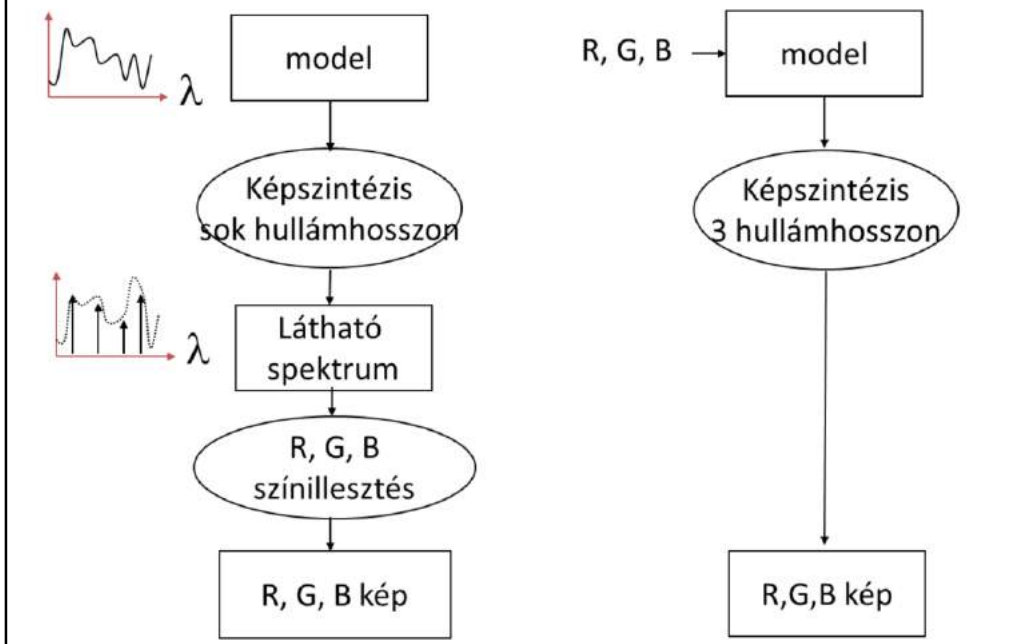
Színérzékelés: nem monokromatikus fény



$$\begin{aligned} r &= \int \Phi(\lambda) r(\lambda) d\lambda \\ g &= \int \Phi(\lambda) g(\lambda) d\lambda \\ b &= \int \Phi(\lambda) b(\lambda) d\lambda \end{aligned}$$

Based on these experiments, we can establish the Grasmann laws of color science. Any spectrum can be matched with three primaries by weighting the monochromatic components by the color matching functions and adding (integrating) different monochromatic components.

Spektrális versus RGB képszintézis



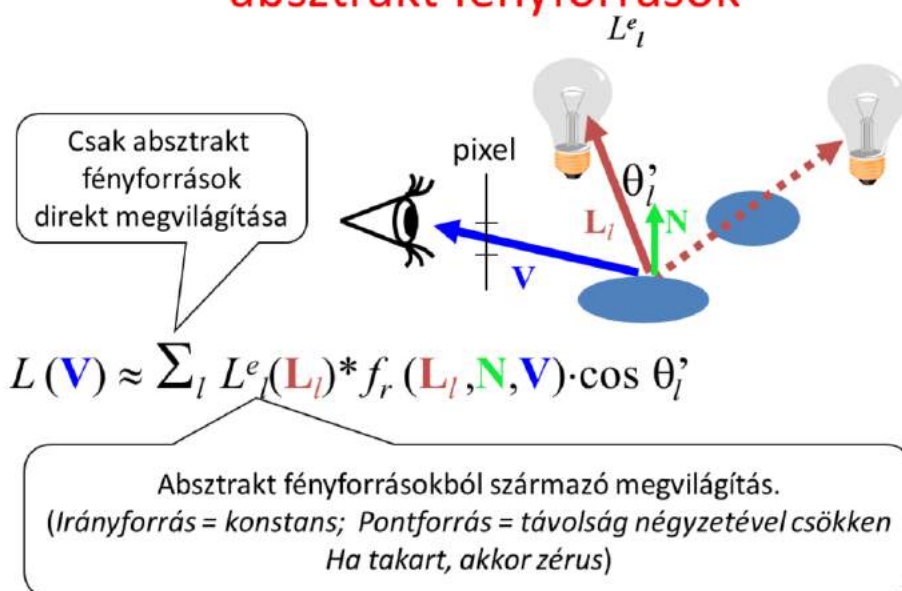
A physically plausible simulation would be executed on many wavelengths (note that wavelengths can be handled independently) resulting in a visible spectrum. The final step of rendering is the conversion of this spectrum to red, green, blue intensities, which can be set in the frame buffer, and ultimately on the display.

However, in many cases, we use an approximation. We assume that light sources emit light directly on the wavelengths of the red, green, blue. Thus, we can immediately get the r,g,b, values without any integration. Note, however, that the rendering process is not linear since the products of radiance values and BRDFs are computed, so this simpler option is just an approximation.

Sugárkövetés: ray-casting, ray-tracing

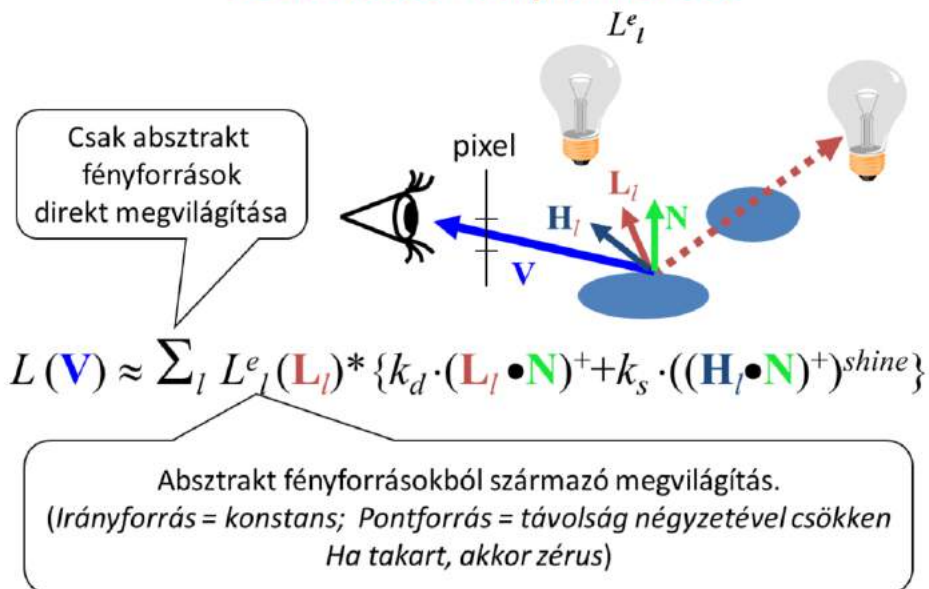
Szirmay-Kalos László

Lokális illumináció: rücskös felületek, absztrakt fényforrások

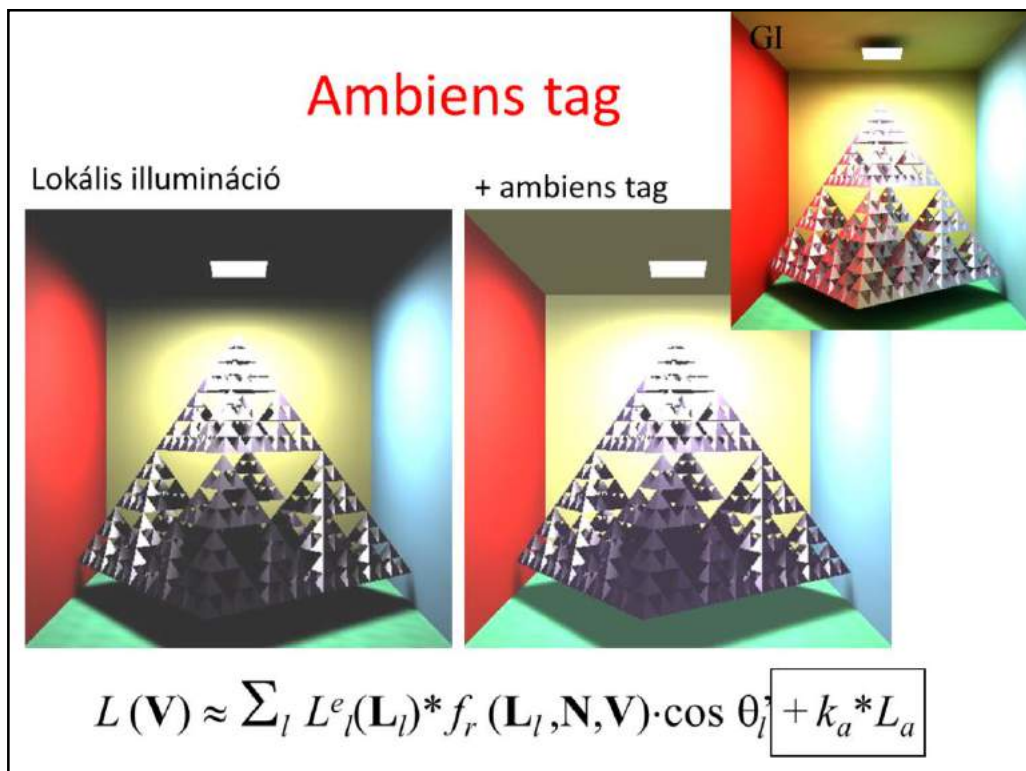


In local illumination rendering, having identified the surface visible in a pixel, we have to compute the reflected radiance due to only the few abstract light sources. An abstract light source may illuminate a point just from a single direction. The intensity provided by the light source at the point is multiplied by the BRDF and the geometry term (cosine of the angle between the surface normal and the illumination direction). The intensity provided by the light source is zero if the light source is not visible from the shaded point. For directional sources, the intensity and the direction are the same everywhere. For point sources, the direction is from the source to the shaded point and decreases with the square of the distance.

Lokális illumináció: rücskös felületek, absztrakt fényforrások



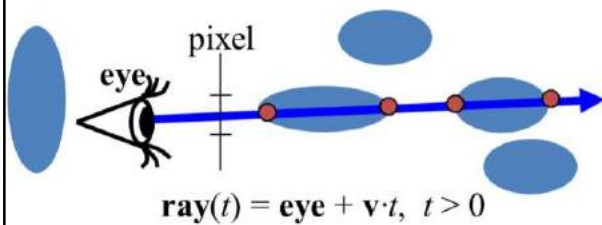
The BRDF times the geometry factor equals to the following expression for diffuse + Phong-Blinn type materials. Here we use different product symbols for different data types; * for spectra, \cdot for multiplying with a scalar, and \bullet for the dot product of two vectors.



In the local illumination model all surfaces that are not directly visible from the light sources are completely back. However, this is against everyday experience when some light indirectly illuminates even hidden regions as well. So, we add an ambient term to the reflected radiance, where the intensity is uniform everywhere and in all directions, and the ambient reflection k_a is a material property (if a physically accurate model is applied, $k_a = k_d * \pi$).

Note, however, that adding the ambient term is a very crude approximation of true indirect lighting, which can be obtained by global illumination algorithms (upper right image).

Láthatóság: trace a ray



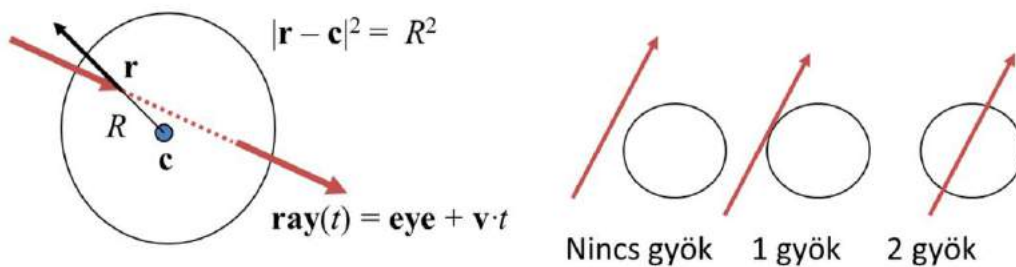
```
struct Hit {  
    float t;  
    vec3 position;  
    vec3 normal;  
    Material* material;  
    Hit() { t = -1; }  
};
```

```
Hit firstIntersect(Ray ray) {  
    Hit bestHit;  
    for(Intersectable * obj : objects) {  
        Hit hit = obj->intersect(ray); // hit.t < 0 if no intersection  
        if(hit.t > 0 && (bestHit.t < 0 || hit.t < bestHit.t))  
            bestHit = hit;  
    }  
    return bestHit;  
}
```

A fundamental operation of ray tracing is the identification of the surface point hit by a ray. The ray may be a primary ray originating at the eye and passing through the pixel, it can be a shadow ray originating at the shaded point and going towards the light source, or even a secondary ray that also originates in the shaded point but goes into either the reflection or the refraction direction. The intersection with this ray is on the ray, thus it satisfies the ray equation, $\text{ray}(t) = \text{eye} + \mathbf{v}t$ for some POSITIVE ray parameter t , and at the same time, it is also on the visible object, so point $\text{ray}(t)$ also satisfies the equation of the surface. A ray may intersect more than one surface, when we need to obtain the intersection of minimal positive ray parameter since this is the closest surface that occludes others.

Function `FirstIntersect` finds this point by trying to intersect every surface with function `Intersect` and always keeping the minimum, positive ray parameter t .

Metszéspont számítás gömbbel



$$|\mathbf{ray}(t) - \mathbf{c}|^2 = (\mathbf{ray}(t) - \mathbf{c}) \bullet (\mathbf{ray}(t) - \mathbf{c}) = R^2$$

$$(\mathbf{v} \bullet \mathbf{v})t^2 + 2((\mathbf{eye} - \mathbf{c}) \bullet \mathbf{v})t + ((\mathbf{eye} - \mathbf{c}) \bullet (\mathbf{eye} - \mathbf{c})) - R^2 = 0$$

Wanted: a pozitív megoldások közül a kisebb

Felületi normális: $(\mathbf{ray}(t) - \mathbf{c})/R$

The implementation of function Intersect depends on the actual type of the surface, since it means the inclusion of the ray equation into the equation of the surface. The first example is the sphere. Substituting the ray equation into the equation of the sphere and taking advantage of the distributivity of the scalar product, we can establish a second order equation for unknown ray parameter t . A second order equation may have zero, one or two real roots (complex roots have no physical meaning here), which corresponds to the cases when the ray does not intersect the sphere, the ray is tangent to the sphere, and when the ray intersects the sphere in two points, entering then leaving it. From the roots, we need the smallest positive one.

Recall that we also need the surface normal at the intersection point. For a sphere, the normal is parallel to the vector pointing from the center to the surface point. It can be normalized, i.e. turned to a unit vector, by dividing by its length, which equals to the radius of the sphere.

Sphere as Intersectable

```
struct Intersectable
{
    Material* material;
    virtual Hit intersect(const Ray& ray)=0;
};

class Sphere : public Intersectable {
    vec3 center;
    float radius;
public:
    Hit intersect(const Ray& ray) {...}
};
```

Implicit felületek

- A felület pontjai: $f(x,y,z) = 0$ vagy $f(\mathbf{r}) = 0$
- A sugár pontjai: $\mathbf{ray}(t) = \mathbf{eye} + \mathbf{v} \cdot t$
- A metszés pont: $f(\mathbf{ray}(t)) = 0$,
 - 1 ismeretlenes, ált. nemlineáris egyenlet: t^*
 - $(x^*, y^*, z^*) = \mathbf{eye} + \mathbf{v} \cdot t^*$
- Normálvektor = $\text{grad } f \Big|_{x^*, y^*, z^*}$
 - $0 = f(x,y,z) = f(x^* + (x-x^*), y^* + (y-y^*), z^* + (z-z^*)) \approx$
 $f(x^*, y^*, z^*) + \frac{\partial f}{\partial x}(x-x^*) + \frac{\partial f}{\partial y}(y-y^*) + \frac{\partial f}{\partial z}(z-z^*)$

Az érintősík
egyenlete:

$$\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \bullet (x-x^*, y-y^*, z-z^*) = 0$$

The equation of the sphere is an example of a more general category, the implicit surfaces that are defined by an implicit equation of the x,y,z Cartesian coordinates of the place vectors \mathbf{r} of surface points. Substituting the ray equation into this equation, we obtain a single, usually non-linear equation for the single unknown, the ray parameter t . Having solved this equation, we can substitute the ray parameter t^* into the equation of the ray to find the intersection point.

The normal vector of the surface can be obtained by computing the gradient at the intersection point. To prove it, let us express the surface around the intersection point as a Taylor approximation. $f(x^*, y^*, z^*)$ becomes zero since the intersection point is also on the surface. What we get is a linear equation of form $\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0) = 0$, which is the equation of the plane, where $\mathbf{n} = \text{grad } f$.

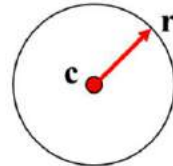
So, the gradient is the normal vector of the plane that approximates the surface locally in the intersection point.

Konkrét példa: gömb normálvektora

$$|\mathbf{r} - \mathbf{c}|^2 = R^2$$

$$|\mathbf{r} - \mathbf{c}|^2 - R^2 = 0$$

$$f(\mathbf{r}) = 0$$



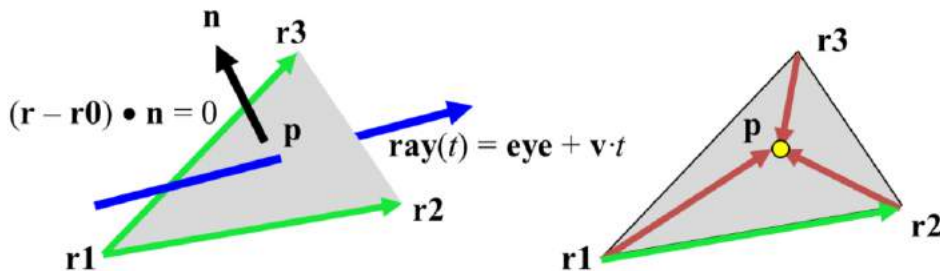
$$f(x,y,z) = (x-c_x)^2 + (y-c_y)^2 + (z-c_z)^2 - R^2 = 0$$

$$\frac{\partial f}{\partial x} = 2(x - c_x) + 0 + 0 - 0$$

$$\frac{\partial f}{\partial x} = 2(x - c_x) \quad \frac{\partial f}{\partial y} = 2(y - c_y) \quad \frac{\partial f}{\partial z} = 2(z - c_z)$$

$$\text{grad } f(x,y,z) = 2 (x-c_x, y-c_y, z-c_z)$$

Háromszög



1. Síkmetszés: $(\text{ray}(t) - \mathbf{r1}) \cdot \mathbf{n} = 0, t > 0$
normál: $\mathbf{n} = (\mathbf{r2} - \mathbf{r1}) \times (\mathbf{r3} - \mathbf{r1})$

$$t = \frac{(\mathbf{r1} - \text{eye}) \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}}$$

2. A metszéspont a háromszögön belül van-e?

$$\begin{aligned} ((\mathbf{r2} - \mathbf{r1}) \times (\mathbf{p} - \mathbf{r1})) \cdot \mathbf{n} &> 0 \\ ((\mathbf{r3} - \mathbf{r2}) \times (\mathbf{p} - \mathbf{r2})) \cdot \mathbf{n} &> 0 \\ ((\mathbf{r1} - \mathbf{r3}) \times (\mathbf{p} - \mathbf{r3})) \cdot \mathbf{n} &> 0 \end{aligned}$$

Felületi normális: \mathbf{n}
vagy árnyaló normálok
(shading normals)

The triangle is the most important primitive because we often use it to approximate arbitrary surfaces. So effective ray-triangle intersection algorithms are still in the focus of research. Now, we present a very simple algorithm, which is far behind the leading methods in terms of efficiency.

The algorithm consists of two steps, first the intersection with the plane of the triangle is found, then we determine whether or not the ray-plane intersection point is inside the triangle. Suppose that the triangle is given by its vertices $\mathbf{r1}$, $\mathbf{r2}$, $\mathbf{r3}$. The equation of its plane is $\mathbf{n} \cdot (\mathbf{r} - \mathbf{r0}) = 0$ where \mathbf{n} is the normal vector and $\mathbf{r0}$ is a point of the plane. Place vector $\mathbf{r0}$ can be any of the three vertices and normal vector \mathbf{n} can be computed as the cross product of edge vectors $\mathbf{r2} - \mathbf{r1}$ and $\mathbf{r3} - \mathbf{r1}$. Substituting the ray equation into this linear equation, we get a linear equation for t , which can be solved. If t is negative, the intersection is behind the eye, so it must be ignored. The positive t is substituted back to the ray equation giving \mathbf{p} as the intersection with the plane.

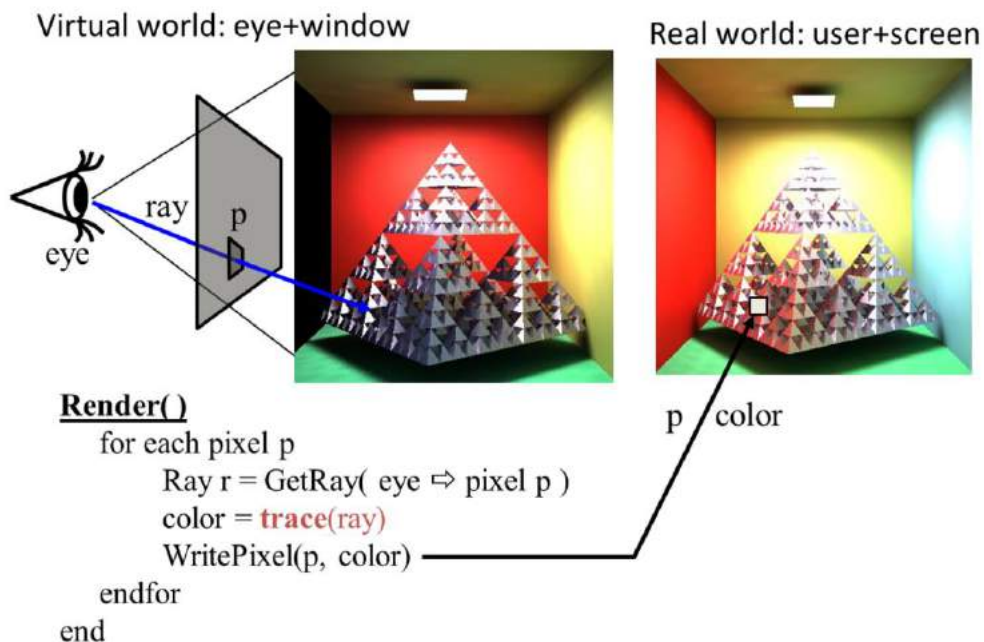
Now we should determine whether \mathbf{p} is inside the triangle. An edge line separates the plane into two half planes, a “good” or one (this is the left one if the edge vector points from $\mathbf{r1}$ to $\mathbf{r2}$) that contains the triangle and the third vertex and a “bad” one that contains nothing. Point \mathbf{p} must be on the good side, i.e. where the third vertex is. Points on the left and right with respect to edge $\mathbf{r1} - \mathbf{r2}$ can be separated using the properties of the cross product.

Assuming that we look at the plane from above, $(\mathbf{r2} - \mathbf{r1}) \times (\mathbf{p} - \mathbf{r1})$ will point towards us if \mathbf{p} is on the left, and it will point down if \mathbf{p} is on the right.

As $\mathbf{n} = (\mathbf{r2} - \mathbf{r1}) \times (\mathbf{r3} - \mathbf{r1})$ points towards us, we can check whether $(\mathbf{r2} - \mathbf{r1}) \times (\mathbf{p} - \mathbf{r1})$ has the same direction by computing their dot product and checking if the result is

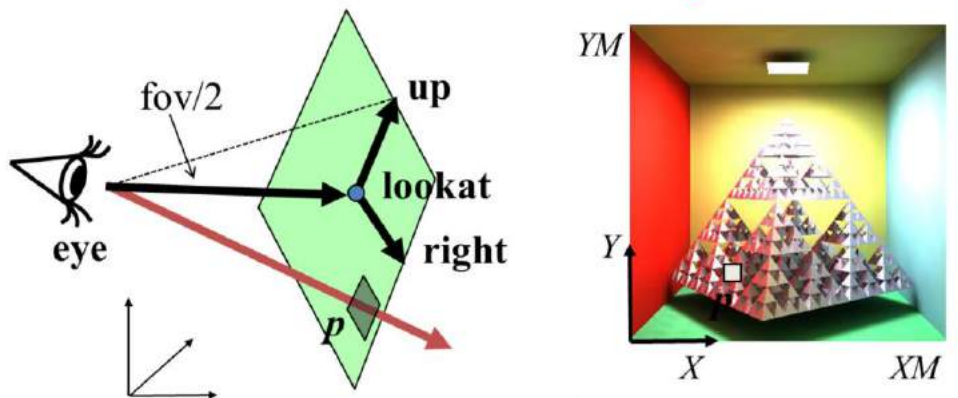
positive (the dot product of two vectors pointing into the same direction is positive, the dot product of two oppositely pointing vectors is negative). A single inequality states that the point is on the good side with respect to a given edge vector. If this condition is met for all three edge vectors, the point is inside the triangle.

Ray tracing: Render



On the top level, ray tracing rendering visits pixels one by one. For every pixel, the virtual camera has a point on its window (in real space we have the user and the screen; in virtual world one of the user's eye is the virtual eye and the display surface is a rectangle). The origin of primary rays is always the eye position. The direction of a ray is from the eye to the center of the pixel on the window rectangle, which is calculated by the GetRay function. With this ray, function Trace is called, which computes the radiance transferred back by this ray (i.e. the radiance of the point hit by this ray in the opposite of the ray direction). The radiance on the wavelengths of r,g,b is written into the current physical pixel.

Kamera: GetRay



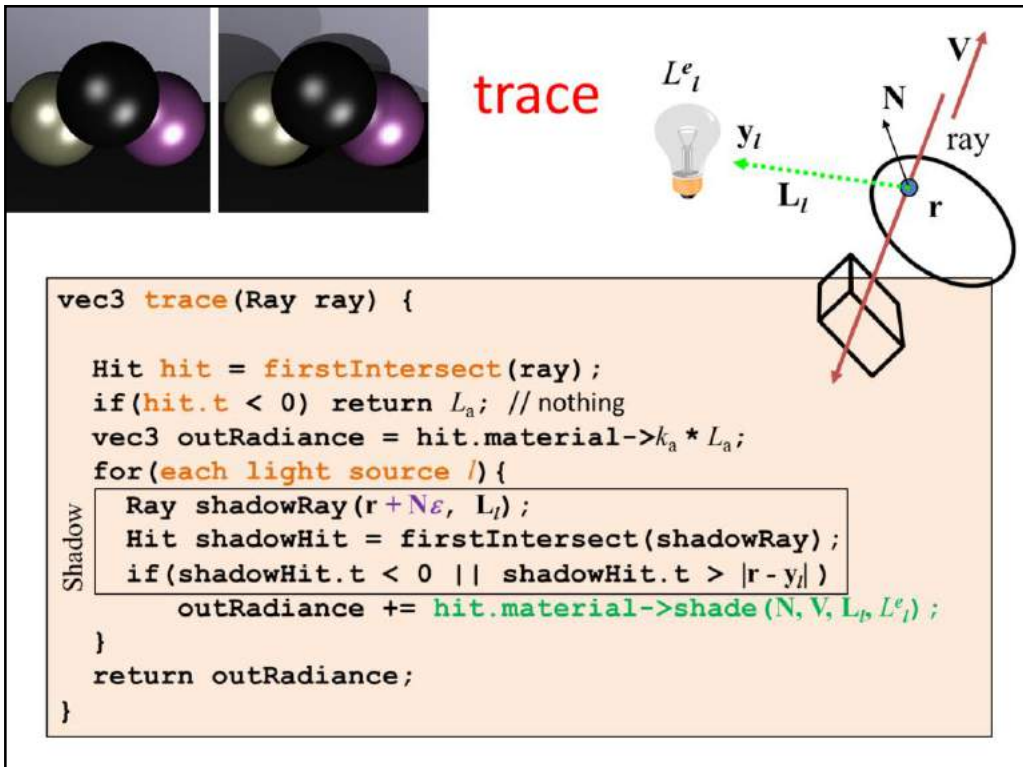
$$\begin{aligned}
 p &= \text{lookat} + \alpha \cdot \text{right} + \beta \cdot \text{up}, \\
 &= \text{lookat} + (2X/XM-1) \cdot \text{right} + (2Y/YM-1) \cdot \text{up}
 \end{aligned}$$

$$\text{Ray dir} = p - \text{eye}$$

Normalizált eszköz koordináták

$$\alpha, \beta \in [-1, 1]$$

To implement the GetRay function, the virtual camera should be defined in the virtual space. The user's location is specified by the place vector called eye. The display surface is represented by a 2D rectangle in the virtual world coordinates. The center of this rectangle is specified by the lookat point, and its orientation and size are defined by two vectors. Right points from the center of the window to the right edge, up from the center to the top edge. If the resolution of the target image is $XM \times YM$, then the center of pixel (X, Y) in world coordinates is $p = \text{lookat} + (2X/XM-1) \cdot \text{right} + (2Y/YM-1) \cdot \text{up}$.

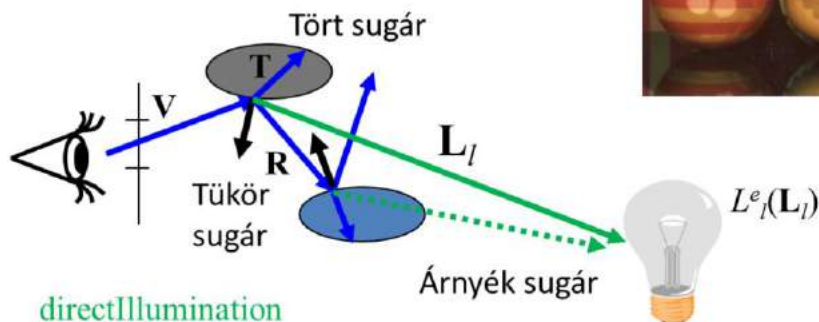


The trace function gets the ray that involves its origin and direction vectors. First we compute the intersection that is in front of the eye and is closest to the eye. The already implemented solution is firstIntersect. This function indicates with a negative value if there is no intersection. In this case, trace returns with the radiance of the ambient illumination.

If some surface is seen, trace computes the contribution of the abstract light sources. To check the visibility of a particular light source, a ray, called shadow ray, is sent from the shaded point towards the light source. If this ray intersects an object and this intersection is closer than the light source, the object occludes the light source so this point is in shadow.

If the surface is smooth and is ideally reflective, then the reflection direction is computed and the trace function is called recursively to compute the radiance of reflection direction. The same is done for the refraction direction if the surface is refractive.

Rekurzív sugárkövetés



direct illumination

$$L(V) \approx \sum_l L^e_l(L_l) * (k_d \cdot (L_l \cdot N)^+ + k_s \cdot ((H_l \cdot N)^+)^{shine} + k_a * L_a$$

$$+ F(V \cdot N) * L^{in}(R) + (1 - F(V \cdot N)) * L^{in}(T)$$

Fresnel Tükör irányból érkező fény 1-Fresnel Törési irányból érkező fény

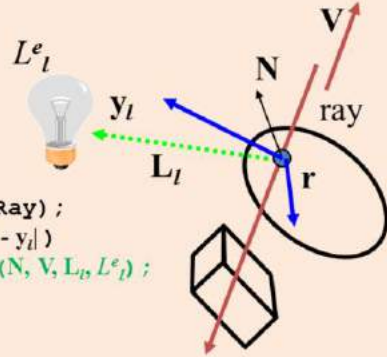
To simulate also smooth surfaces responsible for mirroring and light refraction, the local illumination model should be extended. When the surface visible from the eye is identified, we calculate the radiance as the contribution from abstract light sources but also add the reflection of the radiance from the ideal reflection direction and the refraction of the radiance coming from the ideal refraction direction. According physics, the scaling factors of the radiance values are the Fresnel and 1-Fresnel for reflection and refraction, respectively. However, we do not always insist of physical precision so may use other scaling factors that are set by an artist and not computed as the Fresnel function.

This equation expresses the radiance of a surface point in a given direction as the function of the direct light sources and the radiance coming from the ideal reflection and refraction directions. The question is how these extra terms can be computed.

Let us recognize, that the computation of the radiance delivered back by reflection and refraction rays is essentially the same computation what we are doing right now, just the ray origin and direction should be altered. So the solution of this problem is a recursive function.

trace

```
vec3 trace(Ray ray) {
    Hit hit = firstIntersect(ray);
    if(hit.t < 0) return  $L_a$ ; // nothing
    vec3 outRadiance = hit.material-> $k_a$  *  $L_a$ ;
    for(each light source  $l$ ) {
        Ray shadowRay(r +  $N \epsilon \text{sign}(N \cdot V)$ ,  $L_l$ );
        Hit shadowHit = firstIntersect(shadowRay);
        if(shadowHit.t < 0 || shadowHit.t > |r -  $y_l$ |)
            outRadiance += hit.material->shade( $N$ ,  $V$ ,  $L_l$ ,  $L_l^e$ );
    }
    if(hit.material->reflective) {
        vec3 reflectionDir = reflect( $V$ ,  $N$ );
        Ray reflectedRay(r +  $N \epsilon \text{sign}(N \cdot V)$ , reflectionDir);
        outRadiance += trace(reflectedRay) *  $F(V, N)$ ;
    }
    if(hit.material->refractive) {
        vec3 refractionDir = refract( $V$ ,  $N$ );
        Ray refractedRay(r -  $N \epsilon \text{sign}(N \cdot V)$ , refractionDir);
        outRadiance += trace(refractedRay) * (vec3(1,1,1) -  $F(V, N)$ );
    }
    return outRadiance;
}
```



The trace function gets the ray that involves its origin and direction vectors. First we compute the intersection that is in front of the eye and is closest to the eye. The already implemented solution is firstIntersect. This function indicates with a negative value if there is no intersection. In this case, trace returns with the radiance of the ambient illumination.

If some surface is seen, trace computes the contribution of the abstract light sources. To check the visibility of a particular light source, a ray, called shadow ray, is sent from the shaded point towards the light source. If this ray intersects an object and this intersection is closer than the light source, the object occludes the light source so this point is in shadow.

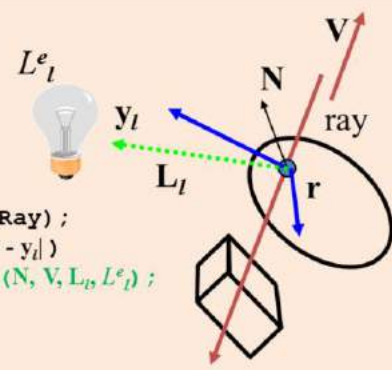
If the surface is smooth and is ideally reflective, then the reflection direction is computed and the trace function is called recursively to compute the radiance of reflection direction. The same is done for the refraction direction if the surface is refractive.

trace

```

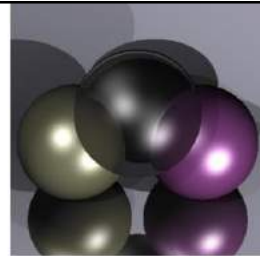
vec3 trace(Ray ray, int depth) {
    if (depth > maxdepth) return La;
    Hit hit = firstIntersect(ray);
    if(hit.t < 0) return La; // nothing
    vec3 outRadiance = hit.material->ka * La;
    for(each light source l) {
        Ray shadowRay(r + Nεsign(N·V), Ll);
        Hit shadowHit = firstIntersect(shadowRay);
        if(shadowHit.t < 0 || shadowHit.t > |r - yl|)
            outRadiance += hit.material->shade(N, V, Ll, Lle);
    }
    if(hit.material->reflective) {
        vec3 reflectionDir = reflect(V, N);
        Ray reflectedRay(r + Nεsign(N·V), reflectionDir);
        outRadiance += trace(reflectedRay, depth+1) * F(V, N);
    }
    if(hit.material->refractive) {
        vec3 refractionDir = refract(V, N);
        Ray refractedRay(r - Nεsign(N·V), refractionDir);
        outRadiance += trace(refractedRay, depth+1) * (vec3(1,1,1) - F(V, N));
    }
    return outRadiance;
}

```



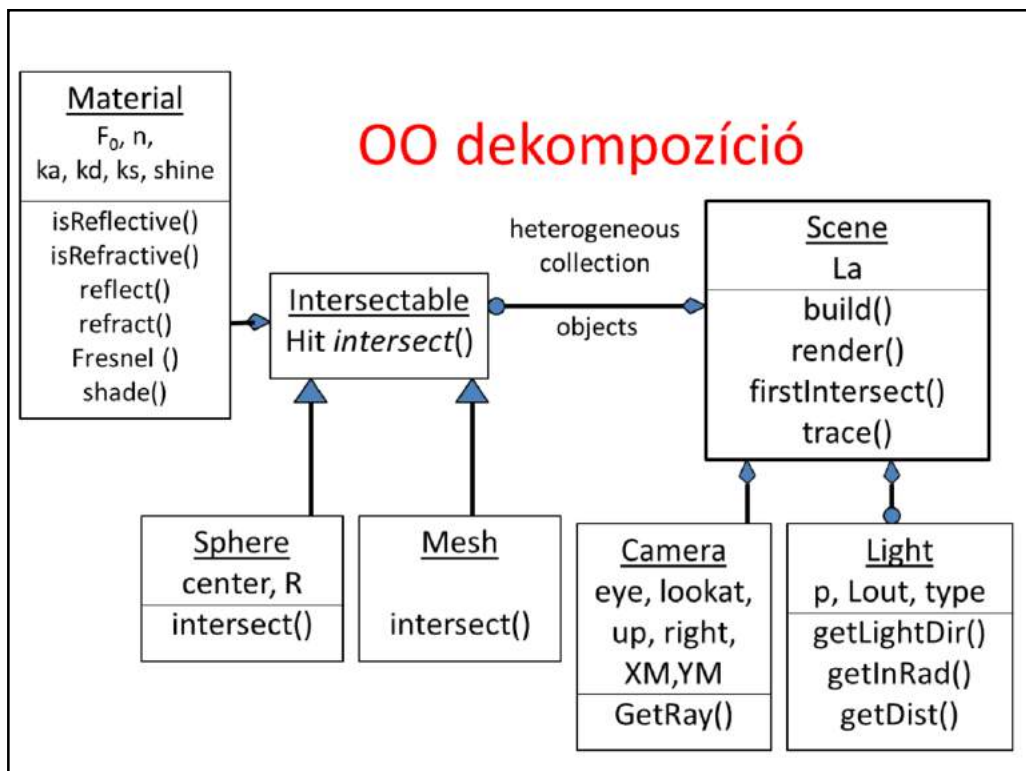
Recursion is a dangerous operation if we cannot make sure that it stops. Assume, for example, that this ellipsoid is made of glass. The ray is refracted into the glass and there can be infinite number of reflections on the internal surface. So our program will surely crash with a stack overflow message. We should limit the recursion depth for any price. This is possible with depth parameter, which is incremented in each recursive call. If depth is greater than the limit, additional calculations are prohibited.

Heckbert Palika házija a névjegyén

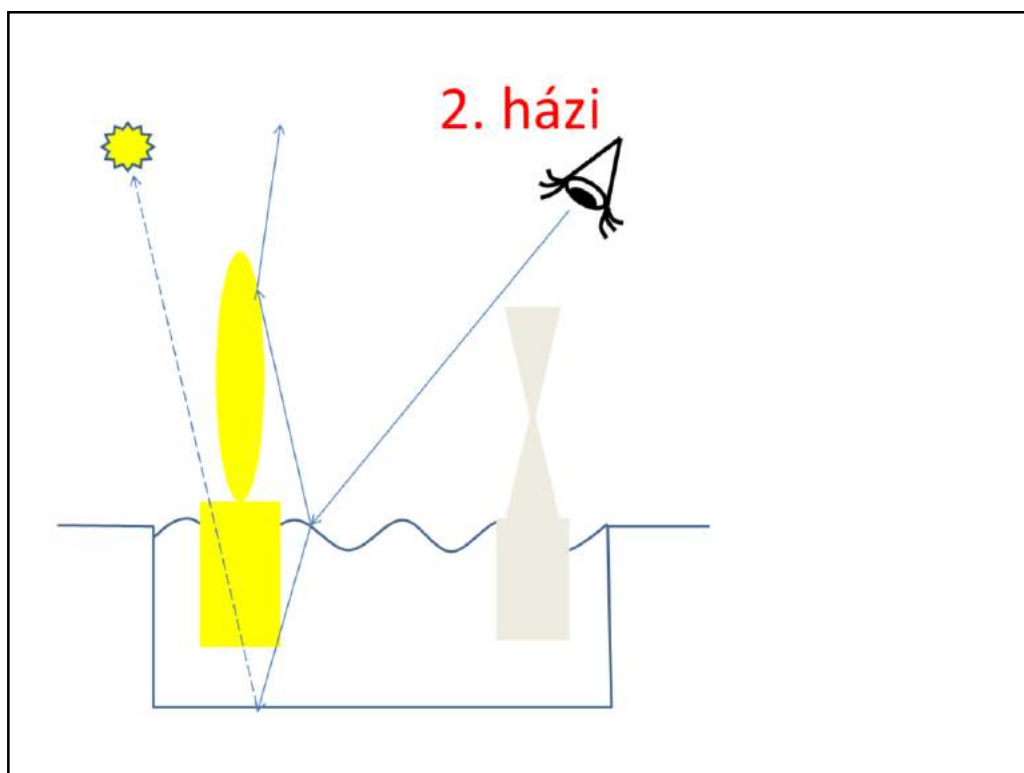


```
typedef struct {double x,y,z;}vec;vec U,black,amb={.02,.02,.02}; struct sphere { vec cen,color;double rad,kd,ks,kt,kl,ir} *s,
*best,sph[]={0.,6.,5.1,1.,1.,9.,.05,2.,85,0.,1.7,-1.,8.,-5.1,5.,2,1.,7.,3,0.,.05,1.2,1.,8.,-5.1,8.,8., 1.,3.,7,0.,0.,1.2,3.,-6.,15.,1.,
.8,1.,7.,0.,0.,6,1.5,-3.,-3.,12.,8,1., 1.,5.,0.,0.,0.,5,1.5,};yx; double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A ,B;
{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B; {B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}
vec vunit(A)vec A;{return vcomb(1./sqrt(vdot(A,A)),A,black);}struct sphere *intersect(P,D)vec P,D;{best=0;tmin=1e30;
s= sph+5;while(s-->sph){b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s ->rad,u=u>0?sqrt(u):1e31,u=b-u>
1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u: tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,c;vec N,color;
struct sphere *s,*l;if(!level--)return black;if(s=intersect(P,D));else return amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb
(-1.,P=vcomb(tmin,D,P),s->cen )));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;l=sph+5;while(l-->sph){if((e=l ->kl*vdot(N,
U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e ,l->color,color);U=s->color;color.x*=U.x;color.y*=
U.y;color.z*=U.z;e=1-eta* eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt (e),N,black))):
black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd, color,vcomb(s->kl,U,black))));}
```

```
main() {printf("%d %d\n",32,32);while(yx<32*32) U.x=yx%32-32/2,U.z=32/2-yx+32,U.y=32/2/tan(25/114.5915590261),
U=vcomb(255., trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/*minray!*/
```



Object oriented decomposition identifies the objects representing the problem. These objects are defined in an abstract way by specifying what operations can be executed on these elements.



Inkrementális 3D képszintézis

Szirmay-Kalos László

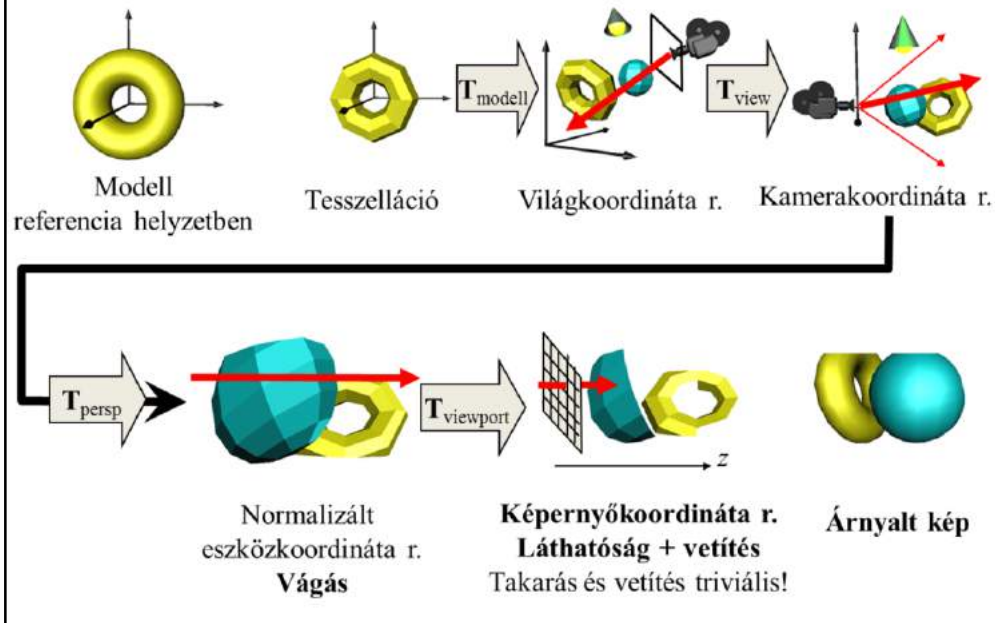
Inkrementális képszintézis

- Sugárkövetés számítási idő \propto
Pixelszám \times Objektumszám \times (Fényforrás szám+1)
-
- koherencia: oldjuk meg nagyobb egységekre
 - feleslegesen ne számoljunk: vágás
 - transzformációk: minden feladathoz megfelelő koordinátarendszert
 - vágni, transzformálni nem lehet akármit: tesszelláció

Ray tracing processes each pixel independently, thus it may repeat calculations that could be reused in other pixels. Consequently, ray tracing is slow, it is difficult to render complex, animated scenes with ray tracing at high frame rates.

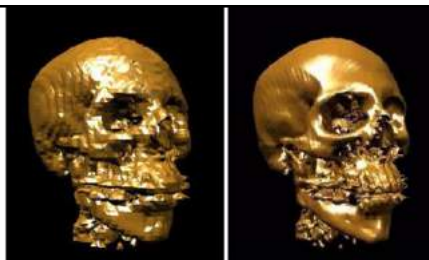
The goal of incremental rendering is speed and it sacrifices everything for it. To obtain an image quickly, it solves many problems for regions larger than a single pixel. This region is usually a triangle of the scene. It gets rid of objects that are surely not visible via clipping. Most importantly, it executes operations in appropriate coordinate systems where this particular operation is simple (recall that ray tracing does everything in world space). Moving object from one coordinate system to another requires transformations. As we cannot transform all types of geometry without modifying the type, we approximate all kinds of geometry with triangles. This process is called tessellation.

3D inkrementális képsztntézis

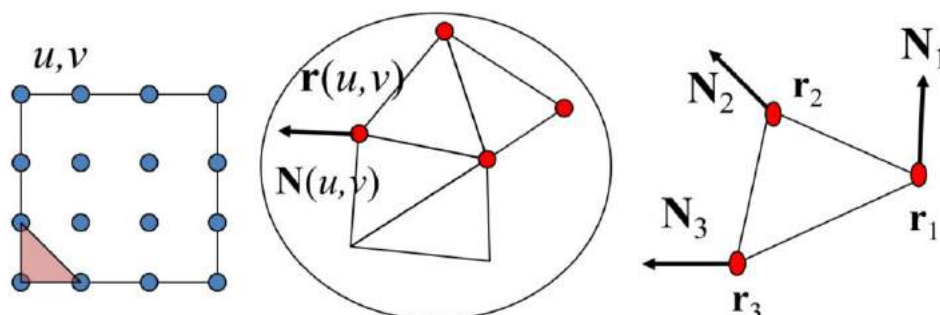


Incremental rendering is a pipeline of operations starting at the model and ending on the image. Objects are defined in their reference state. First surfaces are approximated by triangle meshes. Then the tessellated object is placed in the world, setting its real size, orientation and position. Here, different objects, the camera and light sources meet. Ray tracing would solve the visibility problem here, but incremental rendering applies transformations to find a coordinate system when it is trivial to decide whether two points occlude each other, and projection is also simple. This is the screen coordinate system where rays are parallel with axis z and a ray has the pixel's x, y coordinates. To transform the model to screen coordinates, first we execute the camera transformation which translates and rotates the scene so that the camera is in the origin and looks at the $-z$ direction (the negative sign is due to the fact that we prefer right handed coordinate system here). In the camera coordinate system, projection rays go through the origin and projection is perspective. To simplify this, we distort the space and make rays meet in an ideal point at the end of axis z , so projection rays will be parallel. Clipping is executed here. Finally, we take into account the real resolution of the image and scale the space accordingly.

Tesszelláció

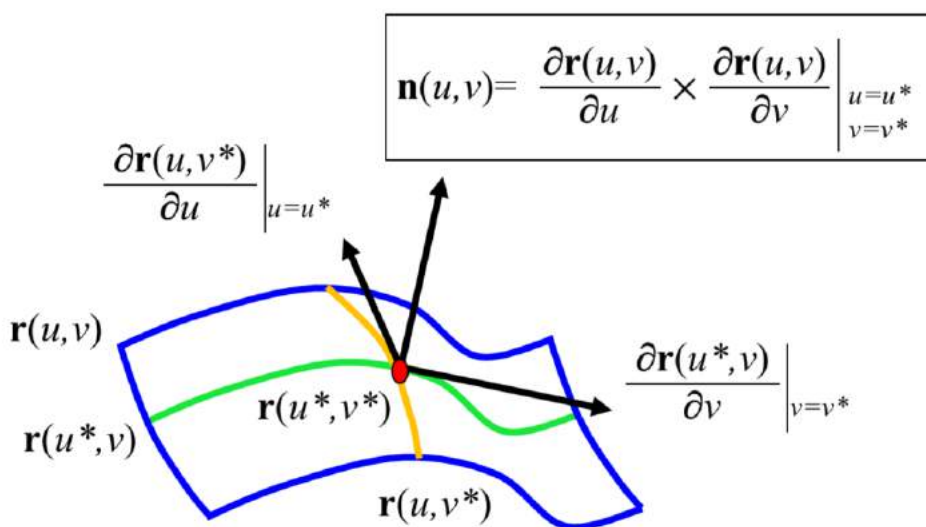


- Felületi pontok: $\mathbf{r}_{n,m} = \mathbf{r}(u_n, v_m)$
- Normálvektor: $\mathbf{N}(u_n, v_m) = \frac{\partial \mathbf{r}(u, v)}{\partial u} \times \frac{\partial \mathbf{r}(u, v)}{\partial v}$
- „Paraméterterben szomszédos” pontokból háromszögek



The tessellation of a parametric surface basically means the evaluation of the surface equation in parameter points that are placed regularly by a grid. Those points form a triangle that are neighbors in parameter space. To obtain the shading normals, the cross product of the partial derivatives of the parametric equation is also computed at the sample points.

Parametrikus felületek normálvektora



To get the normal vector of a parametric surface, we exploit isoparametric lines. Suppose that we need the normal at point associated with parameters u^* , v^* . Let us keep u^* fixed, but allow v to run over its domain. This $\mathbf{r}(u^*, v)$ is a one-variate parametric function, which is a curve. As it always satisfies the surface equation, this curve is on the surface and when $v=v^*$, this curve passes through the point of interest. We know that the derivative of a curve always tangent to the curve, so the derivative with respect to v at v^* will be the tangent of a curve at this point, and consequently will be in the tangent plane.

Similarly, the derivative with respect to u will always be in the tangent plane. The cross product results in a vector that is perpendicular to both operands, so it will be the normal of the tangent plane.

Objektumok az GPU-nak

```
struct Geometry {  
    unsigned int vao, nVtx;  
  
    Geometry( ) {  
        glGenVertexArrays(1, &vao);  
        glBindVertexArray(vao);  
    }  
    void Draw() {  
        glBindVertexArray(vao);  
        glDrawArrays(GL_TRIANGLES, 0, nVtx);  
    }  
};  
  
struct VertexData {  
    vec3 position, normal;  
    float u, v;  
};  
  
struct ParamSurface : Geometry {  
    virtual VertexData GenVertexData(float u, float v) = 0;  
    void Create(int N, int M);  
};
```

Tessellation is done on the CPU with a C++ program. The tessellated triangle mesh is copied to the GPU and assigned to a vao (vertex array object).

The general base class of triangle meshes is the Geometry that stores the vao and the number of vertices. In its constructor, the vao is generated and is bound, i.e. is made active. When a Geometry is drawn, the vao is bound again since other vaos may become active between the construction of this one and its drawing. Then the vao is drawn stating that its vertices define a set of triangles where the first three vertices define the first triangle, the fourth, fifth, sixth the second triangle, etc. Note that this is not the most efficient way of encoding a triangle mesh since a vertex is stored as many times as many triangles it participates, but this is the simplest one.

A parametric surface, called ParamSurface is a special type of Geometry, where Create function creates the vao. During creation we need the equation of the parametric surface which is different for different types (e.g. sphere, torus, flag, etc.). So here, we declare GenVertexData as a pure virtual function, which returns the position, normal and the parameter pair for a given parameter pair.

Parametrikus felület GPU-nak

```
void ParamSurface::Create(int N, int M) {
    nVtx = N * M * 6;
    unsigned int vbo;
    glGenBuffers(1, &vbo); glBindBuffer(GL_ARRAY_BUFFER, vbo);
    VertexData *vtxData = new VertexData[nVtx], *pVtx = vtxData;
    for (int i = 0; i < N; i++) for (int j = 0; j < M; j++) {
        *pVtx++ = GenVertexData((float)i / N, (float)j / M);
        *pVtx++ = GenVertexData((float)(i + 1) / N, (float)j / M);
        *pVtx++ = GenVertexData((float)i / N, (float)(j + 1) / M);
        *pVtx++ = GenVertexData((float)(i + 1) / N, (float)j / M);
        *pVtx++ = GenVertexData((float)(i + 1) / N, (float)(j + 1) / M);
        *pVtx++ = GenVertexData((float)i / N, (float)(j + 1) / M);
    }

    int stride = sizeof(VertexData), sVec3 = sizeof(vec3);
    glBufferData(GL_ARRAY_BUFFER, nVtx * stride, vtxData, GL_STATIC_DRAW);

    glEnableVertexAttribArray(0); // AttribArray 0 = POSITION
    glEnableVertexAttribArray(1); // AttribArray 1 = NORMAL
    glEnableVertexAttribArray(2); // AttribArray 2 = UV
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, stride, (void*)0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, stride, (void*)sVec3);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, stride, (void*)(2*sVec3));
}
```

Recall that we have activated the vao. Now its corresponding data stored in a single vbo (vertex buffer object) are generated. If the parameter space is decomposed to N columns and M rows, then we have $N * M$ quads. A single quad is composed by two triangles, each having three vertices. Thus the number of vertices is $N * M * 6$.

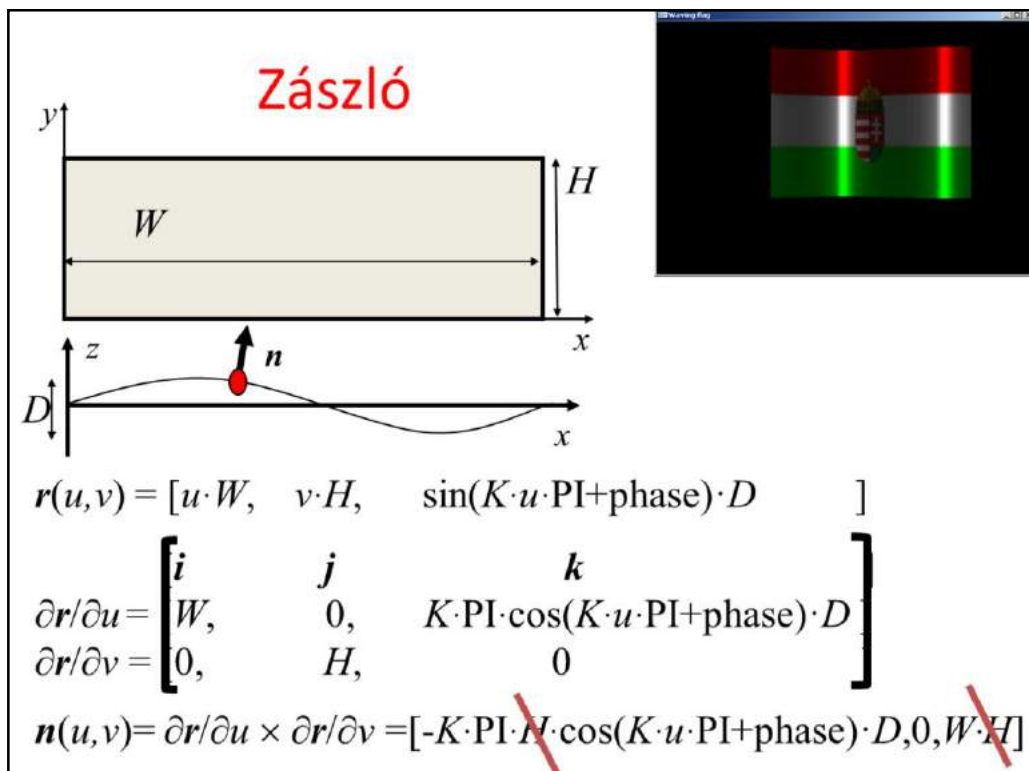
The vbo is generated and bound, then it is filled up by calling the `GenVertexData` virtual function of the vertices of the triangles.

In the GPU, vertex shader input register #0 will get the position (3 floats), #1 the normal (3 floats), and #2 the u,v parameter pair (2 floats).

Gömb

```
class Sphere : public ParamSurface {  
    vec3 center;  
    float radius;  
public:  
    Sphere(vec3 c, float r) : center(c), radius(r) {  
        Create(16, 8); // tessellation level  
    }  
  
    VertexData GenVertexData(float u, float v) {  
        VertexData vd;  
        vd.normal = vec3(cos(u*2*M_PI) * sin(v*M_PI),  
                        sin(u*2*M_PI) * sin(v*M_PI),  
                        cos(v*M_PI));  
        vd.position = vd.normal * radius + center;  
        vd.u = u; vd.v = v;  
        return vd;  
    }  
};
```

The construction of a parametric surface is general, the only specific thing is the GenVertexData. So, if we derive a Sphere class from ParamSurface, this virtual function should be implemented according to the equations of the sphere.



Another example is the waving flag, which is a rectangle that is modulated by a sine wave. As we stated, the normal vector is the cross product of the partial derivatives.

Zászló

```
class Flag : public ParamSurface {
    float W, H, D, K, phase;
public:
    Flag(float w, float h, float d, float k, float p)
        : W(w), H(h), D(d), K(k), phase(p) {
        Create(60, 40); // tessellation level
    }

    VertexData GenVertexData(float u, float v) {
        VertexData vd;
        float angle = u * K * M_PI + phase;
        vd.position = vec3(u * W, v * H, sin(angle)*D);
        vd.normal = vec3(-K * M_PI * cos(angle) * D, 0, W);
        vd.u = u; vd.v = v;
    }
};
```


Transzformációk

Modellezési transzformáció:

$$\begin{aligned} [\mathbf{r}, 1] \mathbf{T}_{\text{Model}} &= [\mathbf{r}_{\text{world}}, 1] \\ [\mathbf{N}, 0] (\mathbf{T}_{\text{Model}}^{-1})^T &= [\mathbf{N}_{\text{world}}, d] \end{aligned}$$

Kamera transzformáció:

$$[\mathbf{r}_{\text{world}}, 1] \mathbf{T}_{\text{View}} = [\mathbf{r}_{\text{camera}}, 1]$$

Perspektív transzformáció:

$$[\mathbf{r}_{\text{camera}}, 1] \mathbf{T}_{\text{Persp}} = [\mathbf{r}_{\text{screen}}, h, h]$$

MVP transzformáció: $\mathbf{T}_{\text{Model}} \mathbf{T}_{\text{View}} \mathbf{T}_{\text{Persp}} = \mathbf{T}_{\text{MVP}}$

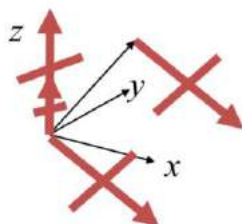
Using the modeling transformation, the object is mapped to world coordinates. Recall that the transformation of the shading normals requires the application of the inverse-transpose of the 4x4 matrix, or if the normal is a part of a column vector, we should multiply it with the inverse.

From world coordinates, we go to camera space, where the camera is at the origin and looks at the $-z$ direction. The transformation between world and camera coordinates is a translation and a rotation.

After camera transformation, the next step is perspective transformation, which distorts the objects in a way that the original perspective projection will be equivalent to the parallel projection of the distorted objects. This is a not-affine transformation, so it will not preserve the value of the fourth homogeneous coordinates (which has been 1 so far).

The three transformation matrices (model, camera, perspective) can be concatenated, so a single composite transformation matrix takes us from the reference state directly to normalized screen space.

Modellezési transzformáció



1. skálázás: s_x, s_y, s_z
2. orientáció: w_x, w_y, w_z, α
3. pozíció: p_x, p_y, p_z

$$\mathbf{T}_M = \begin{bmatrix} s_x & & & \\ & s_y & & \\ & & s_z & \\ & & & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ p_x & p_y & p_z & 1 \end{bmatrix}$$

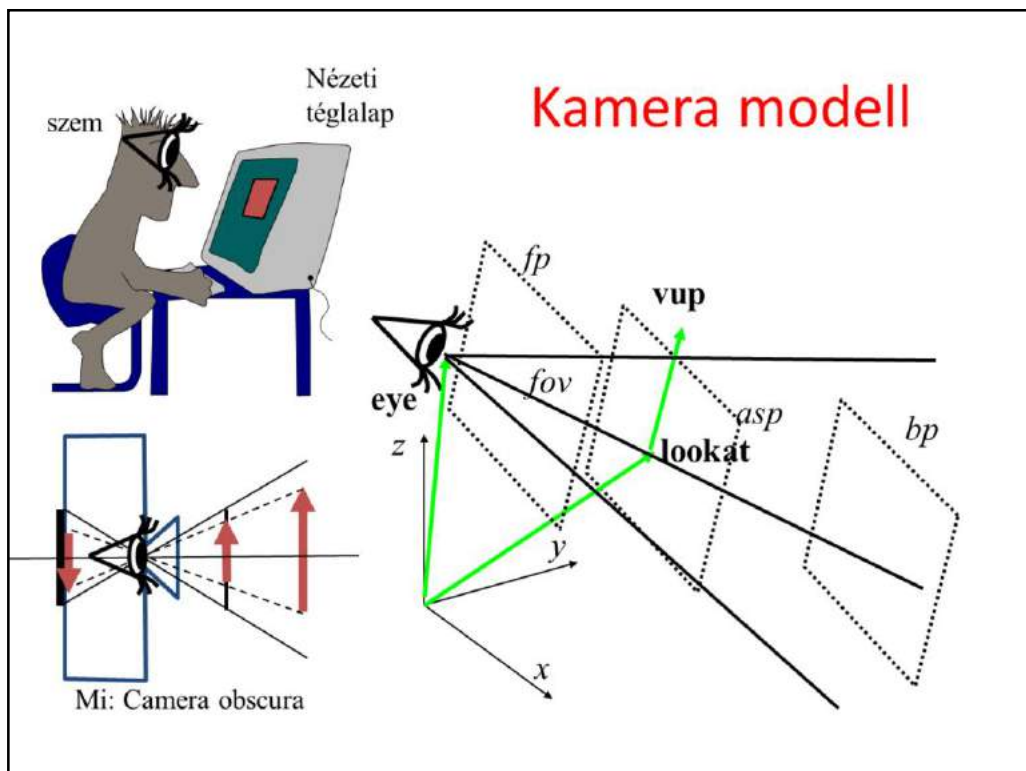
\uparrow
 $\mathbf{r}' = \mathbf{r} \cos(\alpha) + \mathbf{w}^0 (\mathbf{r} \cdot \mathbf{w}^0) (1 - \cos(\alpha)) + \mathbf{w}^0 \times \mathbf{r} \sin(\alpha)$

Modeling transformation sets up the object in the virtual world. This means scaling to set its size, then rotation to set its orientation, and finally translation to place it at its position. All three transformations are affine and can be given as a homogeneous transformation matrix. Concatenating the matrices, we obtain a single modeling transformation matrix, which maps the object from its reference state to its actual state.

4x4-es mátrix

```
struct mat4 {  
    float m[4][4];  
    mat4(float m00,..., float m33) { ... }  
    mat4 operator*(const mat4& right);  
  
    void SetUniform(unsigned shaderProg, char * name) {  
        int loc = glGetUniformLocation(shaderProg, name);  
        glUniformMatrix4fv(loc, 1, GL_TRUE, &m[0][0]);  
    }  
};  
  
mat4 Translate(float tx, float ty, float tz) {  
    return mat4(1,    0,    0,    0,  
                0,    1,    0,    0,  
                0,    0,    1,    0,  
                tx,   ty,   tz,    1);  
}  
  
mat4 Rotate(float angle, float wx, float wy, float wz) {...}  
mat4 Scale(sx, sy, sz) {...}
```

On the CPU side, we need a mat4 class to handle matrices (on the GPU side we program in GLSL where this is already a built in type). We need to construct the 4x4 matrix and implement the matrix multiplication for concatenation. As we already have formulae for translation, rotation, scaling, the corresponding matrices can easily be given.



The definition of the camera transformation depends on the parameters of the camera. In computer graphics the camera is the eye position that represents the user in the virtual world and a window rectangle that represents the screen.

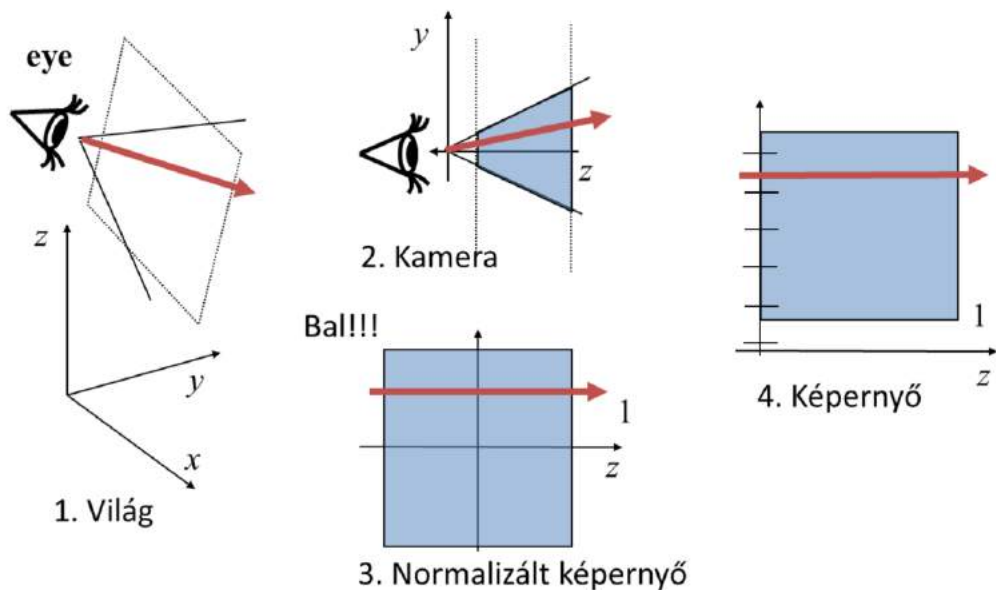
It is often more intuitive to think of a virtual camera as being similar to a real camera. A real camera has a focus point or pin-hole, where the lens is, and a planar film where the image is created in a bottom-up position. In fact, this model is equivalent to the model of the user's eye and the screen, just the user's eye should be imagined in the lens position and the film mirrored onto the lens as the screen.

So in both cases, we need to define a virtual eye position and a rectangle in 3D. The **eye** position is a vector in world coordinates. The position of the window is defined by the location of its center, which is called **lookat** point or view reference point. We assume that the **main viewing direction** that is between the eye and the lookat positions is perpendicular to the window. To find the vertical direction of the window, a **view up (vup)** vector needs to be specified. If it is not exactly perpendicular to the viewing direction, then only its perpendicular component is used.

The vectors defined so far specify the window plane and orientation, but not the size of the rectangle. To set the vertical size, the **field of view angle (fov)** is given. For the horizontal size, the **aspect** ratio of the vertical and horizontal window edge sizes should be specified.

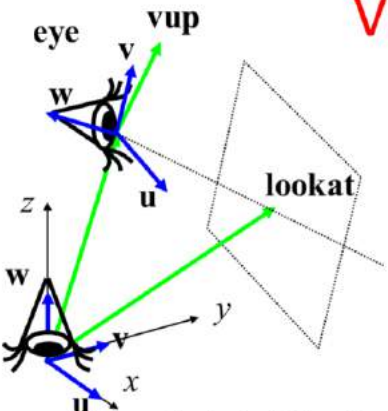
Objects being very close to the eye are not visible and leads to numerical inaccuracy. So we also set up a front clipping plane that is parallel to the window and ignore everything that is behind this plane. Similarly, objects that are very far are not visible and may lead to numerical representation problems. So we also introduce a back clipping plane to limit the space for the camera.

Világból a képernyőre



Our goal is to transform the scene from world coordinates to screen coordinates, where visibility determination and projection are trivial. This transformation is built as a sequence of elementary transformations because of pedagogical reasons, but we shall execute all transformations at once, as a single matrix-vector multiplication.

View transzformáció



$$\mathbf{w} = (\text{eye} - \text{lookat}) / |\text{eye} - \text{lookat}|$$

$$\mathbf{u} = (\mathbf{vup} \times \mathbf{w}) / |\mathbf{w} \times \mathbf{vup}|$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

$$[x', y', z', 1] = [x, y, z, 1]$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\text{eye}_x & -\text{eye}_y & -\text{eye}_z & 1 \end{pmatrix} \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

$$\begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

First we apply a transformation for the scene, including objects and the camera, that moves the camera to the origin and rotates it to make the main viewing be axis $-z$ and the camera's vertical direction be axis y . To find such transformation, we assign an orthonormal basis to the camera so that its first basis vector, u , is the camera's horizontal direction, the second, v , is the vertical direction, and the third, w , is the opposite of the main viewing direction (we reverse the main viewing direction to maintain the right handedness of the system).

Vector w can be obtained from the main viewing direction by a simple normalization. The application of normalization to get v from vup is also tempting, but simple normalization would not guarantee that basis vector v is orthogonal to basis vector w . So instead of directly computing v from vup , first we obtain u as a vector that is orthogonal to both w and vup . Then, v is computed indirectly through w and u to make it orthogonal to both of them.

The transformation we are looking for is a translation then a rotation. The translation moves the eye position to the origin. The translation has a simple homogeneous linear transformation matrix. Having applied this translation, the orientation should be changed to align vector w with axis z , vector v

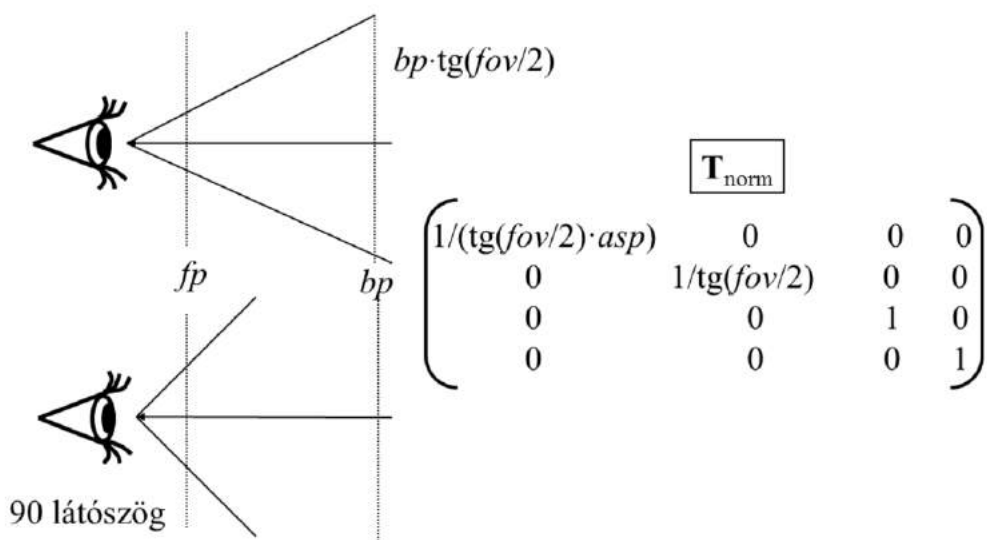
with axis y, and vector u with axis x. Although this transformation is non-trivial, its inverse that aligns axis x with u, axis y with v, and axis z with w is straightforward.

Its basic idea is that the rows of an affine transformation (fourth column is $[0,0,0,1]^T$) are the images of the three basis vectors and the origin respectively.

So, the transformation of x,y,z axes to u,v,w is the matrix that contains u,v,w as the row vectors of the 3x3 minor matrix of the 4x4 transformation matrix.

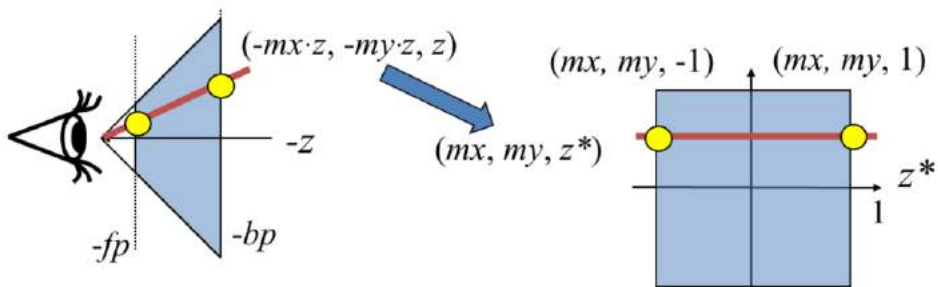
As we need the inverse transformation, this matrix needs to be inverted. Such matrices – called orthonormal matrices – are easy to invert, since their transpose is their inverse.

Látószög normalizálás



In camera space, the camera is in the origin and the main viewing direction is axis $-z$. The normalization step distorts the space to make the viewing angle be equal to 90 degrees. This is a scaling along axes y and x . Considering scaling along axis y , before the transformation the top of the viewing pyramid has y coordinate $bp \cdot \text{tg}(fov/2)$, and we expect it to be bp . So, y coordinates must be divided by $\text{tg}(fov/2)$. Similarly, x coordinates must be divided by $\text{tg}(fov/2) \cdot asp$.

Normalizálás utáni perspektív transzformáció



$$(-mx \cdot z, -my \cdot z, z) \Rightarrow (mx, my, z^*)$$

$$[-mx \cdot z, -my \cdot z, z, 1] \Rightarrow [mx, my, z^*, 1] \sim [-mx \cdot z, -my \cdot z, -z \cdot z^*, -z]$$

Perspective transformation makes the rays meeting in the origin be parallel with axis z , i.e. meeting in infinity (an ideal point at the “end” of axis z).

Additionally, we expect the viewing frustum to be mapped to an axis aligned cube of corner points $(-1, -1, -1)$ and $(1, 1, 1)$. There are infinitely many solutions for this problem. However, only that solution is acceptable for us which maps lines to lines (and consequently triangles to triangles) since when objects are transformed, we wish to execute the matrix vector multiplication only for the vertices of the triangle mesh and not for every single point (there are infinite of them). Homogeneous linear transformations are known to map lines to lines, so if we can find a homogeneous linear transformation that does the job, we are done. To find the transformation matrix, we consider how a ray should be mapped. A ray can be defined by a line of explicit equation

$$x = -mx \cdot z, \quad y = -my \cdot z$$

where coordinate z is a free parameter (mx and my are the slopes of the line). In normalized camera space the slopes are between -1 and 1 . We expect this line to be parallel with axis z^* after the transformation (z^* is the transformed z to resolve ambiguity), so its x and y coordinates should be independent of z .

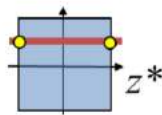
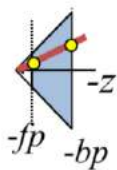
As x, y must also be in $[-1, 1]$, the transformed line is

$$x^* = mx, \quad y^* = my,$$

and z^* is a free parameter.

The mapping from $(x,y,z) = (-mx \cdot z, -my \cdot z, z)$ to $(x^*,y^*,z^*)=(mx, my, z^*)$ cannot be linear in Cartesian coordinates, but is linear (we hope) in homogeneous coordinates. So, we are looking for a linear mapping from $[x,y,z,I] = [-mx \cdot z, -my \cdot z, z, 1]$ to $[x^*,y^*,z^*,I] = [mx, my, z^*, 1]$. To make it simpler, we can exploit the homogeneous property, i.e. the represented point remains the same if all coordinates are multiplied by a non-zero scalar. Let this scalar be $-z$. So our goal is to find a linear mapping from $[x,y,z,I] = [-mx \cdot z, -my \cdot z, z, 1]$ to $[x^*,y^*,z^*,I] \sim [-mx \cdot z, -my \cdot z, -z \cdot z^*, -z]$.

Perspektív transzformáció



T_{persp}

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & -1 \\ 0 & 0 & \beta & 0 \end{pmatrix}$$

$$[-mx \cdot z, -my \cdot z, z, 1] \Rightarrow [-mx \cdot z, -my \cdot z, -z \cdot z^*, -z]$$

$$-z \cdot z^* = \alpha z + \beta$$

\Rightarrow

$$z^* = -\alpha - \beta/z$$

$$fp \cdot (-1) = \alpha(-fp) + \beta$$

$$bp \cdot (1) = \alpha(-bp) + \beta$$

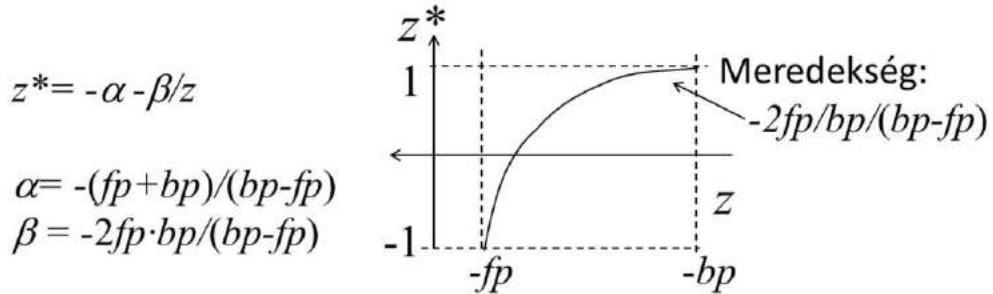
$$\alpha = -(fp + bp)/(bp - fp)$$

$$\beta = -2fp \cdot bp/(bp - fp)$$

A homogeneous linear transformation is a 4x4 matrix, i.e. sixteen scalars, which need to be found. The requirement is that for arbitrary mx , my , z , when multiplying with $[-mx \cdot z, -my \cdot z, z, 1]$, the result must be $[-mx \cdot z, -my \cdot z, -z \cdot z^*, -z]$. The first two elements are kept for arbitrary mx , my , z , which is possible if the first two columns of the matrix are $[1, 0, 0, 0]$ and $[0, 1, 0, 0]$. As mx and my do not affect the third and the fourth elements in the result, the corresponding matrix element must be zero. The fourth element of the result is $-z$, so the fourth column is $[0, 0, -1, 0]$. We are left with only two unknown parameters alpha and beta. They can be found by considering the requirements that the entry point that is the intersection of the ray and the front clipping plane is mapped to $z^* = -1$ and the exit point to $z^* = 1$.

Z-fighting

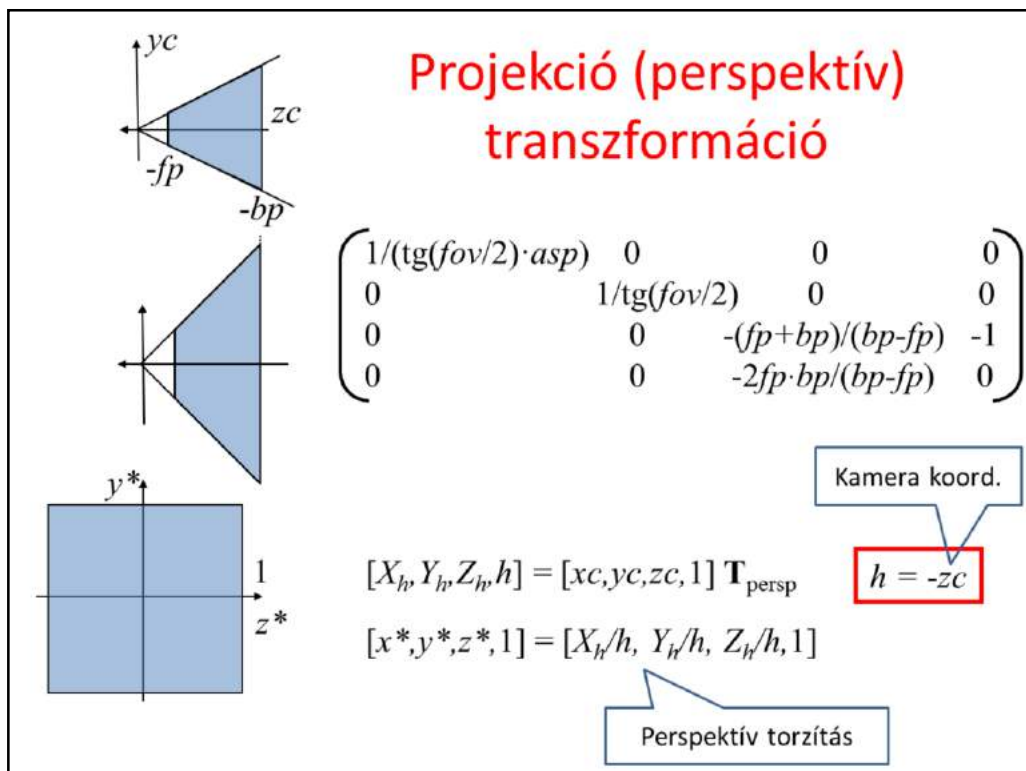
fp/bp nem lehet kicsi!



Note that the expression of z^* as a function of z is not a linear (which we knew from the very beginning), but a reciprocal function. Recall that this $1/x$ function changes quickly where x is small but will be close to constant where x is large. Value z^* is used to determine visibility. To determine visibility robustly, the difference of z^* for two points must be large enough (otherwise comparison fails due to numerical inaccuracies and number representation limitations). This is not the case if β is small and z is large, e.g. when $z \approx -bp$ approximately, i.e. when $\beta/z \approx 2fp/(bp-fp)$. If bp is much greater than fp , then

$\beta/z \approx 2fp/(bp-fp) \approx 2fp/bp$ is very small, prohibiting to robustly distinguish two occluding surfaces.

Never specify fp and bp such that fp/bp is too small (e.g. less than 0.01). If you do, occluded surfaces will randomly show up because of numerical inaccuracy. This phenomenon is called **z-fighting**.



Normalization and perspective transformation are usually combined and the composed transformation is set directly.

It is worth noting that this transformation sets the fourth homogeneous coordinate to the camera coordinate depth value. It is also notable that this transformation maps the eye ($[0,0,0,1]$ in homogeneous coordinates) to the ideal point of axis z , i.e. to $[0,0, -2fp \cdot bp/(bp-fp), 0] \sim [0,0, 1, 0]$.

Camera osztály

```
class Camera {  
    vec3 wEye, wLookat, wVup;  
    float fov, asp, fp, bp;  
public:  
    mat4 V() { // view matrix  
        vec3 w = (wEye - wLookat).normalize();  
        vec3 u = cross(wVup, w).normalize();  
        vec3 v = cross(w, u);  
        return Translate(-wEye.x, -wEye.y, -wEye.z) *  
            mat4( u.x,  v.x,  w.x,  0.0f,  
                  u.y,  v.y,  w.y,  0.0f,  
                  u.z,  v.z,  w.z,  0.0f,  
                  0.0f, 0.0f, 0.0f, 1.0f );  
    }  
    mat4 P() { // projection matrix  
        float sy = 1/tan(fov/2);  
        return mat4(sy/asp, 0.0f, 0.0f, 0.0f,  
                    0.0f, sy, 0.0f, 0.0f,  
                    0.0f, 0.0f, -(fp+bp)/(bp - fp), -1.0f,  
                    0.0f, 0.0f, -2*fp*bp/(bp - fp), 0.0f);  
    }  
};
```

Our 3D Camera class stores parameters need to define a camera, and implement two transformation functions. Transformation V is the view transformation that takes a point from world space to camera space, and transformation P is the projection or perspective transformation that takes a point from camera space to normalized device space.

```

const char *vertexSource = R"(
uniform mat4 M, Minv, MVP;
in vec3 vtxPos;
in vec3 vtxNorm;
out vec4 color;

void main() {
    gl_Position = vec4(vtxPos, 1) * MVP;
    vec4 wPos = vec4(vtxPos, 1) * M;
    vec4 wNormal = Minv * vec4(vtxNorm, 0);
    color = Illumination(wPos, wNormal);
})";

```

Transzformációk a GPU-n

```

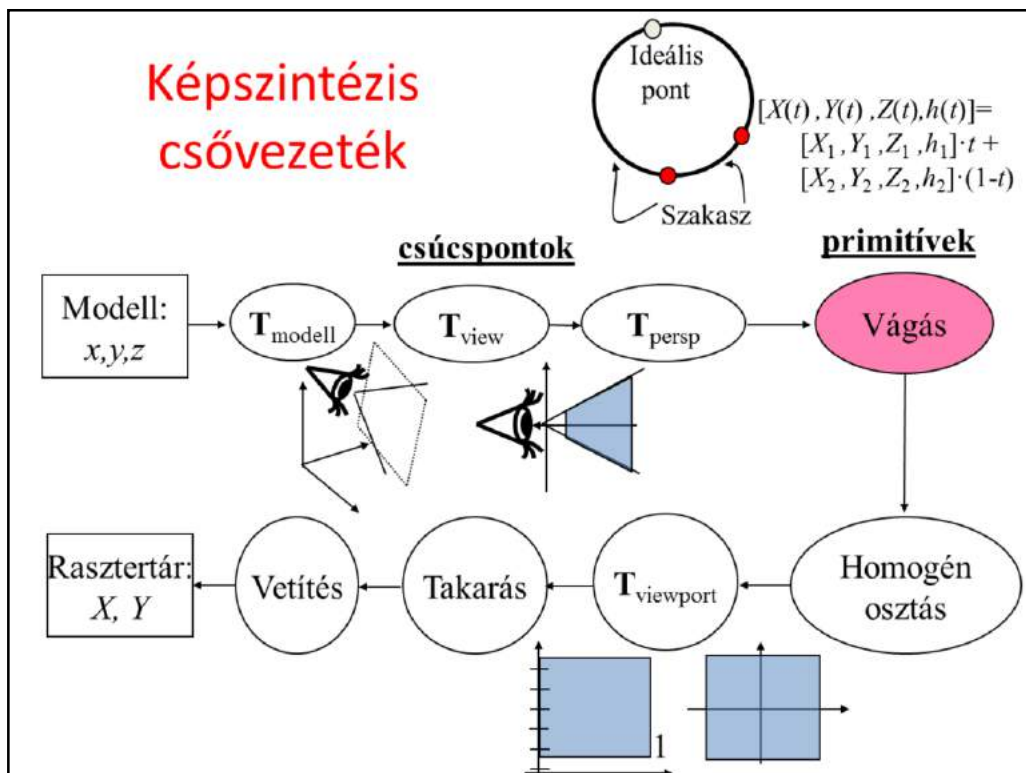
void Draw() {
    mat4 M = Scale(scale.x, scale.y, scale.z) *
        Rotate(rotAng, rotAxis.x, rotAxis.y, rotAxis.z) *
        Translate(pos.x, pos.y, pos.z);
    mat4 Minv = Translate(-pos.x, -pos.y, -pos.z) *
        Rotate(-rotAngle, rotAxis.x, rotAxis.y, rotAxis.z) *
        Scale(1/scale.x, 1/scale.y, 1/scale.z);
    mat4 MVP = M * camera.V() * camera.P();

    M.SetUniform(shaderProg, "M");
    Minv.SetUniform(shaderProg, "Minv");
    MVP.SetUniform(shaderProg, "MVP");

    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, nVtx);
}

```

Transformations are set on the CPU side but are applied to points by the GPU. Transformations are uniform variables of the vertex shader, which are set from the CPU program and used by the vertex shader. We always need the MVP, i.e. model-view-projection transformation that is the concatenation of the modeling transformation and the two phases of the camera transformation. If illumination is also computed in world coordinate system, object should also be transformed there. For points, we need the modeling transformation M, for normal vectors its inverse. Note that normal vectors stand on the right side of the matrix, i.e. they form column vectors.



Clipping must be done before the homogeneous division, preferably in a coordinate system where it is the simplest. The optimal choice is the normalized device coordinates where the view frustum is a cube. However, clipping must be done before the homogeneous division. This might be surprising but becomes clear if we consider the topology of a projective line and the interpretation of a line segment. A projective line is like a circle, the ideal point attaches the two “endpoints”. As on a circle, two points do not unambiguously identify an arc (there are two possible arcs), on a projective line, two points may define two complements “line segments” (one of them looks as two half lines). To resolve this ambiguity, we specify endpoints with positive fourth homogeneous coordinates, and define the line segment as a convex combination of two endpoints. If fourth coordinates h are positive for both end points, then the interpolated h cannot be zero, so this line segment does not contain the ideal point (it is a “normal line segment”). Affine transformations usually do not alter the fourth homogeneous coordinate, so “normal line segments” will remain normal.

Perspective transformation may map a line segment to a line segment that contains the ideal point, which is clearly indicated in the different signs of the fourth homogeneous coordinates of the two endpoints. So, if the two h coordinates have the same sign, then the line segment is normal. If the two h

coordinates have different sign, then the line segment is wrapped around (it contains the ideal point so we would call it two half lines and not a line segment).

One way to solve this problem is to clip away everything that is behind the front clipping plane. This clipping must be executed before the homogeneous division since during this operation we lose the information regarding the sign of the fourth homogeneous coordinate.

Vágás homogén koordinátákban

Cél: $-1 < X = X_h/h < 1$
 $-1 < Y = Y_h/h < 1$
 $-1 < Z = Z_h/h < 1$
 Vegyük hozzá: $h > 0$ (mert $h = -zc$)



$$\begin{aligned} -h < X_h < h \\ -h < Y_h < h \\ -h < Z_h < h \end{aligned}$$

	$h = X_h$
Kívül ●	Belül ●
$[3, 0, 0, 2]$	$[2, 0, 0, 3]$
$h = 2 < X_h = 3$	$h = 3 > X_h = 2$

In Cartesian coordinates the limits of the viewing frustum are -1 and 1 in all three coordinates. As the clipping operation will be executed in homogeneous coordinates, we should find the equation of the viewing frustum in homogeneous coordinates. Substituting Cartesian coordinate X by X_h/h , etc. these equations can be obtained. To make it simpler we wish to multiply both sides by h . However, an inequality cannot be multiplied by an unknown variable since should this variable be negative, the relations must be negated. So we add requirement $h > 0$ which means that the volume must be in front of the eye. Note that this fact is due to the specific selection of the perspective transformation matrix. If h is surely positive, we can safely multiply the inequalities by h .

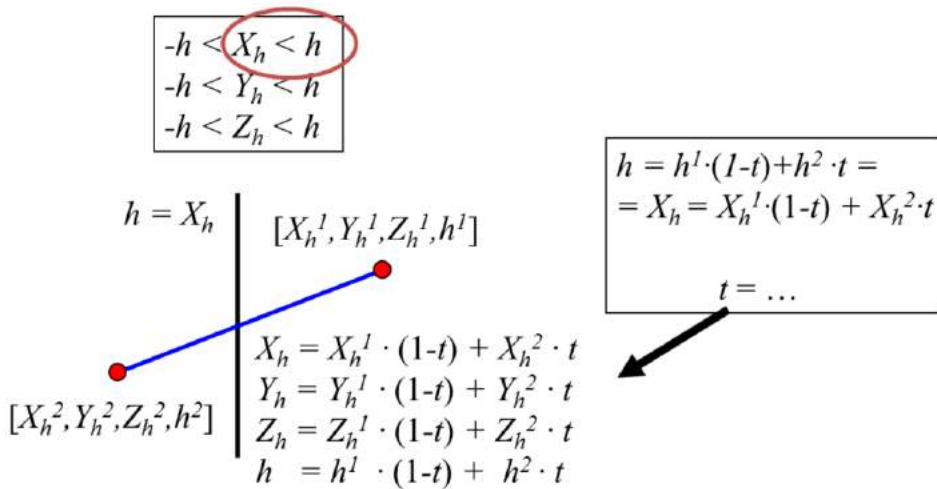
The collection of six inequalities defines a cube. A point is inside the cube if all inequalities are met.

To make trivial things complicated, we can also imagine that instead of clipping on a cube, we clip onto 6 half-spaces one after the other. The intersection of these half-spaces is the cubical view frustum. Each half space is associated with a single inequality and the border plane of the half-space is defined by the equation where $<$ is replaced by $=$. The advantage of this

approach is that when more complex objects (like lines or polygons) are clipped, the fact that the vertices are outside the cube provides no information whether the line or polygon overlaps with the clipping region.

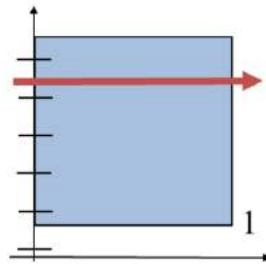
However, if both endpoints of a line are outside of the half-space, then the line segment is totally out. If both endpoints are in, then the line segment is totally in. When one is out while the other is in, the intersection of the boundary plane and the line segment should be computed, and the outer point should be replaced by the intersection.

Szakasz/poligon vágás



So line clipping is executed 6 times for the half-spaces. We consider here just one half-space of inequality $X_h < h$, whose boundary is the plane of equation $X_h = h$. The half-space inequality is evaluated for both endpoints. If both of them are in, the line segment is completely preserved. If both of them are out, the line segment is completely ignored. If one is in and one is out, we consider the equation of the boundary plane ($X_h = h$), and the equation of the line segment (a line segment is the convex combination of its two endpoints), and solve this for unknown combination parameter t . Substituting the solution back to the equation of the line segment, we get the homogeneous coordinates of the intersection point. This intersection point replaces the endpoint that has been found outside.

Takarás



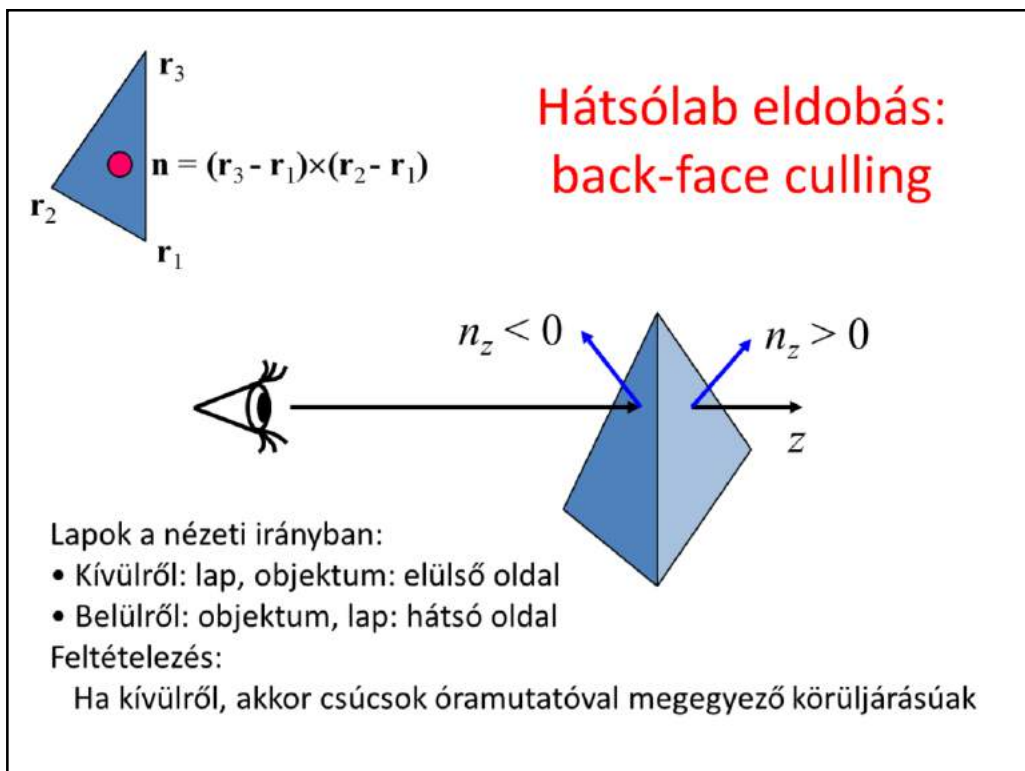
- Képernyő koordinátarendszerben
 - vetítősugarak a z tengellyel párhuzamosak!
- Objektumtér algoritmusok (folytonos):
 - láthatóság számítás nem függ a felbontástól
- Képtér algoritmusok (diszkrét):
 - mi látszik egy pixelben
 - Sugárkövetés ilyen volt!

What we need is solid rendering which renders filled polygons and also considers occlusions. Many polygons may be mapped onto the same pixel. We should find that polygon (and its color), which occludes the others, i.e. which are the closest to the eye position. This calculation is called **visibility determination**.

We have to emphasize that visibility is determined in the screen coordinate system where rays are parallel with axis z and the x , y coordinates are the physical pixel coordinates. We made all the complicated looking homogeneous transformations just for this, to calculate visibility and projection in a coordinate system where these operations are simple.

There are many visibility algorithms (from which we shall discuss only 1.5 :-). The literature classifies them as object space (aka object precision or continuous) and screen space (aka image precision or discrete). Object space visibility algorithms find the visible portions of triangles in the form of (possibly clipped and subdivided) triangles, independently of the resolution of the image. Screen space algorithms exploit the fact that the image has finite precision and determine the visibility just at finite number of points, usually through the centers of the pixels. Recall that ray tracing belongs to

the category of image precision algorithms.

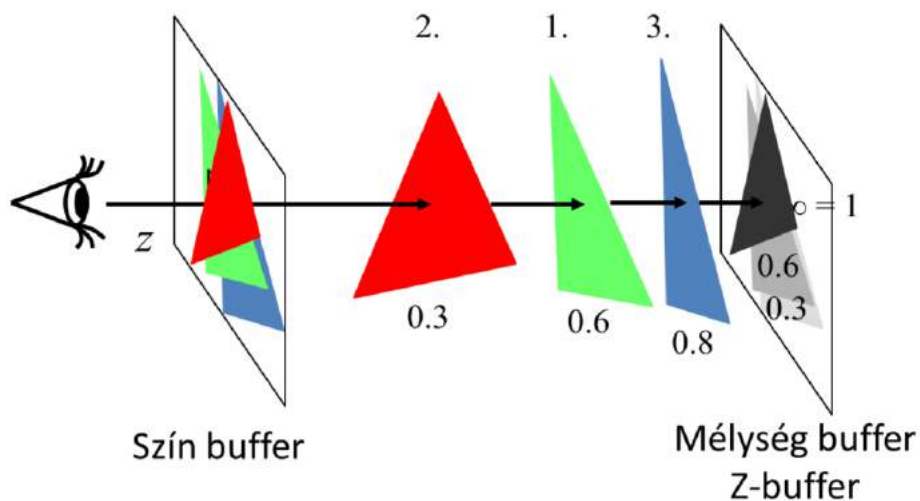


First an object space method is presented, which gives only partial solution.

Assume that the triangle mesh is the boundary of a valid 3D object! In this case, we can see only one side or face of each boundary polygon. The potentially visible face is called **front-face**, the other face or side where the polygon is "glued" to its object is called **back-face**. A back-face is never visible since the object itself always occludes it. So, when a polygon shows its back-face to the camera, we can simply ignore it since we know that there are other polygons of the same object, which will occlude it. To help the determination whether a face is front or back, we use a coding scheme that must be set during the modeling of the geometry (or during tessellation). In theory, triangle or polygon vertices can be specified either in clock-wise or counter-clock-wise order. Let us use this ordering to indicate which face is glued to the object. For example, we can state that vertices must be specified in clock-wise order when the object is seen from outside, i.e. when we see the front-face of the polygon. Computing a geometric normal with $\mathbf{n} = (\mathbf{r}_3 - \mathbf{r}_1) \times (\mathbf{r}_2 - \mathbf{r}_1)$, the clock-wise order means that \mathbf{n} points towards the viewer. As in screen coordinates the viewing rays are parallel with axis z , it is easy to decide whether a vector points towards the viewer, simply the z coordinate of \mathbf{n} must be negative.

With back-face culling, we can get rid of the surely non-visible polygons, which always helps. However, front-faces may also occlude each other, which should be resolved by another algorithm. Still, back-face culling is worth applying, since it roughly halves the number of the potentially visible polygons.

Z-buffer algoritmus



And now let us meet the far most popular visibility algorithm of incremental image synthesis.

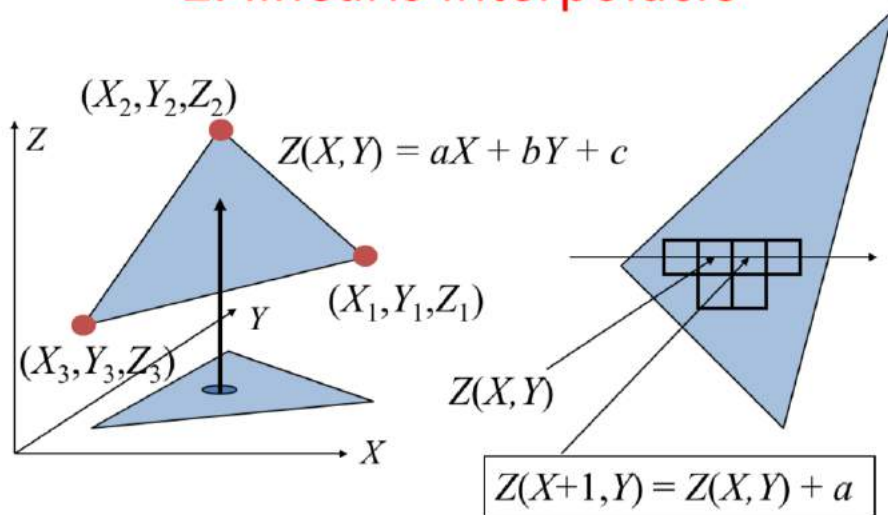
Ray tracing considers pixels one by one and for a single pixel, it checks each object for intersection while always keeping the first intersection found so far.

In incremental rendering, we would like to exploit the similarities of different points on an object (on a triangle), so instead of processing every pixel independently, we take an object centric approach. Let us swap the order of the for loops of ray tracing and take triangles one by one and for a single triangle, find visibility rays intersecting it, always keeping the first intersection in every pixel.

As all pixels are processed simultaneously, instead of a single minimum ray parameter, we need to maintain as many ray parameters as pixels the viewport has. In screen coordinates, the ray parameter, i.e. the distance of the intersection, is represented by the z coordinate of the intersection point. Therefore, the array storing the ray parameters of the intersections is called the z-buffer or depth buffer.

The z-buffer algorithm initializes the depth buffer to the largest possible value, since we are looking for a minimum. In screen coordinates, the maximum of z coordinates is 1 (the back clipping plane is here). The algorithm takes triangles one-by-one in an arbitrary order. A given triangle is projected onto the screen and rasterized, i.e. we visit those pixels that are inside the triangle's projection (this step is equivalent to identifying those rays which intersect the given object). At a particular pixel, the z coordinate of the point of the triangle visible in this pixel is computed (this is the same as ray triangle intersection), and the triangle's z coordinate is compared to the z value stored in the z buffer at this pixel. If the triangle's z coordinate is smaller, then the triangle point is closer to the front clipping plane at this pixel than any of the triangles processed so far, so the triangle's color is written into the frame buffer and its z coordinate to the depth buffer. Thus, the frame buffer and the depth buffer always store the color and depth of that triangles which are visible from the triangles processed so far.

Z: lineáris interpoláció

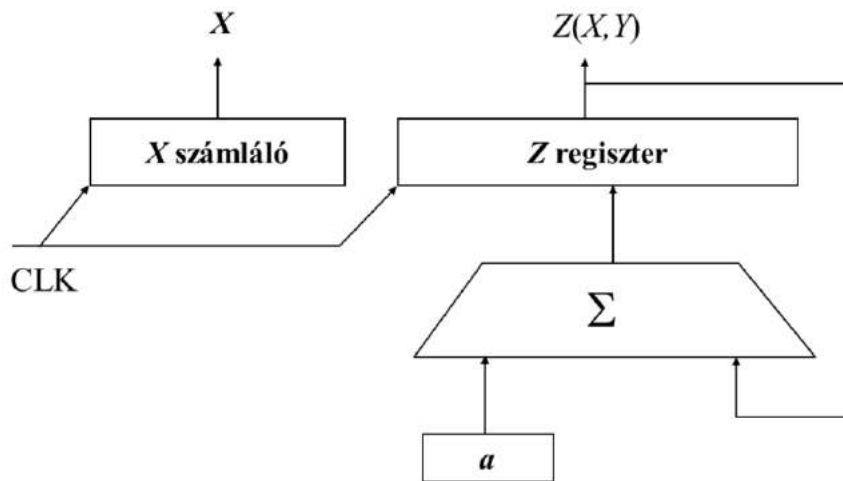


Projected triangles are rasterized by visiting scan lines (i.e. rows of pixels) and determining the intersection of this horizontal line and the edges of the triangle. Pixels of X coordinates that are in between the X coordinates of the two edge intersections are those where the triangle is potentially visible.

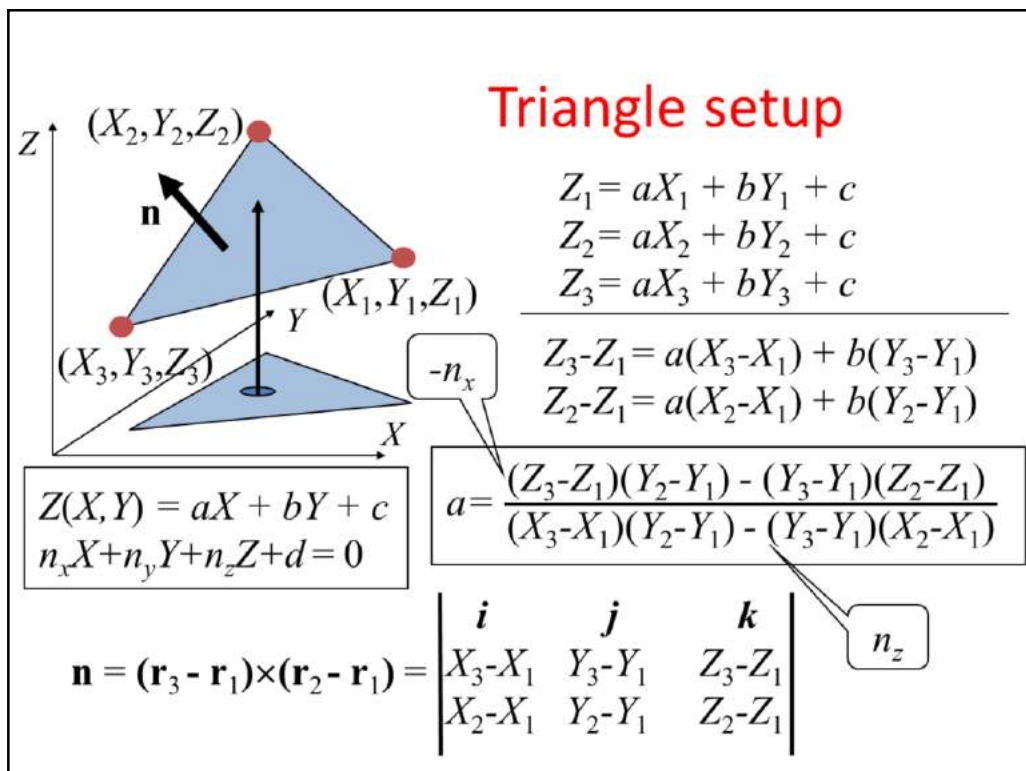
Pixels in this row are visited from left to right, and the z coordinate of the triangle point whose projection is this pixel center is calculated. This z coordinate is compared to the value of the depth buffer.

The first question is how the Z coordinates of the triangle points are computed for every pixel of coordinates X and Y. The triangle is on a plane so X, Y, Z satisfy the plane equation, i.e. a linear equation, so Z will also be a linear function of pixel coordinates X and Y. The evaluation of this linear function requires two multiplications and two additions, which are too much if we have just a few nanoseconds for this operation. To speed up the process, **the incremental principle** can be exploited. As we visit pixels in the order of coordinate X, when a pixel of coordinates X+1,Y is processed, we have the solution, Z, for the previous pixel. It is easier to compute only the increment than obtaining Z from scratch. Indeed, the difference between $Z(X+1, Y)$ and $Z(X, Y)$ is constant so the new depth can be obtained from the previous one as a single addition with a constant.

Z-interpolációs hardver



Such an incremental algorithm is easy to be implemented directly in hardware. A counter increments its value to generate coordinate X for every clock cycle. A register that stores the actual Z coordinate in fixed point, non-integer format (note that increment a is usually not an integer). This register is updated with the sum of its previous value and a for every clock cycle.



The final problem is how Z increment a is calculated. One way of determining it is to satisfy the interpolation constraints at the three vertices.

The other way is based on the recognition that we work with the plane equation where X,Y,Z coordinates are multiplied by the coordinates of the plane's normal. The normal vector, in turn, can be calculated as a cross product of the edge vectors.

Note that the coordinates of the screen space normal are needed not only here, but also in back-face culling. So, triangle set up is also responsible for the front-face, back-face classification.

Takarás OpenGL-ben

```
int main(int argc, char * argv[]) {  
    ...  
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |  
                        GLUT_DEPTH);  
    glEnable(GL_DEPTH_TEST); // z-buffer is on  
    glDisable(GL_CULL_FACE); // backface culling is off  
    ...  
}  
  
void onDisplay() {  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
  
    rajzolás...  
  
    glutSwapBuffers(); // exchange the two buffers  
}
```

Árnyalás

$$L(V) \approx \sum_l L_l(L_l) * f_r(L_l, N, V) \cdot \cos \theta'$$

- Koherencia: ne mindent pixelenként
- Csúcspontonként:
belül az L „szín” interpolációja:
Gouraud árnyalás (per-vertex shading)
- Pixelenként:
belül a Normál (V iew, L ight) vektort interpoláljuk:
Phong árnyalás (per-pixel shading)



When a triangle point turns out to be visible (at least among the triangles processed so far), its color should be computed and written into the frame buffer.

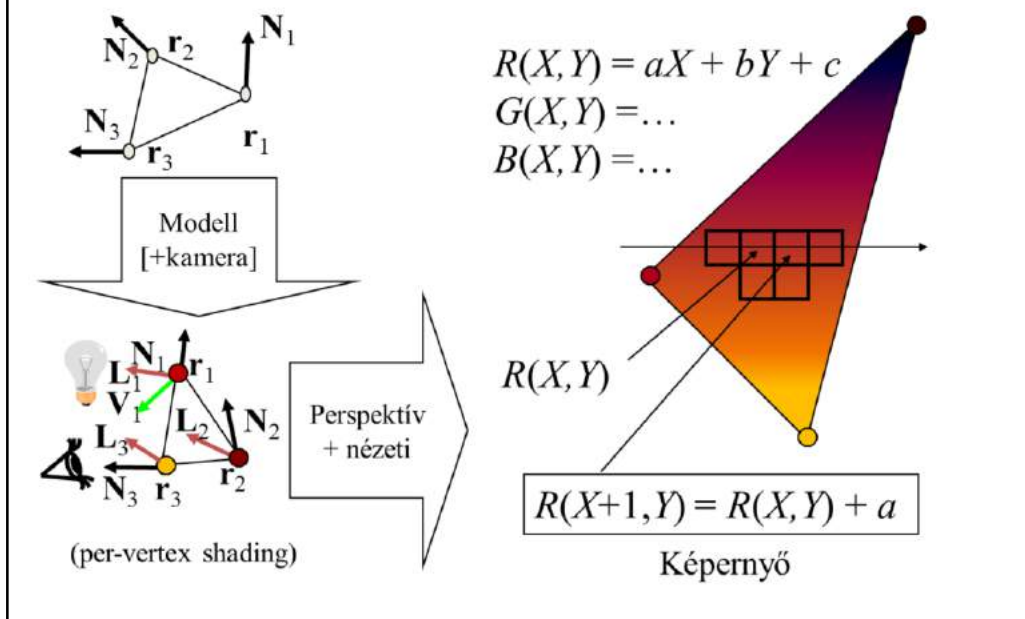
According to optics, the color is the radiance calculated in the direction of the eye, which is the sum of the contributions of abstract light sources in the local illumination model. For a single light source, the radiance is the light source intensity times the BRDF times the geometry factor.

In incremental rendering, we try to reuse calculation done in other pixels, so the evaluation of this formula for different pixels shares computations.

For example, we can evaluate this illumination formula only for the three vertices and apply linear interpolation in between the vertices (Gouraud shading or per-vertex shading).

Or, while the illumination formula is evaluated at every pixel, the vectors needed for the calculation are interpolated from the vectors available at the vertices (Phong shading or per-pixel shading).

Per-vertex (Gouraud) árnyalás



The radiance can be computed in world coordinates since here we have everything together needed by the illumination calculation, including the illuminated objects, light sources, and the camera. Alternatively, radiance can also be computed in camera space since the illumination formula depends on angles (e.g. between the illumination direction and the surface normal) and distances (in case of point light sources), and the transformation between world space and camera space is angle and distance preserving (congruence), so we obtain the same results in the two coordinate systems. However, radiance computation must not be postponed to later stages of the pipeline (e.g. normalized camera space, normalized device space, or screen space) since the mapping to these coordinate systems may modify angles.

After tessellation, the object is a set of triangles with vertices and shading normals. These are transformed to world space (or to camera space) by multiplying the vertices by the modeling transform (or by the modeling and camera transforms) and the normals by the inverse-transport of this matrix.

In this space, the view and light directions are computed at the triangle vertices. If per-vertex shading is applied, these vectors are immediately inserted into the illumination formula obtaining the reflected radiance at the

vertices. The triangle vertices with their computed colors are mapped to screen space, where rasterization takes place. The color of the internal pixels is computed by linear interpolation of the r,g,b values of the colors at the vertices. This is very similar to the interpolation of depth values.

Per-vertex shading: Vertex shader

```
uniform mat4 MVP, M, Minv; // MVP, Model, Model-inverse
uniform vec4 kd, ks, ka;    // diffuse, specular, ambient ref
uniform vec4 La, Le;        // ambient and point sources
uniform vec4 wLiPos;        // pos of light source in world
uniform vec3 wEye;          // pos of eye in world
uniform float shine;        // shininess for specular ref

in vec3 vtxPos;             // pos in modeling space
in vec3 vtxNorm;            // normal in modeling space
out vec4 color;             // computed vertex color

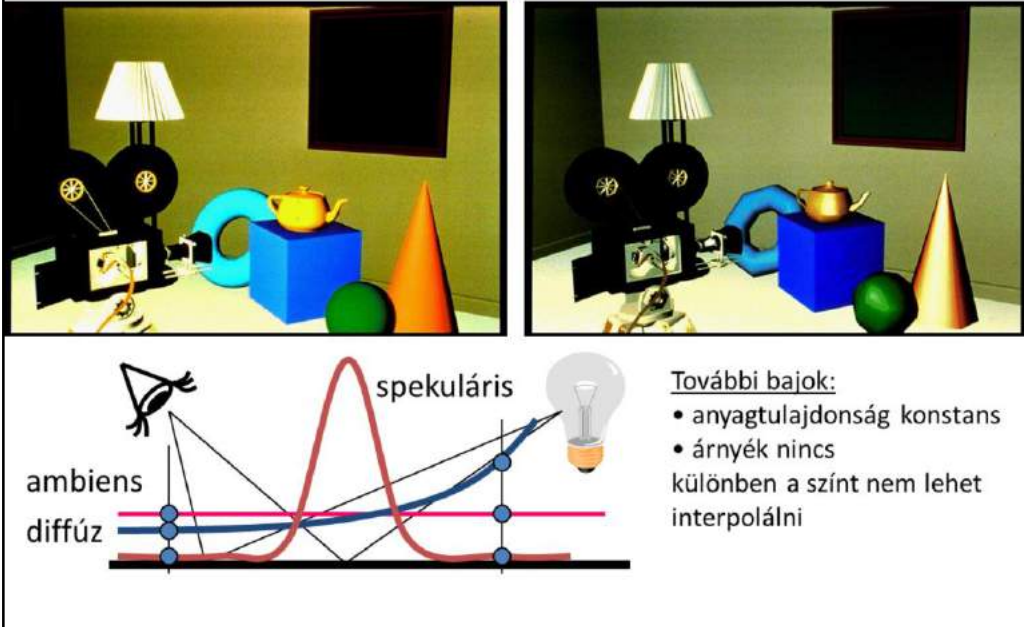
void main() {
    gl_Position = vec4(vtxPos, 1) * MVP; // to NDC
    vec4 wPos = vec4(vtxPos, 1) * M;
    vec3 L = normalize(wLiPos.xyz/wLiPos.w - wPos.xyz/wPos.w);
    vec3 V = normalize(wEye * wPos.w - wPos.xyz);
    vec4 wNormal = Minv * vec4(vtxNorm, 0);
    vec3 N = normalize(wNormal.xyz);
    vec3 H = normalize(L + V);
    float cost = max(dot(N, L), 0), cosd = max(dot(N, H), 0);
    color = ka * La + (kd * cost + ks * pow(cosd, shine)) * Le;
}
```

Per-vertex shading: Pixel shader

```
in vec4 color;           // interpolated color of vertex shader
out vec4 fragmentColor; // output goes to frame buffer

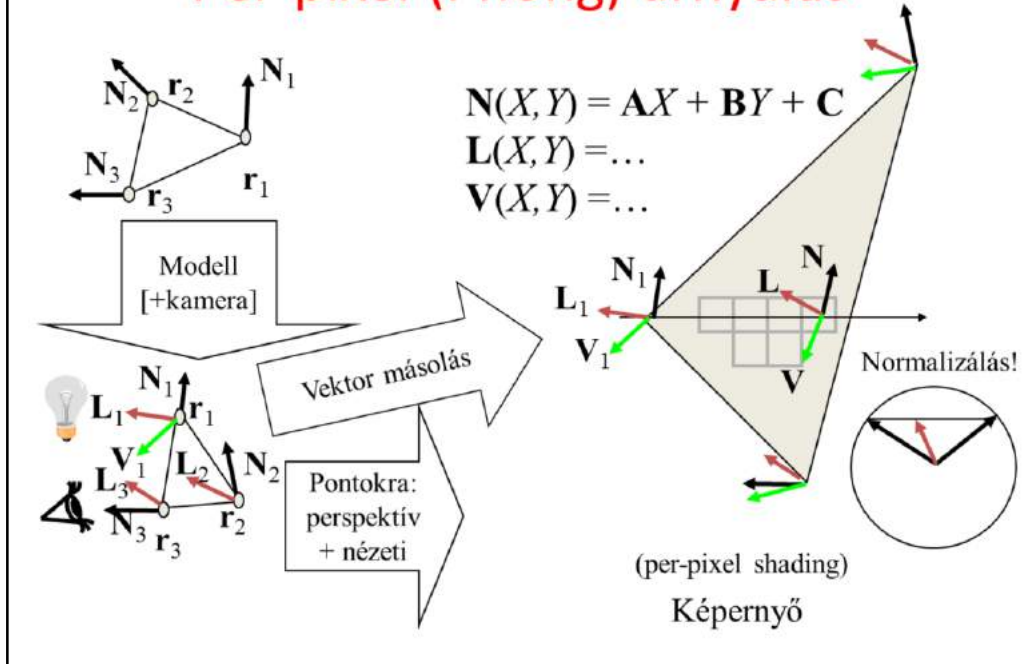
void main() {
    fragmentColor = color;
}
```

Gouraud árnyalás hasfájásai



Gouraud shading produces satisfactory results if the surfaces are finely tessellated and not strongly glossy or specular. If the triangles are large and the material is highly specular, the highlights will be strangely deformed revealing the underlying triangular approximation. The problem is that glossy reflection is strongly non-linear, which cannot be well represented by linear interpolation.

Per-pixel (Phong) árnyalás



The solution to this problem is Phong shading, which interpolates only those variables which are smoothly changing inside a triangle. Phong shading interpolates the vectors (normal, view, and illumination) and evaluates the radiance from the interpolated vectors at each pixel.

The triangle vertices and their shading normals are transformed to world space (or camera space). Here, illumination and view directions are obtained for each vertex. The vertices are transformed further to screen space, and the computed normal, illumination and view directions will follow them, but without any transformation (they still represent world (or camera) space directions). These vectors are linearly interpolated inside the triangle and for every pixel, the radiance is obtained.

Per-pixel shading: Vertex shader

```
uniform mat4 MVP, M, Minv; // MVP, Model, Model-inverse
uniform vec4 wLiPos;       // pos of light source
uniform vec3 wEye;         // pos of eye

in  vec3 vtxPos;           // pos in modeling space
in  vec3 vtxNorm;          // normal in modeling space

out vec3 wNormal;          // normal in world space
out vec3 wView;            // view in world space
out vec3 wLight;           // light dir in world space

void main() {
    gl_Position = vec4(vtxPos, 1) * MVP; // to NDC

    vec4 wPos = vec4(vtxPos, 1) * M;
    wLight = wLiPos.xyz * wPos.w - wPos.xyz * wLiPos.w;
    wView = wEye * wPos.w - wPos.xyz;
    wNormal = (Minv * vec4(vtxNorm, 0)).xyz;
}
```

Per-pixel shading: Pixel shader

```
uniform vec3 kd, ks, ka; // diffuse, specular, ambient ref
uniform vec3 La, Le;    // ambient and point source rad
uniform float shine;    // shininess for specular ref
in  vec3 wNormal;       // interpolated world sp normal
in  vec3 wView;         // interpolated world sp view
in  vec3 wLight;        // interpolated world sp illum dir
out vec4 fragmentColor; // output goes to frame buffer

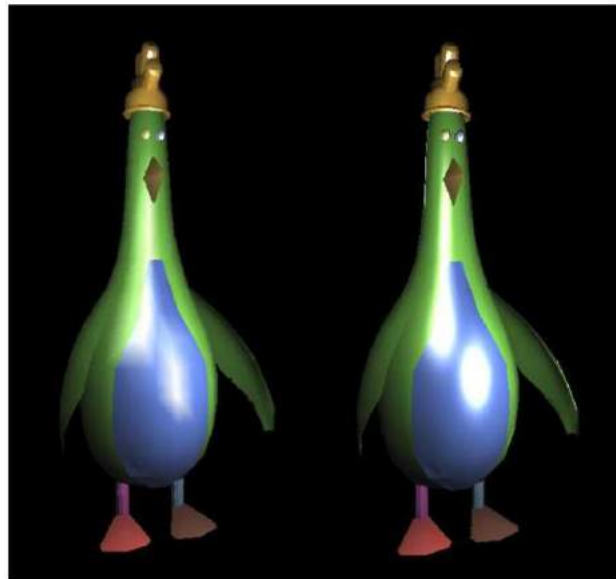
void main() {
    vec3 N = normalize(wNormal);
    vec3 V = normalize(wView);
    vec3 L = normalize(wLight);
    vec3 H = normalize(L + V);
    float cost = max(dot(N,L), 0), cosd = max(dot(N,H), 0);
    vec3 color = ka * La +
                (kd * cost + ks * pow(cosd,shine)) * Le;
    fragmentColor = vec4(color, 1);
}
```


Gouraud versus Phong

Gouraud



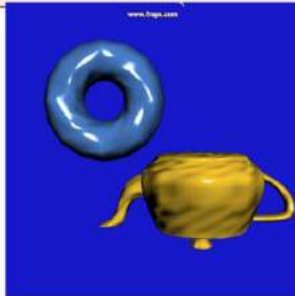
Phong



NPR: Non-Photorealistic Rendering

```
uniform vec3 kd; // diffuse ref
in  vec3 wNormal, wView, wLight; // interpolated
out vec4 fragmentColor; // output goes to frame buffer

void main() {
    vec3 N = normalize(wNormal);
    vec3 V = normalize(wView);
    vec3 L = normalize(wLight);
    float y = (dot(N, L) > 0.5) ? 1 : 0.5;
    if (abs(dot(N, V)) < 0.2) fragmentColor = vec4(0, 0, 0, 1);
    else
        fragmentColor = vec4(y * kd, 1);
}
```



NPR



Lichthof Productions

Bump/displacement mapping



texture mapping



bump mapping



parallax mapping



FPS = 100



FPS = 50



parallax mapping with offset limiting



parallax mapping with slope



iterative parallax mapping



FPS = 60



FPS = 30



ray marching



binary search



sphere tracing



relief mapping



cone stepping



per-vertex displacement mapping

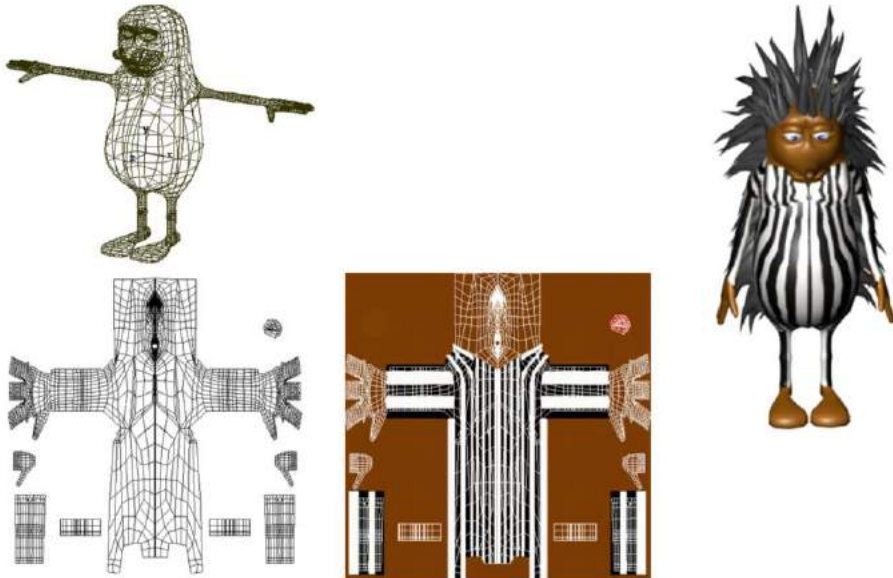


FPS = 85



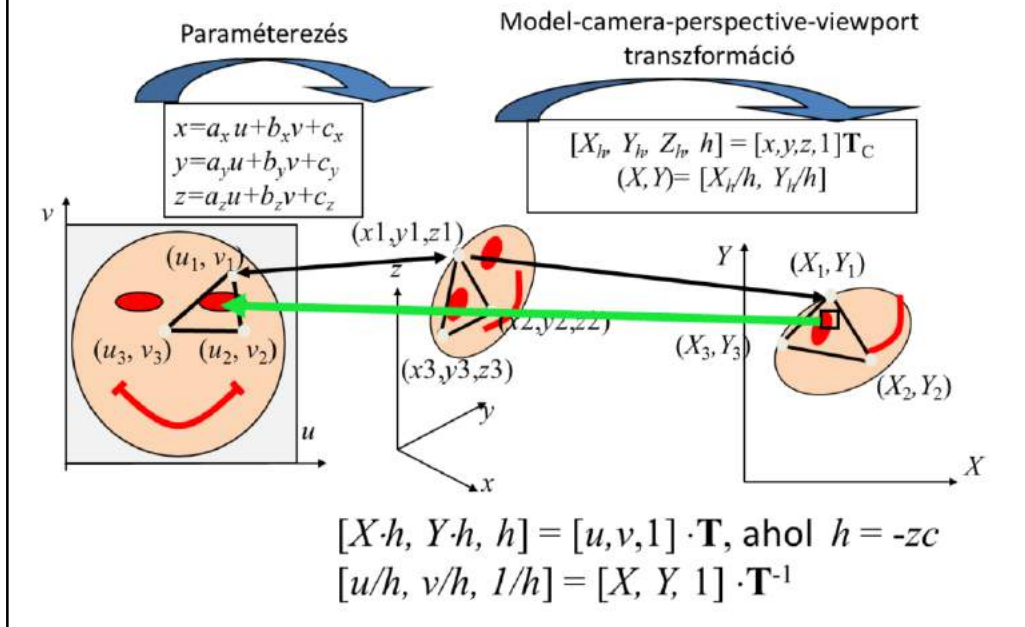
FPS = 42

Textúra leképezés: anyagjellemzők változnak a felületen



So far, we assumed that material properties are constant on a surface, i.e. inside a triangle. Texture mapping eliminates this restriction and makes material properties varying on the surface.

Textúrázás

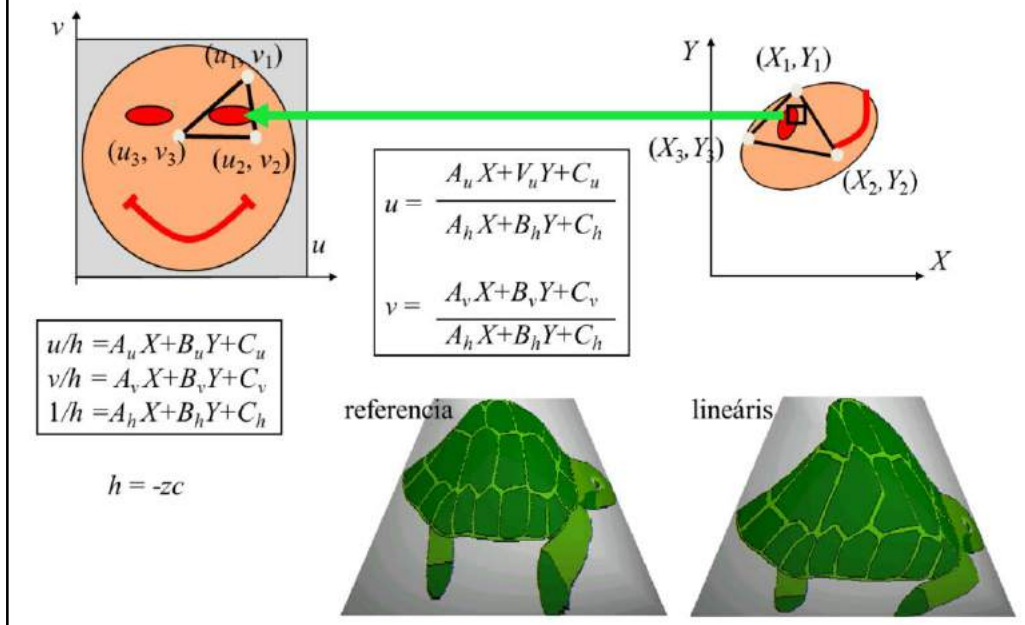


2D texture mapping can be imagined as wallpapering. We have a wallpaper that defines the image or the function of a given material property. This image is defined in texture space as a unit rectangle. The wallpapering process will cover the 3D surface with this image. This usually implies the distortion of the texture image. To execute texturing, we have to find a correspondence between the 3D surface and the 2D texture space. After tessellation, the 3D surface is a triangle mesh, so for each triangle, we have to identify a 2D texture space triangle, which will be painted onto the 3D triangle. The definition of this correspondence is called **parameterization**.

A triangle can be parameterized with an affine transformation (x, y, z are linear functions of u, v). Screen space coordinates are obtained with homogeneous linear transformation from x, y, z . Thus the mapping between texture space and screen space is also a homogeneous linear transformation: $[X \cdot h, Y \cdot h, h] = [u, v, 1] \cdot \mathbf{T}$, where X, Y are the pixel coordinates, $h = -zc$, the negative camera space z coordinate.

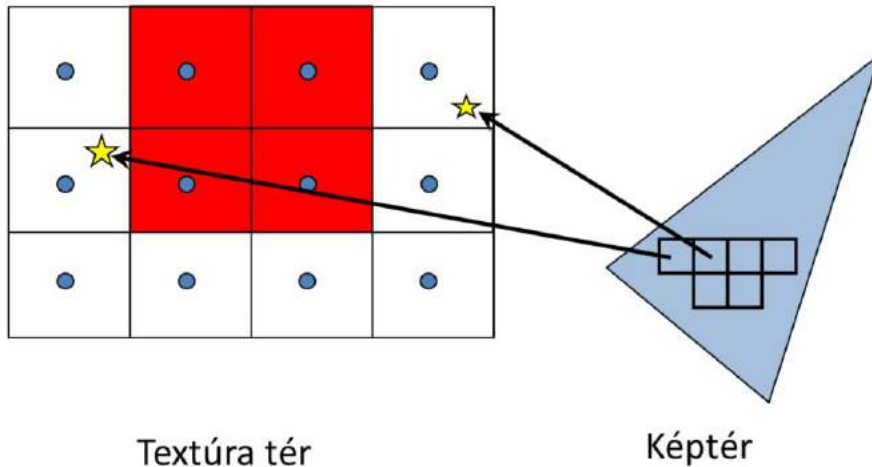
The triangle is rasterized in screen space. When a pixel is processed, texture coordinate pair u, v must be determined from pixel coordinates X, Y .

Perspektíva helyes textúrázás



The correct solution is homogeneous linear interpolation, aka perspective correct texture mapping, which linearly interpolates u/h , v/h , and $1/h$ and obtains u, v as two per-pixel divisions. Current GPUs do this perspective correct interpolation automatically.

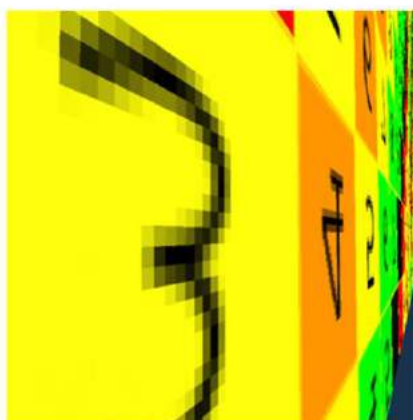
Textúra szűrés



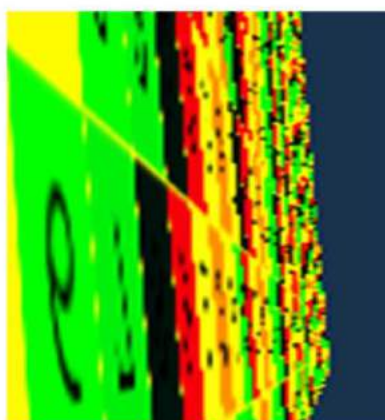
Rasterization visits pixels inside the projection of the triangle and maps the center of the pixel from screen space to texture space to look up the texture color. This mapping will result in a point that is in between the texel centers. More importantly, this mapping may be a magnification, which means that a single step in screen space results in a larger step in texture space, so we may skip texels, and the result will be a mess or noise.

From signal processing point of view, in this case, the texture is a high frequency signal which is sampled too rarely, resulting in sampling artifacts.

Textúratér és képtér kapcsolata

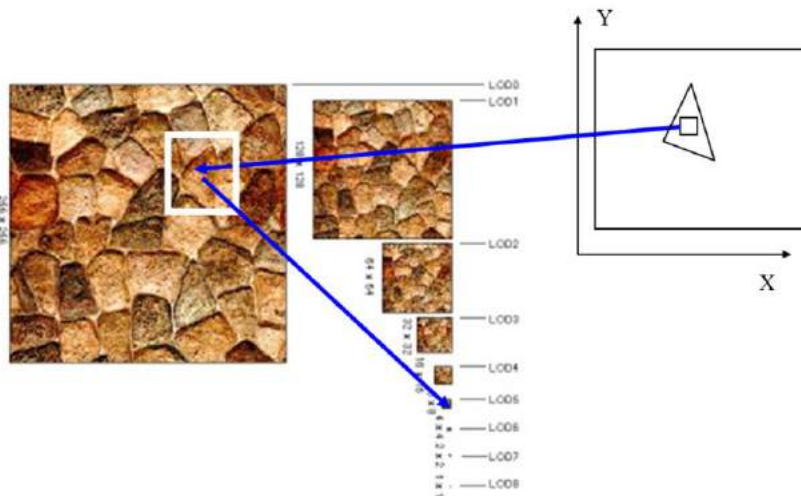


Magnification



Minification

Mip-map (multum in parvo)



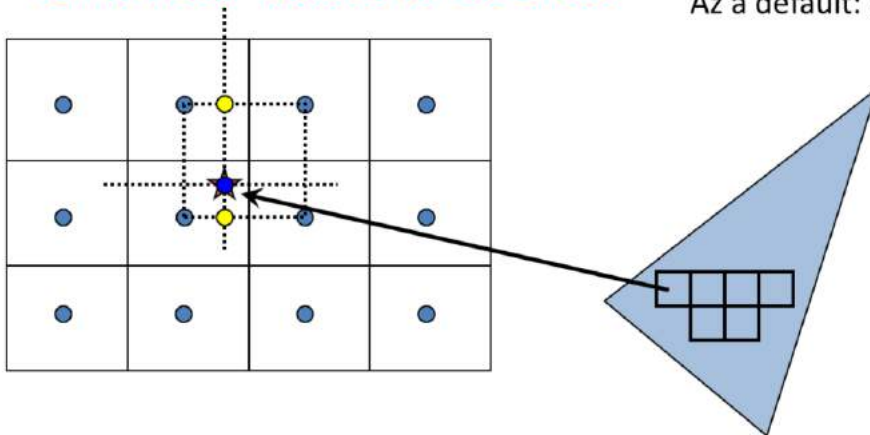
The solution for such sampling problems is filtering. Instead of mapping just the center of the pixel, the complete pixel rectangle must be mapped to texture space at least approximately, and the average of texels in this region should be returned as a color. However, it would be two time consuming.

One efficient approximation is to prepare the texture not only in its original resolution, but also in half resolution, quarter resolution, etc. where a texel represents the average color of a square of the original texel. During rasterization, OpenGL estimates the magnification factor, and looks up the appropriate version of filtered, downsample texture. The collection of the original and downsampled textures is called mip-map.

Bi-linear textúra szűrés

Mip-map is van: ☺

Az a default: ☹

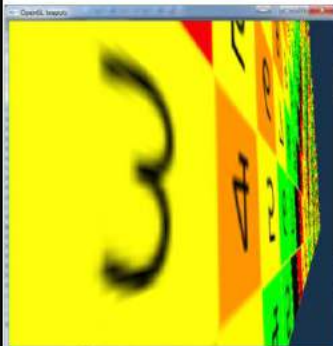
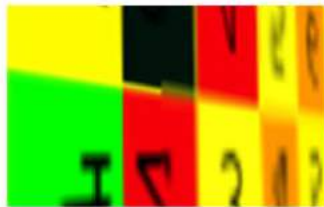


```
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D,  
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

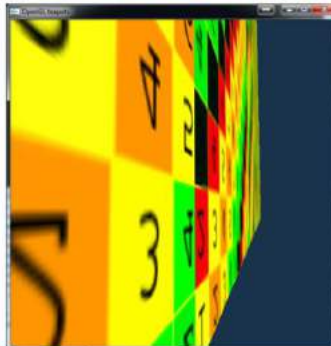
If we do not like to prepare our texture with reduced resolution, there is another simpler filtering scheme. When a pixel center is mapped to texture space, not only the closest texel is obtained but the four closest ones, and the filtered color is computed as the bi-linear interpolation of their colors.

The filtering method can be set separately when the pixel to texture space is a magnification or when it is a minification.

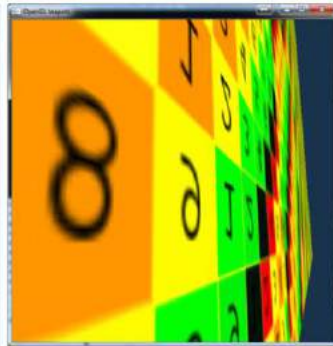
Textúra szűrési módok



Bi-linear filtering

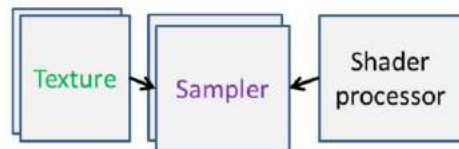


Mip-mapping



Linear mip-map szint

Textúrázás



```
struct Texture {
    unsigned int textureId;
    Texture(char * fname) {
        glGenTextures(1, &textureId);
        glBindTexture(GL_TEXTURE_2D, textureId);    // binding
        int width, height;
        float *image = LoadImage(fname,width,height); // megírni!
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
                    0, GL_RGB, GL_FLOAT, image); //Texture -> OpenGL
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    }
};
```

```
void Geometry::Draw( ) {
    int samplerUnit = 0;
    int location = glGetUniformLocation(shaderProg, "samplerUnit");
    glUniform1i(location, samplerUnit);
    glActiveTexture(GL_TEXTURE0 + samplerUnit);
    glBindTexture(GL_TEXTURE_2D, texture.textureId);
    glBindVertexArray(vao); glDrawArrays(GL_TRIANGLES, 0, nVtx);
}
```

Vertex és Pixel Shader

```
in vec3 vtxPos;  
in vec3 vtxNorm;  
in vec2 vtxUV;  
out vec2 texcoord;  
  
void main() {  
    gl_Position = vec4(vtxPos, 1) * MVP;  
    texcoord = vtxUV;  
    ...  
}
```

```
uniform sampler2D samplerUnit;  
in vec2 texcoord;  
out vec4 fragmentColor;  
  
void main() {  
    fragmentColor = texture(samplerUnit, texcoord);  
}
```

Átlátszóság: Sorrend számít!

első

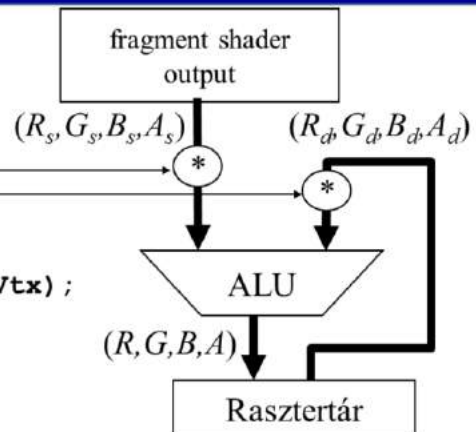
első

```
glEnable(GL_BLEND);
```

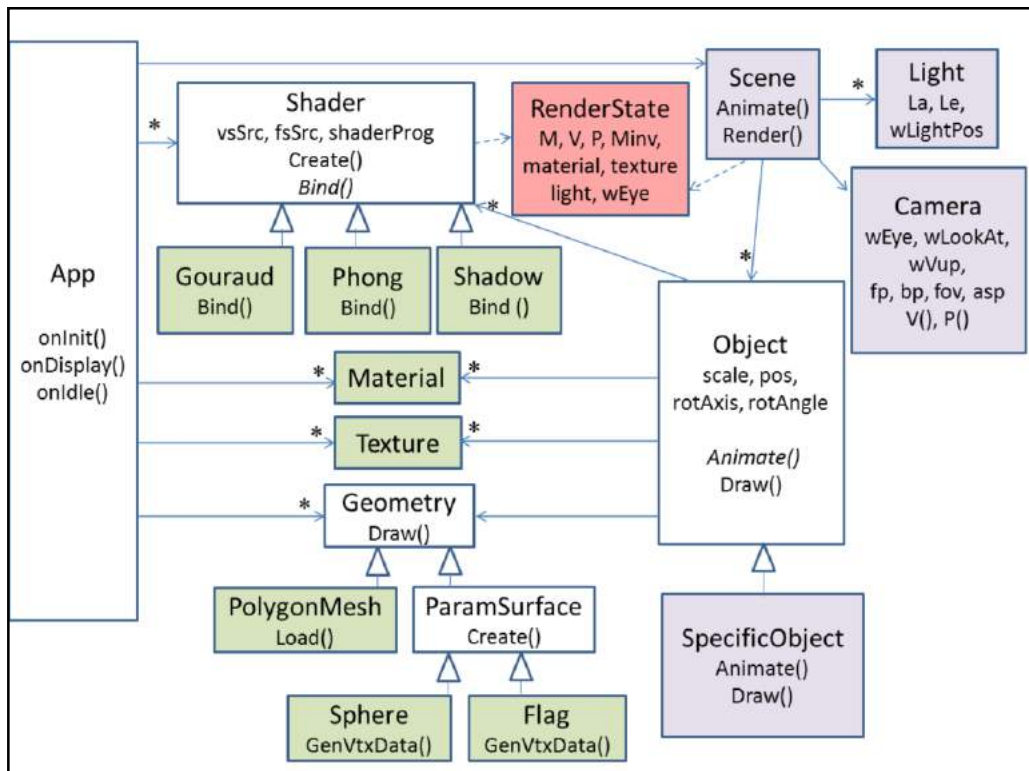
```
glBlendFunc(  
    GL_SRC_ALPHA, _____  
    GL_ONE_MINUS_SRC_ALPHA  
);
```

```
glDrawArrays(GL_TRIANGLES, 0, nVtx);
```

```
glDisable(GL_BLEND);
```



$$(R, G, B, A) = (R_s A_s + R_d (1 - A_s), G_s A_s + G_d (1 - A_s), B_s A_s + B_d (1 - A_s), A_s A_s + A_d (1 - A_s))$$



Scene

```
class Scene {
    Camera camera;
    vector<Object *> objects;
    Light light;
    RenderState state;
public:
    void Render() {
        state.wEye = camera.wEye;
        state.V = camera.V;
        state.P = camera.P;
        state.light = light;
        for (Object * obj : objects) obj->Draw(state);
    }
    void Animate(float dt) {
        for (Object * obj : objects) obj->Animate(dt);
    }
};
```

Object

```
class Object {
    Shader *  shader;
    Material * material;
    Texture * texture;
    Geometry * geometry;
    vec3 scale, pos, rotAxis;
    float rotAngle;
public:
    virtual void Draw(RenderState state) {
        state.M = Scale(scale.x, scale.y, scale.z) *
            Rotate(rotAngle, rotAxis.x, rotAxis.y, rotAxis.z) *
            Translate(pos.x, pos.y, pos.z);
        state.Minv = Translate(-pos.x, -pos.y, -pos.z) *
            Rotate(-rotAngle, rotAxis.x, rotAxis.y, rotAxis.z) *
            Scale(1/scale.x, 1/scale.y, 1/scale.z);
        state.material = material; state.texture = texture;
        shader->Bind(state);
        geometry->Draw();
    }
    virtual void Animate(float dt) {}
};
```

Érték szerinti paraméterátadás:
Objektumok nem zavarják egymást

Shader

```
struct Shader {
    unsigned int shaderProg;

    void Create(const char * vsSrc, const char * vsAttrNames[],
               const char * fsSrc, const char * fsOutputName) {
        unsigned int vs = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vs, 1, &vsSrc, NULL); glCompileShader(vs);
        unsigned int fs = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fs, 1, &fsSrc, NULL); glCompileShader(fs);
        shaderProgram = glCreateProgram();
        glAttachShader(shaderProg, vs);
        glAttachShader(shaderProg, fs);

        for (int i = 0; i < sizeof(vsAttrNames)/sizeof(char*); i++)
            glBindAttribLocation(shaderProg, i, vsAttrNames[i]);
        glBindFragDataLocation(shaderProg, 0, fsOutputName);
        glLinkProgram(shaderProg);
    }
    virtual
    void Bind(RenderState& state) { glUseProgram(shaderProg); }
};
```

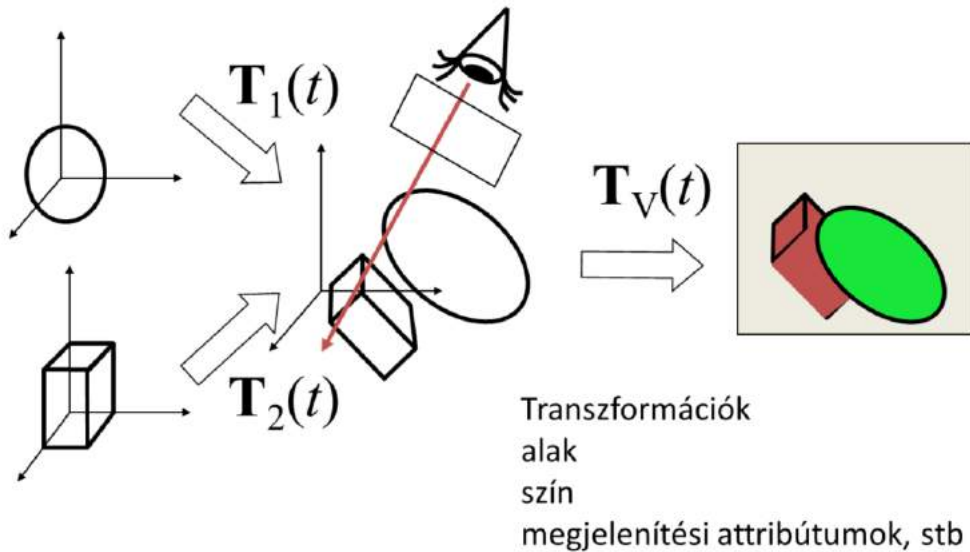
ShadowShader

```
class ShadowShader : public Shader {
    const char * vsSrc = R"(
        uniform mat4 MVP;
        in vec3 vtxPos;
        void main() { gl_Position = vec4(vtxPos, 1) * MVP; }
    )";
    const char * fsSrc = R"(
        out vec4 fragmentColor;
        void main() { fragmentColor = vec4(0, 0, 0, 1); }
    )";
public:
    ShadowShader() {
        static const char * vsAttrNames[] = { "vtxPos" };
        Create(vsSrc, vsAttrNames, fsSrc, "fragmentColor");
    }
    void Bind(RenderState& state) {
        glUseProgram(shaderProg);
        mat4 MVP = state.M * state.V * state.P;
        MVP.SetUniform(shaderProg, "MVP");
    }
};
```

Animáció

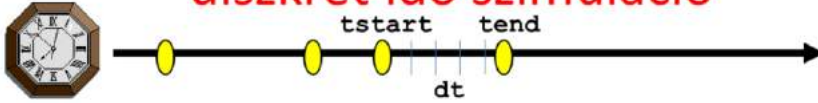
Szirmay-Kalos László

Animáció = időfüggés



Animation means that the properties of objects, light sources, or the camera change in time. Any property may change, but the most important case is when transformations are functions of time. If modeling transformation depends on time, we animate objects. If camera transformations are time dependent, we animate the camera.

Valós idejű animáció és diszkrét idő szimuláció



```
void IdleFunc( ) {           // idle call back
    static float tend = 0;
    const float dt = 0.01; // dt is "infinitesimal"
    float tstart = tend;
    tend = glutGet(GLUT_ELAPSED_TIME)/1000.0f;

    for(float t = tstart; t < tend; t += dt) {
        float Dt = min(dt, tend - t);
        for (Object * obj : objects) obj->Control(Dt);
        for (Object * obj : objects) obj->Animate(Dt);
    }
    glutPostRedisplay();
}
```

Animation also means that when some time elapses, the state of the virtual world catches up with the elapsed time. This should be happening even if the user does not even touch the computer. The event handling system provides the idle callback for this purpose. So in an idle call back, the elapsed times is computed, and the time interval $[tstart, tend]$ elapsed since the last idle callback is simulated. As there is no upper limit for the length of the simulated interval, it is decomposed to dt steps that are sufficiently small. Here small means that time differentials can be well approximated differences and in dt we can suppose that the velocity and acceleration are constants.

The simulation of an object is subdivided to control and animate. Control means the preparation for the state change, and animate means the execution of the state change.

If we merged the two operations together, then the simulation would depend on the processing order of objects.

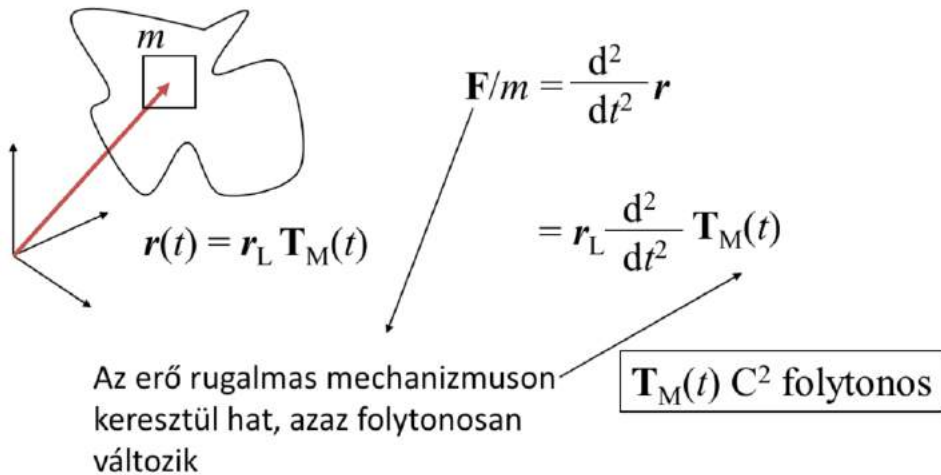
Valószerű mozgás

- Fizikai törvények:
 - Newton törvény
 - Impulzus deriváltja az erő; Perdület deriváltja a forgatónyomaték
 - ütközés detektálás és válasz:
 - impulzus és perdület megmaradás
 - energia részleges megmaradás
- Fiziológiai törvények
 - csontváz nem szakad szét
 - meghatározott szabadságfokú ízületek
 - bőr rugalmasan követi a csontokat
- Energiafelhasználás minimuma



The task of animation is the definition of the time dependence of transformations. We hope for realistic animations that do not contradict to the physical laws (acceleration is proportional to the force, linear momentum and angular momentum are conserved) or to the physiological laws (bones are connected by joints that do not allow bones to separate).

Newton törvény



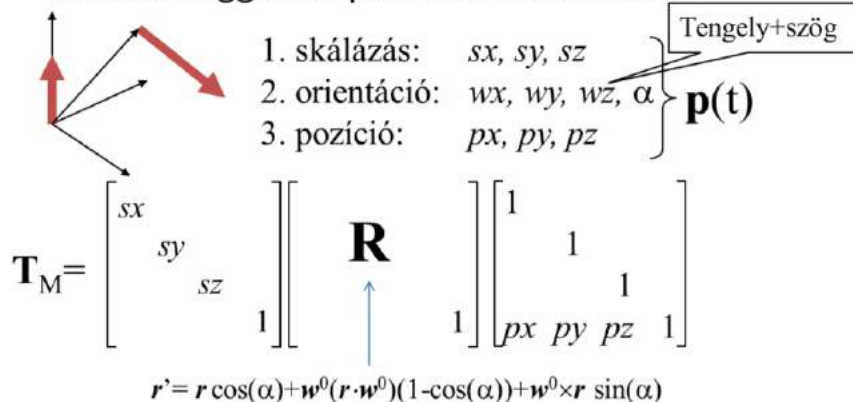
According to the Newton's law, acceleration, i.e. the second derivative of the motion is proportional to the force. As forces act on some elastic mechanism, they cannot change abruptly. So generally, the path should have a continuous second derivative, i.e. of C^2 type curve.

However, if the body is really rigid (and not elastic), then force can change abruptly. In case of collisions very large, i.e. infinite forces may occur. So in special situations, the path can be of C^1 or C^0 type.

$T_M(t)$: Mozgástervezés

- Követelmény: ált. C^2 , néha (C^1, C^0) folytonosság
- Mátrixelemek nem függetlenek

– Tervezés független paraméterek terében



So we need motion curves that are generally of C^2 type, occasionally of C^1 and C^0 . Motion is the time dependence of 4×4 homogeneous linear transformation matrices, having 16 elements. However, typical motions like translation, translation+rotation etc. Have less degrees of freedom, translation has 3, rotation has also 3, thus the 16 matrix elements are not independent. Should we interpolate them independently, the transformed object would be distorted. Therefore, we never specify time dependency directly for the transformation matrix elements. Instead, the space of independent motion parameters is found, the independent motion parameters are given time functions and are calculated first, then the matrix elements are obtained from the independent motion parameters. For rigid body motion, the independent motion parameters include the Cartesian coordinates of the translation, the direction vector of the rotation axis, and the angle of rotation. If the size can also change in time, three additional scaling parameters are added. From these, the modeling transformation matrix can be obtained.

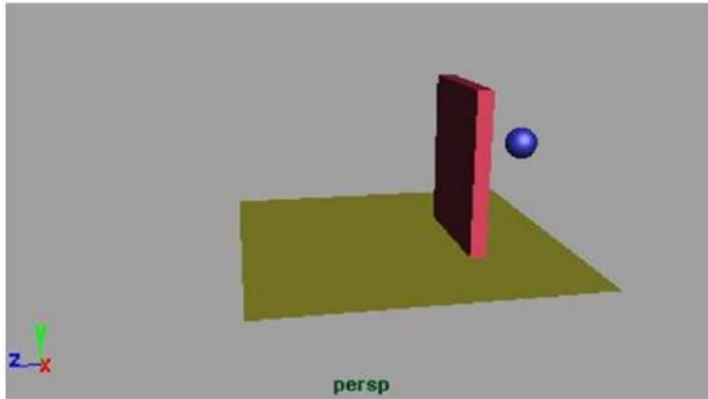
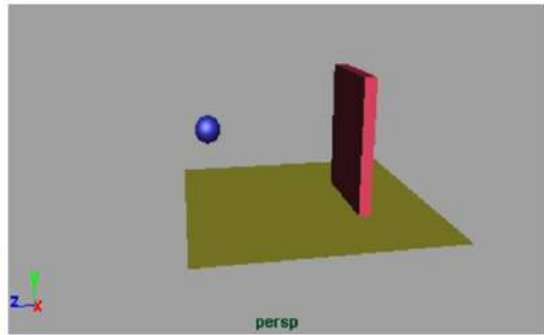
Mozgástervezés a paraméterterben

- $\mathbf{p}(t)$ elemei ált. C^2 , néha (C^1, C^0) folytonosak
- $\mathbf{p}(t)$ elemeinek a definíciója:
 - görbével direkt módon (**spline**)
 - képlettel: **script animation**
 - kulcsokból interpolációval: **keyframe animation**
 - görbével indirekt módon: **path animation**
 - mechanikai modellből az erők alapján: **physical animation**
 - mérésekből: **motion capture animation**

Let us concentrate on the definition of the vector of independent motion parameters. There are various possibilities to define time functions in them.

Parametric curves use the analogy of motion, thus they can be directly used to describe motion. The time functions can also be given by formulae (e.g. motion of a bullet from a canon). Keyframe animation requires the setting of objects at discrete points of time, called keyframes, and the inbetweening process finds curves that interpolate the discrete poses. Path animation defines the path with a motion curve also requiring that the orientation of the object changes according to the motion, i.e. the object follows its own „head” and tries to preserve a vertical direction. Physical animation does not specify motion directly, but provides the physical parameters like mass, friction, initial velocity and position etc. and solves the laws of motion to simulate motion. Finally motion capture animation measures the motion of a real-character (motion artist) and uses the measured data to control a virtual character.

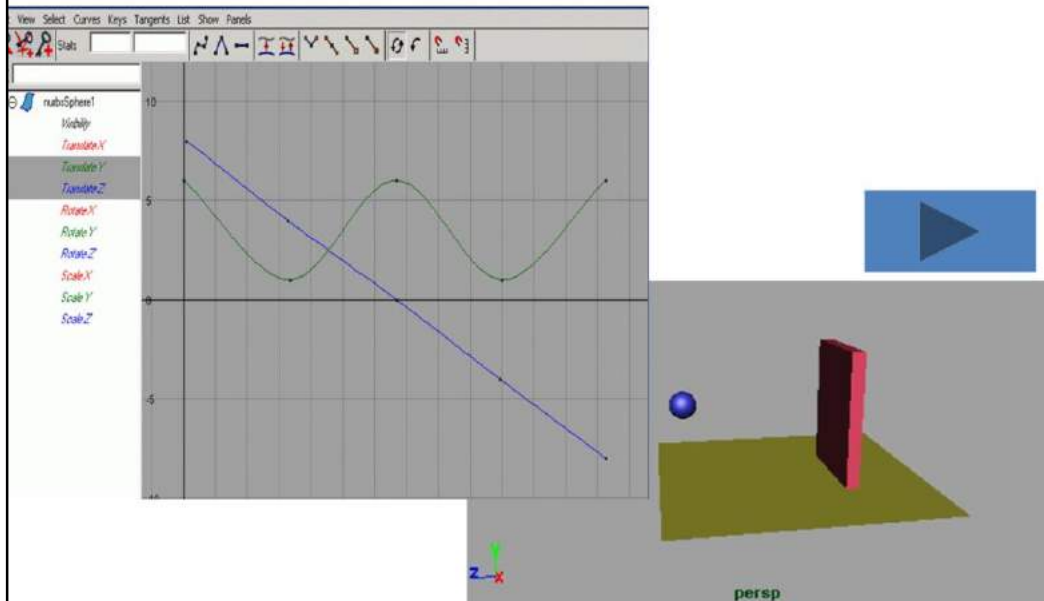
Keyframe animáció



5.key

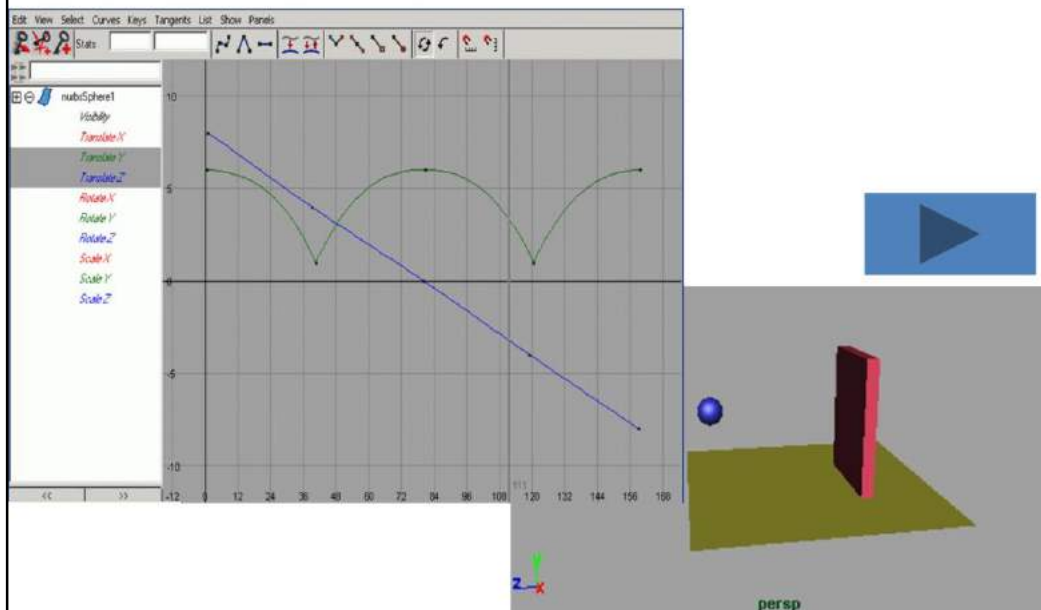
Let us define a clip by keyframe animation, where a ball bounces on the floor while a door opens and lets the ball in. At 5 points of time, the ball and the door are positioned.

Keyframe animáció görbéi



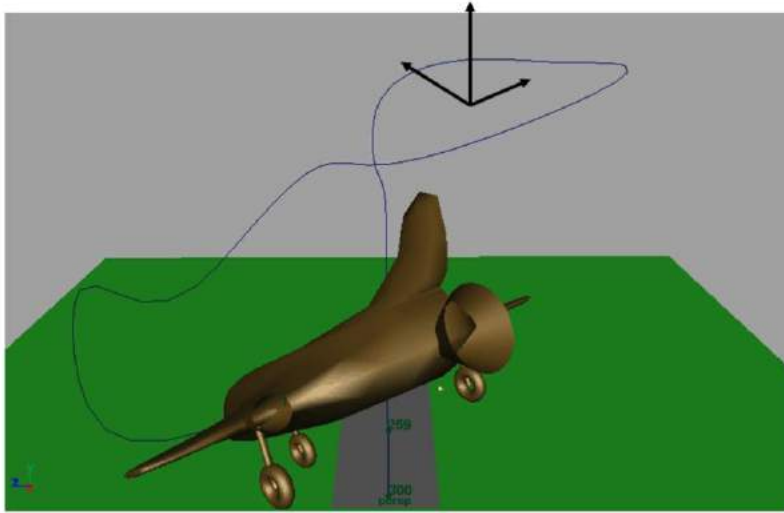
The discrete positions are black dots, i.e. Points to be interpolated on the yet unknown motion curves (y and z coordinates of the ball are shown). The interpolation is done with Catmull-Rom spline, so we get two functions interpolating the key values. The resulting animation is not realistic since it does not provide the bouncing effect.

Görbék megváltoztatása



So the splines are modified by hand. We know from physics that the y coordinate should follow a parabola.

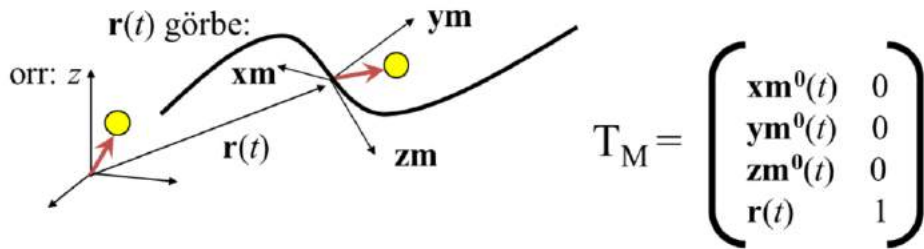
Pálya (path) animáció



t = spline paraméter vagy az ívhossz

For path animation a single 3D curve should be drawn, which directly defines the position and indirectly the orientation.

Pálya animáció: Transzformáció



Explicit up vektor

$$z_m = r'(t)$$

$$x_m = z_m \otimes \text{up}$$

$$y_m = z_m \otimes x_m$$

Frenet keretek:

$$z_m = r'(t)$$

$$x_m = z_m \otimes r''(t)$$

$$y_m = z_m \otimes x_m$$

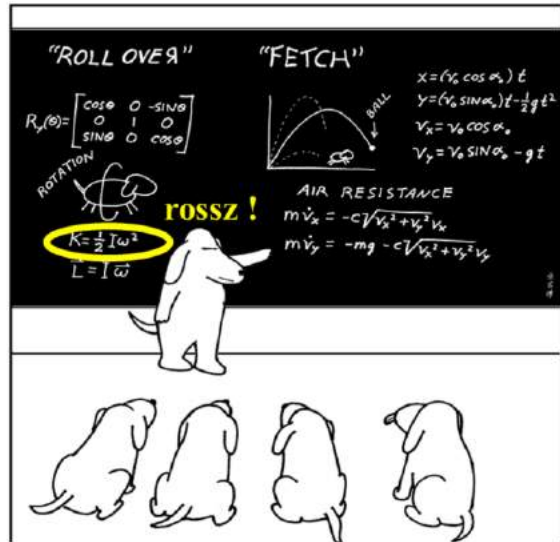
A függőleges,
amerre az erő hat

The indirect orientation definition is based on the recognition that objects like airplanes, birds, cars, people, animals etc. follow their own „nose”, meaning that their nose always point into the direction of the motion, which is the current velocity vector. The velocity is the derivative of the path. The nose direction is not enough to define the full orientation, so we also specify a „preferred vertical” direction. This can be a fixed direction or the current acceleration (this option is called Frenet frames in differential geometry).

If in modeling space, the nose direction is axis z , the vertical direction is axis y , then the transformation matrix can be directly obtained from their transformed versions.

Fizikai animáció

- Erők (pl. gravitáció, turbulencia stb.)
- Tömeg, tehetetlenségi nyomaték ($F = ma$)
- Ütközés detektálás (metszéspontszámítás)
- Ütközés válasz
 - rugók, ha közel vannak
 - impulzus megmaradás



In physical animation the forces, masses, inertia and the initial conditions are defined and motion is obtained simulating the physical laws:

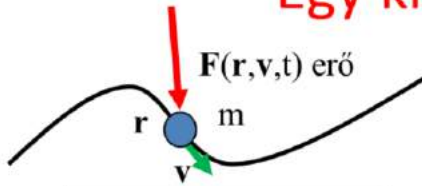
- Linear momentum is the product of the mass and the velocity of the center of mass
- The time derivative of the linear momentum is the force
- Angular momentum is the product of the inertial matrix and the angular velocity
- The time derivative of the angular momentum is the torque
- Collision happens when objects are about to penetrate into each other
- Upon collision, linear momentum and angular momentum are conserved, the kinetic energy is conserved only in case of elastic collision.

Note that in dog school the formula of the kinetic energy of rotational motion is wrong. What would be the right one?

Haladó és forgó mozgás

Pozíció:	\vec{r}	Orientáció:	\mathbf{R}
Összefűzés:	$\vec{r}_1 + \vec{r}_2$	Összefűzés:	$\mathbf{R}_1 \cdot \mathbf{R}_2$
Sebesség:	$\vec{r}(t + dt) =$ $\vec{r}(t) + \vec{v}dt$	Szögsebesség:	$\mathbf{R}(t + dt) =$ $\mathbf{R}(t)R(\vec{\omega} dt, \vec{\omega})$
Tömeg:	m	Tehetlenségi nyom:	\mathbf{I}
Lendület:	$\vec{p} = m\vec{v}$	Perdület:	$\vec{L} = \mathbf{I}\vec{\omega}$
Erő:	$\vec{F} = \frac{d\vec{p}}{dt}$	Forgató nyom:	$\vec{M} = \frac{d\vec{L}}{dt}$
Energia:	$K = \frac{1}{2}mv^2$	Energia:	$K = ?$

Egy kis mechanika

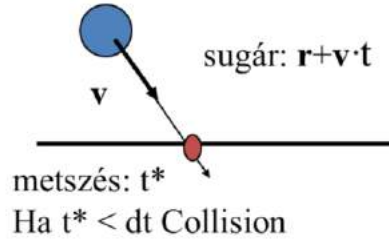


$$\frac{dr}{dt} = v$$

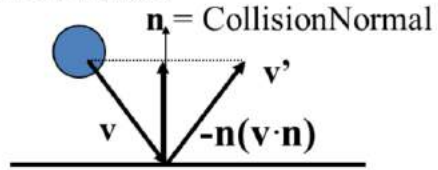
$$\frac{dv}{dt} = \mathbf{F}(\mathbf{r}, \mathbf{v}, t)/m$$

```
for( t = 0; t < T; t += dt) {
     $\mathbf{F} = \text{Eredő erő}(\mathbf{r}, \mathbf{v})$ 
     $\mathbf{a} = \mathbf{F}/m$ 
     $\mathbf{v} += \mathbf{a} \cdot dt$ 
    if ( ÜtközésDetektál )
        ÜtközésVálasz
     $\mathbf{r} += \mathbf{v} \cdot dt$ 
}
```

ÜtközésDetektál



ÜtközésVálasz



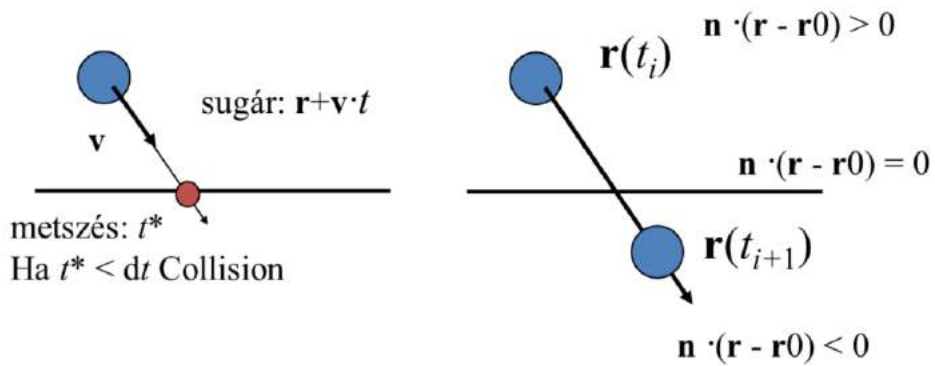
$$\mathbf{v}' = [\mathbf{v} - \mathbf{n}(\mathbf{v} \cdot \mathbf{n})] - [\mathbf{n}(\mathbf{v} \cdot \mathbf{n}) \cdot \text{bounce}]$$

For the sake of simplicity, we consider only translational motion and point like objects. The force field in nature may depend on the time (e.g. wind), position (e.g. gravity) and the velocity (e.g. air resistance), but not on higher derivatives. According to the Newton's law, the time derivative of the velocity is the force divided by the mass. According to the definition of the velocity, it is the time derivative of the position. So we have two linear differential equations that need to be solved. If the time is decomposed to small steps dt , differentials are approximated by differences, so the change of velocity will be the acceleration times dt , while the change of position the velocity time dt , since we assume that dt is small enough so acceleration and velocity are constants.

This is not the case for collision, so this should be checked and if collision happens, the modified velocity should be directly computed from the preservation laws (linear momentum and angular momentum are preserved, kinetic energy is preserved only for elastic collision).

For the motion of a point like object in interval dt when we can assume that the velocity is constant, collision detection is equivalent to ray tracing. Collision response is similar to mirror like reflection if the collision is elastic, and we should reduce the perpendicular component if it is inelastic.

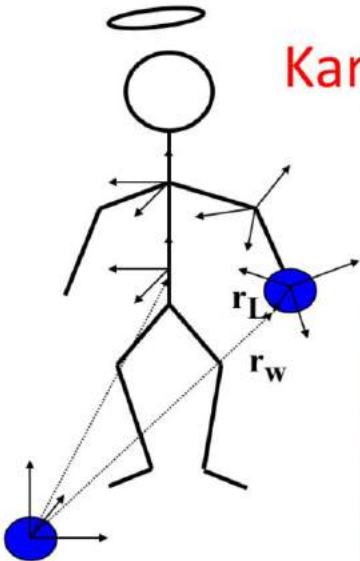
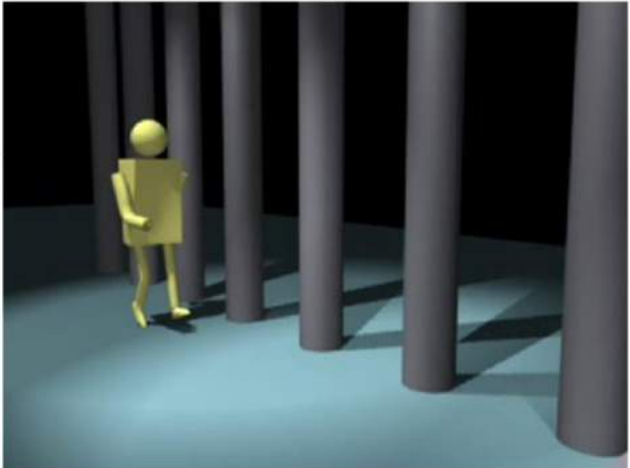
Folytonos-Diszkrét ütközés detektálás pontra és féltérre



Continuous collision detection is equivalent to ray tracing since the path of a point is assumed to be linear in dt .

Discrete collision detection checks whether the objects have penetrated into each other by containment test.

Karakter animáció

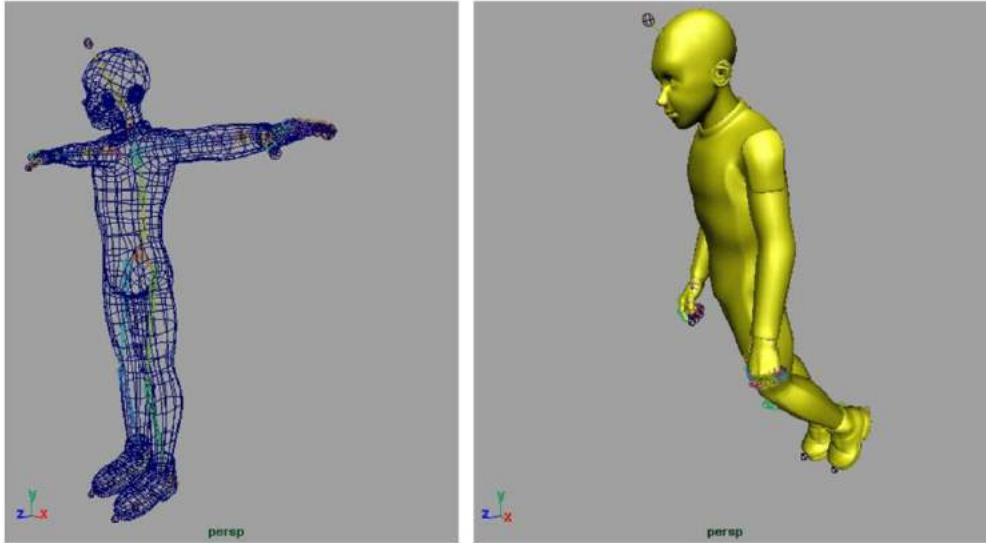



$$\mathbf{r}_w = \mathbf{r}_L \cdot \mathbf{R}_{\text{kéz}} \cdot \mathbf{T}_{\text{alkar}} \cdot \mathbf{R}_{\text{könyök}} \cdot \mathbf{T}_{\text{felkar}} \cdot \mathbf{R}_{\text{váll}} \cdot \mathbf{T}_{\text{gerinc}} \cdot \mathbf{T}_{\text{ember}}$$

homogén koordináta 4-es

In character animation, the skeleton defines the motion. Every bone in the skeleton can introduce a rotation around the joint and a translation depending on the length of the bone.

Csontváz



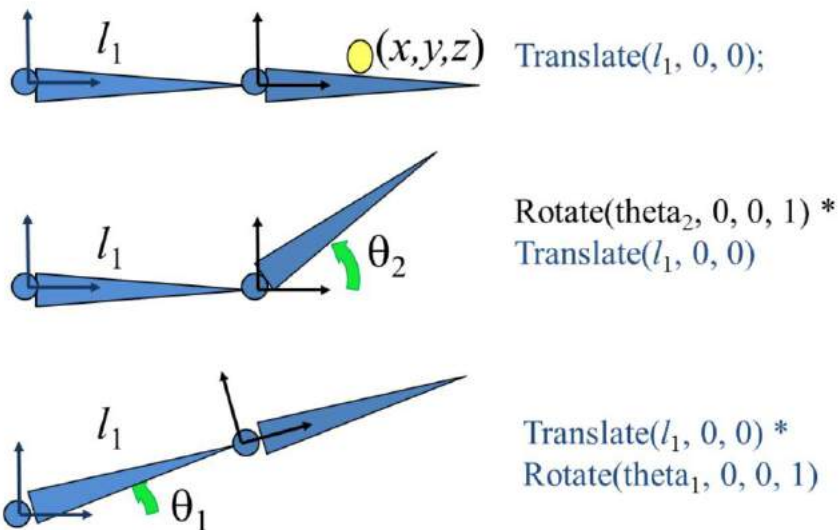
When bone animation is defined, the bones and joints connecting them are also included in the mesh representing the skin. By default, a skin vertex will be transformed by the transformation of that bone which is closest to it.

During animation, the bones are rotated in the joints and upper level bones naturally modify all other bones connected to it via joints. The skin is deformed with the resulting transformation matrices of the bones.

Séta



Transzformáció hierarchia



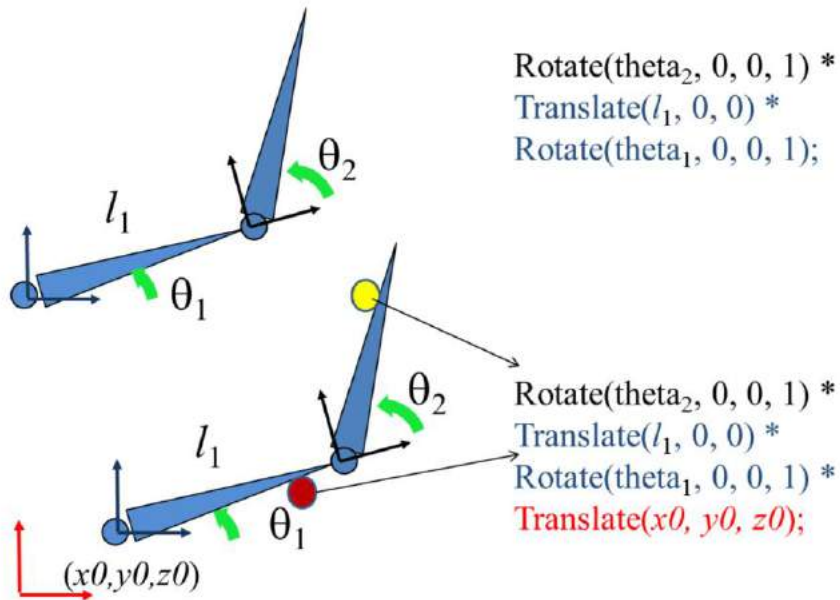
To examine how this can be done in OpenGL, let us consider a skin vertex attached to the cyan bone. When this correspondence is made, the skin vertex is expressed relative to its bone, i.e. in the coordinate system of the bone, resulting in (x, y, z) .

This point can also be expressed in the coordinate system of the parent bone (brown bone), just the transformation between the reference systems of the two bones should be executed. This is currently a translation along axis x with the length of the bone l_1 .

If the child (cyan) bone is rotated, this rotation applies to the skin in its coordinate system, thus rotation happens before applying the translation to the parent system.

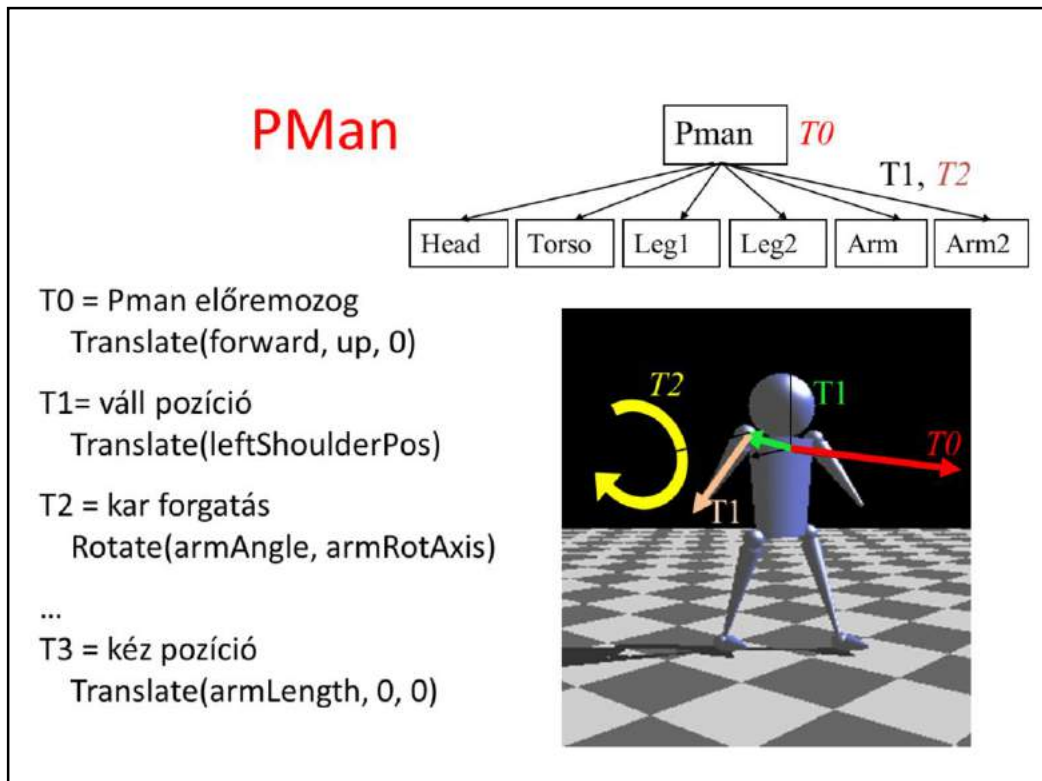
If the parent bone is rotated, the skin is translated first to the parent's coordinate system, then rotation takes place.

Transzformáció hierarchia



If both bones are rotated, first child rotation, then translation to the parent's system, finally rotation of the parent are executed. If the parent is also a child of some other node, or the parent is placed in the world, then new transformations must be added on the top of the hierarchy.

Generally, a bone is a rotation transformation for its own skin and a rotation + translation to its children, which should be applied recursively on hierarchical characters.



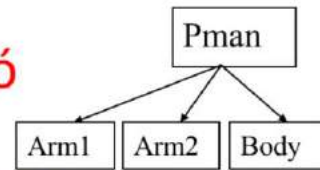
Our simple example is the primitive man, which consists of a head, torso, two independent legs and two arms. Let us consider just an arm.

Pman swings his arm while walking. The swing rotation is defined in a coordinate system where the origin is the shoulder position. Then, the position of the shoulder with respect to Pman should be defined, i.e. the swinging arm should be expressed in a coordinate system having the origin in the center of Pman. This is a translation. Finally, Pman moves forward, i.e. its center is translated in the world coordinate system. So the arm is first rotated ($T2$), then translated ($T1$), and translated again ($T0$). From these transformations, $T0$ and $T2$ change in time, but $T1$ remains constant, which defines the physiological constraints of the body:

Pman can move its complete body and can swing its arm, but cannot remove its arm from its shoulder.

Pman rajzolás és animáció

```
class Pman {
    float armAngle, dArmAngle, forward, up;
    const ... armLength, armRotAxis, rightArmJoint, ...;
public:
    void DrawArm(float dt, mat4 M) {
        M = Rotate(armAngle, armRotAxis) * M; // T2
        DrawRefArm(M);
    }
    void DrawPman( ) { // set matrices from animation parameters
        mat4 M = Translate(forward, up, 0); // T0
        DrawRefBody(M);
        DrawArm(dt, Translate(rightArmJoint) * M); // T1
        DrawLeg(dt, Translate(rightLegJoint) * M);
        ...
    }
    void Animate(float dt) { // calculate animation parameters
        armAngle += dArmAngle * dt;
        if (armAngle > 0.5 || armAngle < -0.5) dArmAngle *= -1;
        forward += 5 * dt;
    }
};
```

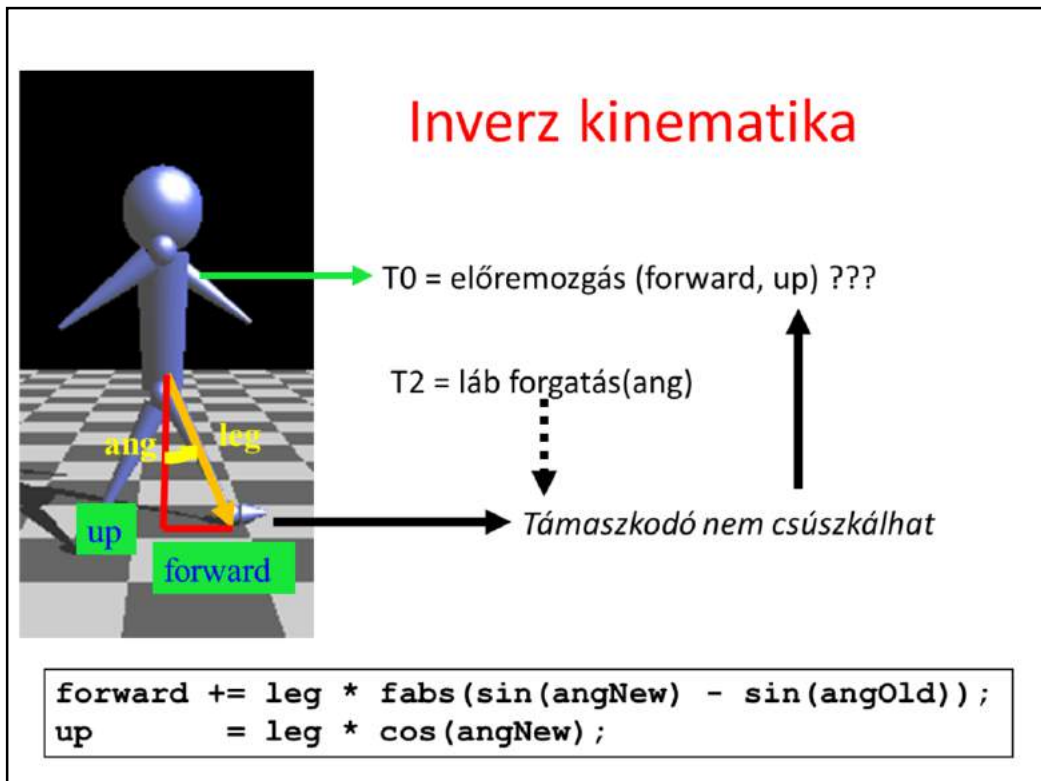


Push/Pop

Only the parameters of T0 and T2 are updated.

T2 is a periodic swinging rotation, which is defined by two key frames defining the two extreme angles and the rotation angle is linearly interpolated in between according to the elapsed time.

Note that when we step on a lower hierarchy level, the transformation matrix is pushed on stack, and then restored since objects on the same level should not interfere (arms are independent, so are legs). However, the parent affects all its children.



The motion defined by constant forward moving velocity and constant angular speed in the hips and shoulders is not realistic since the leg will slip on the floor (like a break dancer) and the body will fly over the floor.

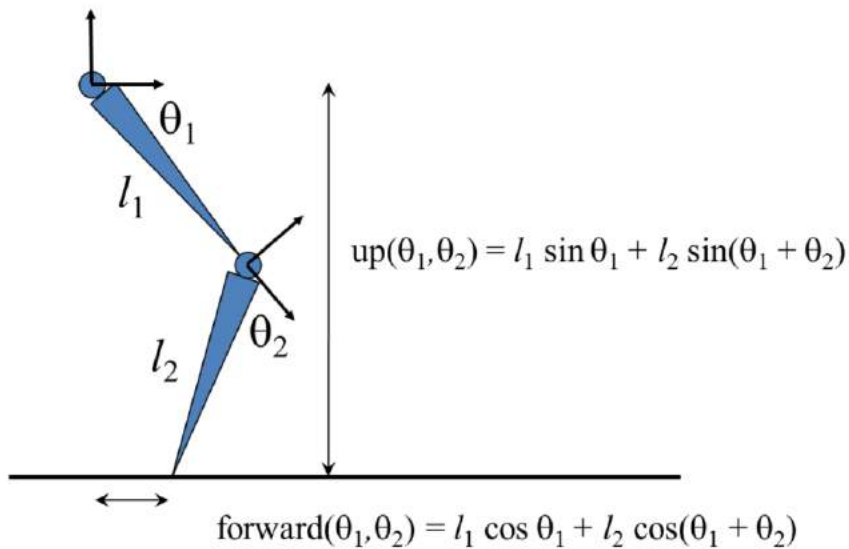
In realistic walking one leg should stand still on the floor. In an animation, the point of interest for which constraints are given is called the **end effector** (a term inherited from robotics). The end effector of the walking is the leg holding the weight of the body.

The problem is that we define the character state from top to bottom by setting a sequence of transformations. This is called **forward kinematics**. The end effector at the end of the transformation sequence will be affected by all transformations. The task is to determine the upper level transformations in a way that the resulting end effector position meets the specified constraint. Such problems are called **inverse kinematics**.

In this simple problem, we can explicitly solve the inverse kinematics problem since the relationship between the position of the character (forward and up) a hip rotation (ang) is defined by a right triangle of hypotenuse equal to the leg length. The end effector is always on the ground, so up directly specifies the distance from the floor. However, Pman walks forward, so its location along the forward direction is not constant. The actual forward

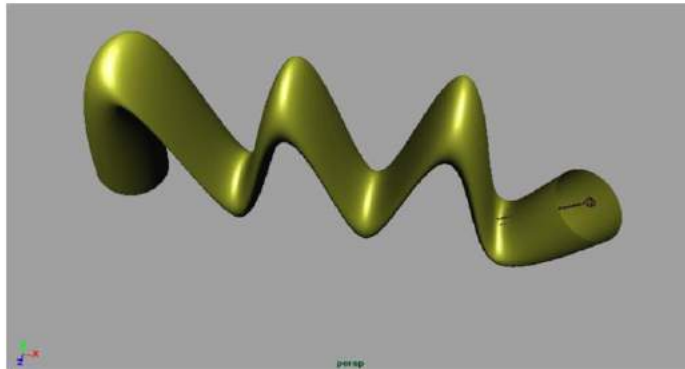
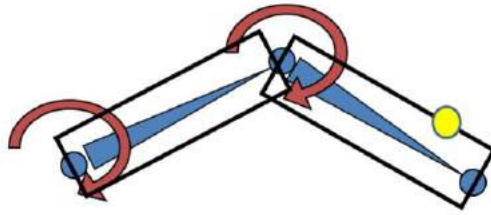
position is just relative to the leg, which means that forward is updated incrementally.

Inverz kinematika



In case of multiple joints and bones, the correspondence between the rotation angles and the relative position of the origin and the end effector may become complicated if the rotation axes are different. However, when rotation axes are parallel (more or less, this is the case for hip and knee rotations), a simple analytic expression can be elaborated.

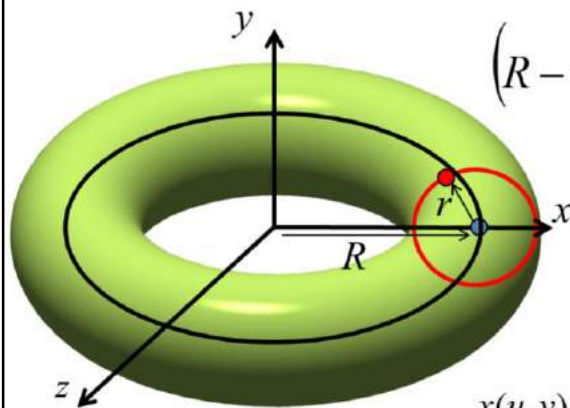
Bőrözés



3. házi: Tóruszba zárt Spiderman

Egy procedurálisan textúrázott, diffúz-spekuláris tórusz belsejében egy ugyancsak procedurálisan textúrázott golyó gördül, egy cián és egy sárga fényű fényforrás labda pattog a tórussszal rugalmasan ütközve és a mechanikai energiáját megtartva, valamint spiderman avatárunk várja a sorsát, akinek a szemszögévől követjük az eseményeket és gyönyörködünk a Phong árnyalt színtérben. A tórusz rögzített, golyó anti-gravitációs készülékkel van ellátva, így nem szakad el a tórusz falától. A többiekre pedig hat a homogén nehézségi erőter. A golyó pályája a tórusz falán periódikus és nem kör (pályaanímáció). Spiderman mindig a golyó irányába néz, mert szeretnénk elkerülni, hogy a golyó legázolja. Ha a tórusz belső falának egy pontjára mutatunk a bal egérgomb lenyomásával, akkor oda egy nem zérus nyugalmi hosszúságú gumikötelet lő ki, ami megnyúlás esetén a Hooke törvény és a dinamika alaptörvénye szerint magával rántja, így a közeledő golyó elől el tud ugrani. Minden újabb gumilövés a régit oldja.

Tórusz



$$\left(R - \sqrt{x^2 + z^2}\right)^2 + y^2 - r^2 = 0$$

$$x(u, v) = (R + r \cos(2\pi u)) \cos(2\pi v)$$

$$y(u, v) = r \sin(2\pi u)$$

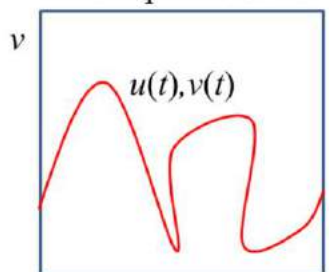
$$z(u, v) = (R + r \cos(2\pi u)) \sin(2\pi v)$$

Meglőtt pont előállítása

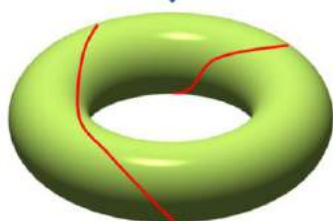
- Irány visszatranszformálása világkoordinátarendszerbe és ray tracing
- Irány és mélység visszatranszformálása világkoordináta rendszerbe (glReadPixels)
- Képszintézis színek helyett világkoordináták rajzolásával (vigyázat 8 bites rasztertár vagy *render-to-texture*)

Golyó gördül a tóruszon

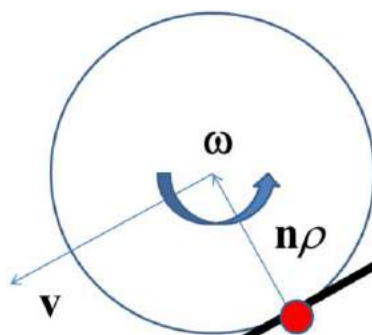
Tórusz paraméterter



$\mathbf{r}(u, v)$



$$\mathbf{v} = \frac{d\mathbf{r}(u(t), v(t))}{dt} = \frac{\partial \mathbf{r}}{\partial u} \frac{du}{dt} + \frac{\partial \mathbf{r}}{\partial v} \frac{dv}{dt}$$



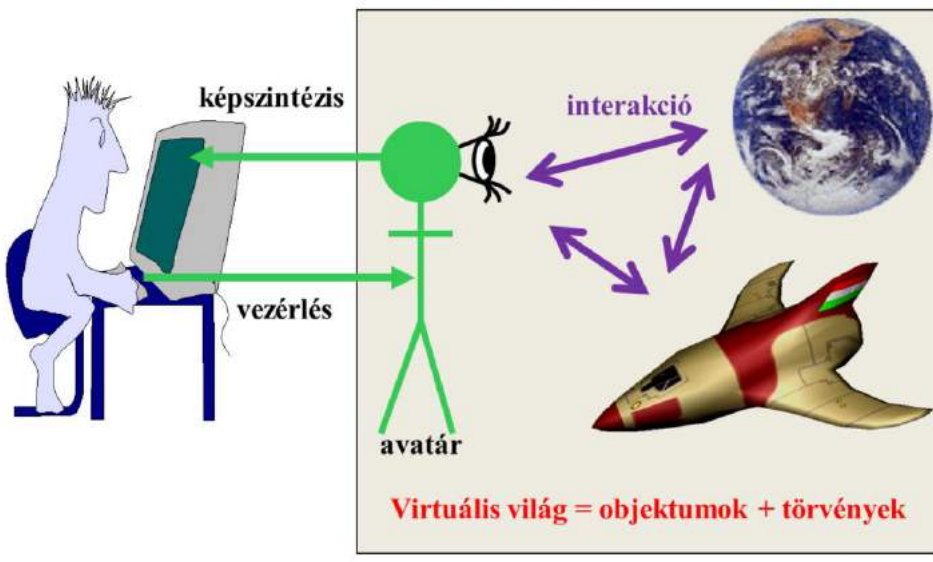
$$\mathbf{v} + \omega \times (-\mathbf{n}\rho) = \mathbf{0}$$

Játékfejlesztés



Szirmay-Kalos László

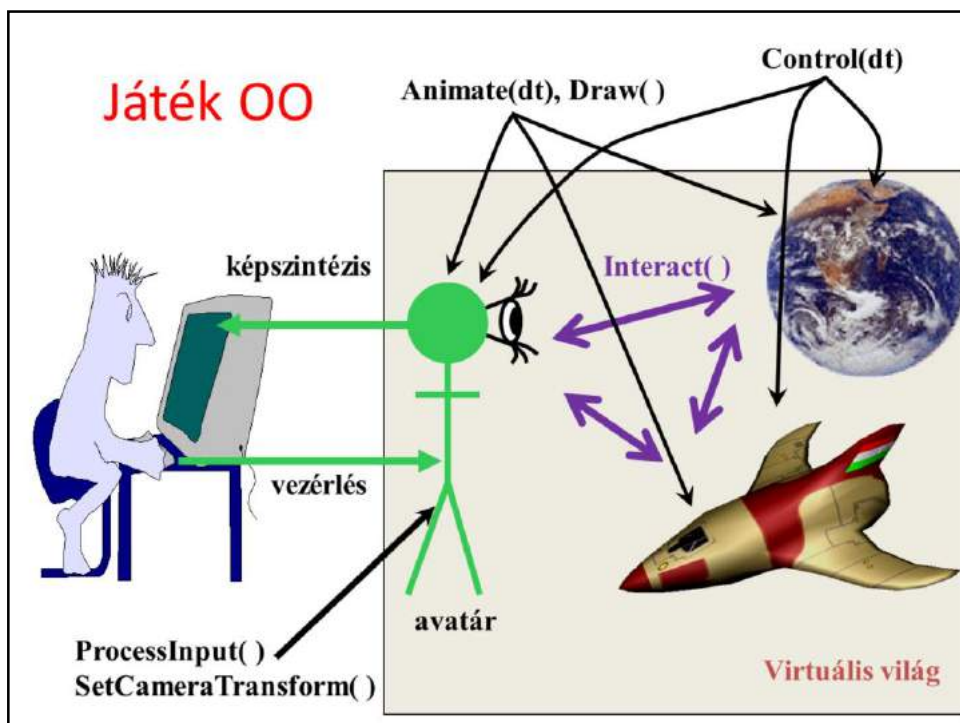
Virtuális valóság



Játékok feladatai

- Képszintézis az avatar nézőpontjából
- Az avatar vezérlése a beviteli eszközökkel (keyboard, mouse, Wii, gépi látás, Kinect, stb.)
- Az „intelligens” objektumok vezérlése (AI)
- A fizikai világ szimulációja

Játék OO



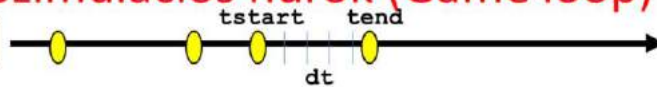
Játékobjektum (GameObject)

```
class GameObject {  
protected:  
    Shader *    shader;  
    Material * material;  
    Texture *   texture;  
    Geometry *  geometry;  
    vec3 pos, velocity, acceleration;  
public:  
    GameObject(Shader* s, Material* m,  
               Texture* t, Geometry* g) { ... }  
    virtual void Control(float dt) { }  
    virtual void Animate(float dt) { }  
    virtual void Draw(RenderState state) { }  
};
```

Virtuális világ

```
std::vector<GameObject*> objects;
```


Szimulációs hurok (Game loop)



```

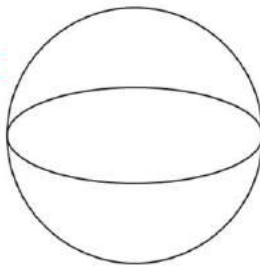
void onIdle ( ) {    // idle call back
    static float tend = 0;
    float tstart = tend;
    tend = glutGet(GLUT_ELAPSED_TIME)/1000.0f;
    avatar->ProcessInput( );
    for(float t = tstart; t < tend; t += dt) {
        float Dt = min(dt, tend - t);
        for (GameObject * obj : objects) obj->Control(Dt);
        for (GameObject * obj : objects) obj->Animate(Dt);
    }
    glutPostRedisplay();
}

void onDisplay(){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    avatar->SetCameraTransform(state);
    for (GameObject * obj : objects) obj->Draw(state);
    glutSwapBuffers( );
}
    
```



Bolygó: Planet

- Geometria: gömb
- Textúra
- Fizikai vagy
 - Tájékozódik majd követi a gravitációs törvényt
- Képletanimáció:
 - „beégetett pálya”
 - Többiek érdektelenek
 - Nincs respektált törvény



Planet class



```
class Planet : public GameObject {
    float rotAngle; // animation state
    float rotSpeed; // animation parameter
public:
    Planet(Gouraud* s, Diffuse* m, Texture* t, Sphere* s)
    : GameObject(s, m, t, s) { rotAngle = 0; rotSpeed = ...; }

    void Animate(float dt) { rotAngle += rotSpeed * dt; }

    void Draw(RenderState state) {
        state.M = Rotate(rotAngle, 0, 0, 1)
        state.Minv = Rotate(-rotAngle, 0, 0, 1)
        state.material = material;
        state.texture = texture;
        shader->Bind(state);
        geometry->Draw();
    }
};
```

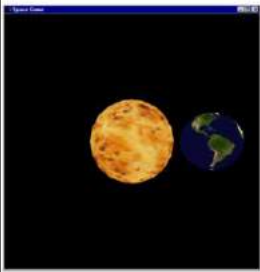
To draw the textured sphere of the planet, we can follow the general strategy of drawing parametric surfaces since the sphere can also be expressed in parametric form. The parameter space is tessellated and u, v samples are inserted into the parametric equations defining point on the surface. Those points that are neighbors in parameter space form triangles. The u, v pairs are used directly to pass texture coordinates to OpenGL.

Putting these together, we can implement the **Planet class**, which is derived from the general **GameObject class**, and implements its virtual functions like `ControlIt`, `InteractIt`, `AnimateIt`, and `DrawIt`.

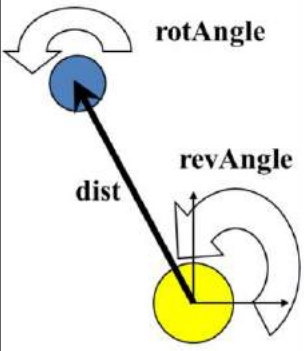
`ControlIt` and `InteractIt` are empty since a planet does not have to control its path, neither does it have AI, it simply responds to physics laws.

First, we consider a simplified case when the planet does not move, it only rotates around its axis. The current state of the planet is represented by rotation angle **rot_angle**, which is update in **AnimateIt** according to rotation speed **rot_speed** and the elapsed time **dt**.

The DrawIt function gets the planet to be drawn by OpenGL, first setting the Earth's texture (loaded probably in the constructor) as the active texture, asking for a rotation by rot_angle around vector (0,0,1) (which is supposed to be the Earth's axis of rotation), and finally sending the tessellated mesh to OpenGL by **gluSphere**. Note that this rotation should not be applied on other objects of the virtual world, so the transformation is saved in the transformation stack by **glPushMatrix** and restored after drawing by **glPopMatrix**.



A Föld kering a Nap körül



```

void Planet::Animate(float dt){
    rotAngle += rotSpeed * dt;
    revAngle += revSpeed * dt;
}

void Planet::Draw(RenderState state) {
    state.M = Rotate(rotAngle,0,0,1)*
               Translate(dist, 0, 0) *
               Rotate(revAngle,0,0,1);

    state.Minv = ...;
    state.material = material;
    state.texture = texture;
    shader->Bind(state);
    geometry->Draw();
}

```

Earth not only rotates around its axis but also revolves around the Sun. So, we need two animation parameters, rotation angle and revolution angle, which are updated according to the rotation speed (1/day in real world) and the revolution speed (1/year in real world), respectively.

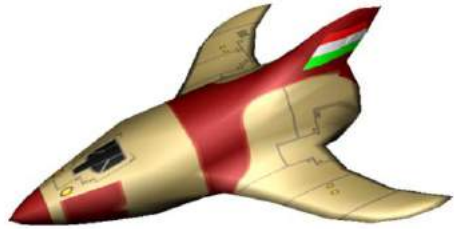
Let us read the DrawIt function backwards since transformation should be read in this order. SphereInOrigin passes a triangular mesh approximation of a sphere centered in the origin to OpenGL. `glRotatef(rot_angle, 0, 0, 1)` rotates the sphere around axis z.

`glTranslatef(dist, 0, 0)` translates the sphere to distance `dist` from the origin where the Sun is supposed to be. Finally, we apply another rotation `glRotatef(rev_angle, 0, 0, 1)`, around axis z, which keeps the Earth on the circle of radius `dist` and in the xy plane around the Sun.

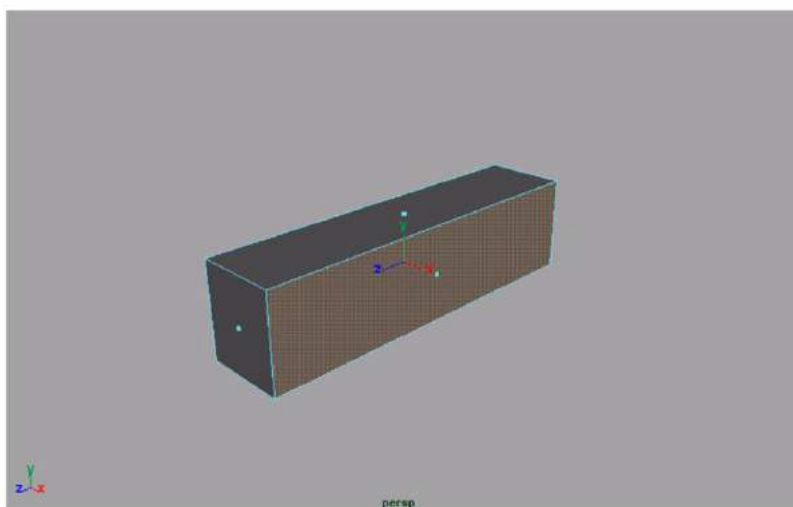
By the way, the real Earth's rotation axis is not perpendicular to the plane of revolution, but is tilted by about 23 degrees. The addition of this transformation is a homework.

Az űrhajó

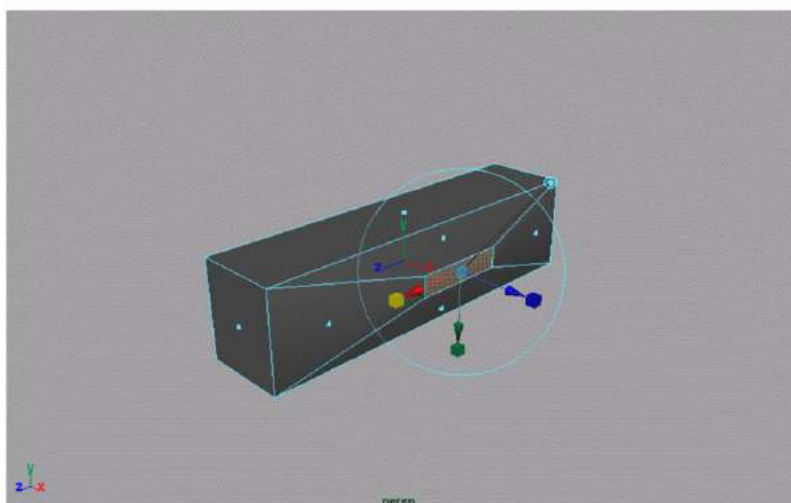
- Komplex geometria
 - négyszögháló
- Komplex textúra
- Fizikai animáció
 - erők (gravitáció, rakéták)
 - ütközések
- Viselkedés (AI)
 - A rakéták vezérlése
 - Ütközés elkerülés, avatártól menekülés, avatár üldözése



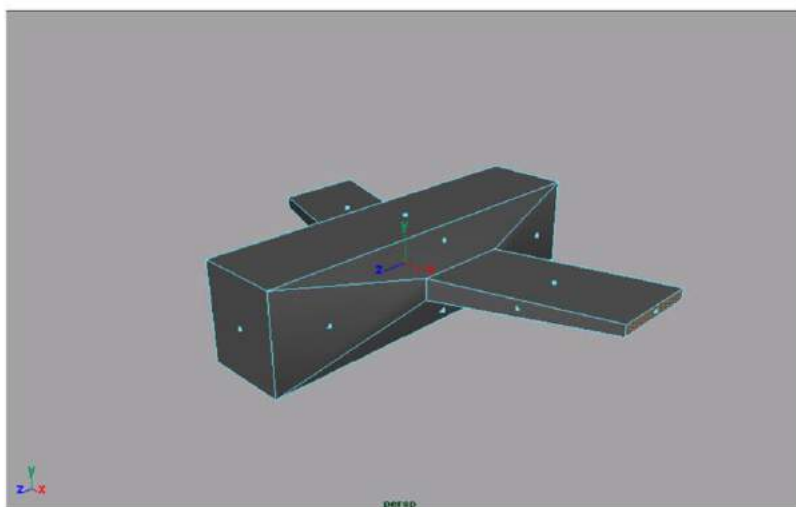
Úrhajó geometria



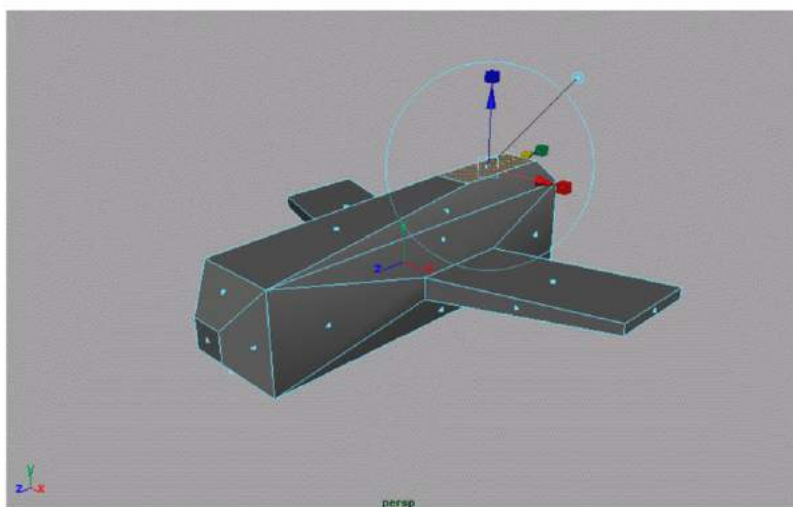
Úrhajó geometria



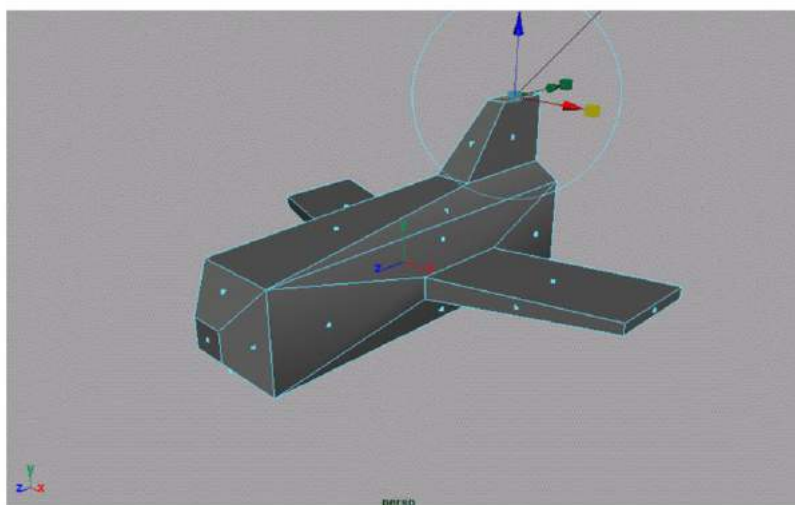
Úrhajó geometria



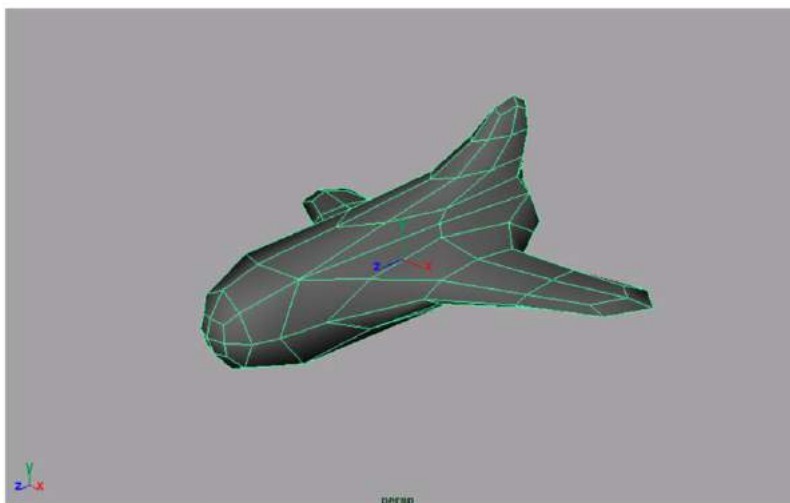
Úrhajó geometria



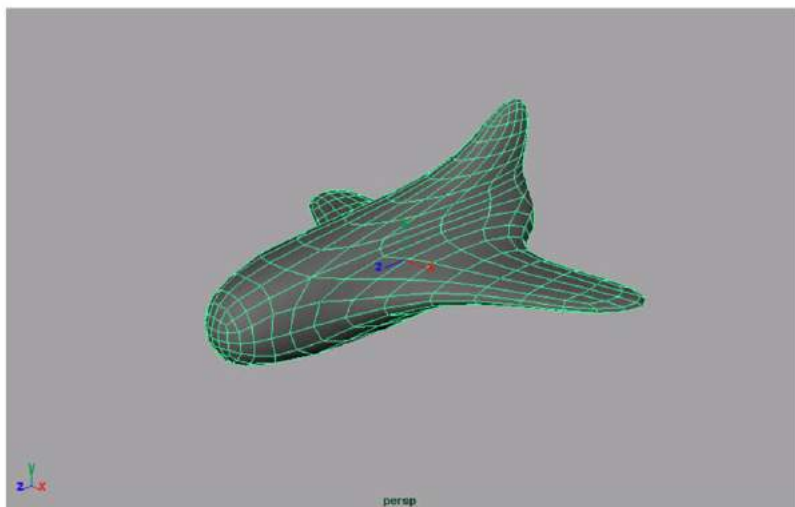
Úrhajó geometria



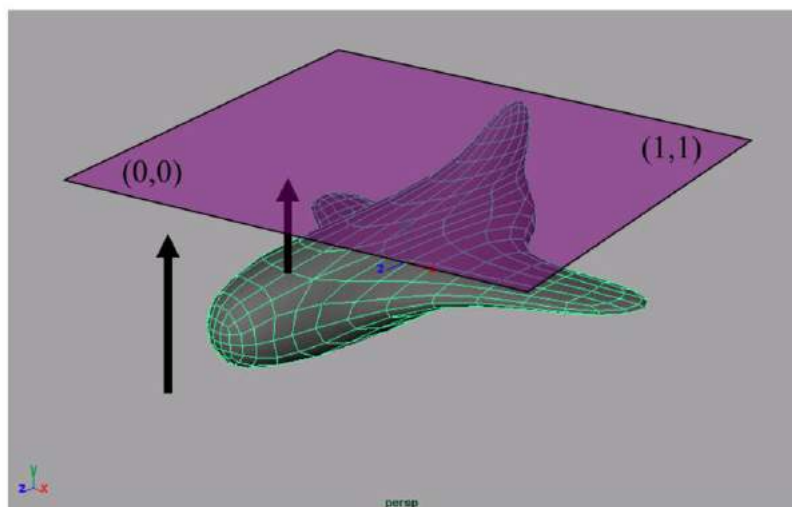
Úrhajó geometria



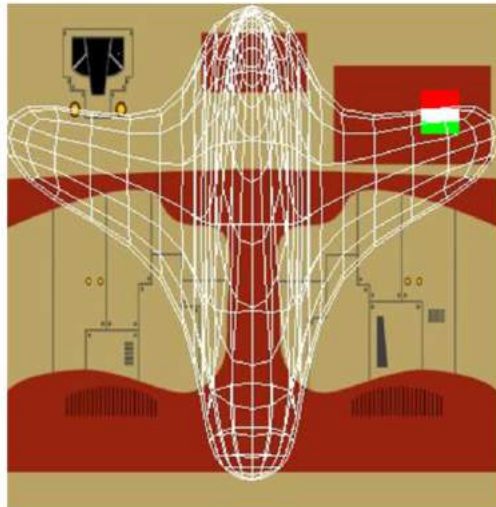
Úrhajó geometria



Textúrához paraméterezés



Textúrához paraméterezés



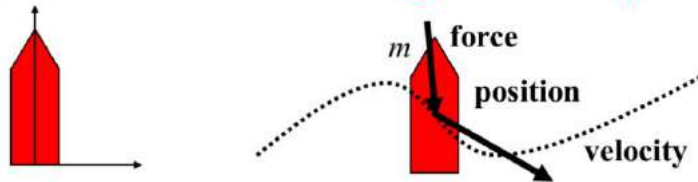
Textúrázott űrhajó



OBJ formátumban

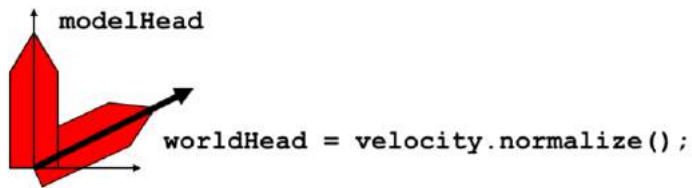
```
v -0.708698 -0.679666 2.277417
v 0.708698 -0.679666 2.277417
v -0.735419 0.754681 2.256846
...
vt 0.510655 0.078673
vt 0.509594 0.070000
vt 0.496429 0.079059
...
vn -0.843091 0.000000 0.537771
vn -0.670151 -0.543088 0.505918
vn -0.000000 -0.783747 0.621081
...
f 65/1/1 37/2/2 62/3/3 61/4/4
f 70/8/5 45/217/6 67/218/7 66/241/8
f 75/9/9 57/10/10 72/11/11 71/12/12
...
```

Animate: Newton mozgástörvényei



```
void Ship :: Animate( float dt ) {  
    acceleration = force/m;  
    velocity += acceleration * dt;  
    pos += velocity * dt;  
}  
  
void Ship :: Draw(RenderState state) {  
    state.M = Translate(pos.x, pos.y, pos.z);  
    shader->Bind(state);  
    geometry->Draw();  
}
```

Orientáció beállítása



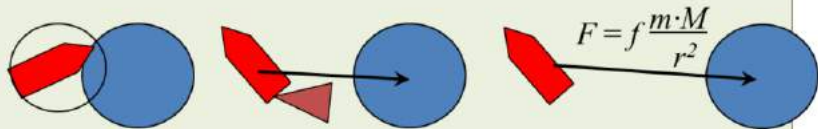
```
void Ship :: Draw(RenderState state) {  
    vec3 modelHead( 0, 0, 1 );  
    vec3 worldHead = velocity.normalize();  
    vec3 rotAxis = cross(modelHead, worldHead);  
  
    float rotAng = acos(dot(worldHead, modelHead));  
    state.M = Rotate(rotAng,rotAxis.x,rotAxis.y,rotAxis.z) *  
               Translate(pos.x, pos.y, pos.z);  
    shader->Bind(state);  
    geometry->Draw();  
}
```

Ship :: Control

```

void Ship :: Control( float dt ) {
    force = vec3(0, 0, 0);
    for (GameObject * obj : objects) {
        if (dynamic_cast<Planet*>(obj)) {

```



```

    }
    if (dynamic_cast<Avatar*>(obj)) {

```

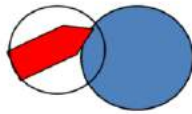


```

    }
}

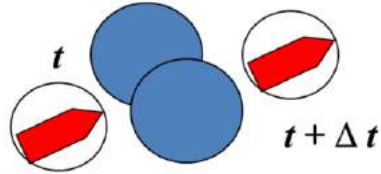
```

Ütközésdetektálás: lassú objektumok



adott t

Probléma, ha az objektum gyors

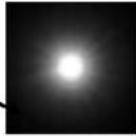


```
dist = obj1.pos - obj2.pos  
min = obj1.BoundingRadius() + obj2.BoundingRadius()  
if (dist.Length() < min) Collision!
```

Foton torpedó

- Nagyon komplex geometria
- Hasonló kinézet minden irányból
- Könnyebb a képét használni

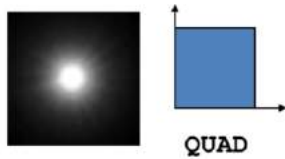
átlátszó



- Ütközésetektálás = gyors mozgás

Billboard

Egyetlen félig átlátszó textúra egy téglalapon



```

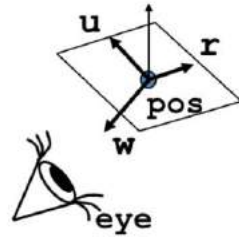
void Bullet :: Draw(RenderState state) {
    Vector w = eye - pos;
    Vector r = w % Vector(0, 1, 0);
    Vector u = r % w;
    r = r.normalize() * size;
    u = u.normalize() * size;

    glEnable(GL_BLEND);          // átlátszóság
    glBlendFunc(GL_SRC_ALPHA, GL_ONE); // hozzáadás

    state.M = mat4(r.x,  r.y,  r.z,  0,
                  u.x,  u.y,  u.z,  0,
                  0,    0,    1,    0,
                  pos.x, pos.y, pos.z, 1);

    shader->Bind(state);
    geometry->Draw();
    glDisable(GL_BLEND);
}

```

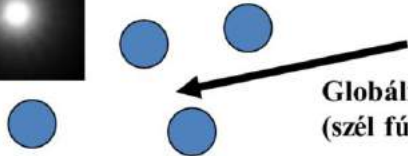


Robbanás

- Nagyon komplex geometria
- Hasonló kinézet minden irányból
- Plakátgyűjtemény
- Részecske rendszer



Részecske rendszerek



Globális erőter
(szél fújja a füstöt)



Véletlen
Kezdeti
értékek



pos:
velocity:
acceleration:
lifetime
age:
size, dsize:
weight, dweight:
color, dcolor:

pos += velocity * dt
velocity += acceleration * dt
acceleration = force / weight

age += dt; if (age > lifetime) Kill();
size += dsize * dt;
weight += dweight * dt
color += dcolor * dt

Robbanás paraméterei



```
pos = center; // kezdetben fókuszált
lifetime = Rand(2, 1);

size = 0.001; // kezdetben kicsi
dsize = Rand(0.5, 0.25) / lifetime;

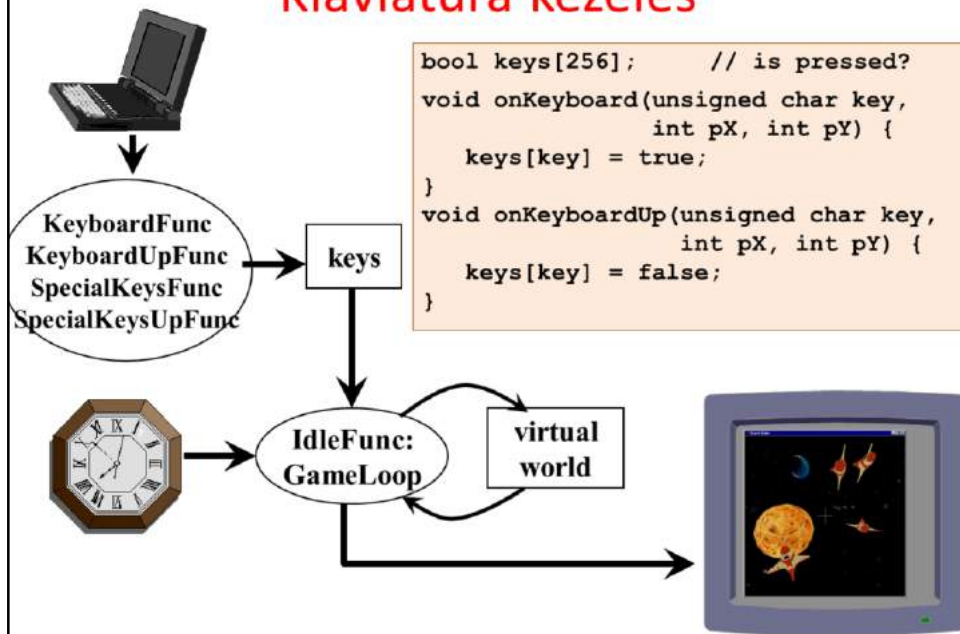
velocity = Vector(Rand(0,0.4),Rand(0,0.4),Rand(0,0.4));
acceleration = Vector(Rand(0,1),Rand(0,1),Rand(0,1));

// Planck törvény: sárga átlátszatlanból vörös átlátszóba
color = Color(1, Rand(0.5, 0.25), 0, 1);
dcolor = Color(0, -0.25, 0, -1) / lifetime;
```

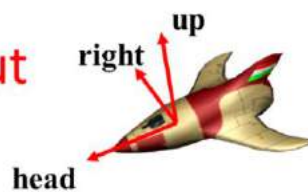
Avatár

- A viselkedését a klaviatúra vezérli:
 - ProcessInput
- A helye és iránya viszi a kamerát
 - SetCameraTransform
- Olyan mint egy űrhajó, de nem rajzoljuk
 - Control: gravitáció, lövedék ütközés

Klaviatúra kezelés



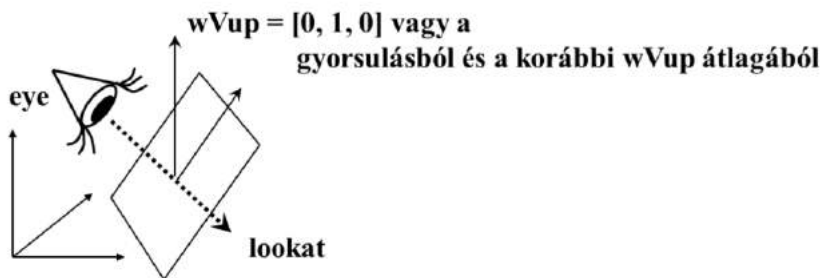
Avatar :: ProcessInput



```
Avatar :: ProcessInput() {  
    if ( keys[' '] ) // fire!  
        objects.push_back(new Bullet(pos, velocity));  
  
    // Kormányzás: az avatar koordinátarendszerében!  
    vec3 head = velocity.normalize( );  
    vec3 right = cross(wVup, head).normalize();  
    vec3 up = cross(head, right);  
  
    if (keys[KEY_UP])    force -= up;  
    if (keys[KEY_DOWN])  force += up;  
    if (keys[KEY_LEFT])  force -= right;  
    if (keys[KEY_RIGHT]) force += right;  
}
```

Avatar::SetCameraTransform

```
Avatar :: SetCameraTransform(RenderState& state) {  
    Camera camera(pos, pos + velocity, wVup,  
                  fov, asp, fp, bp);  
    state.V() = camera.V();  
    state.P() = camera.P();  
}
```



Egy földi lövöldözős játék

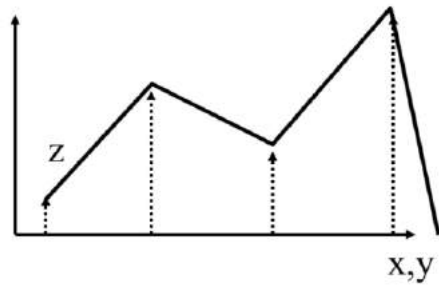
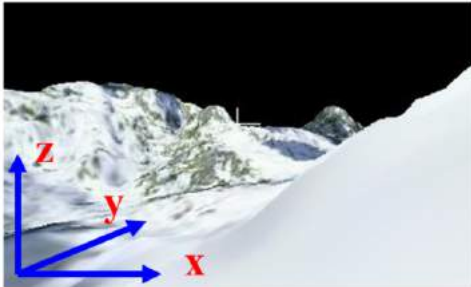


Terepek

- Komplex geometria
 - magasságmező
- Bonyolult textúra
- Nem gondolkodik
- Nem mozog
- Ütközés detektálás kell
- Megemeli az objektumokat



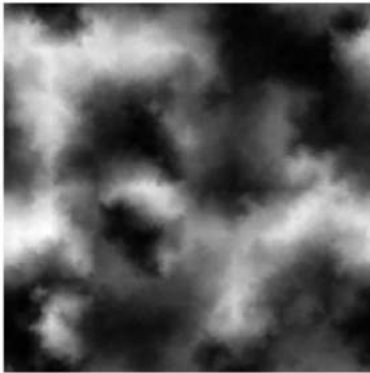
Terep geometria



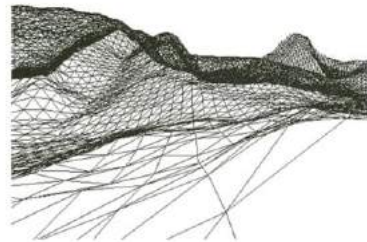
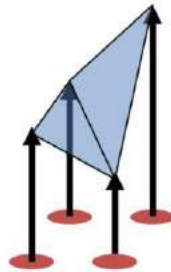
$$z = \text{height}(x,y)$$

Magasságmező:
Diszkrét minták +
Lineáris interpoláció

Diszkrét minták



Magasság mező



Háromszög háló

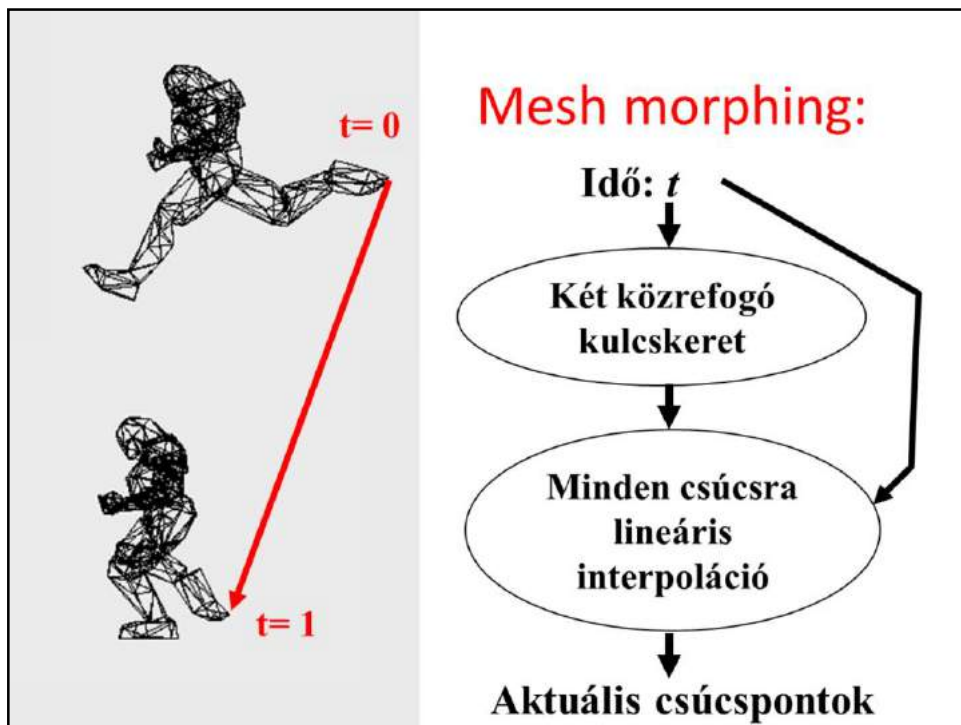
Ellenség

- Animált geometria
 - Kulcskeretekkel (clip-enként)
 - Áll, fut, támad, meghal
 - poligonháló deformáció
- Textúrák (animált)
- AI
- Ütközés detektálás



Kulcskeret animáció: futás





Futás poligonháló deformációval



+ pozíció animáció:

position += velocity * dt



Mozgás definíció

- Clip-ek definíciója kulcskeretekkel
- Összes clip összes kulcskeretek fájlban: MD2, MD3
- Tipikus clip-ek:
 - Run, stand, attack, die, pain, salute, crouch, wave, point, taunt, etc.

Clip-ek



▶
All
40 kulcskeret

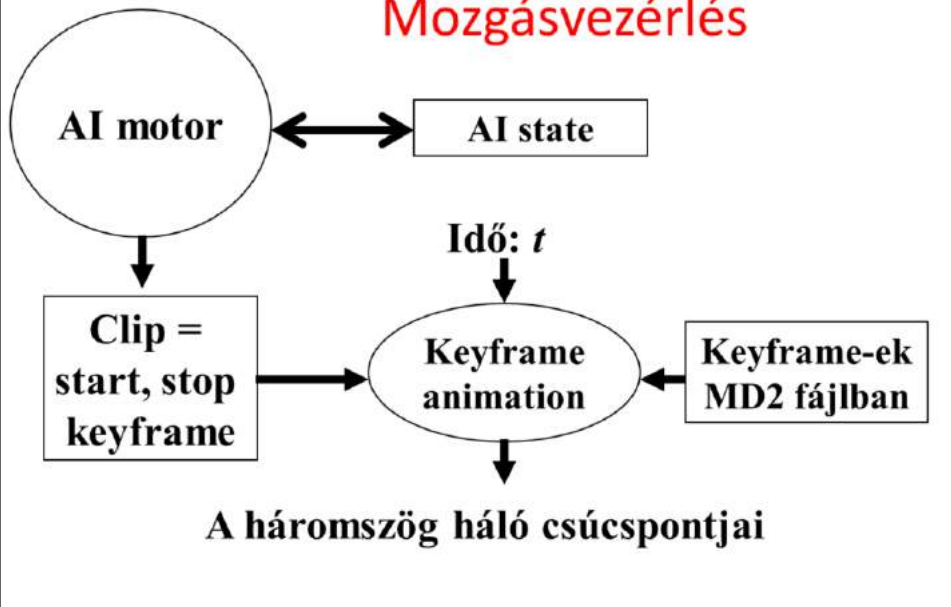


▶
Fut
5 kulcskeret

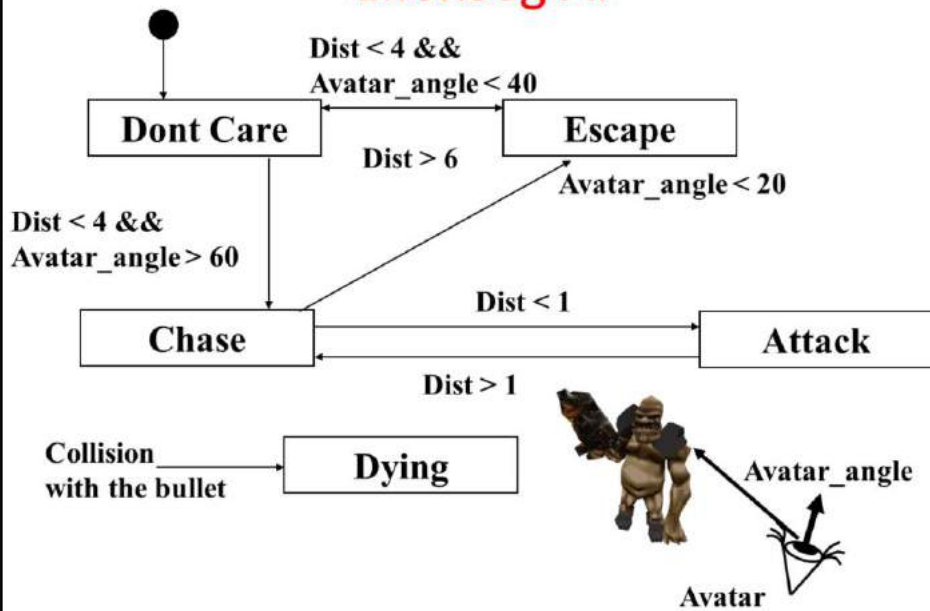


▶
Szalutál
11 kulcskeret

Mozgásvezérlés

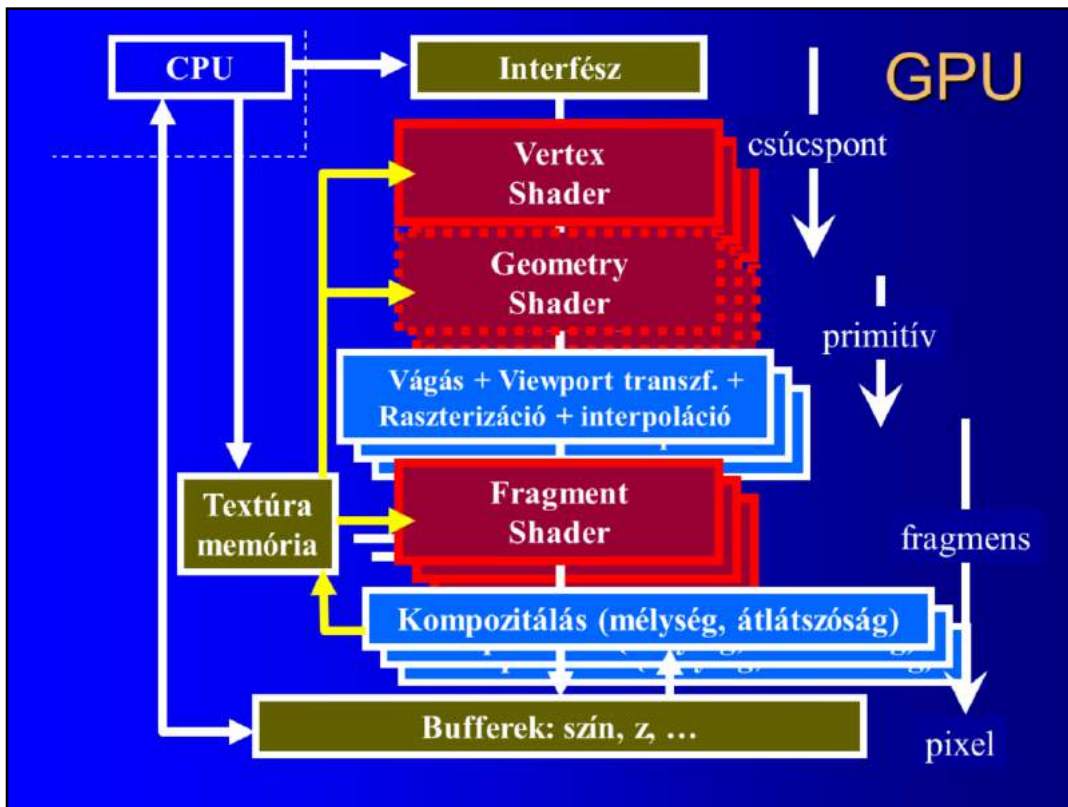


Ellenség AI



GPU

Szirmay-Kalos László



By the end of the last century, GPUs were direct hardware implementations of the incremental image synthesis algorithm, or the OpenGL pipeline. Vertices arriving from the CPU are processed by transforming them to normalized device space (matrix-vector multiplication), and – if lighting is enabled – vertices and normals are also transformed to camera space where the diffuse+Phong-Blinn illumination formula is evaluated replacing the vertex color with the computed result. In normalized device space triangles are clipped and having transformed the vertices to screen space, they are rasterized while vertex properties (color and texture coordinates) are interpolated for every internal pixel. If texturing is enabled, the texture memory is addressed by interpolated texture coordinates and the interpolated color is replaced (or modulated) by the color fetched from the texture. The pixel color goes through the compositing phase, where alpha blending and depth testing take place, and is finally written into the frame buffer. OpenGL makes a clear distinction between pixels that are in the frame buffer, and call them **pixels**, and candidate pixels (pixel-wanna-be), called **fragments**, that enter the composition phase and hopes that it will pass the depth test and will be written into the buffer.

Note that this already was a parallel hardware. On the one hand, it is a **pipeline** since while the pixels of a triangle are rasterized and textured, the vertices of the subsequent triangles are transformed and illuminated. On the other hand, this is also **parallel**, since vertices and pixels are processed independently, so multiple

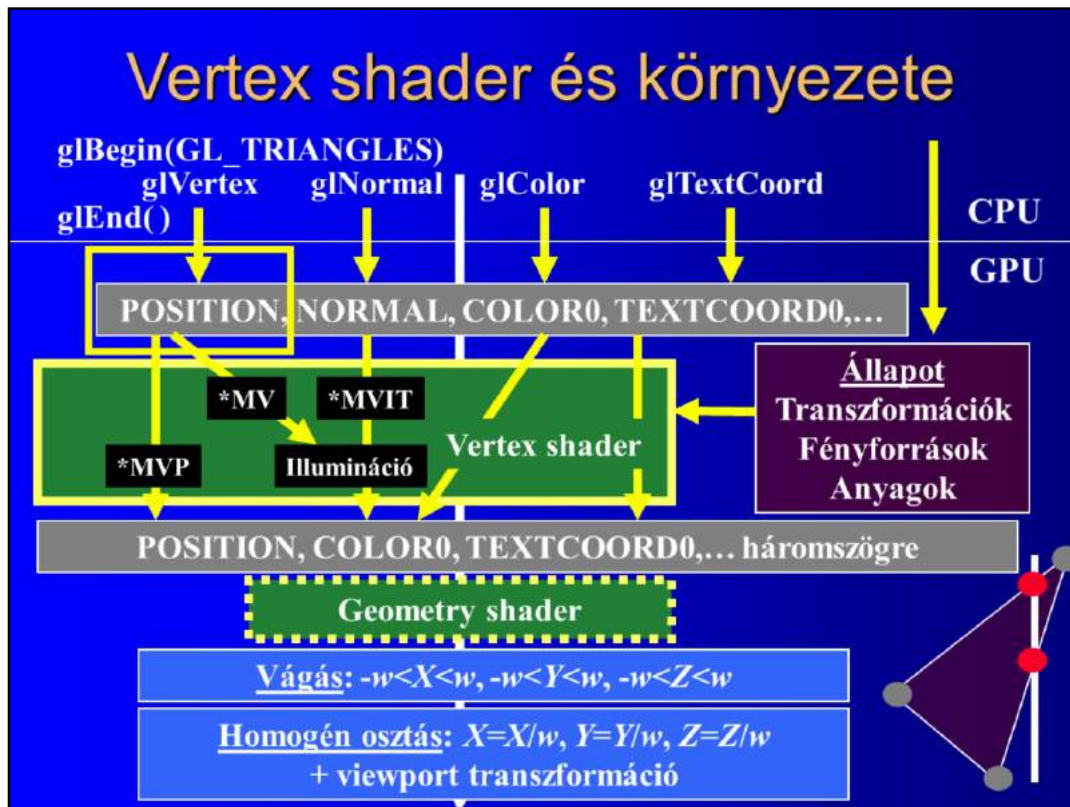
transformation+lighting units and texturing units can run in parallel.

At the turn of the new century, two stages of this pipeline became programmable, the vertex transformation+lighting unit, which is called later as **vertex shader**, and the texturing unit, which got the name **fragment shader** (or pixel shader). A few years later a new processing element, called the **geometry shader** was introduced, which processes primitives (e.g. points, triangles or line segments) and may change the topology of these primitives. For example, when a point is processed, a triangle fan may be output, or a triangle may result in a polyline). The geometry shader has no OpenGL interpretation, it is a more advanced issue. More recently, new shader stages responsible for tessellation have been included between the vertex and the geometry shaders.

Shaders may read the texture memory and the rendering results may also be directed to the texture memory instead of the frame buffer (this feature is called **render-to-texture**). However, a texture may only be read only or write only at a time. In a pass, the texture of the **render target** is written, but cannot be read back (this way we can get rid of synchronization problems and no write cache is necessary).

The geometry shader may also write out data that is fed back in a later pass to the vertex shader.

Note that with the introduction of the programability of these stages, their interface and other stages remained fixed. So, for example, we cannot modify the rasterization or clipping algorithm, and cannot say that a fragment shader will not produce fragment colors (and optionally depth). Note also that the processing of vertices, primitives and fragments is still independent, we cannot establish dependencies in a pass.



Let us explore the GPU starting at the beginning where the CPU feeds it during a pass and where the vertex shader processes the vertices. When, in a pass, `glNormal`, `glTexCoord`, or `glColor` functions are executed, the parameters of these functions will be written into the input registers of the vertex shader. The vertex shader is triggered with the modification of the input `POSITION` register occurring during a `glVertex` call. The triggered vertex shader executes its program for the vertex assuming that its properties are in other registers (`NORMAL`, `COLOR0`, `TEXCOORD0`, etc.). The vertex shader should write out the modified vertex properties into its output registers (`POSITION`, `COLOR0`, `TEXCOORD0`). If we one to simulate standard OpenGL operation, then the input position is multiplied by the concatenated `MODELVIEW` and `PROJECTION` transformations (matrix `MVP`), and if illumination is disabled, then the input `COLOR0` and `TEXCOORD0` registers are copied to the output `COLOR0` and `TEXCOORD0` registers, respectively. If illumination is enabled, the input `POSITION` is multiplied with the `MODELVIEW` (`MV`), the input `NORMAL` is multiplied with its inverse-transpose (`MVIT`), and the illumination formula is evaluated in camera space, writing out this result to the output `COLOR0` instead of copying the input `COLOR0`. During this computation, the vertex shader needs "global variables", called uniform variables that are constant during the pass, like transformation matrices, light source and material definitions.

When the vertices (together with vertex properties) are available for a primitive

(when a triangle is processed, we wait for 3 vertices), the geometry shader processes the triangle. We shall assume its default operation, which is just the copy of its input to its output.

“Standard” vertex shader (Cg, Shader Model 3.0)

	<code>void main(</code>	
	<code>in float4 position</code>	<code>: POSITION,</code>
Változó bemenet	<code>in float3 normal</code>	<code>: NORMAL,</code>
	<code>in float4 color</code>	<code>: COLOR0,</code>
	<code>in float2 texcoord</code>	<code>: TEXCOORD0,</code>
Uniform	<code>uniform float4x4 modelviewproj</code>	<code>: state.matrix.mvp,</code>
Változó kimenet	<code>out float4 hposition</code>	<code>: POSITION,</code>
	<code>out float4 ocolor</code>	<code>: COLOR0,</code>
	<code>out float2 otexcoord</code>	<code>: TEXCOORD0)</code>
	<code>{</code>	
	<code>hposition = mul(modelviewproj, position);</code>	
	<code>otexcoord = texcoord;</code>	
	<code>ocolor = color;</code>	
	<code>}</code>	
		<code>glDisable(GL_LIGHTING);</code>
Mezők elérése:		
	<code>v.x, v.y, v.z, v.w, v.xy, v.wxx</code>	
	<code>c.r, c.rgb, c.ar, ...</code>	

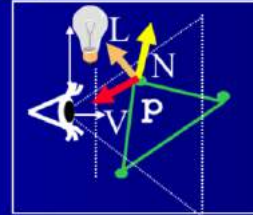
Our first vertex shader program simulates the behavior of OpenGL when lighting is disabled. A vertex shader is the program run for a single vertex and it computes the content of the output registers (POSITION, COLOR0, TEXCOORD0, etc.) from the input registers, called varying input and from the state called uniform input. All registers are of float4 type, so they can hold four float variables.

Registers have fixed names, but we can refer to them as arbitrary variable names. Registers may be declared not only float4, but also float, float2, float3, which means that only a part of the register is utilized. We can refer to the fields of a float4 variable similarly to struct field reference.

When OpenGL lighting is disabled, the input point should be transformed to normalized device space, which is a matrix vector multiplication with the modelviewproj matrix (this is passed as a uniform parameter and is the combination of OpenGL's MODELVIEW and PROJECTION). In Cg, the multiplication of an at most 4x4 matrix and a 4 element vector is a single instruction (**mul**). The texture coordinates and the vertex color are copied.

glEnable(GL_LIGHTING);

```
main( in float4 position : POSITION,
      in float4 normal   : NORMAL,
      uniform float4x4 modelview, modelviewIT, modelviewproj, Ⓢ
      uniform float4 lightpos, Idiff, Iamb, Ispec, Ⓢ
      uniform float4 em, ka, kd, ks, Ⓢ
      uniform float shininess,
      out float4 hposition: POSITION,
      out float4 ocolor   : COLOR0
  ) {
    hposition = mul(modelviewproj, position);
    float3 N = mul(modelviewIT, normal).xyz;
    N = normalize(N); // glEnable(GL_NORMALIZE)
    float4 p = mul(modelview, position);
    float3 L = normalize(lightpos.xyz/lightpos.w - p.xyz/p.w);
    float costheta = dot(N, L); if (costheta < 0) costheta = 0;
    float3 V = normalize(-p.xyz);
    float3 H = normalize(L + V);
    float cosdelta = dot(N, H); if (cosdelta < 0) cosdelta = 0;
    ocolor = em + Iamb * ka + Idiff * kd * costheta +
              Ispec * ks * pow(cosdelta, shininess);
  }
```



To simulate what OpenGL would do when the illumination is enabled, the vertex shader should also transform the point and the shading normal to camera space and compute the reflected radiance there. We need a lot of uniform parameters describing transformation matrices, light sources, and material properties. The smiley indicates that the syntax in these lines is incorrect, we should have repeated the type (e.g. **uniform float4x4**) for every variable (but there is not enough space on this slide).

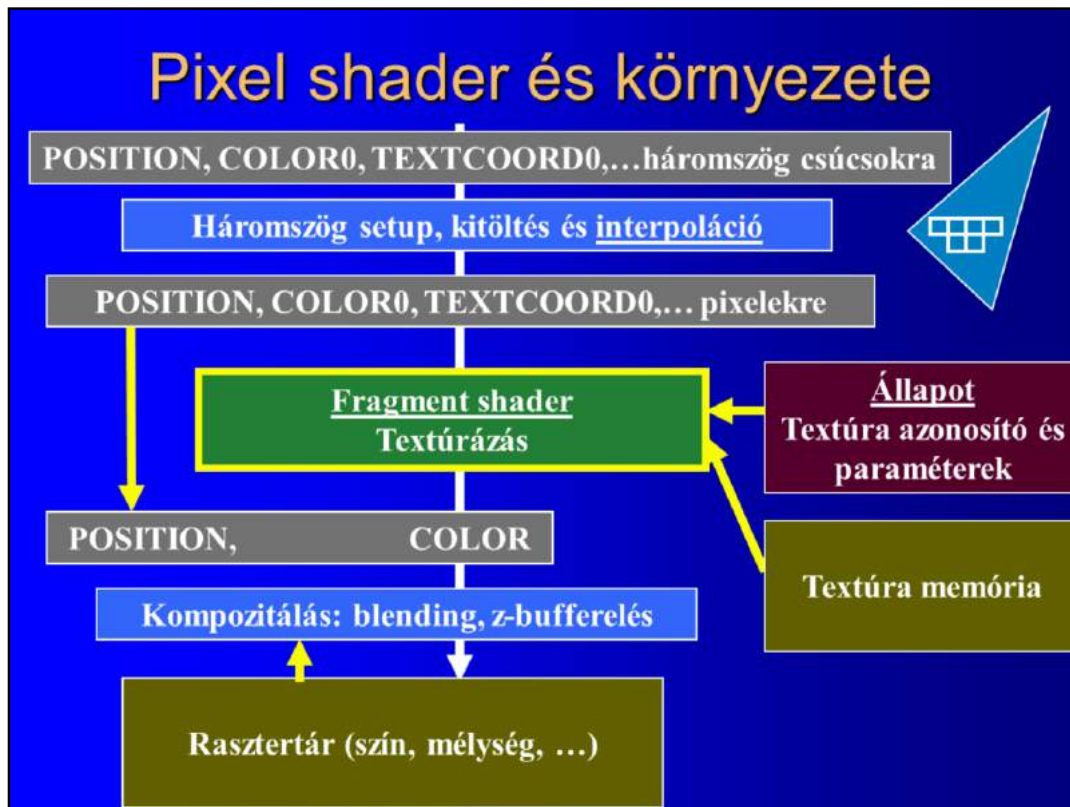
For the sake of simplicity, we assume that only one light source exists.

The program first transforms the point to normalized device space as in the previous case (this line is a part of almost all vertex shader programs). Then the normal vector is transformed the camera space, and then is normalized with **normalize** Cg instruction, i.e. scaled to have unit length (recall that this line corresponds to the enabling of the GL_NORMALIZE switch).

The point is also transformed to camera space and expressed in Cartesian coordinates (cpo). Illumination direction L is computed from the position of the light source and the point (both of them are in camera space). Geometry factor $\cos\theta$ is computed as a dot product for diffuse reflection with the **dot** Cg function.

As in camera space, the eye is in the origin, so the viewing direction of point `cpos` is $-\text{cpos}$.

Similarly, halfway vector `H` is computed, and the Phong-Blinn specular term is evaluated with a C-like power (**pow**) function. Emission, ambient reflection, diffuse reflection and specular reflection are added to get the output color. Note that `+` and `*` are evaluated as needed for spectra in Cg (and we also have **dot** and **cross** for vectors).



Rasterization produces fragments that are inside the projection of the 2D triangle and interpolates all properties. For every fragment, the fragment shader is called to compute the final color from fragment properties. Note that the rasterization decides which fragments should be changed, and the fragment shader computes just the color (optionally the depth) of the given fragment, while it is NOT allowed to modify the target pixel (this is why we drew the arrow of the POSITION outside of the fragment shader). The classical function of the fragment shader is the texture lookup and optional modulation with the interpolated color.

The computed fragment goes into the compositing phase, which applies alpha blending or depth buffering if enabled.

“Standard” pixel shader

```
void main( in float4 color : COLOR0,
           out float4 ocolor : COLOR )
{
    ocolor = color;
}

glDisable(GL_TEXTURE_2D);
```

```
void main( in float2 texcoord      : TEXCOORD0,
           in float3 color        : COLOR0,
           uniform sampler2D texture_map,
           out float4 ocolor      : COLOR )
{
    ocolor = tex2D(texture_map, texcoord) * color;
}

glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV,
          GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

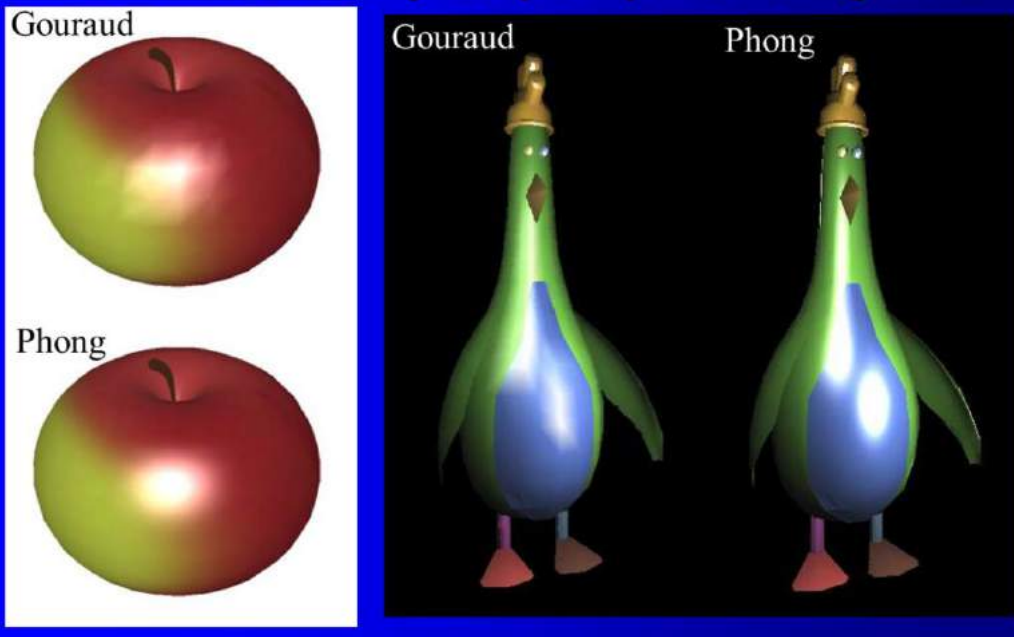
Again, we first implement the same functions that would be enforced by the classic OpenGL operation. If texturing is disabled, the output fragment color is color interpolated from the vertex colors, which is in the input COLOR0 register.

If texturing is enabled, the texture map is looked up with the interpolated texture coordinates passed in TEXCOORD0, and the fragment color is the fetched value. The texture id is a uniform parameter of the fragment shader, of type **sampler2D**. This name indicates that a texture object is more than just an array of texels, it also stores whether filtering or mipmapping is enabled and how texture coordinates outside the [0,1] range should be handled.

The texture fetch is done with **tex2D** Cg function, which returns the color that is already filtered if filtering is enabled.

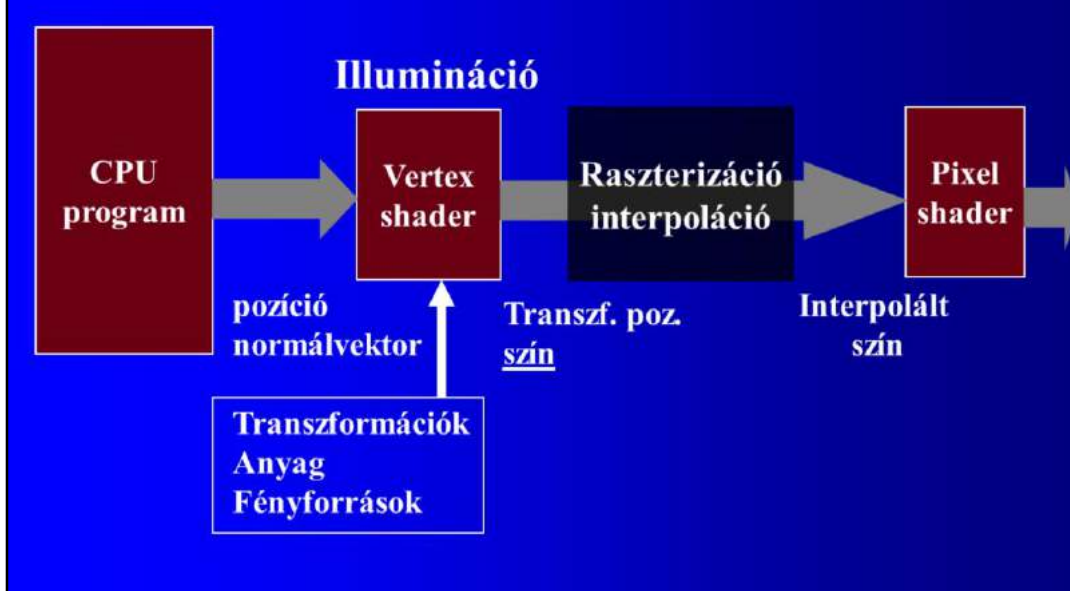
In case of modulate texture environment, the interpolated color and the texel are multiplied.

Gouraud helyett Phong árnyalás Per-vertex helyett per-pixel árnyalás



Our first Cg program is going to be the implementation of Phong shading (per-fragment lighting), which is not available in OpenGL implementing Gouraud shading (per-vertex lighting).

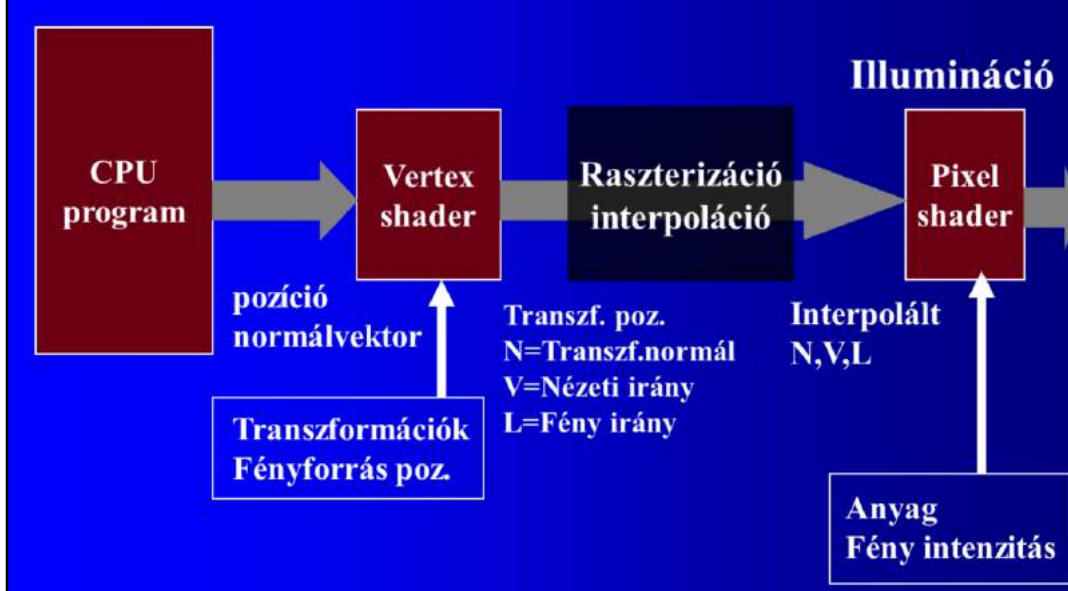
Gouraud (per-vertex) árnyalás



When the GPU is programmed, we develop three programs in parallel: a CPU program that looks similar to a standard OpenGL application and is written, e.g. in C++, a Vertex shader and a Fragment shader program written in Cg.

The rendering pass of the CPU program initiates `glNormal` and `glVertex` calls that will send vertices with their normals to the vertex shader. In standard OpenGL, Gouraud shading would calculate the point transformed to normalized device space and the color based on the illumination formula in the vertex shader program. The fragment shader would just pass the interpolated vertex color to the compositing phase.

Phong (per-pixel) árnyalás



To execute Phong shading, the illumination evaluation should be transported to the fragment shader. On the CPU level, the program still passes normals and vertices. The vertex shader still transforms the point to normalized device space, but instead of evaluating the illumination formula, it just computes the vectors (normal, view and lighting) needed by the evaluation. These vectors are interpolated during rasterization, and the fragment shader evaluates the illumination formula using the interpolated vectors.

Transformations (ModelViewProjection, ModelView, ModelViewIT) are still needed in the vertex shader, as well as the light source position. However, material properties and light intensity are used in the fragment shader, so these will be uniform parameters there.

Programok (lábbal hajtós megoldás)

• .cpp CPU program:

- Shader environment létrehozás
- GPU képesség beállítás (profile)
- Vertex/fragment program betöltés és fordítás: CREATE
- Vertex/fragment program átadás a GPU-nak: LOAD
- Vertex/fragment program kiválasztás: BIND
- Uniform vertex/fragment input változó létrehozás
- Uniform vertex/fragment változó értékadás
- Változó input változó értékadás (glVertex, glColor, glTexCoord)

• .cg vertex program

- Fragment program változó input + homogén pozíció

• .cg fragment program

- Szín kimenet

We shall put together the programs according to the requirements of the Cg toolkit, which was the first and is the most complicated solution (a more comfortable alternative would be the GLSL framework). We insist on using the Cg toolkit, because it does not hide details, so it reveals what is going on. This approach clearly separates the three components, the CPU program, the vertex program, and the fragment program, and expects them in separate files.

The CPU program is responsible for compiling and loading the GPU programs supplied in two additional files (named usually with .cg extension).

The CPU program starts with the definition of the **Shader environment**, which allocates a structure in the CPU memory where GPU related information is stored (similar to opening a file). GPUs are advancing, currently we have Shader Model 1..5 class GPUs, so the compilation should also depend on what the GPU in the computer is capable of, which is specified during profile setting. **Creating** a vertex (or fragment) program usually means the loading the source code from a file into the Shader environment structure of the CPU memory and its compilation based on the profile setting. **Loading** the compiled GPU program means the transfer of the executable code to the GPU memory. A GPU may store different vertex (or fragment) programs at a time, so we should specify which is the active one, which is called **binding** (the meaning and the name are similar to those of texture mapping).

The CPU can send information to the shader via **uniform variables**, which have a representative on the CPU and on the GPU as well, so during their creation, the correspondence should also be established.

Before starting rendering, the global variables of the shaders, i.e. the uniform variables should be set. Rendering is the executing a pass, including `glNormal`, `glColor`, `glTexCoord`, and `glVertex` calls, which write their parameters into registers.

CPU program - Inicializálás

```
...
#include <Cg/cgGL.h>                                // Cg függvények

CGparameter LightPos, LightInt, Shine, Ks, Kd; // uniform par a CPU-n

int main( ) {
    CGcontext shaderContext = cgCreateContext(); // árnyaló kontextus

    CGprofile vertexProf = cgGLGetLatestProfile(CG_GL_VERTEX);
    cgGLEnableProfile(vertexProf);

    CGprogram vertexProgram = cgCreateProgramFromFile(
                                shaderContext,
                                CG_SOURCE, "vertex.cg",
                                vertexProf, NULL, NULL);

    cgGLLoadProgram(vertexProgram); // GPU-ra töltés
    cgGLBindProgram(vertexProgram); // ez legyen a futó program

    // vertex program uniform paraméterek. CPU-n Lightpos; GPU-n clightpos
    LightPos = cgGetNamedParameter(vertexProgram, "clightpos");
}
```

Vertex shader betöltés

To access Cg functions, the Cg toolkit should be downloaded and installed. The declarations are in the cgGL.h file (stands for Cg for OpenGL).

First the **shaderContext** structure is allocated, then the profile for the vertex program is set. Note that with **cgGLGetLatestProfile**, we find the most powerful profile the GPU inside the computer can offer.

The vertex program is created with **cgCreateProgramFromFile**, including loading it from a file, stating that the name of the file is vertex.cg and it is a Cg source file, and compiling it in the shaderContext according to the vertex profile. With the additional parameters of this function, we can set the **entry point** of the vertex shader program, the default is the main function like in C.

The compiled program is transferred to the GPU with **cgGLLoadProgram**, and is made active with **cgGLBindProgram**.

To implement Phong shading, the vectors needed for the illumination calculation are obtained in the vertex shader, for which we need the light source position in camera space. This information is passed from the CPU

as a uniform variable. To do that, we should define a **CGparameter** in the CPU program (we call it Lightpos), and connect it with a uniform parameter of the vertex program (which is called clightpos) using the **cgGetNamedParameter** function.

Fragmens árnyaló betöltés

```
CGprofile fragmentProf = cgGLGetLatestProfile(CG_GL_FRAGMENT);
cgGLEnableProfile(fragmentProf);

CGprogram fragmentProgram = cgCreateProgramFromFile(
    shaderContext,
    CG_SOURCE, "fragment.cg",
    fragmentProf,
    NULL, NULL);

cgGLLoadProgram(fragmentProgram); // GPU-ra töltés
cgGLBindProgram(fragmentProgram); // ez a program fusson

// fragmens program uniform paraméterek
Shine = cgGetNamedParameter(fragmentProgram, "shininess");
Kd = cgGetNamedParameter(fragmentProgram, "kd");
Ks = cgGetNamedParameter(fragmentProgram, "ks");
LightInt = cgGetNamedParameter(fragmentProgram, "lightint");

... OpenGL inicializálás
```

The fragment shader is set in a similar way, first the profile is enabled, then the program is loaded from file and compiled, then uploaded to the GPU, and finally set to run with binding operation.

The fragment shader is controlled via three uniform parameters that have Lightint, Shine, Kd, and Ks names on the CPU and lightint, shininess, kd, and ks on the GPU.

CPU program - OpenGL display

```
void Display( ) {
    glLoadIdentity(); // állapot (implicit uniform) paraméterek beállítása
    gluLookAt(0, 0, -10, 0, 0, 0, 0, 1, 0);
    glRotatef(angle, 0, 1, 0);

    // explicit uniform paraméterek beállítása
    cgGLSetParameter4f(LightPos, 10,20,30,1); // clightpos
    cgGLSetParameter3f(LightInt, 1, 1, 1);    // lightint
    cgGLSetParameter1f(Shine, 40);            // shininess
    cgGLSetParameter3f(Kd, 1, 0.8, 0.2);      // kd
    cgGLSetParameter3f(Ks, 2, 2, 2);          // ks

    // változó paraméterek, a PASS
    glBegin( GL_TRIANGLES );
    for( ... ) {
        glNormal3f(nx, ny, nz); // NORMAL regiszter
        glVertex3f(x, y, z);    // POSITION regiszter + trigger
    }
    glEnd();
}
```

OpenGL rendering usually takes place in the Display callback. Here we set the transformation matrices, which are immediately uploaded to the GPU by OpenGL, where our own shader programs may also access them as uniform parameters. User defined uniform parameters are set by **cgGLSetParameter[1..4]f**, depending on how many float fields this parameter has.

Finally, a conventional OpenGL pass is executed, which sets input register NORMAL when glNormal is called and POSITION when glVertex is called. Function glVertex also gets the vertex shader to execute its program once.

Phong árnyalás: vertex shader

```
void main(  
    in float4 position      : POSITION,  
    in float4 normal        : NORMAL,  
    uniform float4x4 MVP    : state.matrix.mvp,  
    uniform float4x4 MV     : state.matrix.modelview,  
    uniform float4x4 MVIT   : state.matrix.modelview.invtrans,  
    uniform float4  clightpos, // fényforrás poz. kamerak-ban  
    out float4 hposition    : POSITION,  
    out float3 cnormal      : TEXCOORD0,  
    out float3 cview        : TEXCOORD1,  
    out float3 clight       : TEXCOORD2  
)  
{  
    hposition = mul(MVP, position);  
    float4 cp = mul(MV, position);  
    cnormal = mul(MVIT, normal).xyz;  
    clight = clightpos.xyz * cp.w - cp.xyz * clightpos.w;  
    cview = -cp.xyz;  
}
```



The supplied vertex position is transformed to normalized clipping phase as almost always, and additionally, in Phong shading, the vertex shader computes the vectors needed for the illumination. We obtain these vectors in camera space.

So first, the input position is also transformed to camera space using the MODELVIEW matrix. This is set by OpenGL, and we can connect our MVP uniform variable to the OpenGL state variable where this information is stored. Although very unlikely, this matrix multiplication may modify the fourth homogeneous coordinates, so after matrix-vector multiplication, the vertex position is expressed with Cartesian coordinates applying homogeneous division.

The normal vector is also transformed to camera space using the inverse-transpose of the MODELVIEW matrix (fortunately, this matrix is also in the OpenGL state) resulting in **cnormal**. The light source position in camera space is passed from the CPU as a uniform parameter. The difference of the light source position and the vertex position is the camera space illumination direction **clight**. The difference of the origin and the vertex position is the camera space viewing direction **cview**.

Vectors evaluated at the vertices should be interpolated for fragments inside the triangle's projection. So these vectors should be passed to the rasterizer as vertex

properties. Unfortunately, there is no register like “LIGHTDIR” or “VIEWDIR” (and not even “NORMAL” during rasterization), but there are many, general purpose texture coordinate registers, so we utilize them to carry and interpolate these vectors.

Phong árnyalás: fragment shader

```
void main(    in float3 N          : TEXCOORD0,
              in float3 V          : TEXCOORD1,
              in float3 L          : TEXCOORD2,
              uniform float3 lightint,
              uniform float shininess,
              uniform float3 kd,
              uniform float3 ks,
              out float3 ocolor    : COLOR )
{
    N = normalize( N );
    V = normalize( V );
    L = normalize( L );

    float3 H = normalize(V + L);
    float costheta = max(dot(N, L), 0);
    float cosdelta = max(dot(N, H), 0);
    ocolor = lightint *
              (kd * costheta + ks * pow(cosdelta, shininess));
}
```



The triangle is clipped and rasterized, when fragments inside the projection are visited. For each fragment, the fragment shader is executed that gets the interpolated values of the registers (recall that in TEXCOORD0 we passed the cnormal from the vertex shader, so no TEXCOORD0 stores the interpolated normal).

Vectors are normalized since linear interpolation make them have not unit length even if they were normalized before the interpolation. Halfway vector H is computed, then a standard diffuse + Phong-Blinn illumination formula is evaluated.

The result is written into the COLOR register, which will be sent to the frame buffer via compositing.

NPR

```
void main(    in float3 N          : TEXCOORD0,
              in float3 V          : TEXCOORD1,
              in float3 L          : TEXCOORD2,
              uniform float3 kd,
              out float3 ocolor    : COLOR )
{
    N = normalize( N );
    V = normalize( V );
    L = normalize( L );

    float costheta = dot(N, L);
    float y = (costheta > 0.5) ? 1 : 0.5;
    if (abs(dot(N, V)) < 0.2) ocolor = float3(0, 0, 0);
    else                      ocolor = y * kd;
}
```



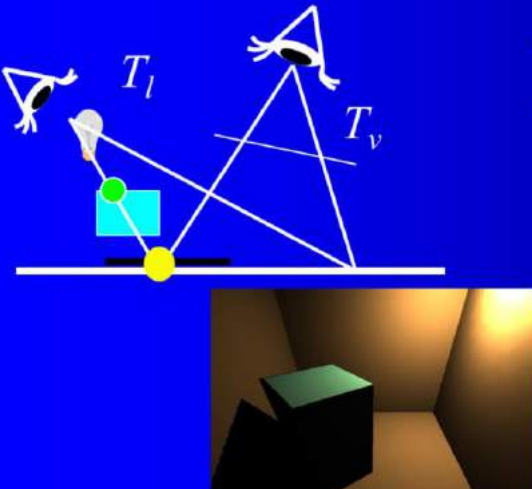
Note that we are not obliged to use the old diffuse + Phong-Blinn model, but arbitrary BRDF models can be implemented. We could even take a non-physical model, for example, use cartoon shading mimicking artistic rendering (aka Non-Photorealistic Rendering). This simple shader assumes that we have just two shades of color (or paint), so illuminated points are painted with light green, points in shadow with dark green. A black silhouette is also drawn to mimic hand-drawn images by checking where front facing and back facing surfaces meet and therefore the angle between the normal and the viewing direction is close to 90 degrees.

NPR



Lichthof Productions

Árnyék térképek



1. pass:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(lightx, lighty, lightz,
           lookatx, lookaty, lookatz,
           lupx, lupy, lupz);
```

Modellezési transzformáció
Képszintézis

Z-buffer -> textúra másolás (v. közv)

2. pass:

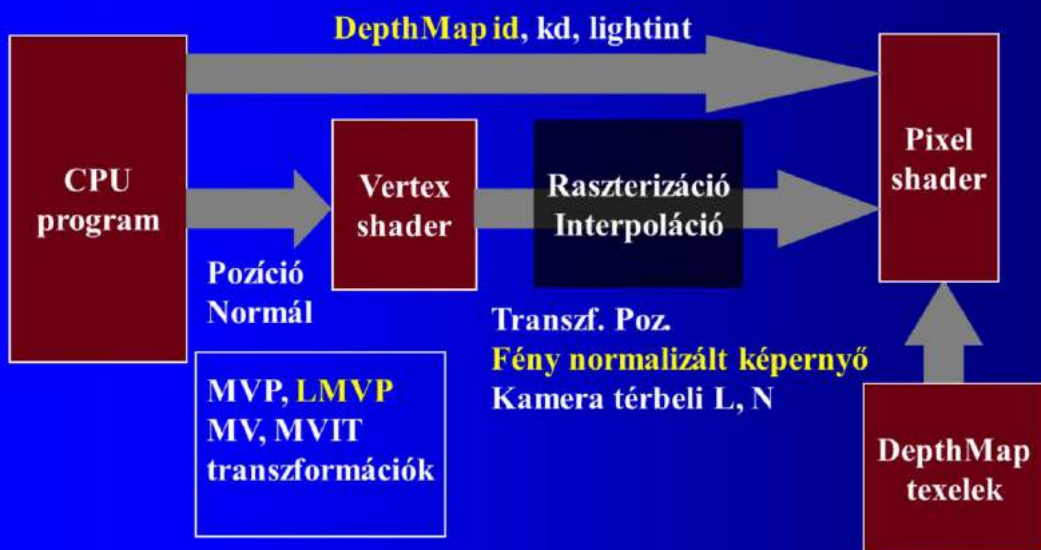
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eyex, eyey, eyez,
           lookatx, lookaty, lookatz,
           upx, upy, upz);
```

Modellezési transzformáció
Képszintézis + árnyék teszt

The second example of shader programming is the implementation of the shadow mapping algorithm. Recall that in OpenGL there is no rendered shadow since all surfaces are processed independently. Independent processing is still the feature of the programmable GPU, so shadows are still impossible in a single pass. However, they can be generated in multipass rendering.

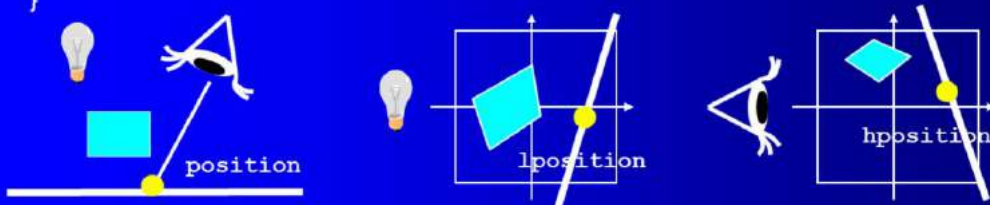
The idea of shadow mapping is that those points are in shadow that are not visible from the light source. So if we render the scene from the point of view of the source, and determine which points are visible from there, this information can be used in the second rendering pass, when the point of view is moved back to the camera.

Képszintézis + árnyékteszt



Vertex shader

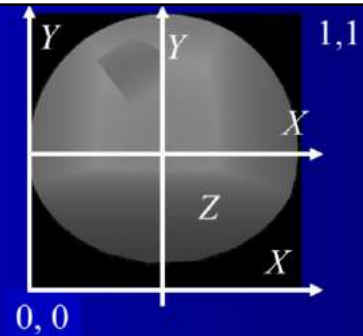
```
void main(
    in float4 position      : POSITION,
    in float4 normal        : NORMAL,
    uniform float4x4 MVP, LMVP, MV, MVIT, @
    uniform float4 clightpos,
    out float4 hPosition    : POSITION,
    out float4 lPosition    : TEXCOORD0,
    out float3 cnormal      : TEXCOORD1,
    out float3 clight       : TEXCOORD2
) {
    hPosition = mul(MVP, position); // to eye's clip space
    lPosition = mul(LMVP, position); // to light's clip space
    float4 cp = mul(MV, position); // camera space (Phong shade)
    cnormal = mul(MVIT, normal).xyz;
    clight = clightpos.xyz * cp.w - cp.xyz * clightpos.w;
}
```



Fragment shader

```

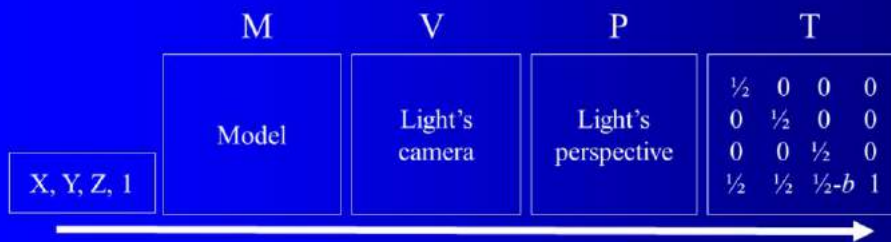
void main(
    in float4 lPosition : TEXCOORD0,
    in float3 N          : TEXCOORD1,
    in float3 L          : TEXCOORD2,
    uniform float4 lightint, kd, ☉
    uniform sampler2D depthMap,
    uniform float bias,
    out float4 ocolor    : COLOR )
{
    float3 lPosCartesian = lPosition.xyz/lPosition.w;
    float2 texcoord;
    texcoord.x = (lPosCartesian.x + 1)/2;
    texcoord.y = (lPosCartesian.y + 1)/2;
    float this_depth = (lPosCartesian.z + 1)/2 - bias;
    float stored_depth = tex2D(depthMap, texcoord);
    if (this_depth <= stored_depth) { // == kéne
        N = normalize(N); L = normalize(L);
        ocolor = lightint * kd * max(dot(N,L), 0);
    } else
        ocolor = float3(0, 0, 0);
}
    
```



Clipping
space to
Texture
space

Projektív textúrázás mátrixa

```
float3 lPosCartesian = lPosition.xyz/lPosition.w;  
  
texcoord.x = (lPosCartesian.x + 1)/2;  
texcoord.y = (lPosCartesian.y + 1)/2;  
float this_depth = (lPosCartesian.z + 1)/2 - bias;
```



Egyszerűbb árnyékteszt

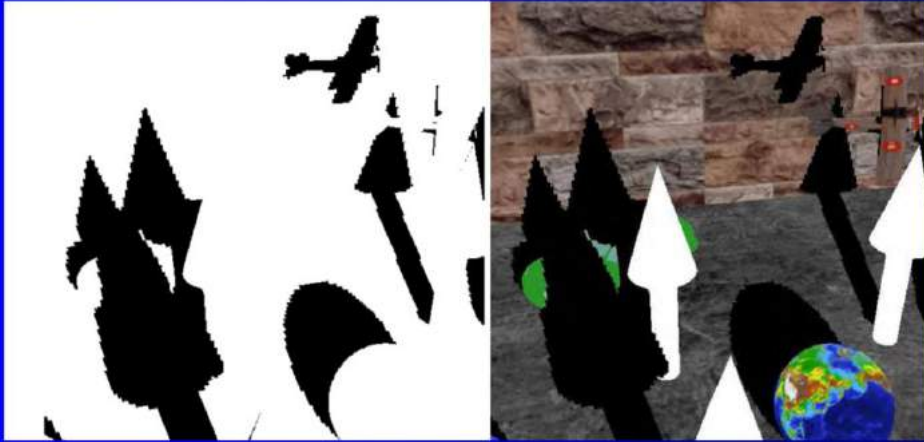
```
void SM_VS(    in float4 position    : POSITION,
               in float4 normal      : NORMAL,
               uniform float4x4 MVP, LMVPT, MV, MVIT, Ⓢ
               uniform float4 lightpos,
               out float4 hPosition  : POSITION,
               out float4 tPosition  : TEXCOORD0,
               out float3 cnormal, clight Ⓢ )
{
    hPosition = mul(MVP, position); // to eye's clip space
    tPosition = mul(LMVPT, position); // to depth texture space
    float4 cp = mul(MV, position); // camera space
    cnormal = mul(MVIT, normal).xyz;
    clight = lightpos.xyz * cp.w - cp.xyz * lightpos.w;
}

void SM_FS (    in float4 tPosition : TEXCOORD0,
               in float3 N, L Ⓢ
               uniform float4 lightint, kd, Ⓢ
               uniform sampler2D depthMap,
               out float4 ocolor      : COLOR )
{
    float costheta = max(dot(normalize(N), normalize(L)), 0);
    ocolor = lightint * kd * costheta * tex2Dproj(depthMap, tPosition);
}
```

Returns 0/1
if it is a
depth texture

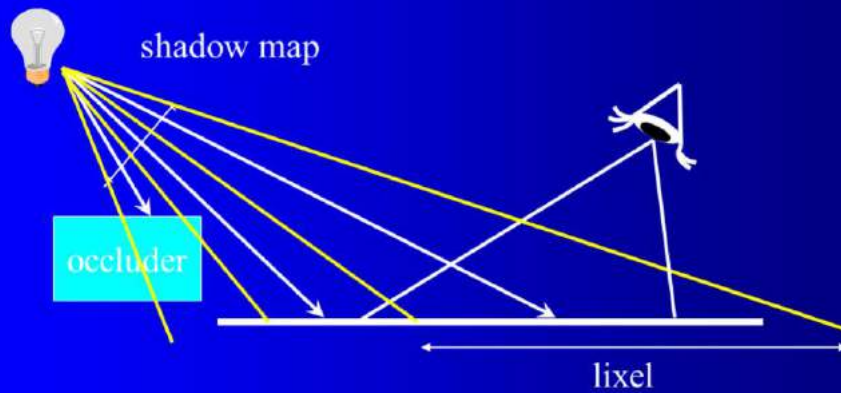
To support shadow mapping, **tex2Dproj** gets a 4 element texture address, executes the homogeneous division, compares the texel to the z component and returns to a 0/1 value indicating whether the z component is smaller than the stored texel value.

Árnyéktérkép aliasing



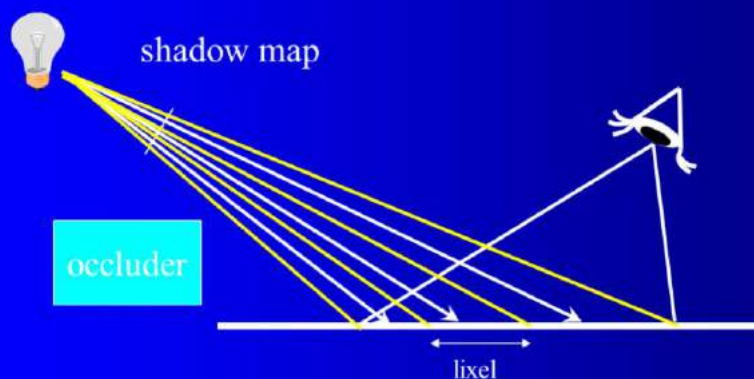
The edges of shadows obtained with the shadow mapping algorithm are jagged, which is due to the aliasing artifact.

Árnyéktérkép aliasing



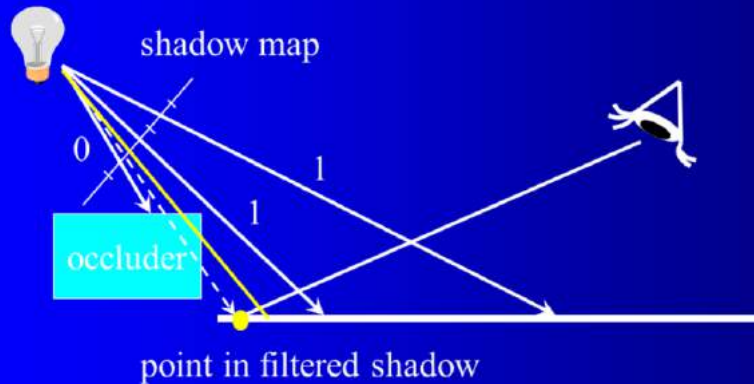
The real depth is available just at discrete points in the shadow map. If the light source has a large field of view and is far away, then a shadow map texel (called **lixel**) is mapped on a larger surface area, where only one depth value is available.

Árnyéktérkép fókuszálás



To improve shadow map quality, a simple technique is **focusing**, i.e. we set the field of view angle of the light pass as small as possible. So first, the region visible from the camera is determined, then this region is focused on during light pass.

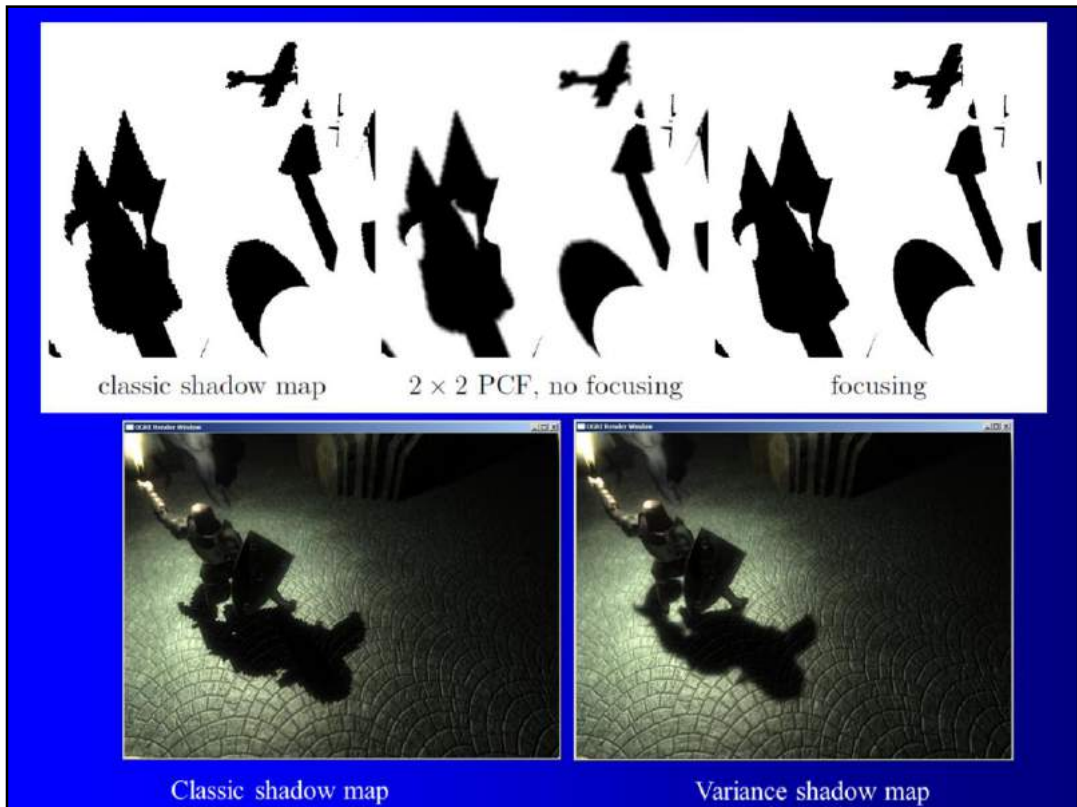
Percentage Closer Filtering



- Shadow map: depths at discrete points
- In between: depth is a random variable Z
- $P(Z > z) = \text{shadowing factor}$

The depths of points that are projected between the texel centers of the shadow map are not known, but can be treated as random variables. The objective is to estimate the probability that at this point shadowing happens and scale down the light intensity accordingly.

Instead of comparing the current depth to the depth in the center of the shadow map pixel, the point's depth is compared to four texels enclosing the current point. This results in four 0/1 values, that are bi-linearly interpolated. This method is called **percentage closer filtering (PCF)**, and is automatically supported by the **tex2Dproj** function of the GPU if the bi-linear filtering is enabled on the depth texture.



Comparing the current depth to a few depth values in the map, we can estimate this probability, using the **one-tailed version of the Chebyshev inequality**. The resulting probability may be between 0 and 1 and can be used to smooth the shadow boundary. This method is called **variance shadow maps**.

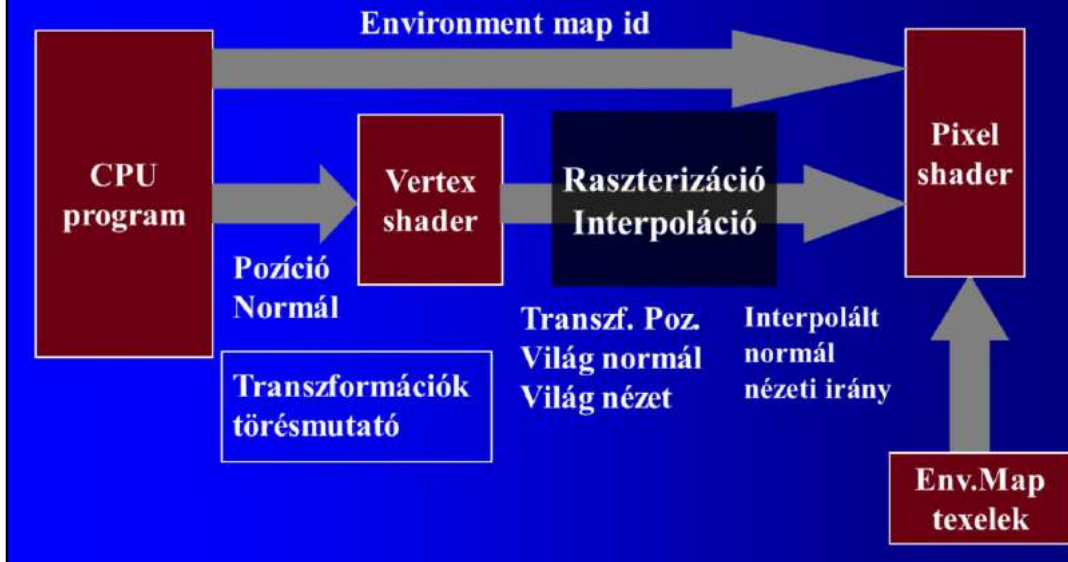


Reflection and refraction also require the consideration of the scene as a whole and do not allow completely independent shading of surface points. Thus these phenomena may only be simulated by multi-pass rendering. Suppose that we have a reflective running man. To find out what may be reflected, first the scene is photographed from the center of this reflective object, and the result is stored in a texture. To get a complete surrounding, the reflective object is removed, and the scene is rendered six times from the center, selecting the faces of a cube as camera windows. The collection of these six images is called the **cube map** or the **environment map**. It is also possible to obtain this cube map in a real environment by taking panoramic images.

Having obtained the cube map, the scene is rendered from the point of view of the real camera. Other objects are rendered in the normal way, but when the reflective object is processed, special vertex and fragment shaders are enabled that compute the reflection with the help of the prepared cube map.

When a reflective surface point is processed, its normal vector is obtained, and the view direction is reflected (and/or refracted) at this point. To simulate reflection, we need to know the radiance coming to this point from the reflection (and/or refraction) direction. Unfortunately, this information is not available, what we store in the cube map is the radiance coming to the center (from where the photographs have been made) from the specified direction. However, if the distance of the shaded point and the cube map center is small with respect to the distance to the environment, then we can look up the environment map with only the reflection direction, and the fetched value is reflected using the Fresnel function.

Visszaverődés/törés számítás: a sima objektum feldolgozása



We discuss only the final rendering of the reflective object (the creation of the cube map and the rendering of the other objects are like a conventional rendering algorithm). The CPU passes the shading normals and vertices of the mesh of the reflective object to the vertex shader. The vertex shader also gets transformation matrices and the eye in world coordinates as uniform parameters. The vertex shader transforms the point to normalized device space for clipping and then rasterization, and also computes the normal and view vectors in world coordinates as vertex properties, which will follow the point and get linearly interpolated.

The fragment shader receives the linearly interpolated world space normal and view vectors associated with the processed fragment, computes the reflection direction, fetches the incident radiance from the environment map, and multiplies it with the Fresnel function to obtain the reflected radiance.

Sima objekum vertex shader

```
void main(    in float4 position : POSITION,
              in float4 normal   : NORMAL,
              uniform float4x4 MVP,      // modelviewproj
              uniform float4x4 M,        // model
              uniform float4x4 MIT,      // IT of model
              uniform float3 eye,        // eye in world
              out float4 hPos : POSITION,
              out float3 V     : TEXCOORD0, // view in world
              out float3 N     : TEXCOORD1) // normal in world
{
    hPos      = mul(MVP, position);

    float3 p = mul(M, position).xyz; // transform to world sp.
    V = eye - p;
    N = mul(MIT, normal).xyz;
}
```

To prepare for clipping, the vertex is transformed to normalized device space (hPos).

The vertex shader computes vectors in the coordinate system of the cube map. The cube map is usually generated by looking left/right, up/down and forward/backward in world space, so the cube map space has the same axes as the world space.

So the vertex is transformed to world space with modeling transform M , resulting in x . The view direction in world space is the difference of world space eye position and the location of the vertex in world space. The normal vector in world space is obtained with the inverse-transpose of the modeling transform.

Viewing direction V and normal N are passed down the pipeline in TEXCOORD0 and TEXCOORD1 registers.

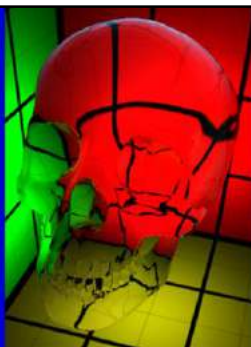
Sima objektum fragment shader

```
void main(  
    in float3 V : TEXCOORD0,  
    in float3 N : TEXCOORD1,  
    uniform float n, // törésmutató  
    uniform float F0, //  $F0 = [(n-1)/(n+1)]^2$   
    uniform samplerCUBE envMap,  
    out float3 color )  
{  
    V = normalize(V);  
    N = normalize(N);  
    float3 T = refract(V, N, 1/n);  
    float3 R = reflect(V, N);  
    float3 refractedRad = texCUBE(envMap, T).rgb;  
    float3 reflectedRad = texCUBE(envMap, R).rgb;  
    float F = F0 + (1-F0) * pow(1-dot(N,V), 5);  
    color = F * reflectedRad + (1-F) * refractedRad;  
}
```

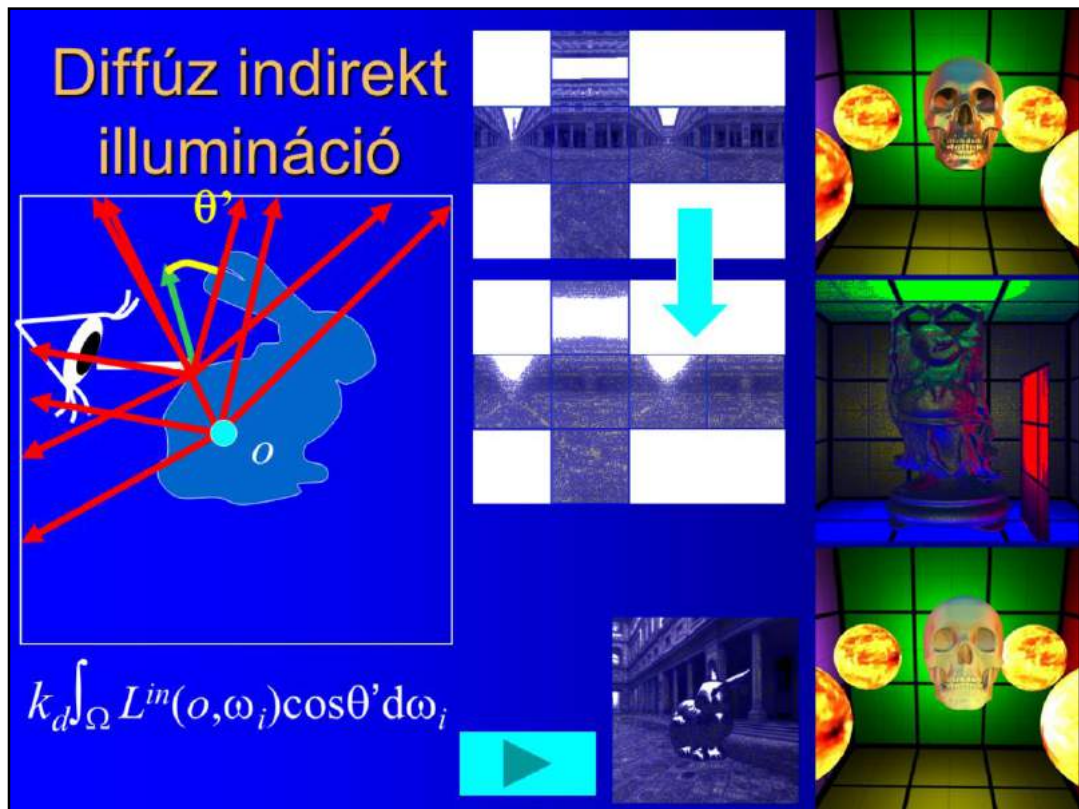


The reflective triangle is rasterized while the world space view direction and normal vector are interpolated. At a given fragment, we normalize these vectors and compute **refraction direction T** and **reflection direction R**. Refraction computation is based on the Snellius-Descartes law and is done by the **refract** Cg function taking also the reciprocal of the index of refraction into account. The reflection computation is based on the reflection law and is done with the **reflect** Cg function. We look up the environment map in the reflection and refraction directions to get the incident illumination. The Fresnel function is computed, and the reflected and refracted radiances are added up weighting them with the Fresnel or 1-Fresnel, respectively.

Lokalizált környezet leképzés

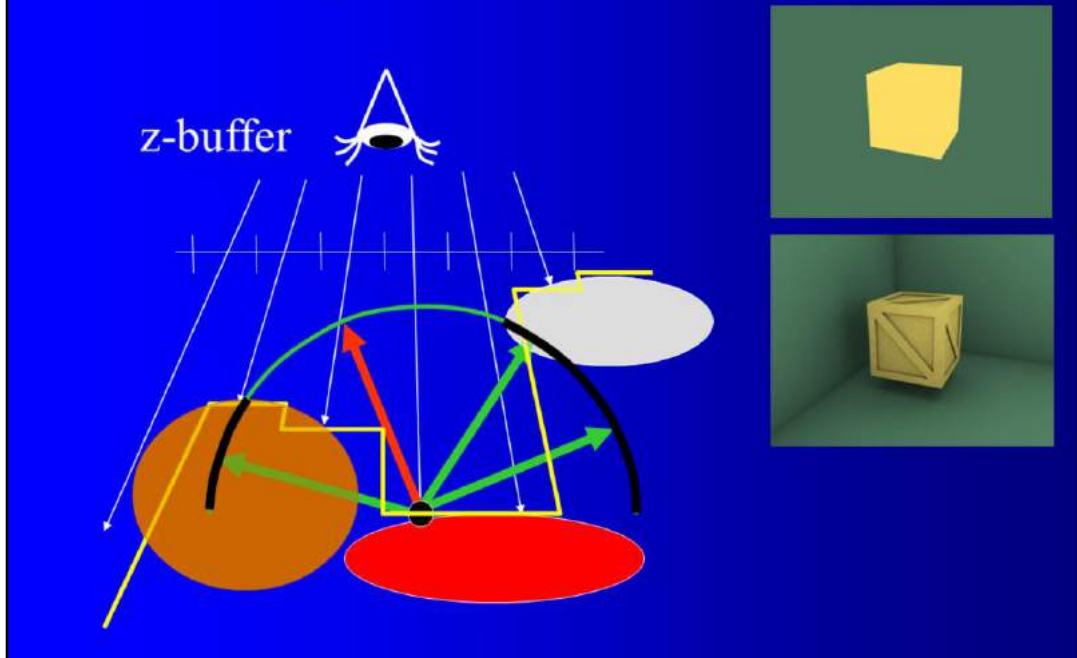


If the cube map texels also store the distance of the visible point from the center of the cube map, the real reflected/refracted point can be searched for. This is called localization.



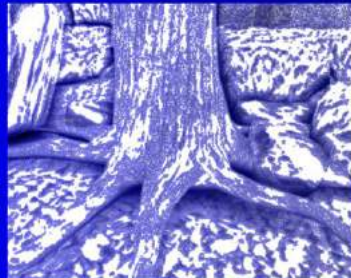
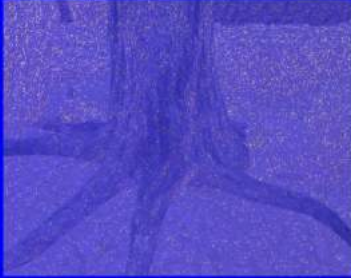
If we blur or convolve the environment map (cube map) with the cosine or with the power of cosine functions, then diffuse or glossy indirect illumination can be simulated.

Screen-Space Ambient Occlusion

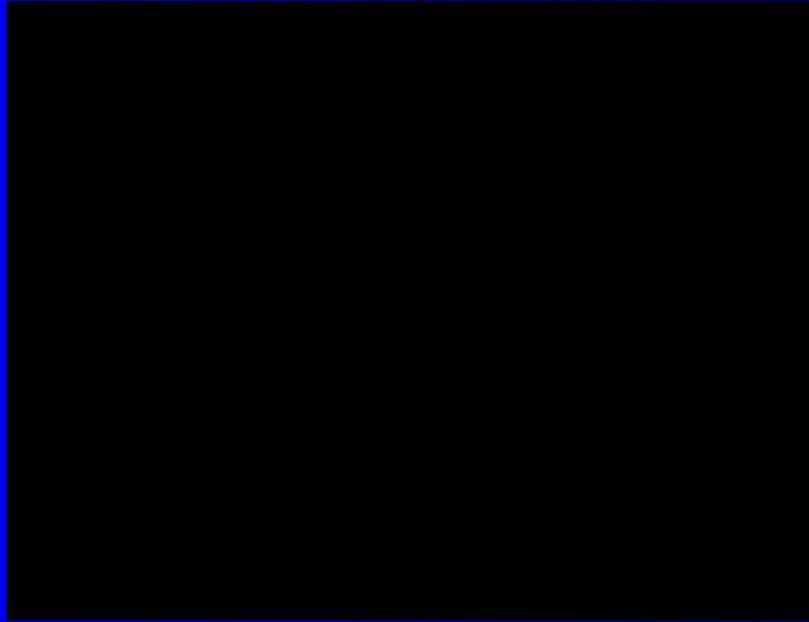


Ambient occlusion computes just how open the scene is around the shaded point and scales ambient illumination accordingly. The computation of the openness may be based on the content of the depth buffer.

Ambient Occlusion



Autós játék

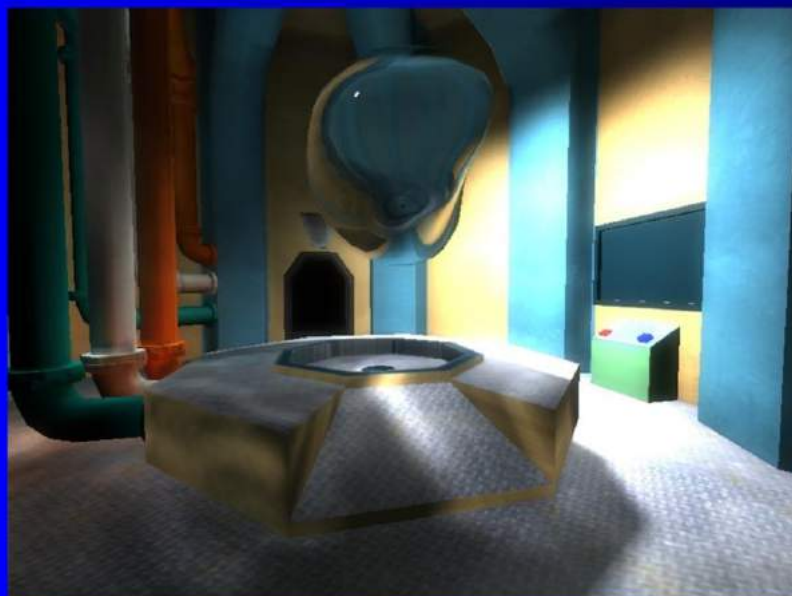


The second demo game is a driving simulator game. Here the car is reflective, and the wheels generate caustics. In this arena the goal is to push bit glass bottles that are reflective/refractive and also caustic generators.

When the car moves in this corridor, we can observe the indirect diffuse/glossy reflections as well.

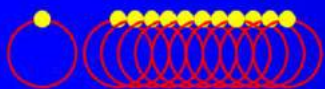
In the other arena, gas tanks should be hit, where the explosions are generated with spherical billboards.

Űrállomás



Moria





Óceán

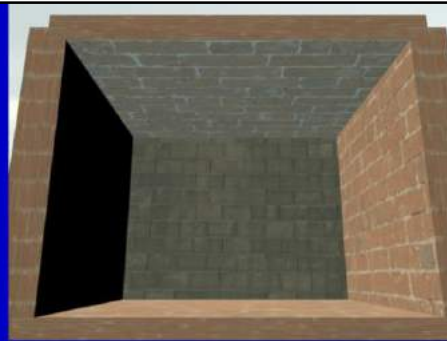


Geometria árnyaló

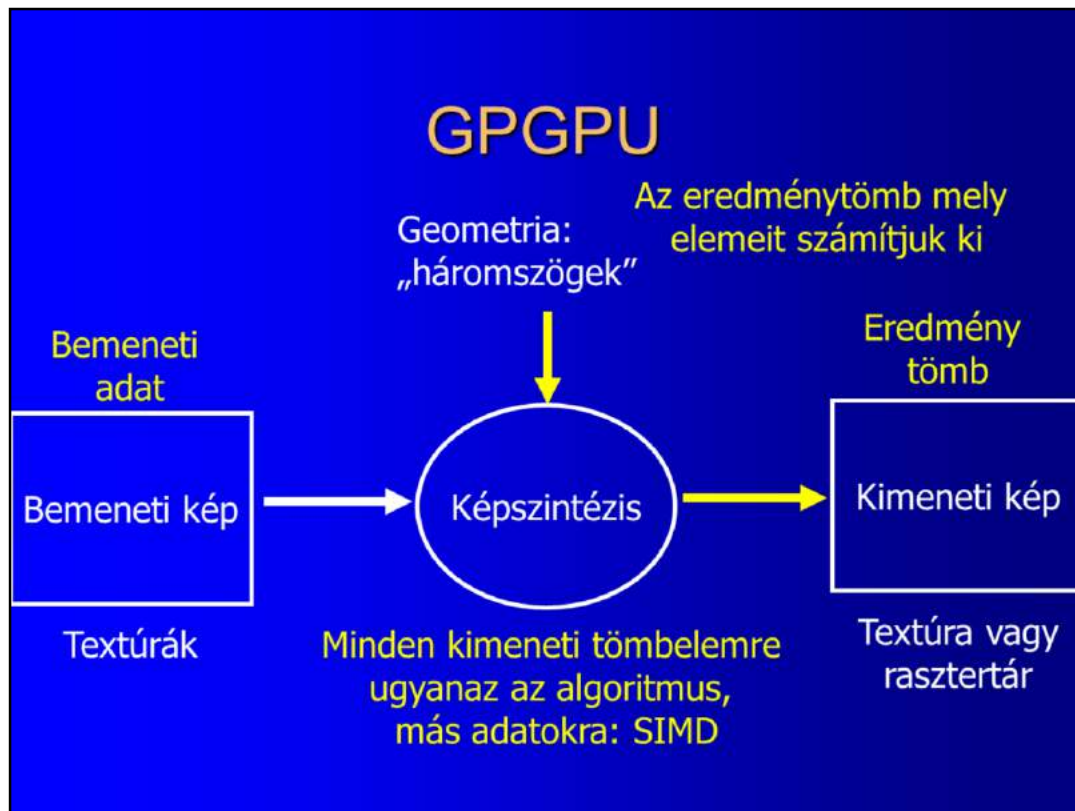


Catmull-Clark
subdivision

Procedurális
geometria



The geometry shader being between the vertex shader and the clipping using, can change the type or topology of the primitive. This can be used, for example, to execute on-the-fly subdivision smoothing, or to produce procedural geometry. Procedural geometry is created inside the GPU and rendered right away, so expensive CPU to GPU transfer can be eliminated, and the storage requirement can be significantly reduced (when an element is created, it is immediately rendered and its storage space is released).



Nowadays, a GPU has supercomputer performance (over two teraflops), which is two magnitude higher than a CPU has, and the gap between GPUs and CPUs grows constantly. So GPUs are worth using not only for graphics but for the solution of general purpose computation as well. As GPUs became programmable, this is a feasible approach.

So far, we assumed that the “main input” of the rendering process is the geometry containing a list of triangles, which is processed by the pipeline, and during the rasterization of this geometry, the fragments onto this geometry is projected are identified. Fragment processing may involve texture fetches, so the texture memory can be imagined as a “secondary input”. The output of rendering is always the 2D array of pixels, which can be the frame buffer or stored in the texture memory as well.

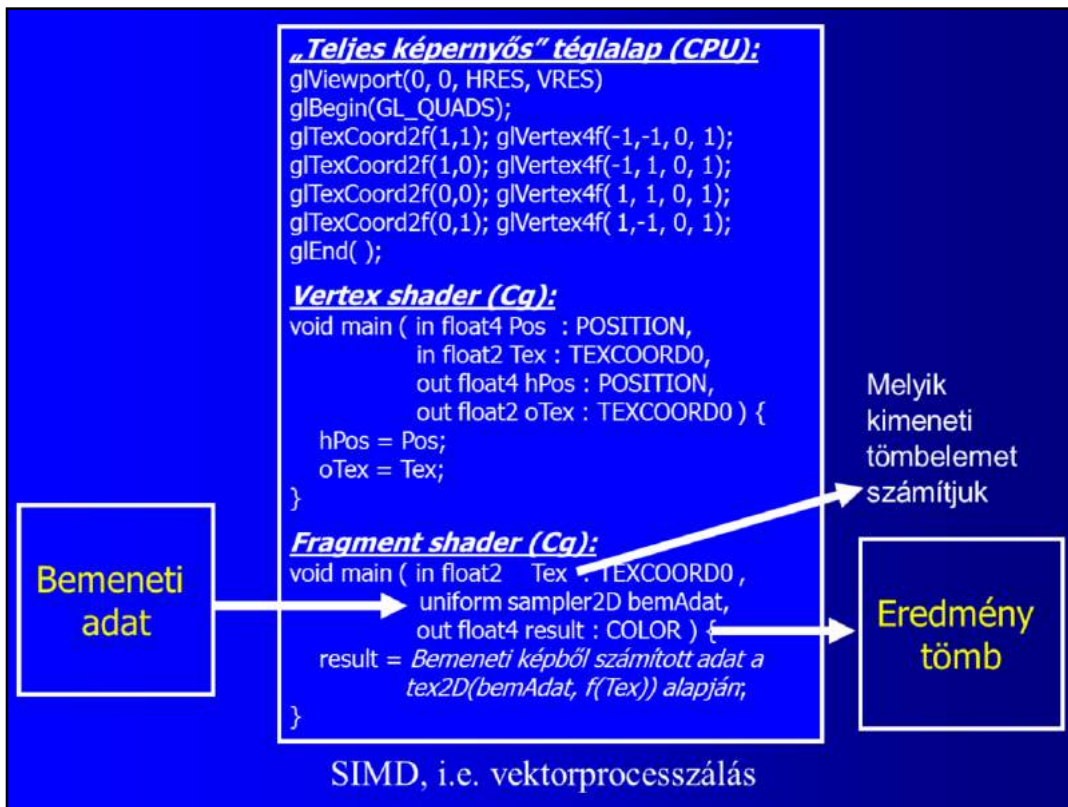
To make this model more appropriate for general purpose, i.e. non-graphics computations, we consider the texture memory, which is a 1D, 2D, or 3D array as the main input of the algorithm and geometry is only supplied to get the fragment shader to be executed for each pixel of the output image. The simplest geometry for such purpose is a quad that covers the full viewport. When this quad is rendered, the fragment shader will run for each of the pixels, where an algorithm can be executed that can access the textures.

Thus, interpreting the texture as an input array, the image as an output array, and the

fragment shader as a function that is computed for every element of the output array, we have a parallel computer system. This system is SIMD (Single Instruction Multiple Data, or more precisely, Single Algorithm Multiple Data) since the same fragment shader program will be executed in parallel computing a result on different data.

SIMD like parallelism is useful in many applications like:

1. Large matrix-vector multiplication,
2. Image filtering,
3. Differential equations on numerical grids,
4. Monte Carlo methods,
5. Etc.



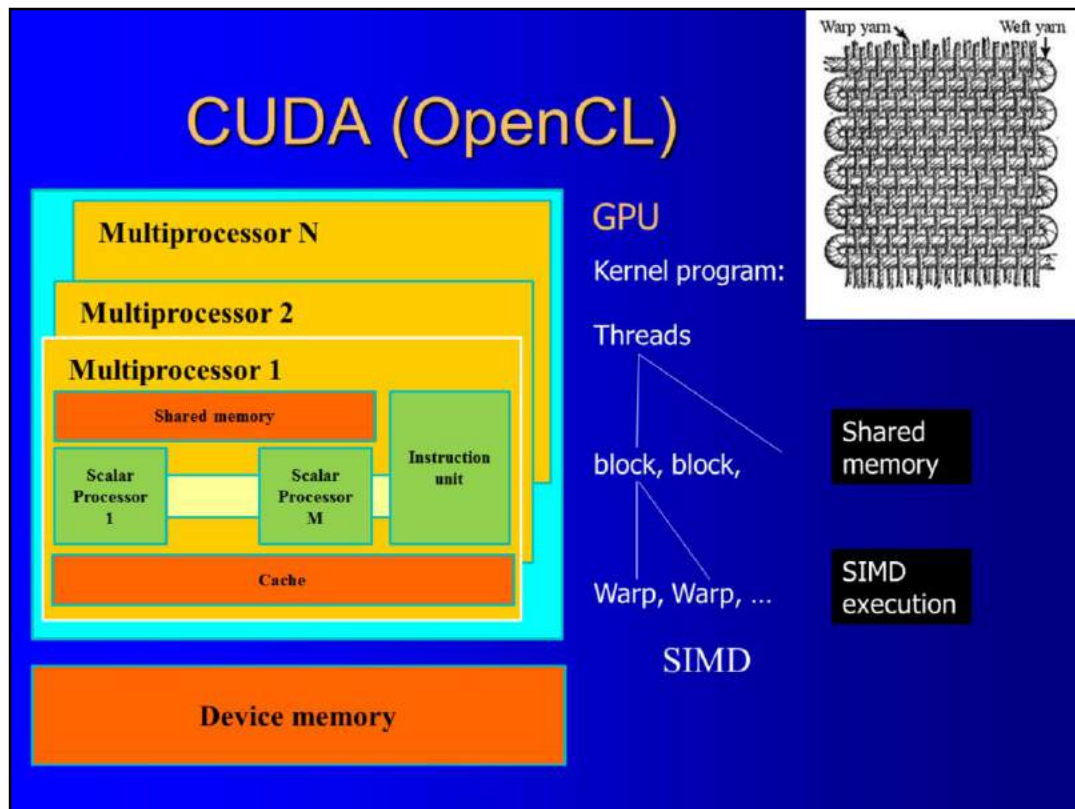
In a general GPGPU application the CPU renders a full viewport quad where vertices are directly specified in normalized device space.

The vertex shader copies the vertex without transformation since it is already in normalized device space and the texture coordinate associated with this quad.

The fragment shader gets the interpolated texture coordinate which tells the shader which output element it computes and thus different fragment shaders would use different input data based on this (it would not make sense to compute the same result many times). The fragment shader can implement any function F that is based in the Input data and also on the Texture coordinate identifying the output index.



An edge detection filter would compute the length of the gradient to locate pixels where the image changes significantly.



The approach discussed so far became very attractive in the community that had some graphics background, since they could understand concepts like normalized device space, clipping, vertex shader, texture coordinates, etc. However, non-graphics programmers did not like it.

To help the development for those who are not familiar with the concepts of computer graphics, NVIDIA developed the **CUDA (Compute Unified Device Architecture)** framework, then a vendor independent version, called OpenCL (supported by AMD) was also born. These GPGPU frameworks present the GPU to the programmer as a large collection of general purpose processors and memory, but do not allow the access of fixed function elements like, depth buffering, alpha blending, clipping or rasterization.

CUDA presents the GPU as a set of N (1..128...) **independent multiprocessors**, where each multiprocessor contains M (e.g. 8) **scalar processors sharing the instruction unit** and are connected by a fast internal **shared memory**. Each scalar processor has local registers that can store the data of many threads at a time. As scalar processors of a single multiprocessor share the instruction unit, they always execute the very same machine instruction in a **SIMD** like fashion. Note that this means that programs having if type branches where different threads may go into different directions are executed in a way that always all branches are executed, but in dummy branches the write operations are disabled. The parallel threads executed on a multiprocessors at a time in a SIMD style are called the **warp** (a word originated in the terminology of weaving). Interestingly the number of threads in a warp is usually larger than M , if $M=8$, then the typical number of

the warp size is 32. The reason is that the execution of the scalar processors is fast, so while a machine instruction is fetched from the memory, it can execute four instructions. So, to keep it busy, each scalar processor runs four threads at a time, and executes an instruction on each of them (their data are stored in the registers, so switching from one thread to the other means just the change of the base address in the register file). Threads of multiple warps can be assigned to a scalar processor, which has the advantage that when a warp is stopped due to a slow memory access, then other warps may run during the memory fetch. The threads assigned to a single multiprocessor are called the **block**.

A GPU has many multiprocessors so they can simultaneously execute many blocks. If the number of blocks is greater than the number of multiprocessors, then they are executed sequentially one after the other.

Két N elemű vektor összeadása

```
#include <cuda.h>
A GPU-n fut, de a CPU-ról is hívható

__global__ void AddVectorGPU( float *C, float *A, float *B, int N ) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // szárazonosító
    if ( i < N ) C[i] = A[i] + B[i];
    0 ,..., gridDim.x-1    0 ,..., blockDim.x-1
}

const int N = 100000;
const int Nb = N * sizeof(float);
float Ccpu[N], Acpu[N], Bcpu[N];

int main ( ) {
    ... // Acpu és Bcpu tömbök feltöltése
    float *Agpu, *Bgpu, *Cgpu;
    cudaMalloc(&Agpu, Nb); cudaMalloc(&Bgpu, Nb); cudaMalloc(&Cgpu, Nb);
    cudaMemcpy(Acpu, Acpu, Nb, cudaMemcpyHostToDevice);
    cudaMemcpy(Bcpu, Bcpu, Nb, cudaMemcpyHostToDevice);

    int blockDim = 256; // #threads egy blokkban: 128, 256, 512
    int gridDim = (N + blockDim - 1) / blockDim; // #blocks

    AddVectorGPU<<<gridDim, blockDim>>>>(Cgpu, Agpu, Bgpu, N);

    cudaMemcpy(Ccpu, Cgpu, Nb, cudaMemcpyDeviceToHost);
    cudaFree(Agpu); cudaFree(Bgpu); cudaFree(Cgpu);
    ... // A Ccpu használata
}
```

As an example, we present a CPU and CUDA program that adds two, large arrays. In the CUDA framework, CPU and GPU functions may be mixed and written in the same file. We can use C (or C++) for all types of functions. Function types can be used to declare whether a function runs on the CPU (this is the default), runs on the GPU but can be called from the CPU (**global**), or runs on the GPU and can only be called from the GPU. Before a conventional C compiler runs, a CUDA pre-processor separates functions for different devices and establishes the proper ways of parameter passing.

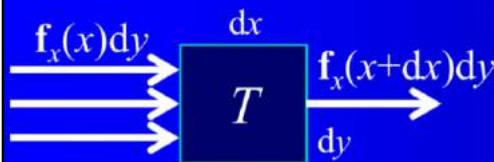
In this example, the **main** function is on the CPU, which calls a parallel GPU function called **AddVectorGPU**. When this function is called, parameters can be passed to it, and we should also specify how many threads of this function should be started in total and how the threads are distributed among the multiprocessors. To do that, enclosed in <<< and >>>, we define the grid dimension specifying how many blocks are started (how many multiprocessors are assumed) and the block dimension describing how many threads a single multiprocessor should run. The **blockDim** is a multiple of the warp size, and 256 is a generally good number.

In this example, we add two N element vectors where a single thread is responsible for adding just a single element of the vector. So, altogether, we need to start N threads, which is distributed into N/blockDim blocks of blockDim threads in a block (the program is a little more complicated, because it also handles the case when N/blockDim is not an integer number).

The GPU function gets not only the passed variables, but also invisible input defining which multiprocessor executes this thread (blockIdx) and also the index of this thread among the threads running on the same multiprocessor (threadIdx). This is very similar to the non-visible this pointer in C++ member functions. These numbers can be used to compute a unique index for the thread, which defines what output this thread should compute.

2D hőáramlás, diffúzió

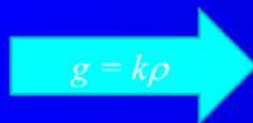
- Anyag-energia megőrzés



$$\frac{\partial T}{\partial t} = -k \left(\frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} \right) = -k \nabla \cdot \mathbf{f}$$

- Hőáramot a hőmérséklet különbség hozza létre

$$\mathbf{f} = (f_x, f_y) = -\rho \left(\frac{\partial T}{\partial x}, \frac{\partial T}{\partial y} \right) = -\rho \nabla T$$



$$g = k\rho$$

$$\frac{\partial T}{\partial t} = g \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) = g \nabla^2 T$$

```

__device__ float T(float* array2d, int x, int y) {
    return (x >= N || y < 0 || y >= N) ? 0 : array2d[y*N+x];
}

__global__ void Diffuse(float* o, float* n, float* b, float g, float dx, float dy, float dt, int N) {
    int x = threadIdx.x, y = blockIdx.y;
    float dTx2 = (T(o, N, x+1, y) - 2*T(o, N, x, y) + T(o, N, x-1, y))/(dx*dx);
    float dTy2 = (T(o, N, x, y+1) - 2*T(o, N, x, y) + T(o, N, x, y-1))/(dy*dy);
    n[y*N+x] = (T(b, N, x, y) == 0) ? T(o, N, x, y) + g * (dTx2 + dTy2) * dt : T(o, N, x, y);
}

```

$$\Delta T = g \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \cdot \Delta t$$

```

const int N = 128, Nb = N*N*sizeof(float); // number of bytes
float Tcpu[N*N] = /*initial+boundary condition*/ , Bcpu[N*N] = /*1-0 mask*/;

int main () {
    float *Tgpu1, *Tgpu2, *Bgpu;
    const float g = 3, dx=1, dy=1, dt = 0.1, tend=10;

    cudaMalloc(&Tgpu1, Nb); cudaMemcpy(Tgpu1, Tcpu, Nb, cudaMemcpyHostToDevice);
    cudaMalloc(&Tgpu2, Nb);
    cudaMalloc(&Bgpu, Nb); cudaMemcpy(Bgpu, Bcpu, Nb, cudaMemcpyHostToDevice);

    for(float t = 0; t < tend; t += dt) {
        Diffuse<<<N, N>>>>(Tgpu1, Tgpu2, Bgpu, g, dx, dy, dt, N);
        Diffuse<<<N, N>>>>(Tgpu2, Tgpu1, Bgpu, g, dx, dy, dt, N);
    }

    cudaMemcpy(Tcpu, Tgpu1, Nb, cudaMemcpyDeviceToHost);
    cudaFree(Tgpu1); cudaFree(Tgpu2);
}

```



Mikor jó?

- Párhuzamos algoritmus ($>10k$ szál):
A probléma elemzésével kezdődik!
- Gyűjtő típusú algoritmus (nincsenek írási memóriaütközések)
- Kevés feltételes utasítás (thread divergencia)
- Adatlokalitás
- Számításintenzív

Tudományos (mérnöki) számítások

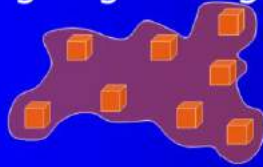
Idő és térváltozók szerinti
differenciálok/integrálok:
Parciális diff egyenletek

Z-tengely körüli forgatás homogén lineáris transzformációs mátrixa



Numerikus megoldás

- Tér-idő differenciál egyenletek
- Idő és térkoordináták diszkretizálása (véges elem)
- Idő: diszkrét idő vagy diszkrét esemény
- Tér: Lagrange-i megközelítés



Mozgó részecskék

Euler-i megközelítés

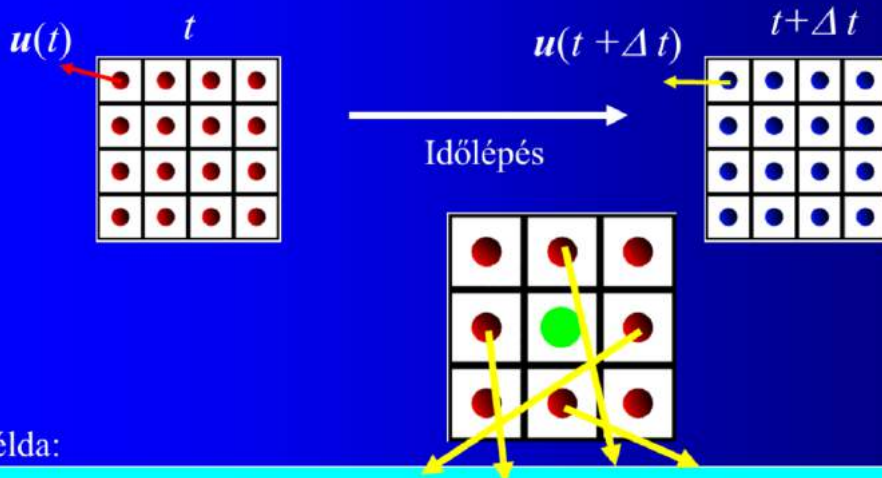


Tér mintavételezése
rögzített rácson

$$\frac{d\vec{u}}{dt} = -(\vec{u} \cdot \nabla) \vec{u} - \frac{1}{\rho} \nabla p + \nu \nabla^2 \vec{u} + \vec{F}$$

$$\nabla \cdot \vec{u} = 0$$

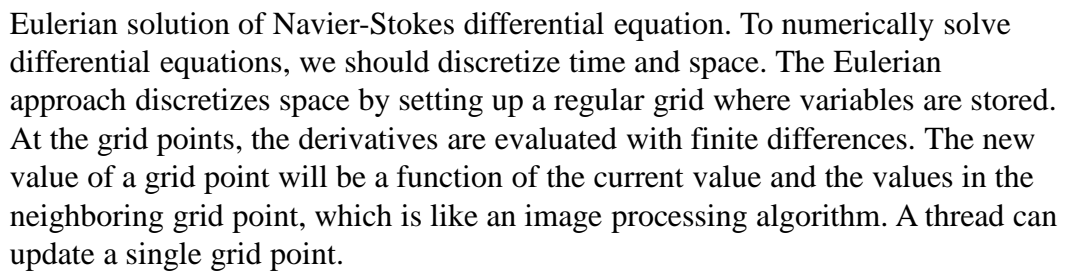
Euler-i folyadék



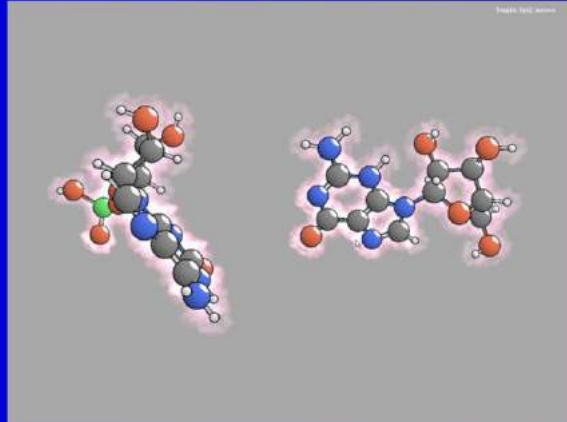
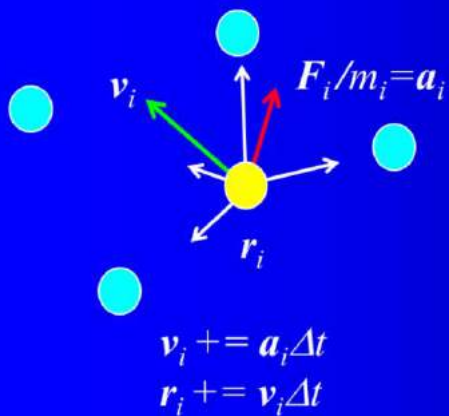
Példa:

$$\nabla \cdot \vec{u} = \text{div } \vec{u} = \frac{\partial \vec{u}_x}{\partial x} + \frac{\partial \vec{u}_y}{\partial y} + \frac{\partial \vec{u}_z}{\partial z} = \frac{\vec{u}_x^{i+1,j,k} - \vec{u}_x^{i-1,j,k}}{2\delta x} + \frac{\vec{u}_y^{i,j+1,k} - \vec{u}_y^{i,j-1,k}}{2\delta y} + \frac{\vec{u}_z^{i,j,k+1} - \vec{u}_z^{i,j,k-1}}{2\delta z}$$

Euler-i folyadék



Lagrange-i módszer: Részecske rendszer, N-body

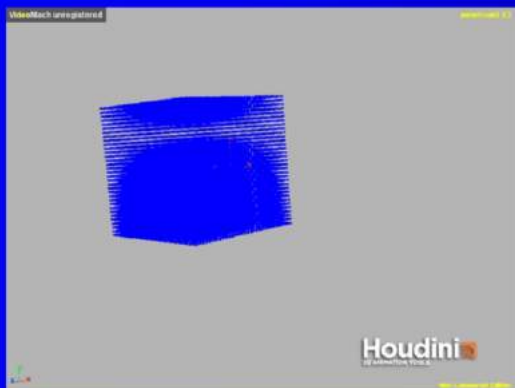


Előrelépő Euler séma.

Helyette: Visszalépő Euler, Runte-Kutta, Midpoint, Verlet, ...

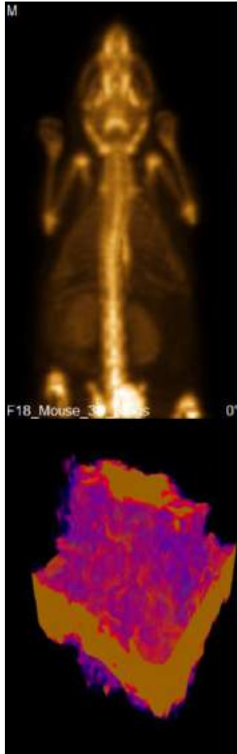
Lagrange-i folyadékáramlás

SPH=Smoothed Particle Hydrodynamics



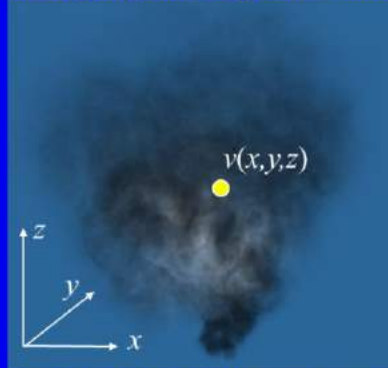
Térfogatvizualizáció

Szirmay-Kalos László

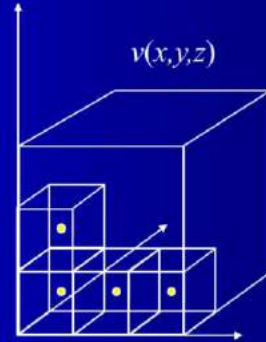


Térfogati modellek (skalár mezők)

3D tér pontjaiban egy skalár érték



- Skalár: hőmérséklet, sűrűség, nyomás, potenciál, ...
- Származás: Euler-i szimuláció, Rekonstrukció (tomográfia)



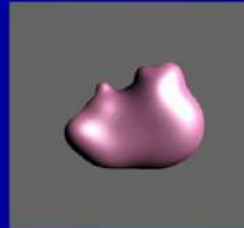
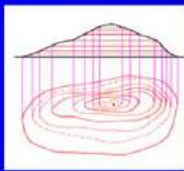
tárolás: 3D textúra
vagy „voxel tömb”

Térfogati modell megjelenítése

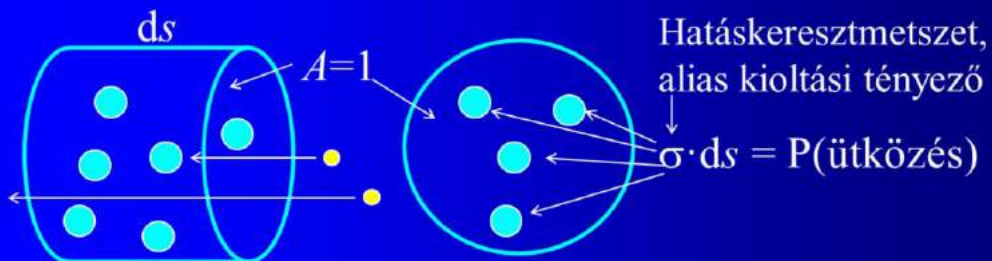
- Megjelenítés fényszóró anyag (participating media) analógiáját felhasználva (belsejébe belelátunk)



- Adott szintfelület kiemelése (külsőt lehámozzuk)

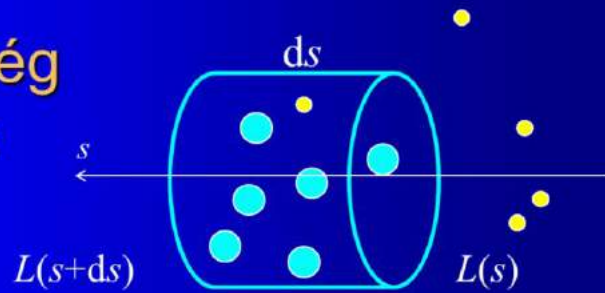


Fényszóró közeg



- Albedo a : a nem-elnyelődés valószínűsége feltéve, hogy az ütközés bekövetkezett
- Fekete test: albedo = 0

Sugársűrűség változása



$$L(s+ds) = L(s) - L(s) \cdot \sigma(s) \cdot ds + \quad // \text{ Kiszóródás+abszorbcio}$$

$$L^e(s) \cdot ds + \quad // \text{ Emisszio}$$

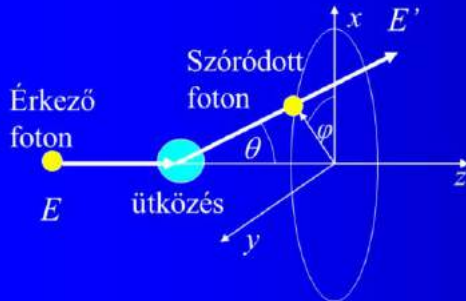
$$\sigma(s) \cdot a(s) \cdot ds \cdot \int f(\omega', \omega) L^i(\omega') d\omega' \quad // \text{ Beszóródás}$$

$$\boxed{dL(s)/ds = -L(s) \cdot \sigma(s) + L^e(s) + L^{inscatter}(s)}$$

Megoldás fényelnyelő közegre
(emisszió és beszóródás nincs):

$$\boxed{L(s) = L(0) \cdot \exp(-\int^s \sigma(s) ds)}$$

Szóródás



Klein-Nishina:

$$P(\theta) = C \left(\frac{E'}{E} + \left(\frac{E'}{E} \right)^3 - \left(\frac{E'}{E} \right)^2 \sin^2 \theta \right)$$

Rayleigh:

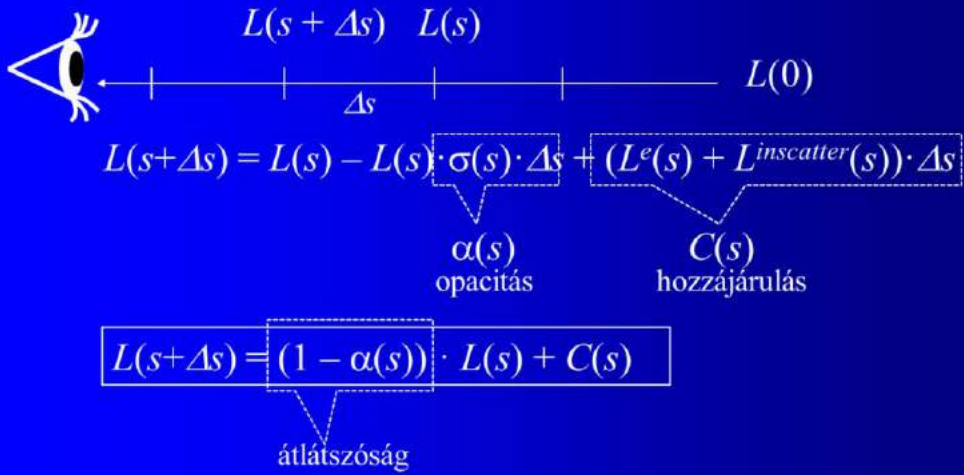
$$P(\theta) = C(1 + \cos^2 \theta)$$

Compton formula:

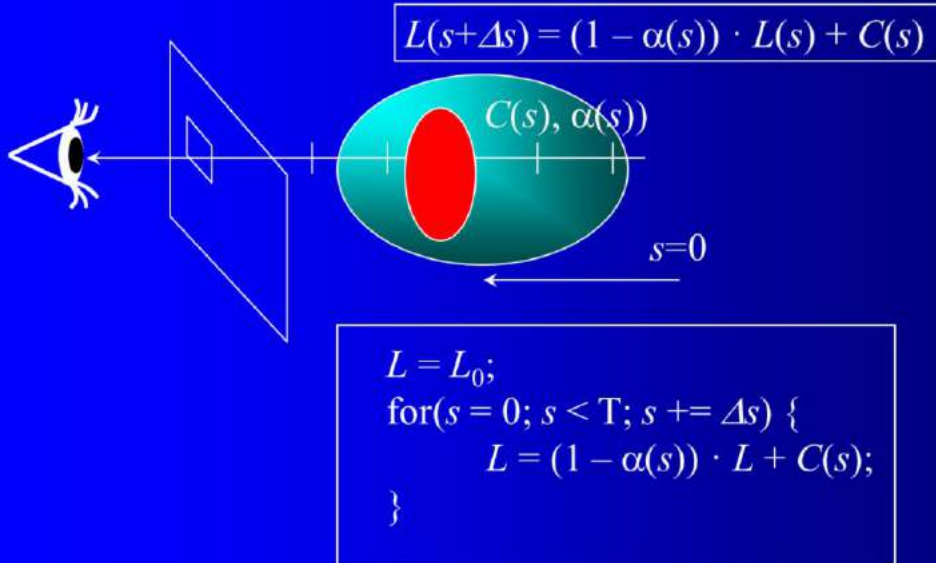
$$\frac{E'}{E} = \frac{1}{1 + \frac{E}{m_e c^2} (1 - \cos \theta)}$$

Sugár masírozás (ray marching)

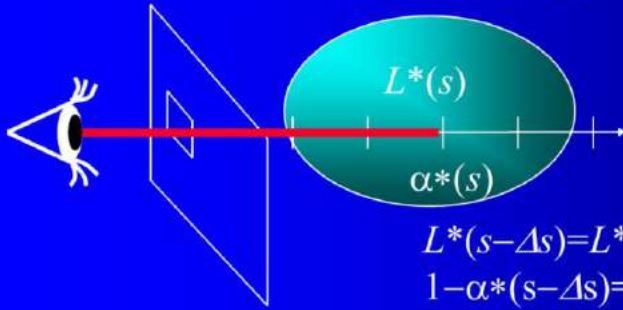
Megoldás: $dL(s)/ds = -L(s) \cdot \sigma(s) + L^e(s) + L^{inscatter}(s)$



Back-to-front ray marching



Front-to-back ray marching



$$L^*(s-\Delta s) = L^*(s) + (1 - \alpha^*(s)) \cdot C(s)$$

$$1 - \alpha^*(s-\Delta s) = (1 - \alpha^*(s)) \cdot (1 - \alpha(s))$$

```
 $L^* = \alpha^* = 0;$   
for(  $s = T$ ;  $s > 0$  ;  $s -= \Delta s$  ) {  
     $L^* += (1 - \alpha^*) \cdot C(s);$   
     $\alpha^* = 1 - (1 - \alpha^*) \cdot (1 - \alpha(s));$   
    if ( $\alpha^* > 1 - \epsilon$ ) break;  
}
```

Voxel szín és opacitás:

Transfer func: $(C, \alpha) = T(v \text{ függv}) \cdot \Delta s$

- Röntgen: $(C, \alpha) = T(v(x, y, z)) \cdot \Delta s$

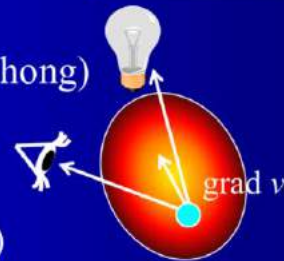
- opacitás = $v(x, y, z) \cdot \Delta s$
- $L(0) = I$, egyébként $C(s) = 0$



- Klasszikus árnyalási modellek

- opacitás: v osztályozása
- C = árnyalási modell (diffúz + Phong)

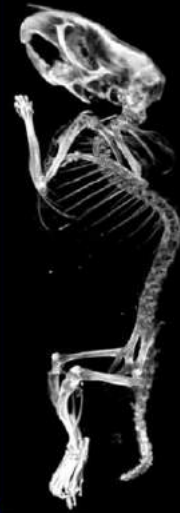
- normál = $\text{grad } v$
- opacitás $\propto |\text{grad } v|$



- Magasabb rendű derivált (görbület)

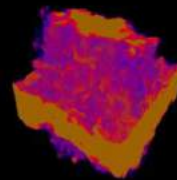
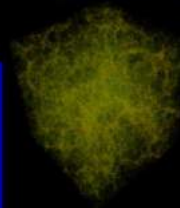
- Transzlucens anyagok (subsurface scattering)

$$(C, \alpha) = T(v) \cdot \Delta s$$



$$\alpha = \text{pow}(v/v_{\text{max}}, \mathbf{aexp}) \cdot \Delta s$$

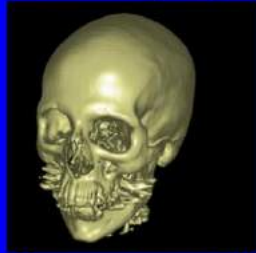
$$C = \text{HLS}((v/v_{\text{max}} + \mathbf{rot}) \cdot 360, 0.5, 1) \cdot \Delta s$$



Klasszikus BRDF modellek

First hit ray casting:

Diffúz+Phong árnyalás



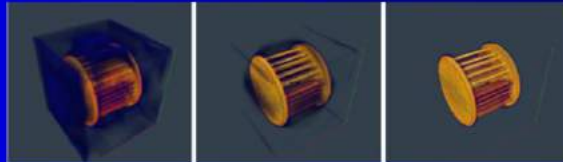
Csont α : 1, másé 0



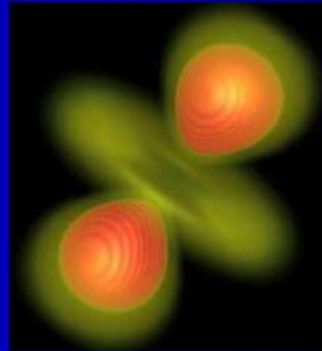
Csont α =1, másé=0



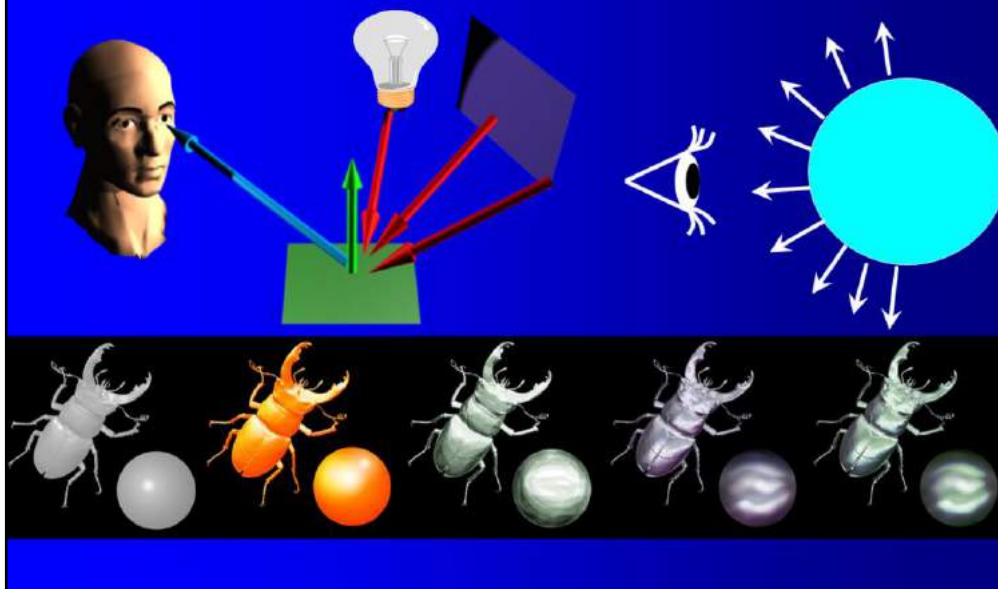
Hús α =1, másé=0



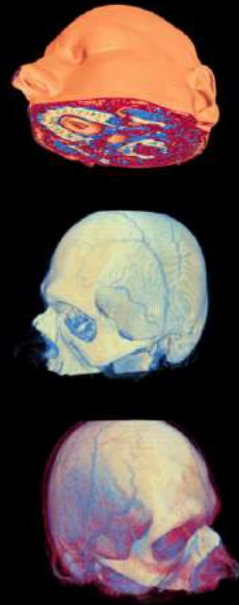
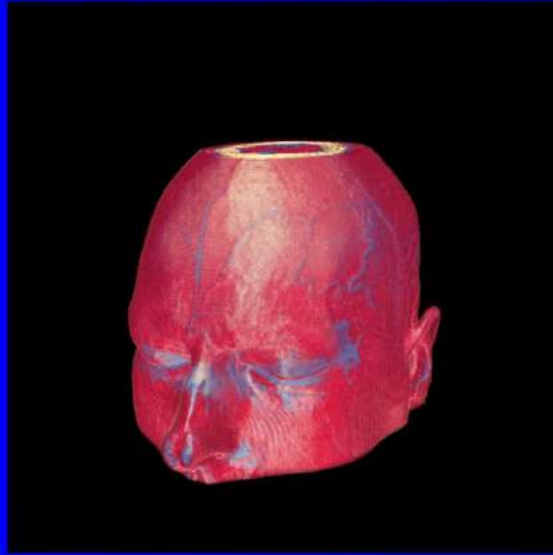
v két osztályba: kék-átlátszó, sárga-átlátszatlan

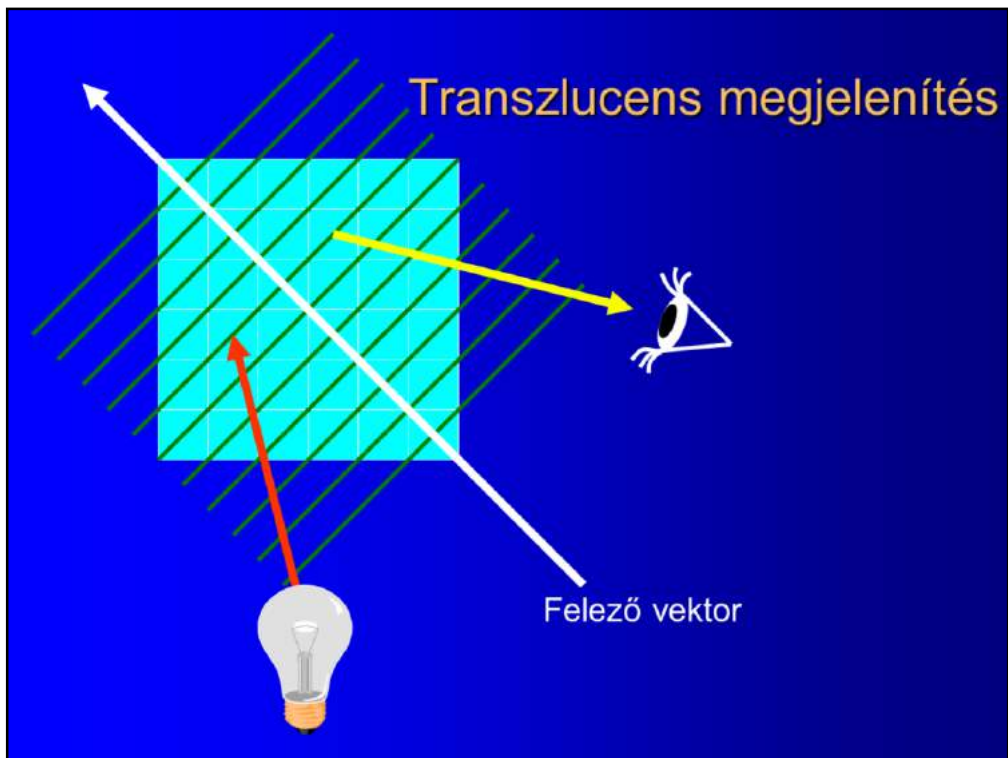


Illusztratív vizualizáció

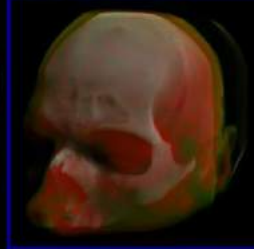
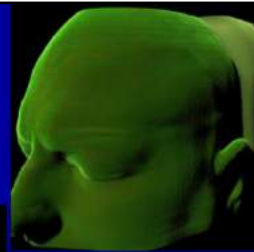


Illusztratív vizualizáció

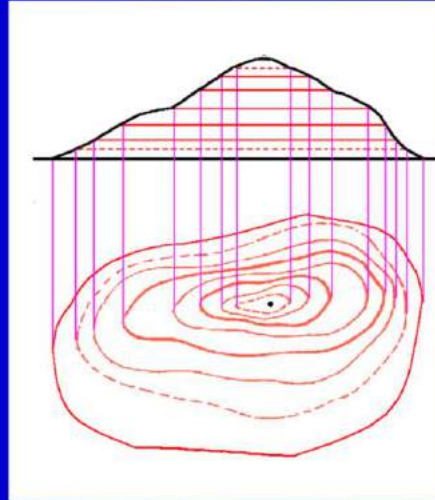




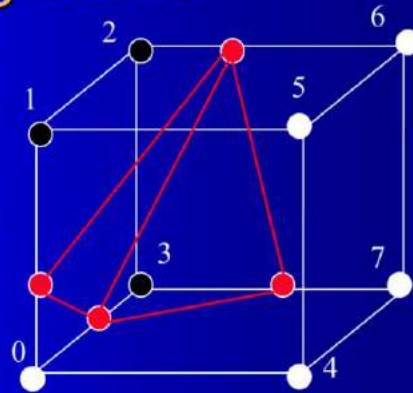
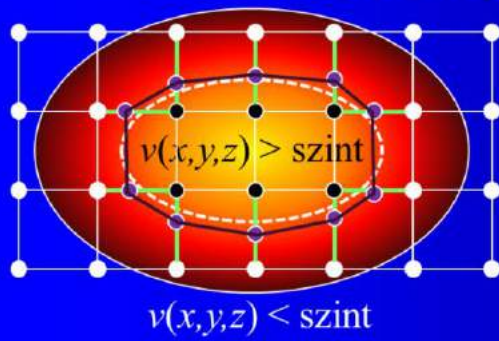
Transzlucens megjelenítés



Szintvonal, szintfelület



Marching cubes

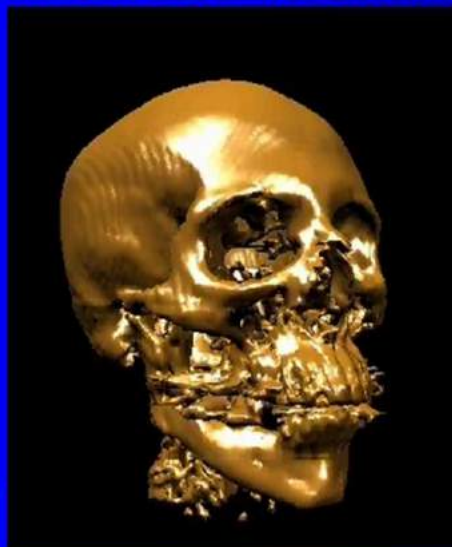


0	0;
14	2; 0-1; 0-3; 2-6; 0-3; 3-7; 2-6
255	0;

Eset: $00001110_2 = 14$

14 ←

Masírozó kockák



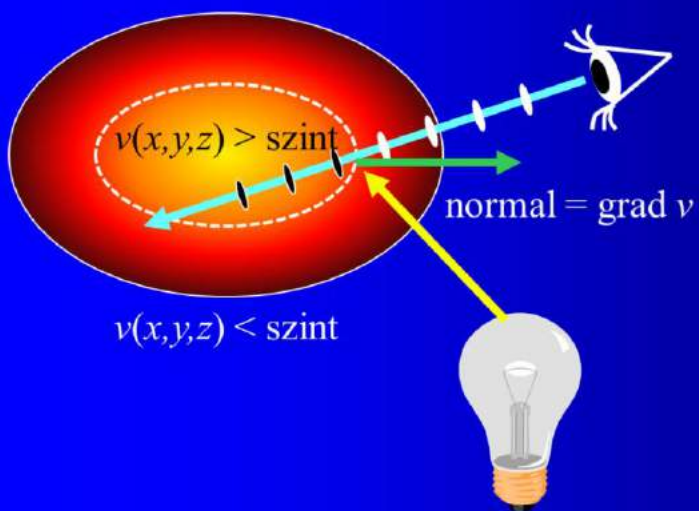
Szintérték = 110



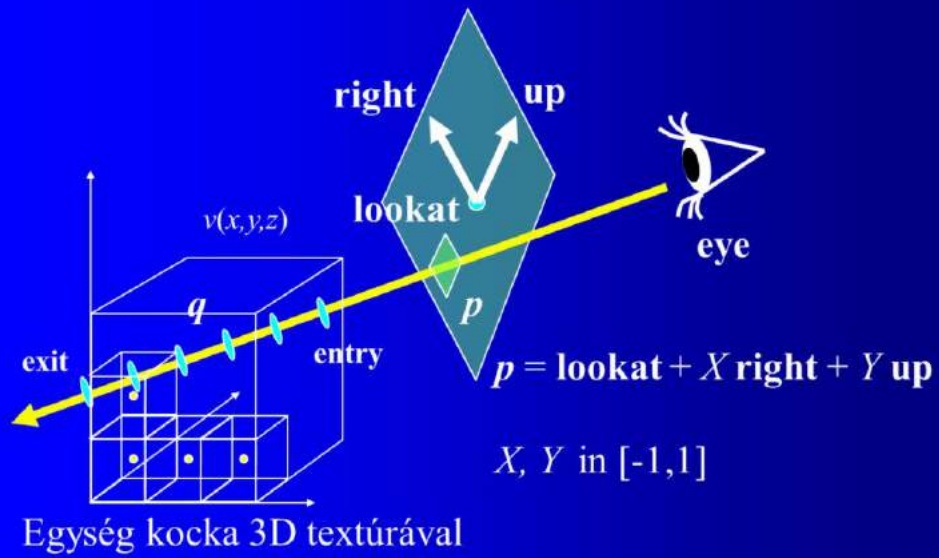
Szintérték = 60



First hit (isosurface) ray casting



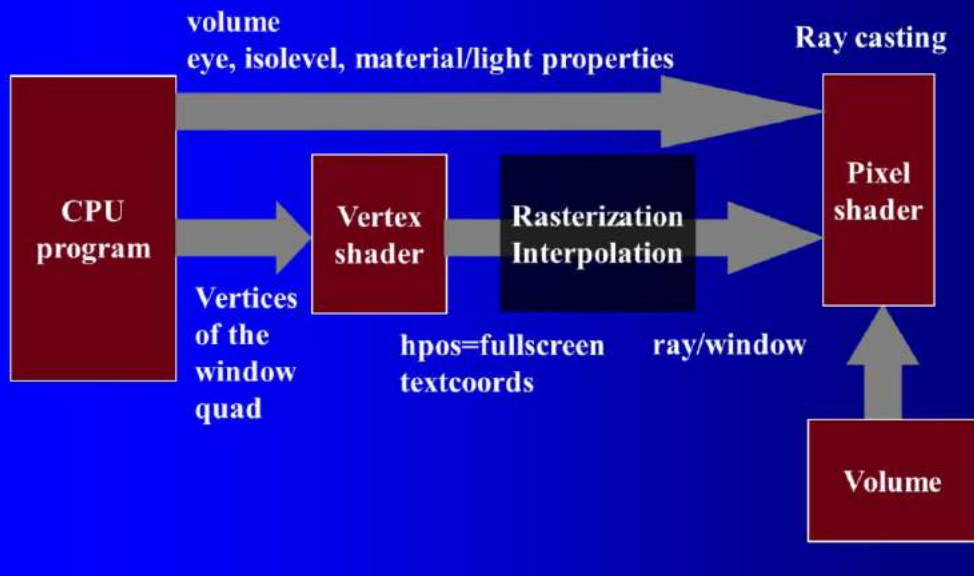
GPU first hit ray-casting



CPU first hit ray-casting

```
For each pixel ← Full screen quad
  Find pixel center p ← Interpolation
  raydir = normalize(p - eye); from the corners
  Find exit and entry
  for(t = entry; t < exit; t+=dt) {
    q = eye + raydir * t;
    if (volume[q] > isovalue) break;
  }
  normal vector estimation; ← central differences
  illumination
}
```

GPU Isosurface ray-casting



CPU program - OpenGL display

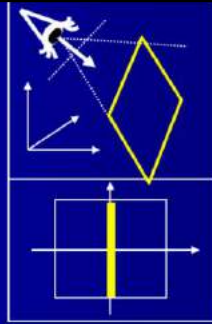
```
void Display( ) {  
    // PASS: non uniform parameters  
    glBegin( GL_QUADS );  
    Vector p = lookat - Right + Up;  
    glTexCoord3f(p.x, p.y, p.z); glVertex3f(-1, 1, 0);  
    p = lookat - Right - Up;  
    glTexCoord3f(p.x, p.y, p.z); glVertex3f(-1, -1, 0);  
    p = lookat + Right - Up;  
    glTexCoord3f(p.x, p.y, p.z); glVertex3f(1, -1, 0);  
    p = lookat + Right + Up;  
    glTexCoord3f(p.x, p.y, p.z); glVertex3f(1, 1, 0);  
    glEnd();  
}
```

Camera window as texture coordinates

Full screen quad

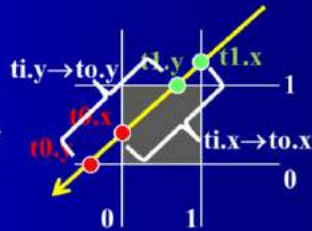
Ray casting: vertex shader

```
void VertexShader(  
    in float4 hPosIn      : POSITION,  
    in float3 wPosIn      : TEXCOORD0,  
  
    out float4 hPosOut     : POSITION,  
    out float3 wPosOut     : TEXCOORD0 )  
{  
    hPosOut = hPosIn;  
    wPosOut = wPosIn;  
}
```



Ray casting: fragment shader

```
void FragmentShader( in float3 p : TEXCOORD0, // point on window
                    uniform float3 eye,
                    uniform sampler3D volume, // voxels
                    uniform float isolevel,
                    uniform float3 lightdir, lightint, kd ☹
                    out float3 color : COLOR )
{
    float3 raydir = normalize(p - eye);
    float3 t0 = (float3(0,0,0)-eye)/raydir;
    float3 t1 = (float3(1,1,1)-eye)/raydir;
    float3 ti = min(t0, t1);
    float3 to = max(t0, t1);
    float entry = max(max(ti.x, ti.y), ti.z);
    float exit = min(min(to.x, to.y), to.z);
    color = float(0, 0, 0);
    bool found = (exit <= entry);
```



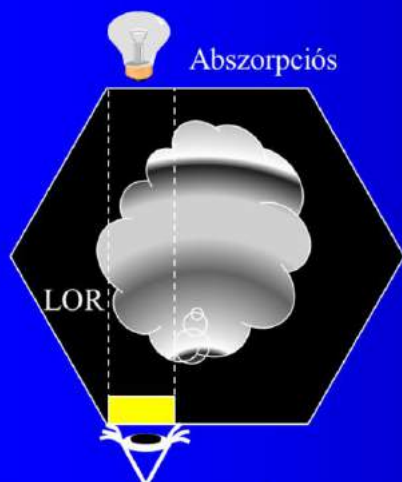
cont'd...

Ray casting fragment shader cont'd

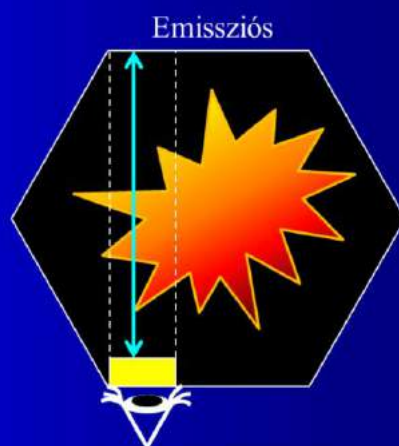
```
if ( !found ) {
    float3 q, normal;
    float dt = (exit - entry) / STEPS;
    for(t = entry; t < exit; t += dt) {
        if ( !found ) {
            q = eye + raydir * t;
            if (tex3D(volume, q).r > isolevel) found = true;
        }
    }
    if ( found ) {
        normal.x = tex3d(volume, q + float3(1/RES,0,0)) -
                   tex3d(volume, q - float3(1/RES,0,0));
        normal.y = tex3d(volume, q + float3(0,1/RES,0)) -
                   tex3d(volume, q - float3(0,1/RES,0));
        normal.z = tex3d(volume, q + float3(0,0,1/RES)) -
                   tex3d(volume, q - float3(0,0,1/RES));
        normal = normalize( normal );
        color = lightint * kd * max(dot(lightdir, normal), 0);
    }
}
```



Tomográfia

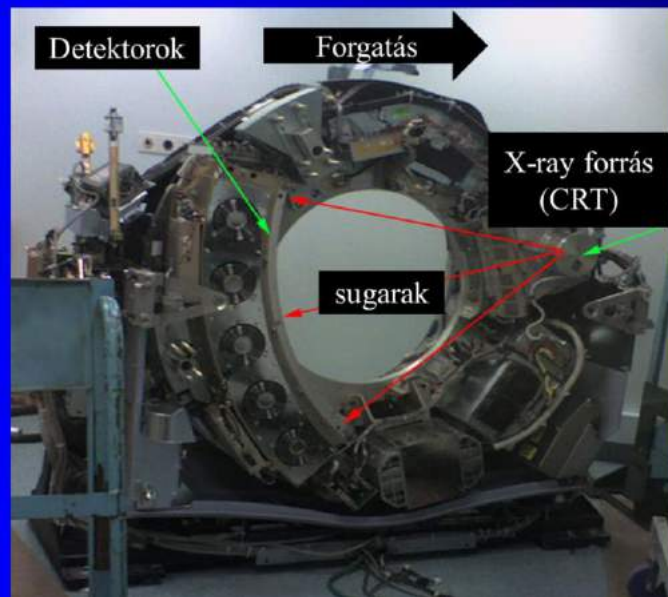


$$L(s) = L(0) \cdot \exp(-\int \sigma(s) ds)$$
$$\int \sigma(s) ds = -\log(L(s)/L(0))$$



$$L(s) = \int L^e(s) ds$$

X-ray Computed Tomography



Mediso NanoPET/CT



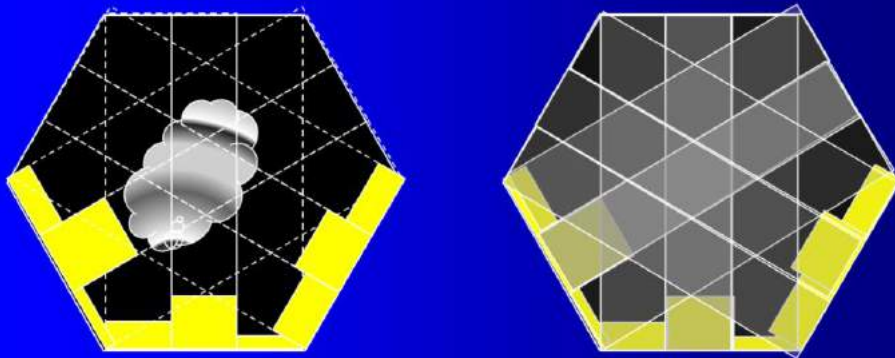
One particular equipment we work with is the NanoPet/CT of Mediso, developed for pharmaceutical research and therefore scans mice and rats. The gamma photon detector structure is shown here, which can measure several hundred million lines, which are distributed not only in 2D slices but forming a complex 3D structure.

From the measurements, we should reconstruct the density of the radiotracer material.

Bigger boys need big toys: AnyScan PET/CT



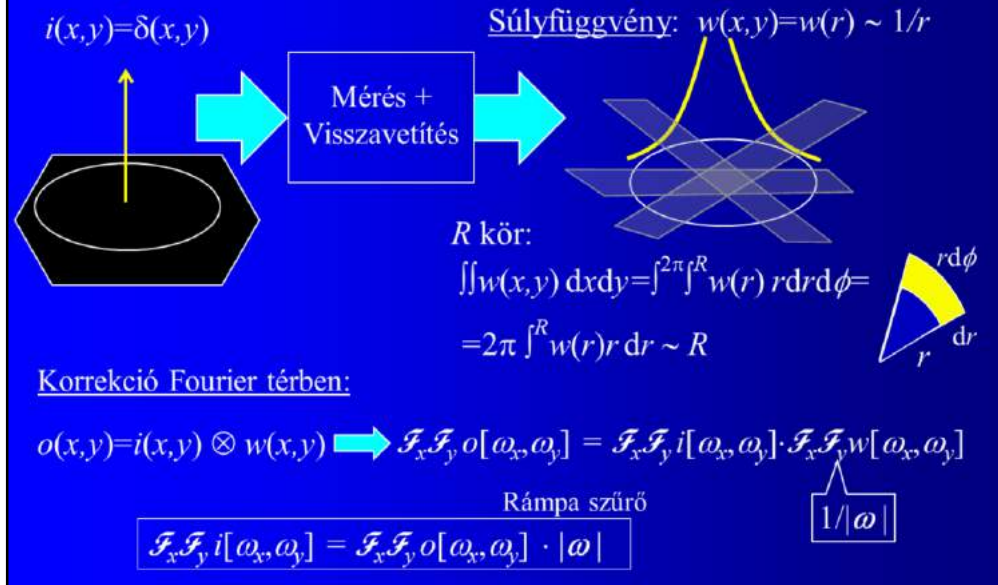
Tomográfiás rekonstrukció



Having made the measurements, we have the integrals of the scalar field along many lines.

The question is how scalar fields can be reconstructed from its line integrals. A very simple approach would be backprojection, which takes a line and distributes the measured integral along it uniformly. As only the integral of the values is available, we cannot do more with a single line. Repeating the same step for all lines, the reconstructed value will be higher where many lines of large integrals meet, so we get a rough approximation of the original scalar field. Clearly, it is blurred and distorted approximation since a point source will be backprojected as a line and the total activity after backprojection will be larger than expected as line cross each other.

Szűrt visszavetítés (FBP=Filtered backprojection)



A standard and old method to improve this naive approach is filtered back projection, which examines what happens with a point source if it is measured and then back projected, and corrects the back projected data accordingly. Let us consider a point source, so the measured scalar field is modeled as a Dirac delta. After backprojection, lines crossing the original point source will have non-zero activity and the reconstructed activity distribution is denoted by function w . This depends just on the distance from the point source due to symmetry, and its integral on a circle of radius R must be proportional to R since the contribution of each constant activity line segment grows linearly with the length. From this, we can easily prove that impulse response of the naive backprojection is a function that decreases proportionally with the distance.

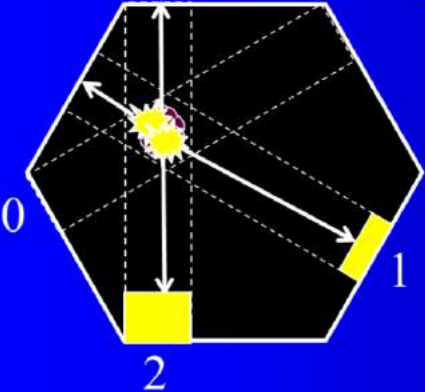
Based on the superposition principle, the measurement and back projection of an arbitrary input signal can be expressed as the convolution of the input signal and the impulse response. In Fourier domain, convolution becomes multiplication and the Fourier transform, so the blurring can be compensated by dividing the Fourier transform of the backprojected signal with the transfer function of the system, which is the Fourier transform of the impulse response. This requires filtering with a so called ramp filter, which is proportional to the absolute value of the frequency vector.

If the signal to noise ratio is not high enough, we get nothing but a mess of noise after reconstruction.

This method is fast if multi-dimensional fast Fourier transformation is applied, but it has a significant problem. Ramp filter is a high-pass filter, so not only the blurring is eliminated but high frequency noise is also amplified.


Zaj!

Becsapódások véletlen események!



Emissziós tomográfia:
Poisson eloszlás

$$P\{d\} = \frac{\lambda^d}{d!} \cdot e^{-\lambda}$$

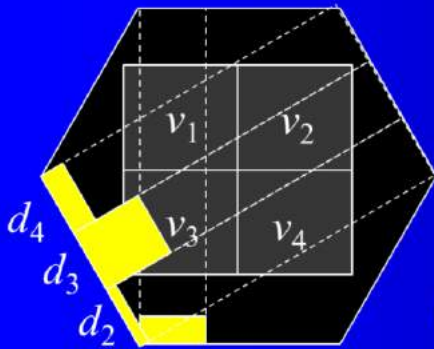
Nagy számok  vénye?

Noise can be significant especially in emission tomography, due to the random nature of the measurement process and the ignored physical effects.

For example, when a positron-electron pair is annihilated, the direction of the generated photon pair will be random due to quantum effects. The number of hits in a LOR is also random and follows Poisson distribution.

It is tempting that we believe in the theorem of large numbers and state that event frequency is close to the expected values, but it is a very bad idea. In practical PET measurements, at least when we do not wish to kill the patient, the number of hits in a detector is small, so we are very far from the comfortable zone offered by the theorem of large numbers.

Algebrai visszavetítés



$$d_1 = A_{11}v_1 + A_{13}v_3$$

Lin egyenlet ($V < D$):

$$\underset{(D)}{d} = \underset{(D \times V)}{A} \cdot \underset{(V)}{v}, \quad d = [d_1, \dots, d_D]$$

$$v = [v_1, \dots, v_V]$$

Moore féle pseudo-inverz:

$$\underset{(V \times D)}{A^T} \cdot \underset{(D)}{d} = \underset{(V \times V)}{\boxed{A^T \cdot A}} \cdot v$$

$$\boxed{v = (A^T \cdot A)^{-1} \cdot A^T \cdot d = A^+ \cdot d}$$

Expectation Maximization



Amelyik éppen a mérési eredményt maximálja $P(d|v)$

- Likelihood maximalizálás: $\log P$
- Előny: a mérés statisztikai modelljét is figyelembe veszi

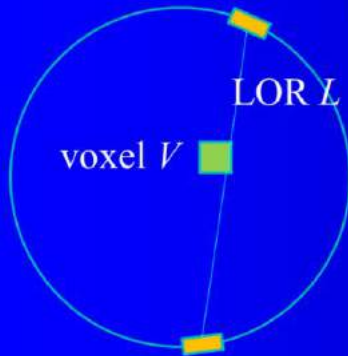
If we have a statistical model of the measurement process, that is we know the distribution of the hit numbers, we can make it even better since this information can be built into the reconstruction and can replace the blind Euclidean distance. The concept is called maximum likelihood estimation in statistics. Based on the observed or measured values, we search for an activity field that would produce this set of measurements with the highest probability. So again, we end up to solve an optimization problem but not the probability needs to be maximized. The typical trick is to maximize the logarithm of this probability which would not modify the optimum since the logarithm is a monotonous function but turns products to sums which makes our life much easier when derivatives are computed during a gradient search.

PET: iteratív séma

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{y}$$

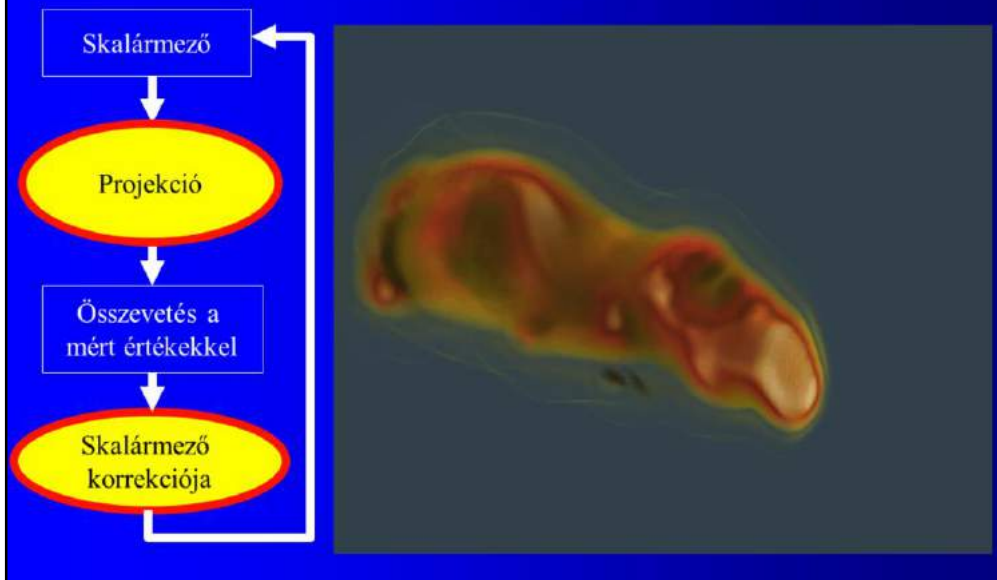
$$\mathbf{x}_V^{(n+1)} = \mathbf{x}_V^{(n)} \frac{\sum_L \mathbf{A}_{LV} \cdot \frac{\mathbf{y}_L}{\sum_{V'} \mathbf{A}_{LV'} \mathbf{x}_{V'}^{(n)}}}{\sum_L \mathbf{A}_{LV}}$$

Poisson és pozitivitási kényszer



\mathbf{A}_{LV} : annak valószínűsége,
hogy a V voxelben
bekövetkezett bomlást az
 L LOR detektálja

Iteratív séma



Inverz problémák

Projekció (Tomográf) Aktivitás eloszlás Beütések száma a detektorokban

$$f(\mathbf{x}) = \mathbf{y} = \tilde{\mathbf{y}} + \mathbf{n} \quad \longrightarrow \quad \mathbf{x} = \dots$$

- Ill-posed: nincs megoldás vagy nem egyértelmű.
- Nem ismerjük \mathbf{n} -et (zaj).
- Közelítő megoldásokból melyik (zaj hatása)?
- Plusz információ bevitele: regularizáció

Megoldási sémák

Algebrai

$$\min \|f(\mathbf{x}) - \mathbf{y}\|$$

2-es norma

Valószínűségi

$$\max \{\log P(\mathbf{y} | \mathbf{x})\}$$

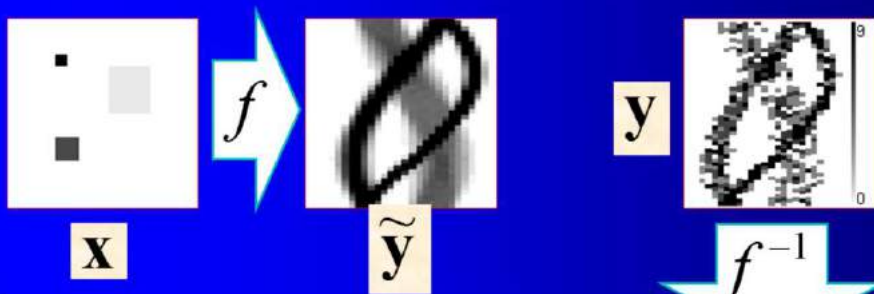
\mathbf{y} Gauss eloszlású

Kullback-Leibler
divergencia

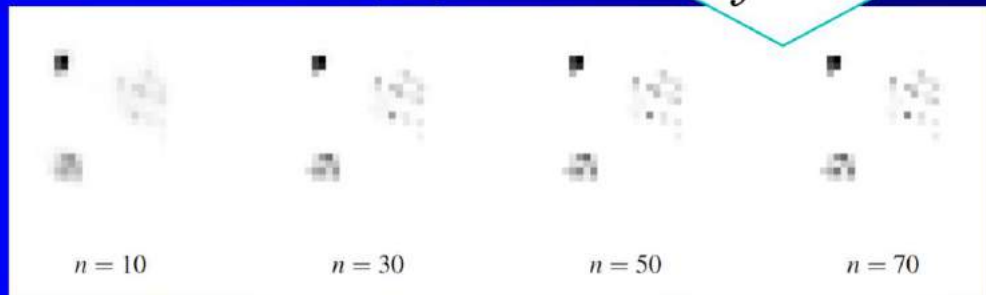
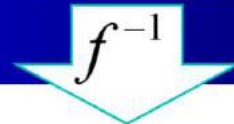
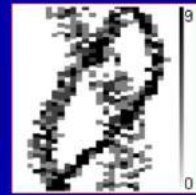


\mathbf{y} Poisson eloszlású

Overfitting



\mathbf{y}



Regularizáció

Likelihood

Regularizációs
paraméter

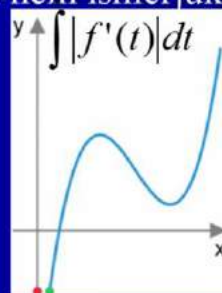
Zajos megoldások
büntetése

$$\min\{-\log P(\mathbf{y} | \mathbf{x}) + \lambda R(\mathbf{x})\}$$

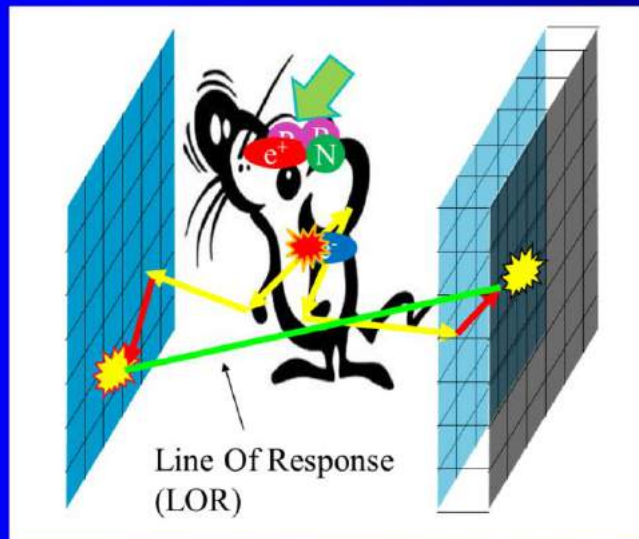
- $R(\mathbf{x})$ „rossz” megoldásoknál nagy, „jó” megoldásoknál kicsi (tökéletes megoldást ne büntesd, de nem ismerjük), konvex függvény.

– Teljes variáció (TV):

$$R(x) = \int_V |\nabla x(v)| dv$$



PET

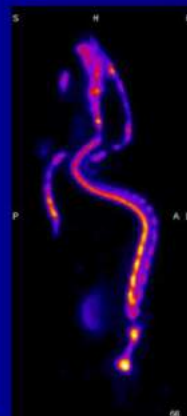
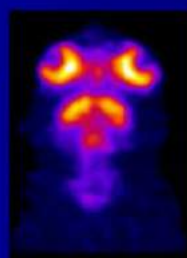
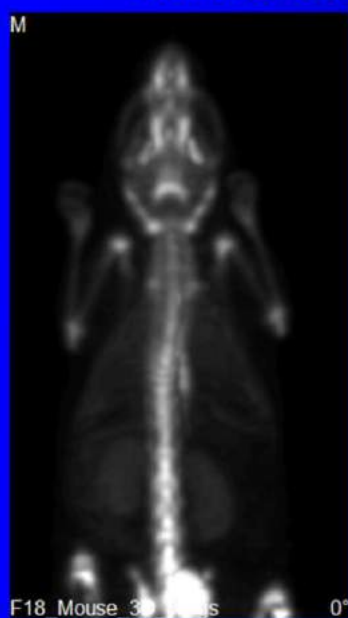


Additionally, noise can also come from the physical phenomena that are ignored in the simple backprojection operation. In reality, when a positron is born, it is not annihilated instantly but it may take an excursion in the material before it meets an electron. This is called positron range, which depends on the isotope and on the material as well. When the photon pair is generated, the two photons are not exactly parallel if the original impulse of the electron and positron was not exactly zero. The photons may get absorbed or scattered in the measured object, so they do not necessarily follow a straight path. Photons may also be scattered in the detector grid, so a different detector will absorb it not the one where it arrived. Finally, the absorption detection may also make random errors.

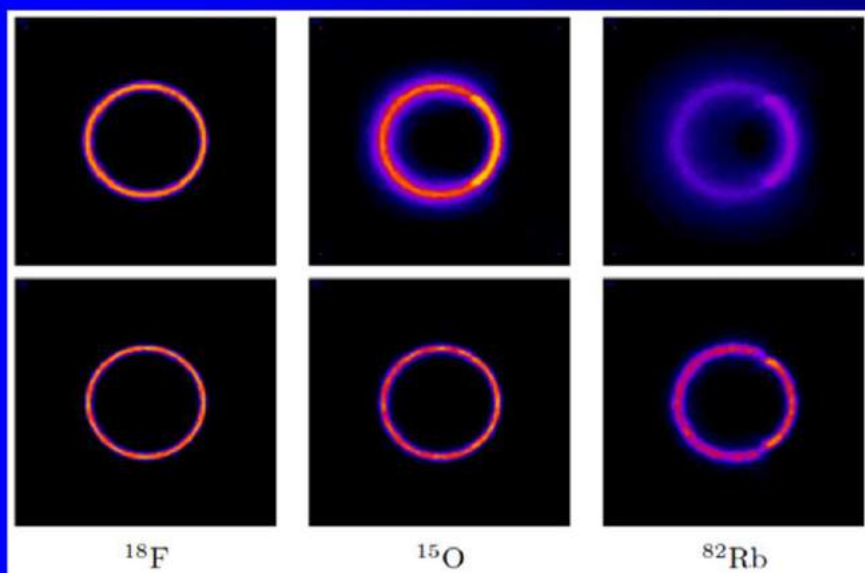
At the end, the LOR reported by the system may be very far and may not go through the point where the positron was born.

Nincs szóródás

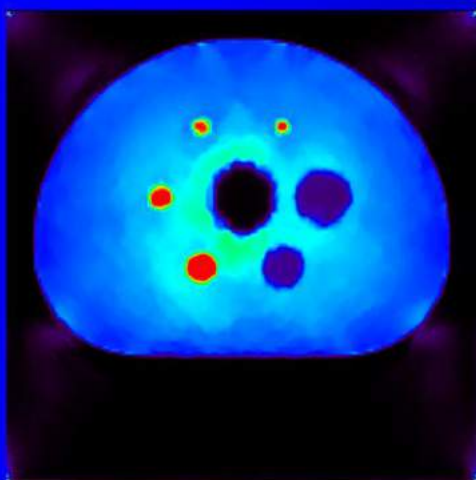
324×315×315 voxels



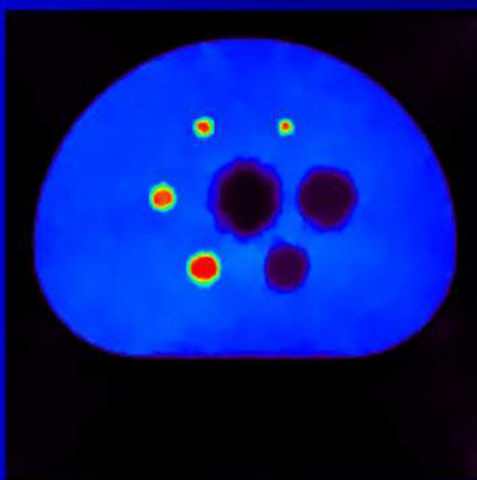
Positron range



Szóródás a testben

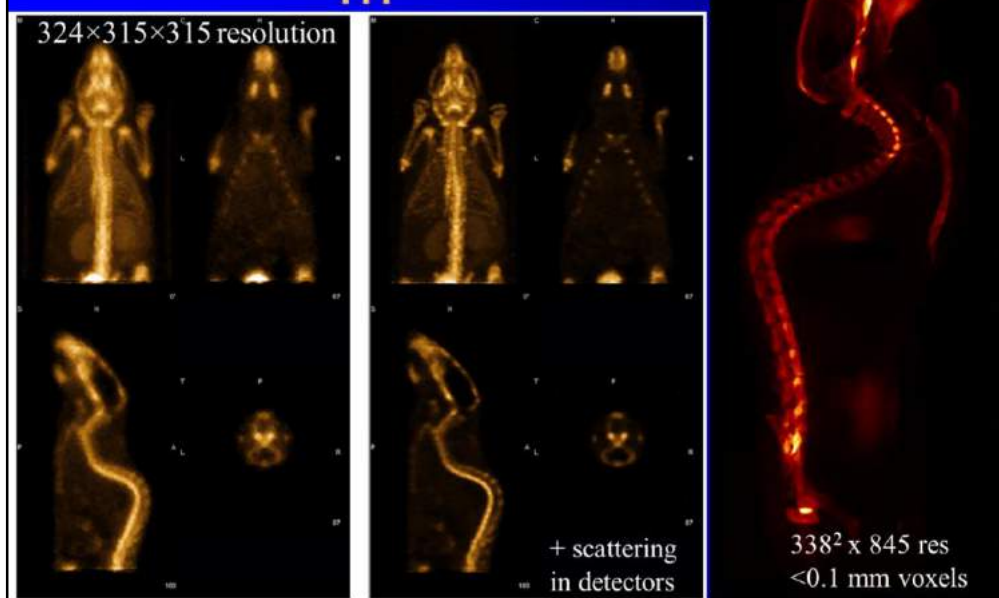


Nélküle



Vele

Szóródás a detektorban



Fraktálok

Szirmay-Kalos László

Fraktálok

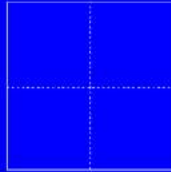
Hausdorff dimenzió

$$N = 2$$



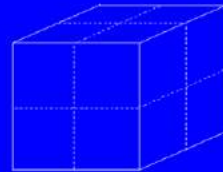
$$r = \frac{1}{2}$$

$$N = 4$$



$$r = \frac{1}{2}$$

$$N = 8$$



$$r = \frac{1}{2}$$

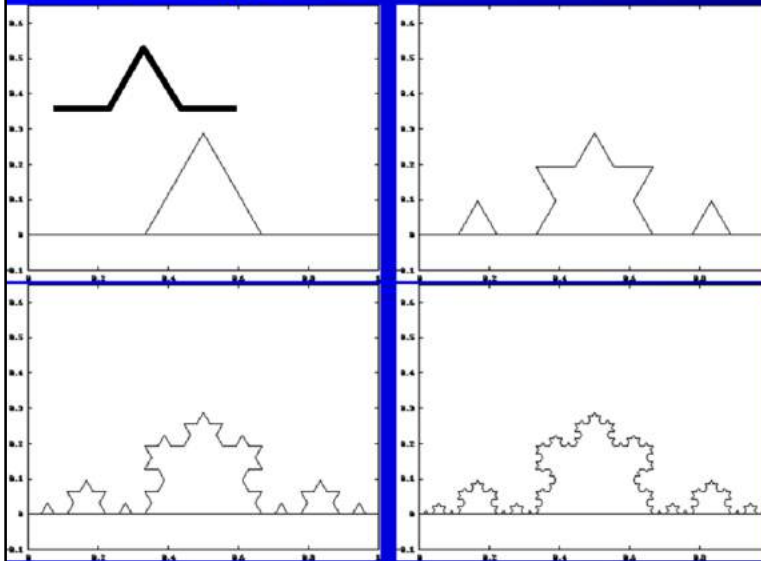
$$N = 1/r^D$$

$$D = (\log N) / (\log 1/r)$$

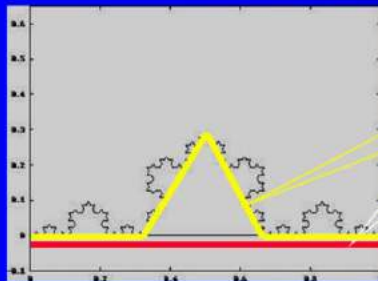
Koch görbe

$$D = (\log 4) / (\log 3) = 1.26$$

$$N = 4,$$
$$r = 1/3$$



Nem önhasznó objektumok dimenziója



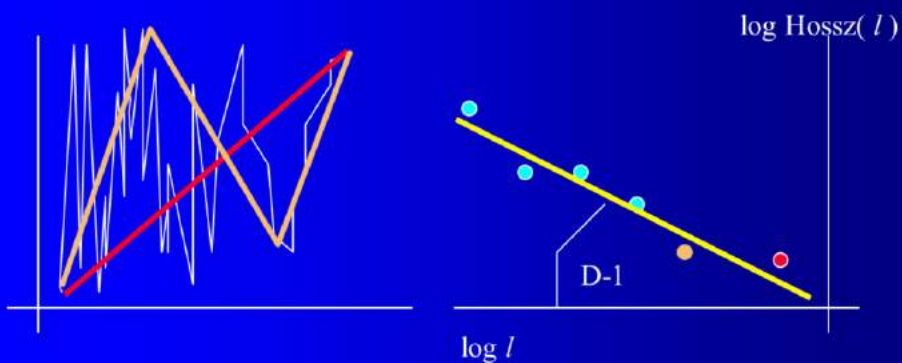
Vonalzó (l)	db
1	1
$r = 1/3$	$N = 4$
r^2	N^2
r^m	N^m

$$\text{Hossz}(l) = l \, db = l N^m = l (1/r^D)^m =$$

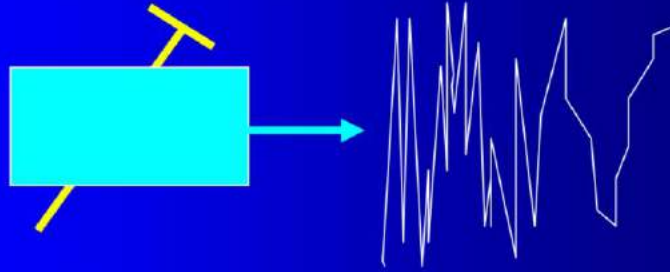
$$= l (1/r^m)^D = 1/l^{D-1}$$

$$D = - \log \text{Hossz}(l) / \log l + 1$$

Dimenziómérés = hossz mérés



Fraktálok előállítása



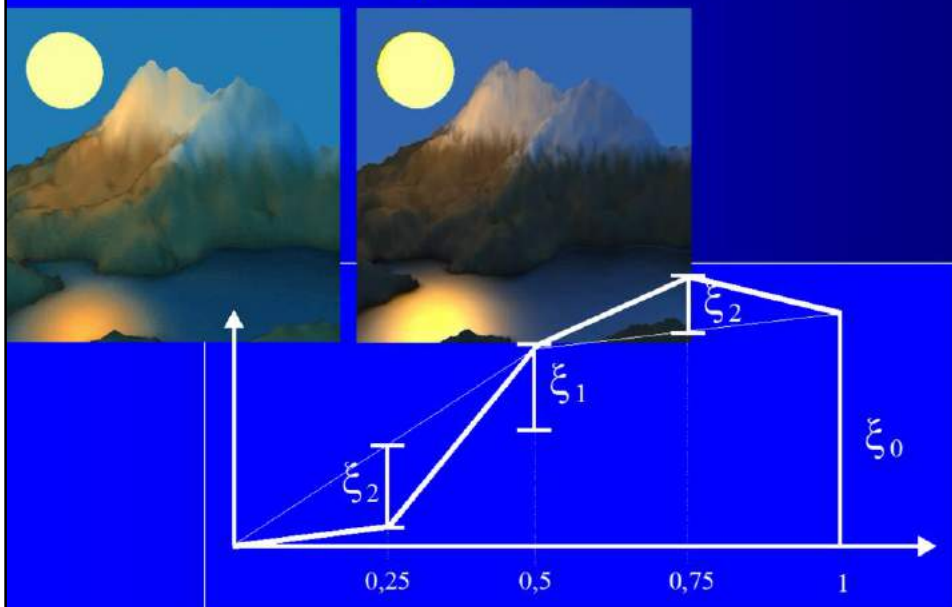
Matematikai gépek:

- Brown mozgás
- Kaotikus dinamikus rendszerek

Brown mozgás - Wiener féle sztochasztikus folyamat

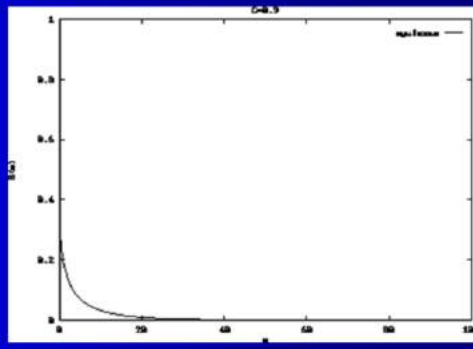
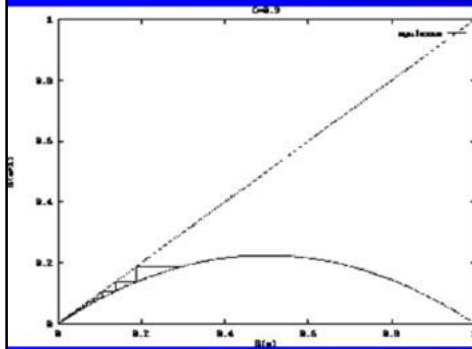
- Sztochasztikus folyamat (véletlen függvény)
- Trajektóriák folytonosak
- Független növekményű folyamat
- Növekmények 0 várható értékű normális eloszlás:
 - a független növekményűségből, a szórás az intervallum hosszával arányos

Brown mozgás alkalmazása

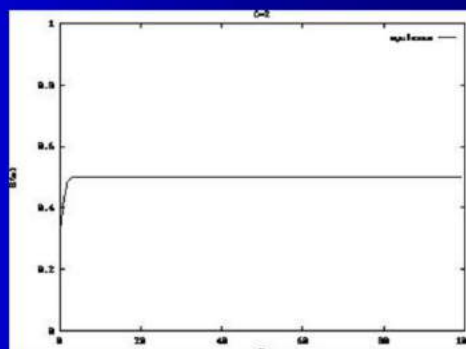
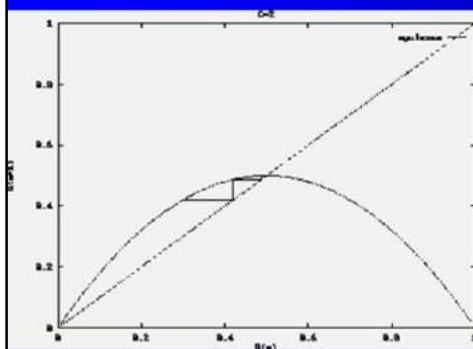


Kaotikus dinamikus rendszer: nyulak kis C értékre

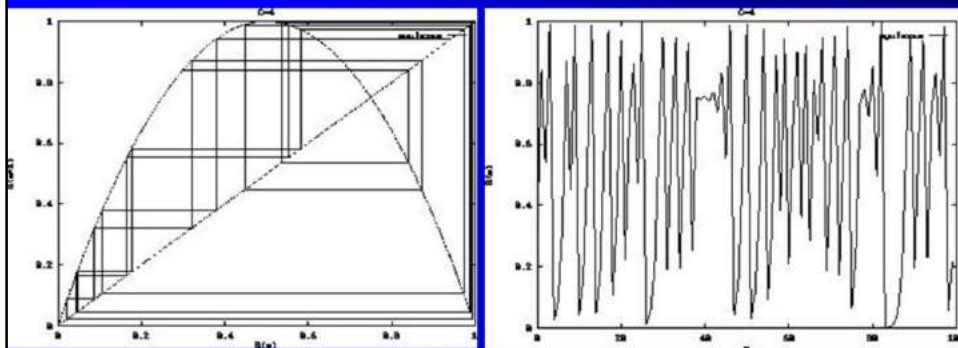
$$S_{n+1} = C S_n (1 - S_n)$$



Kaotikus dinamikus rendszer: nyulak közepes C értékre



Kaotikus dinamikus rendszer: nyulak nagy C értékre

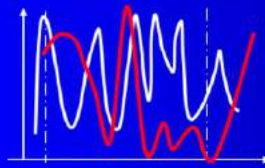


Pseudo véletlenszám generátor

- Iterált függvény:

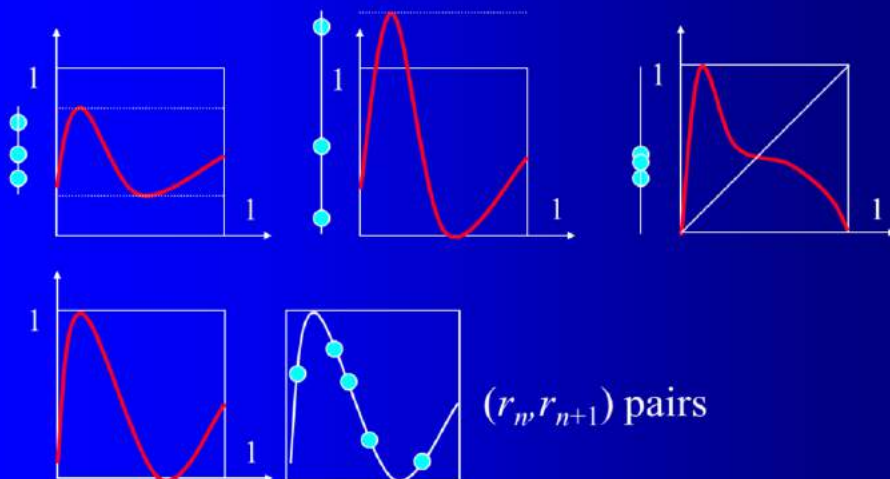
$$r_{n+1} = F(r_n)$$

- véletlenként hat



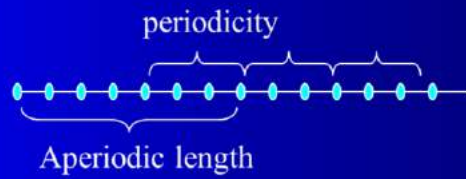
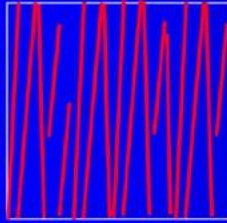
F nagy derivált!

Rossz: $r_{n+1} = F(r_n)$



Jó F

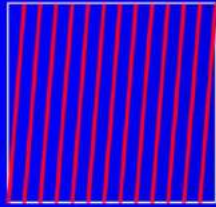
- Sűrűn kitölti a négyzetet
- Mindenütt nagy derivált
- a $[0, 1]$ -ben van



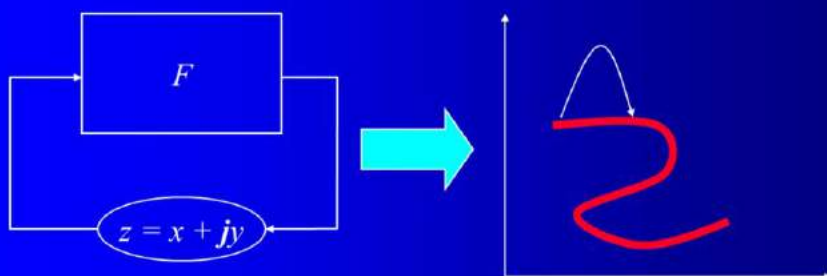
Kongruens generátor

$$F(r) = \{ g \cdot r + c \}$$

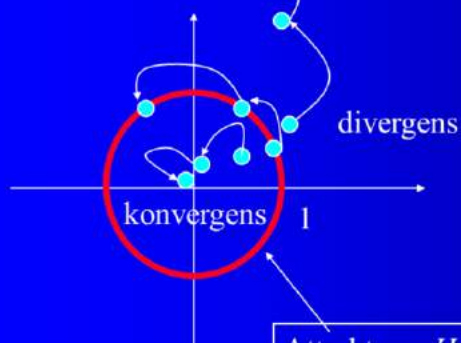
$g \cdot r + c$ tört része
 g nagy



Kaotikus rendszerek a síkon



$$z \rightarrow z^2$$



$$z = r e^{j\phi}$$

$$r \rightarrow r^2$$

$$\phi \rightarrow 2\phi$$

Attraktor: $H = F(H)$

Attraktor előállítás

- Attraktor a labilis és a stabilis tartomány határa: kitöltött attraktor = amely nem divergens
 - $z_{n+1} = z_n^2$: ha $z_\infty < \infty$ akkor fekete
- Attraktorhoz konvergálunk, ha az stabil
 - $z_{n+1} = z_n^2$ attraktora labilis

Inverz iterációs módszer

$$H = F(H)$$

→

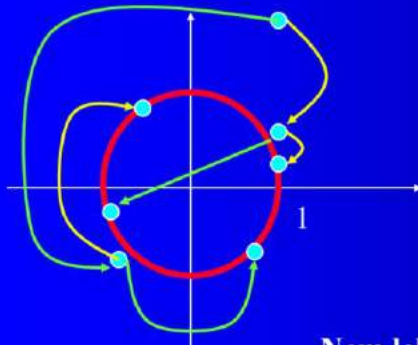
$$H = F^{-1}(H)$$

$$z_{n+1} = z_n^2$$

$$z_{n+1} = \pm \sqrt{z_n}$$

$$r_{n+1} = \sqrt{r_n}$$

$$\phi_{n+1} = \phi_n/2 + \{0,1\} \cdot \pi$$

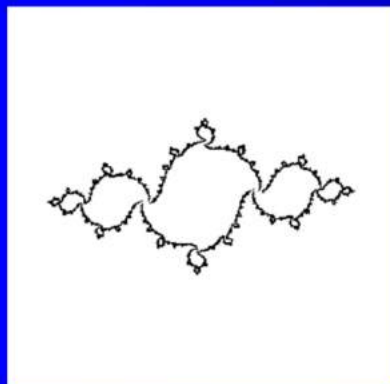


$$r_n \approx 1$$

$$\phi_n \approx \{0,1\}_n \cdot \{0,1\}_{n-1} \cdot \{0,1\}_{n-2} \dots \cdot \pi$$

Nem lehet csak egy értékkel dolgozni ???

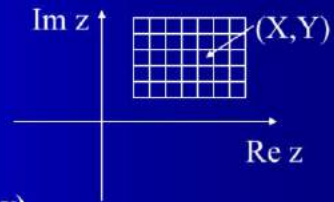
Julia halmaz: $z \rightarrow z^2 + c$



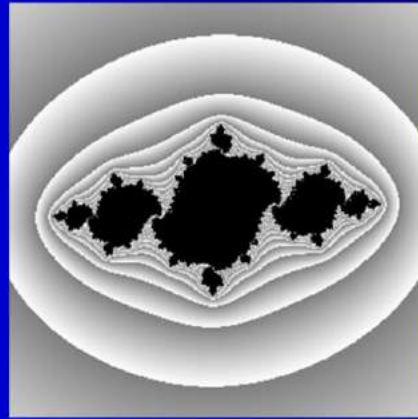
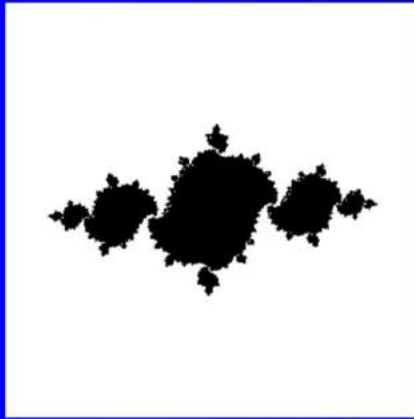
Kitöltött Julia halmaz: algoritmus

FilledJuliaDraw ()

```
FOR Y = 0 TO Ymax DO
  FOR X = 0 TO Xmax DO
    ViewportWindow(X,Y → x, y)
    z = x + j y
    FOR i = 0 TO n DO z = z2 + c
    IF |z| > “infinity” THEN WRITE(X,Y, white)
    ELSE WRITE(X,Y, black)
    ENDFOR
  ENDFOR
END
```



Kitöltött Julia halmaz: kép



Julia halmaz inverz iterációval

JuliaDrawInverseIterate ()

Kezdeti z érték választás

FOR $i = 0$ TO n DO

$x = \text{Re } z, \quad y = \text{Im } z$

IF ClipWindow(x, y)

WindowViewport($x, y \Rightarrow X, Y$)

Pixel(X, Y) = fekete

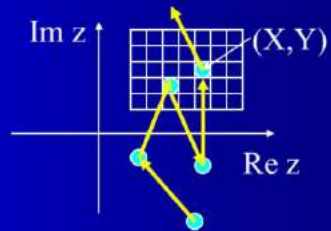
ENDIF

$z = \sqrt{z - c}$

if ($\text{rand}() > 0.5$) $z = -z$

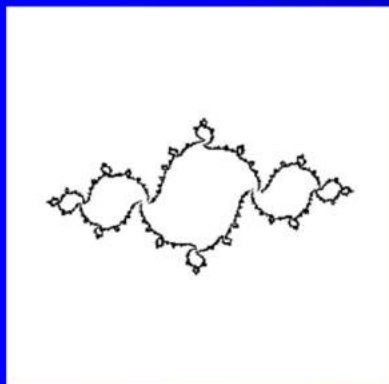
ENDFOR

END



Kezdeti z érték:
 $z^2 = z - c$ gyöke

Julia halmaz

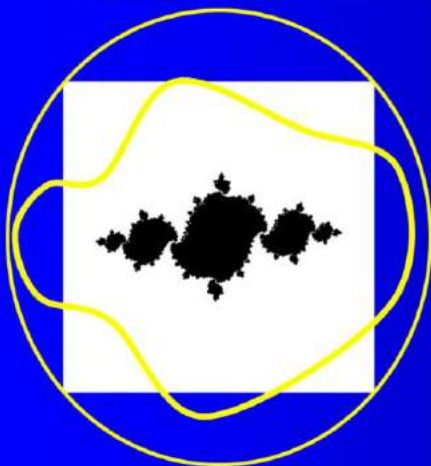


összefüggő

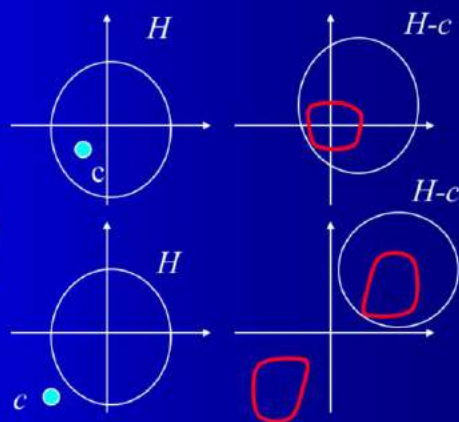


nem összefüggő,
Cantor féle halmaz

Julia halmaz összefüggőse

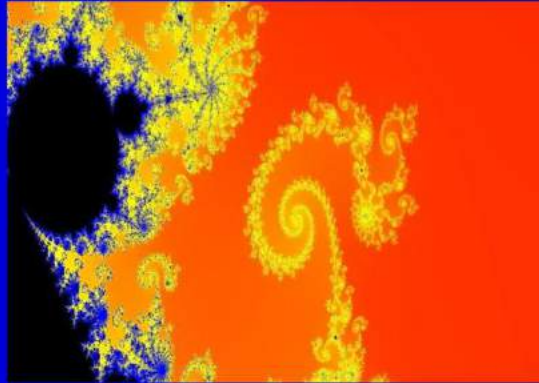
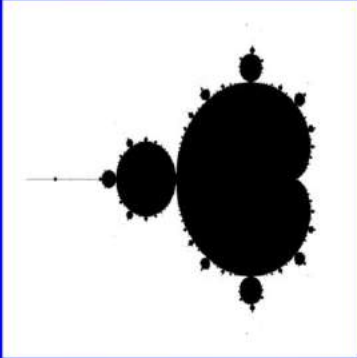


$$z_{n+1} = \pm \sqrt{z_n - c}$$



Mandelbrot halmaz

Azon c komplex számok, amelyekre a
 $z \rightarrow z^2 + c$ Julia halmaza összefüggő

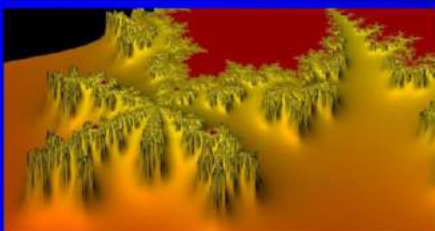
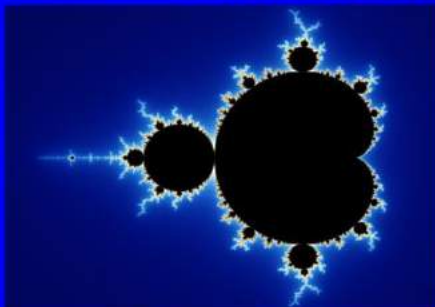


Mandelbrot halmaz, algoritmus

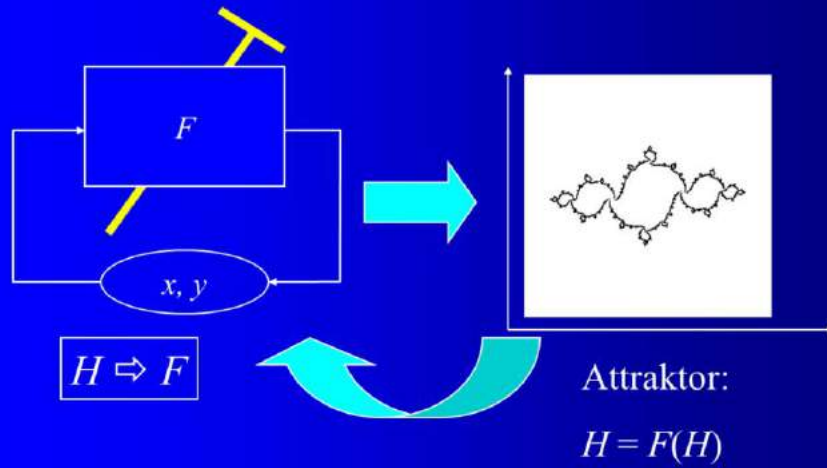
MandelbrotDraw ()

```
FOR Y = 0 TO Ymax DO
  FOR X = 0 TO Xmax DO
    ViewportWindow(X,Y → x, y)
    c = x + j y
    z = 0
    FOR i = 0 TO n DO z = z2 + c
    IF |z| > "infinity" THEN WRITE(X,Y, white)
    ELSE WRITE(X,Y, black)
  ENDFOR
ENDFOR
END
```

„Matematikát játszó” Mandelbrot halmazok



Inverz feladat: IFS modellezés



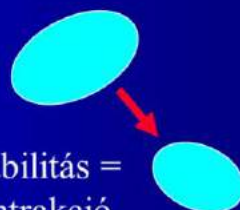
F : szabadon vezérelhető, legyen stabil attraktora

F: többértékű lineáris leképezés

$$F = W_1 \vee W_2 \vee \dots \vee W_n$$

$$W(x,y) = [ax + by + c, dx + ey + f]$$

Stabilitás =
kontrakció



$$H = W_1(H) \cup W_2(H) \cup \dots \cup W_n(H)$$

$$H = F(H)$$



IFS rajzolás: iterációs algoritmus

IFSDraw ()

Legyen $[x,y] = [x,y] A_1 + q_1$ megoldása a kezdő $[x,y]$

FOR $i = 0$ TO n DO

IF ClipWindow(x, y)

WindowViewport($x, y \Rightarrow X, Y$)

Write(X, Y, color);

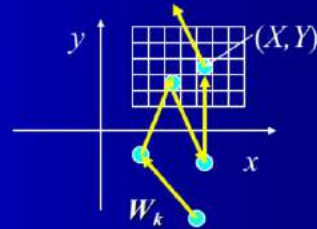
ENDIF

Válassz k -t p_k valószínűséggel

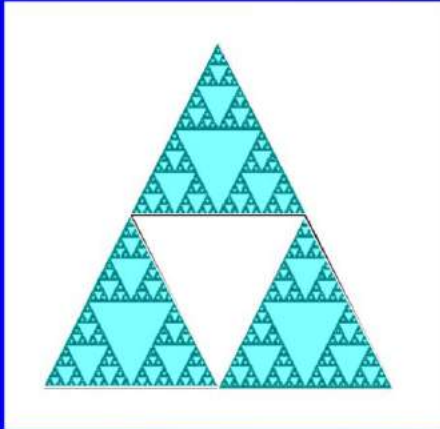
$[x,y] = [x,y] A_k + q_k$

ENDFOR

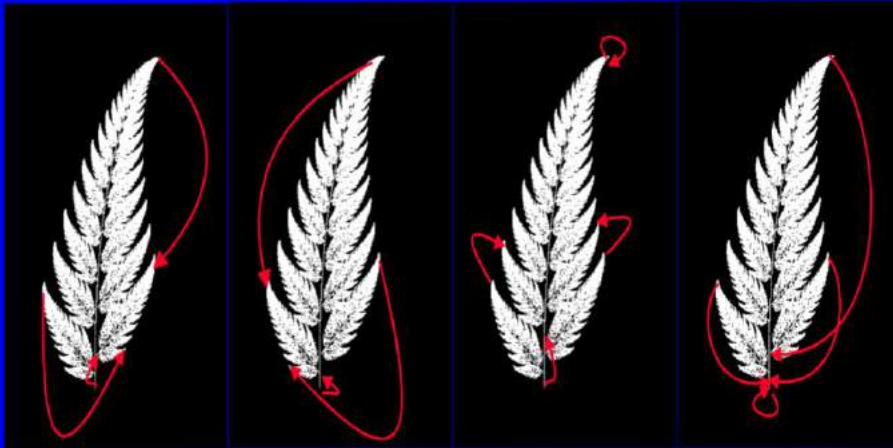
END



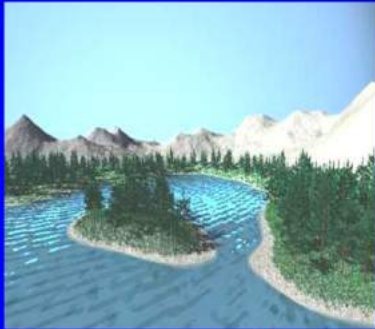
Egyszerű IFS-ek



IFS modellezés



IFS képek



Globális illumináció (GI)

Szirmay-Kalos László

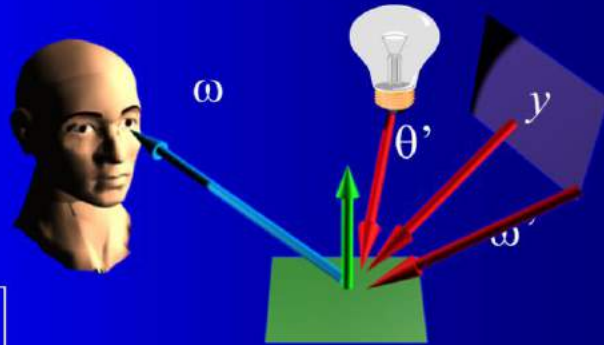
Monte-Carlo methods, random number generation.

Practical application: rendering in computer graphics

Árnyalási egyenlet

Radiancia = Emisszió + Megvilágítás * Visszaverődés

$$L(x, \omega) = L^e(x, \omega) + \int_{\Omega} L(y, \omega') \cdot f_r(\omega', \omega) \cos\theta' d\omega'$$



$$L = L^e + \tau L$$

GI megoldás

- Lokális illumináció

$$L = L^e + \mathcal{T}L \approx L^e + \mathcal{T}L^e$$

- Expanzió

$$L = L^e + \mathcal{T}L = L^e + \mathcal{T}L^e + \mathcal{T}^2L =$$

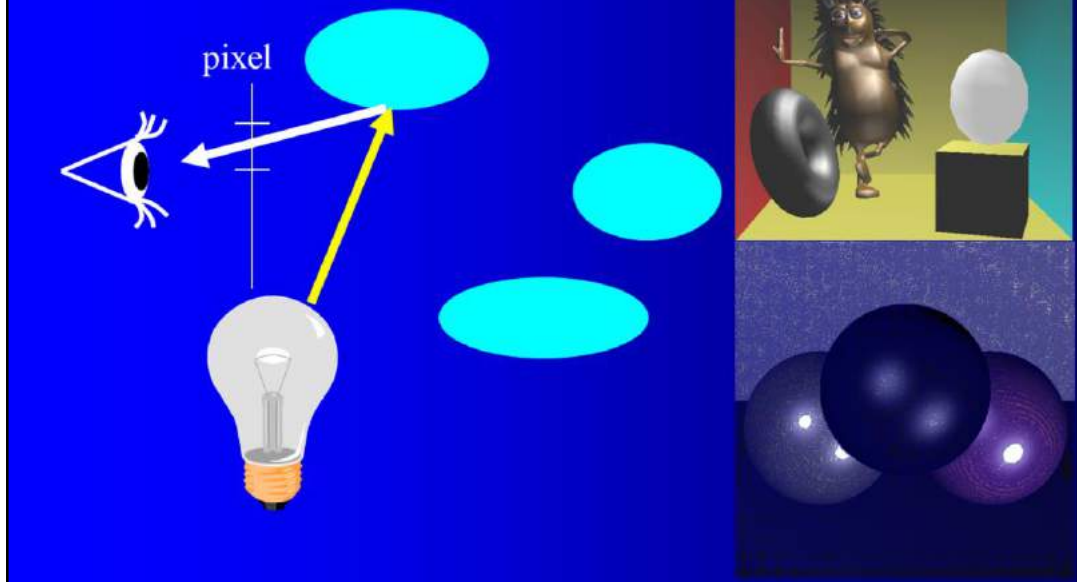
$$L^e + \mathcal{T}L^e + \mathcal{T}^2L^e + \mathcal{T}^3L = \sum \mathcal{T}^n L^e =$$

$$L^e + \mathcal{T}(L^e + \mathcal{T}(L^e + \dots))$$

- Iteráció

$$L_n = L^e + \mathcal{T}L_{n-1}$$

Követett fényutak: Lokális illumináció



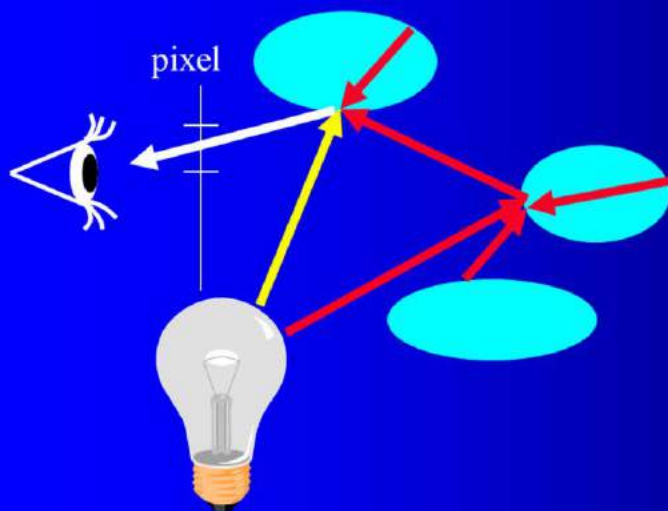
The evaluation of high-dimensional integrals is, of course, rather time-consuming, thus different algorithms take drastic simplifications to increase the computational speed.

Local illumination algorithms consider only those light paths that connect the light source to the eye by a single reflection and ignore indirect illumination completely. The resulting image cannot display mirroring and refracting objects and those points that are not visible from the light sources are dark.

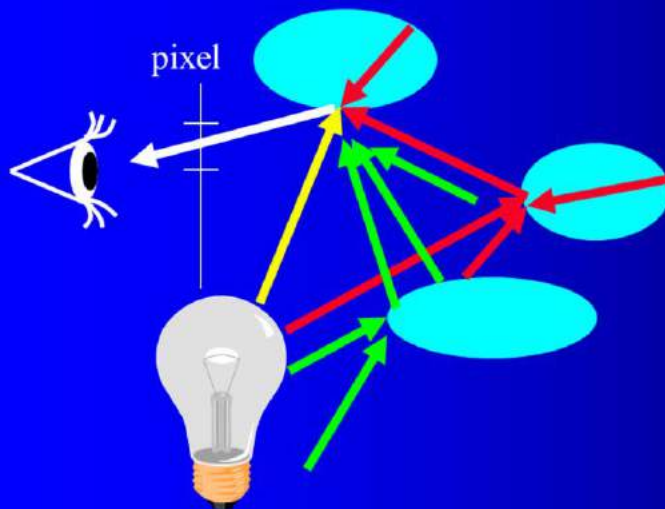
Recursive ray-tracing, on the other hand, allow indirect illumination coming from the ideal reflection and refraction directions, and thus introduces ideal mirrors and refracting objects.

However, this is still just a fraction of the indirect illumination, which could be simulated only by physically valid global illumination algorithms. Note, for example, that the diffuse back wall do not illuminate the spheres in the image obtained with ray-tracing.

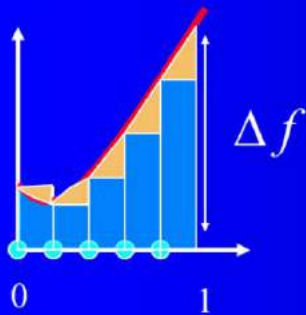
Követett fényutak: Sugárkövetés



Követett fényutak: Globális illumináció



Numerikus integrálás



$$\int f(z) dz \approx 1/M \sum f(z_i)$$

M minta



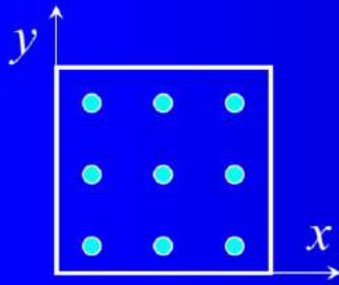
$$\text{Hiba} = \Delta f/2/M \cdot 1/M \cdot M = \Delta f/2/M = O(M^{-1})$$

Let us consider the problem of dense samples used to estimate an integral. Suppose, for the sake of simplicity, that the integral is one-dimensional and the domain is the unit interval. The simplest integration rule places samples regularly in the domain, evaluates the integrand at these samples, and approximates the area below the function by the total area these bricks. This results in the following sum that approximates the integral.

The error of the integration is the total area of the triangle-like objects between the function curve and the bricks, which equals to the product of the average height of the triangles, the base, and the number of triangles.

We can conclude that the error is proportional to the total change, called variation of the function, and inversely proportional to the number of samples.

Magasabb dimenziókban



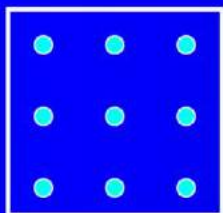
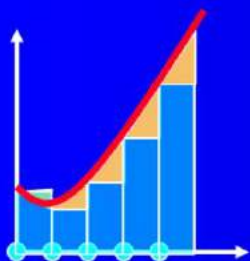
$$\int \int f(x,y) \, dy \, dx = \int F(x) \, dx$$

$F(x)$

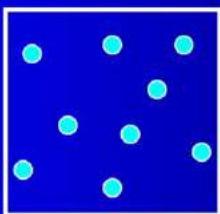
$n = \sqrt{M}$ mintaszám

$$\text{Error} = O(n^{-1}) = O(M^{-0.5})$$

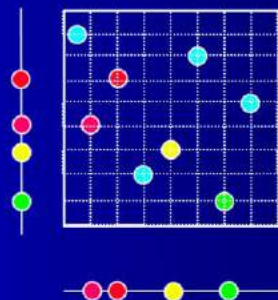
Sűrű minták magasabb dimenziókban



Véletlen:
Monte Carlo method



Determinisztikus,
Alacsony diszkrepancia
quasi Monte Carlo



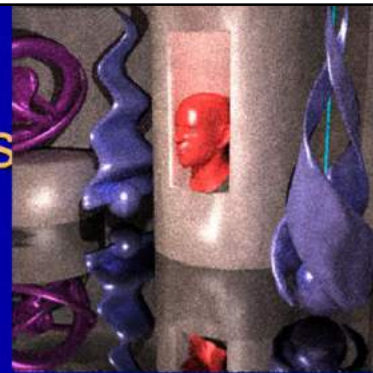
Monte Carlo Integrálás

Integrál = várható érték

$$\int f(z) dz = \int f(z)/p(z) \cdot p(z) dz = \\ = E[f(z)/p(z)] \approx 1/M \sum f(z_m)/p(z_m)$$

$$\text{Hiba} < 3 \cdot (f/p \text{ szórása}) \cdot M^{-1/2}$$

99.7% konfidencia szinttel



100 samples/pixel

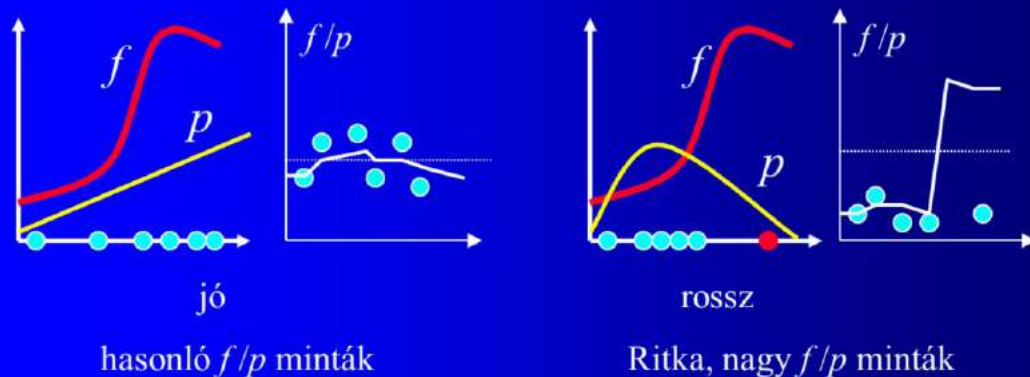
Monte-Carlo integration can also be understood in the following way. In order to evaluate an integral, let us divide and multiply the integrand by a probability density p . Obviously, it does not make any difference.

Looking at this formula, we can realize that this is the expected value of random variable f/p . According to the theorem of large numbers, expected values can be well approximated by averages. Thus taking M samples obtained with probability density p , this average will be a good estimate for the original integral.

The integration error will be proportional to the variation of f/p and inversely proportional to the square root of the number of samples.

Fontosság szerinti mintavételezés

- f/p variációja legyen kicsi:
ahol f nagy p is nagy

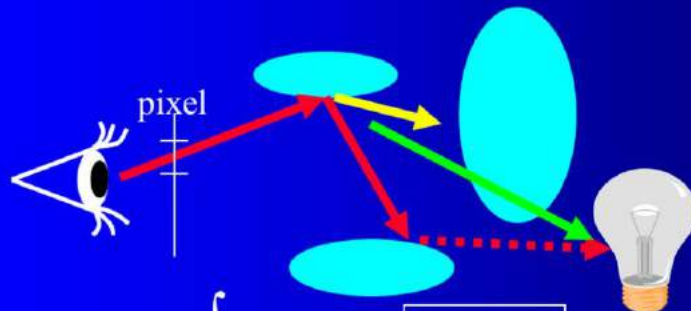


In order to reduce the error, the variation of f/p should be small, that is where the integrand is large, the probability density should also be large. It means that the sampling with this probability density will place more samples where the integrand is large. This concept is called importance sampling.

This figure compares a good and a bad sampling densities. In the first case the f/p terms in the approximating average will be similar, thus the average remains nearly constant as we add new samples.

In the second case, the important region is sampled rarely, thus the approximating average contains many small values when a very large f/p value appears. The corresponding pixel color is dark and suddenly it becomes very bright and remains bright for a long time. This situation should be avoided.

Véletlen bolyongás



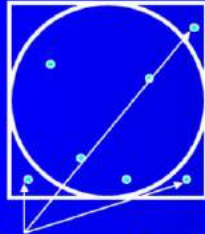
$$L = L^e + \int_{\Omega} L^{\text{in}}(\omega') \underbrace{f_r \cos\theta'}_{w: \text{visszaverődés sűrűség}} d\omega'$$

- BRDF mintavétel: arányos $f_r \cos\theta'$
- Fényforrás mintavétel

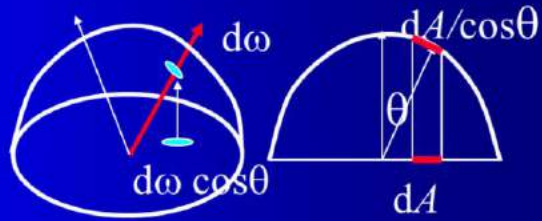
According to importance sampling, when the next step is sampled, we have to prefer those directions from which significant light intensity can be transferred. This depends on two factors, on the incoming intensity and on the material properties that express the ratio of the outgoing and incoming intensities for two particular directions.

Unfortunately, the incoming intensity is not known since we are just about to compute it, thus we have to take approximations. Either we can prefer those continuation directions from which the reflection is likely, or we can continue towards the light sources hoping that the incoming illumination is significant from there. The first approach is called BDF sampling, while the second approach is light source sampling.

Diffúz BRDF mintavételezés



Eldobott minták



1. Egyenletes minták az egységnégyzetben
2. Körön kívüli minták eldobása
do { $x = r_1, y = r_2$ } while ($x^2 + y^2 > 1$)
3. Gömbre vetítés
 $z = \sqrt{1 - x^2 - y^2}$

Végtelen dimenziós integrálok: Orosz rulett

1. MC integrál:

$$\int w_i(L^e + \dots) d\omega_i = E[w_i(L^e + \dots) / p(\omega_i)] = E[L^{\text{refl}}]$$

2. Számítsd ki s valószínűséggel, különben 0

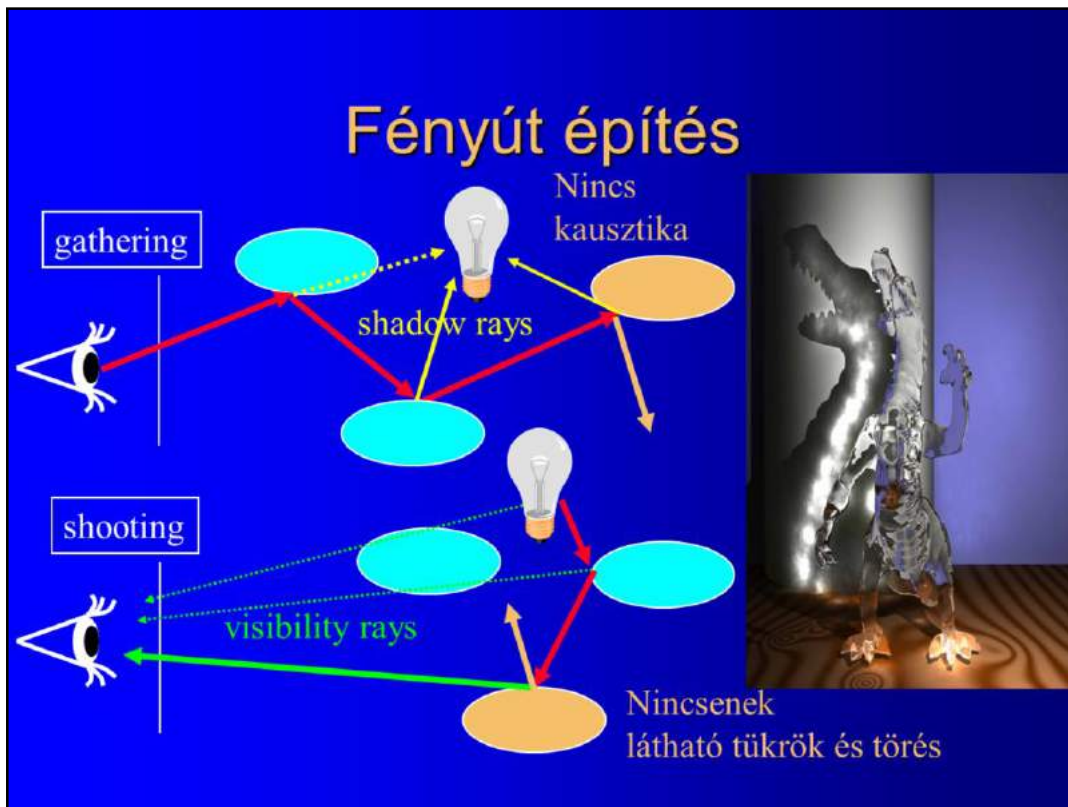
3. Kompenzálj s -sel osztással

Várható érték:

$$E[L^{\text{refl}*}] = s E[L^{\text{refl}}/s] + (1-s) 0 = E[L^{\text{refl}}]$$

Szórás nő:

$$D^2[L^{\text{refl}*}] = s E[(L^{\text{refl}}/s)^2] + (1-s) 0 - E^2[L^{\text{refl}}] = \\ (1/s - 1) E[(L^{\text{refl}})^2] + D^2[L^{\text{refl}}]$$



Path connecting the light sources to the eye can be built from two directions. We can start at the eye and walk in the space opposite to the light generating the next direction with BRDF sampling and then gathering the emission of the visited points. However, if the light sources are small, then only a few walks have non zero contribution, which is responsible for large fluctuations of the average pixel colors. In order to avoid this the walk is forced to go to the light source by connecting the visited points to the light sources by deterministic shadow rays.

Paths can also be obtained starting at the light sources and walking as real photons do. At the reflection points the continuation direction can be samples with BRDF sampling. Again, it is very unlikely that the walks find the eye, thus we force the walk to go to the eye by connecting the visited points by deterministic visibility rays.

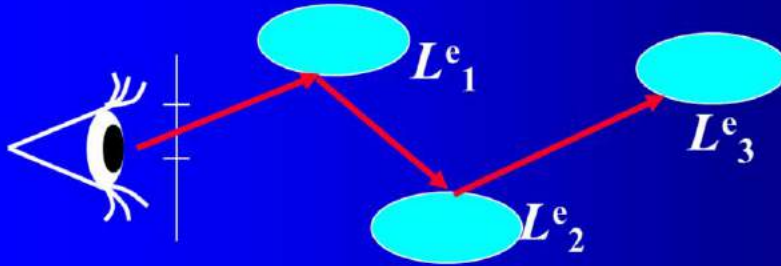
Note that the first approach called gathering or path tracing uses BRDF sampling everywhere except at the last reflection towards the light sources while the second approach called shooting or light tracing uses BRDF sampling everywhere except for the last reflections towards the eye.

If the surfaces at these points are highly specular, or mirror like, then it is very unlikely that the deterministic connection finds that direction that would be preferred by BRDF sampling, which results in high contribution low probability samples that are regarded as bad samples.

Light sources directly illuminating mirrors or ideal refracting objects cause caustics, thus these caustic effects cannot be efficiently rendered by path tracing. For example, this image was obtained with path tracing with 800 samples per pixel. This is quite accurate except for points where the illumination comes from a single reflection of the light source on a close to ideal mirror.

For shooting, visible mirrors and glass objects pose problems, therefore these scenes cannot be efficiently rendered by light tracing.

Path tracing



- BRDF mintavétel: $\Pr\{\text{köv irány}\} \sim \text{Brdf} \cos \theta'$
- Orosz rulett: Befejezés $1 - a_i$ valószínűséggel
- $P = L^e_1 + L^e_2 \underbrace{w_1/p_1/a_1}_{\approx 1} + L^e_3 \underbrace{w_1/p_1/a_1}_{\approx 1} \underbrace{w_2/p_2/a_2}_{\approx 1}$

Path Tracer

Color Trace(ray, depth)

(object, x) = Intersect(ray)

IF no intersection THEN RETURN L_{sky}

color = Direct Lightsource(x, -ray.dir)

if (depth == 0) color += $L_e(x, -\text{ray.dir})$

prob = RussianRoulette (normal, -ray.dir)

IF (prob == 0) RETURN color

prob *= BRDFSampling (newdir, normal, -ray.dir)

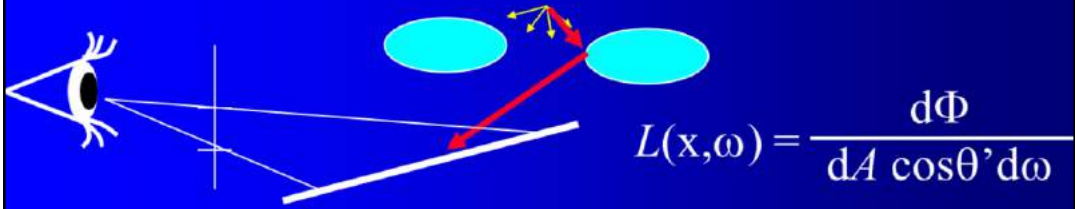
IF (prob == 0) RETURN color

color += Trace(Ray(x, newdir), depth+1) *

$\text{Brd}f(\text{newdir}, \text{normal}, -\text{ray.dir}) \cos\theta' / \text{prob}$

RETURN color

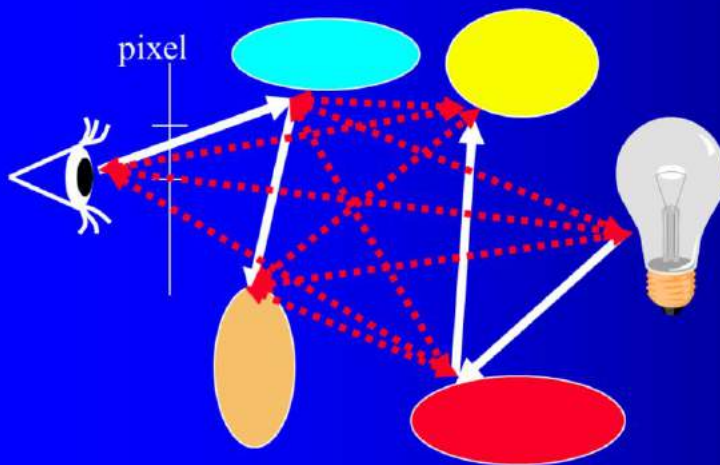
Light tracing



- Source is selected with probability $p^e \propto L^e(x, \omega_1) \cos\theta$
- $\Pr\{\text{next direction}\} \sim \text{BRDF} \cos\theta$
- Termination with $1 - a_i$

$$\bullet P = \underbrace{L^e \cos\theta / p^e}_{\approx \Phi_{\text{total}}} \underbrace{w_1 / p_1 / a_1}_{\approx 1} \underbrace{w_2 / p_2 / a_2}_{\approx 1} \dots w_{\text{eye}} g$$

Bi-directional path tracing

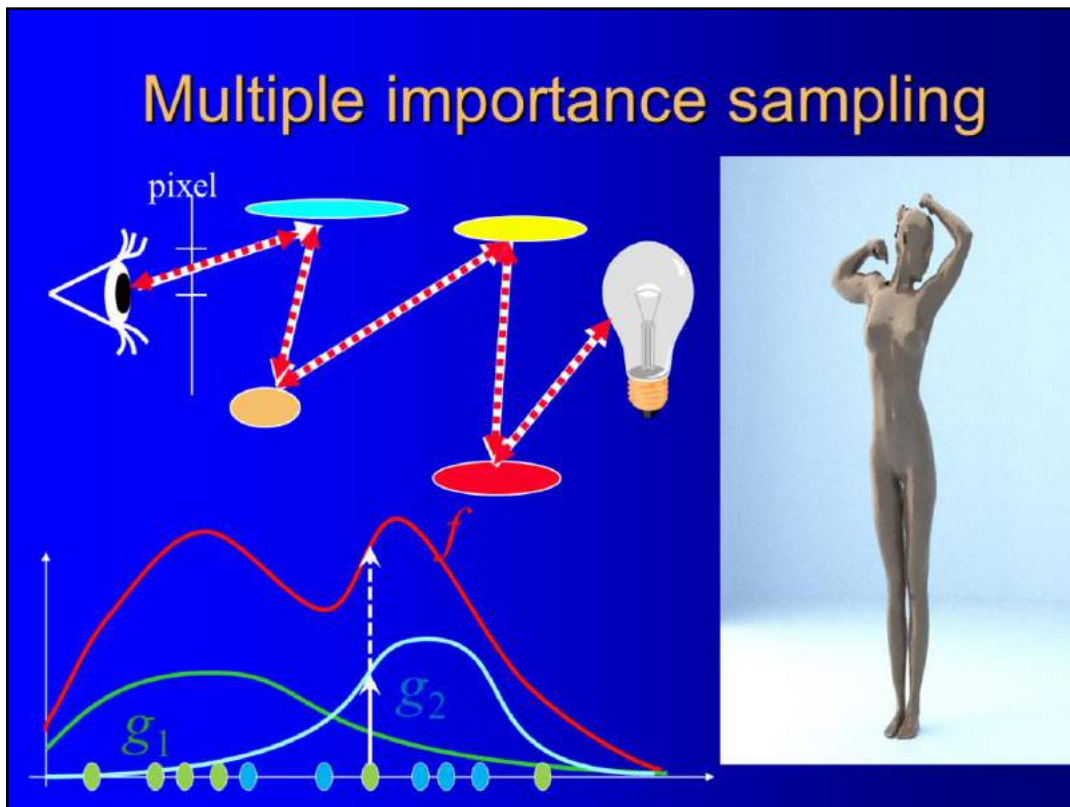


$$L = \Phi / (dx \cos \theta d\omega) = \Phi_{in} \frac{f_r(y) \cos \theta_v f_r(x) \cos \theta_x}{|x - y|^2}$$

Such problematic cases can be solved by bi-directional strategies. Bi-directional path tracing starts a gathering walk from the eye, and a shooting walk from the light source, then connects all visited points of the gathering walk to all visited points of the shooting walk deterministically, generating a complete family of paths.

For those members of this family where the deterministic shadow ray connects two not highly specular objects and the deterministic ray is long, the sample will be good, otherwise the sample will be bad.

The task is then to keep those members of the family that are good and to get rid of bad samples.



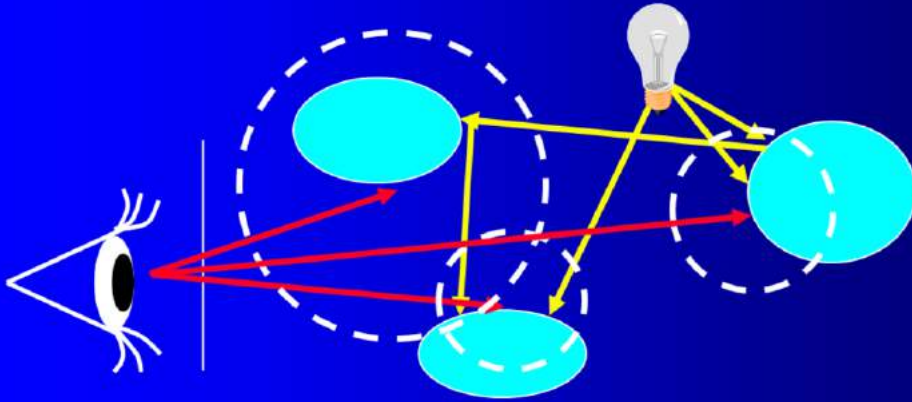
Let us consider a single path connecting the light source to the eye. Bi-directional path tracing could generate this paths by many different ways. For example, it can happen that the length of the light tracing part 4 and a deterministic connecting takes the shooting path to the eye. Or the gathering path can be 1 ray long, placing the deterministic connection between the one ray long gathering path and the three ray long shooting path. Similarly the deterministic connection can appear anywhere in the walk.

These versions correspond to the same sample thus the contributions of these versions are equal. However, the sampling probabilities differ.

How do we know when a particular sample is generated, then it can be considered as a good sample? Recall that what we are afraid of is a high contribution low probability sample. Using this heuristics, we can say that the best version is that one which has the highest sampling probability.

It means that when a bi-directional path is built, we have to compute the generation probability for not only this particular version but for all other versions that would place the deterministic step at different positions. If the probability of the particular version is the maximum, that the sample is kept, otherwise, the contribution of the sample is not computed. This strategy is called maximum heuristics.

Foton térkép

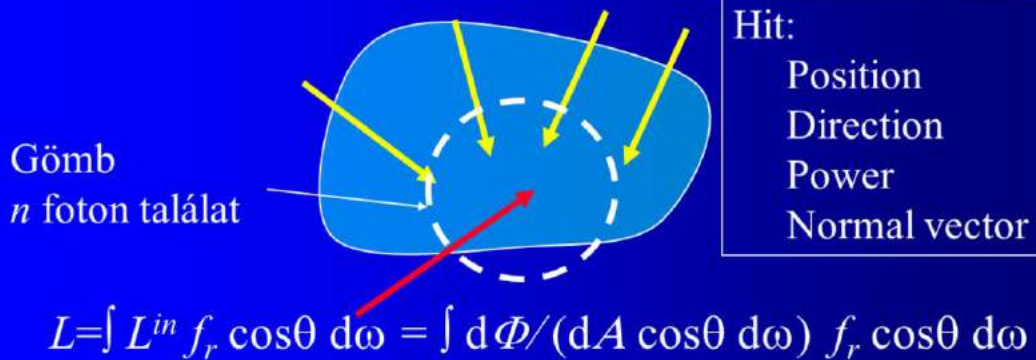


Bi-directional path tracing connects a single gathering path to a single shooting path to obtain complete light paths that are used in the integral quadrature.

If we could store shooting paths somehow, we could connect a single gathering path to all shooting paths simultaneously, thus we could gain much more samples for the integral quadrature. This is the basic idea of the photon map algorithm proposed by Jensen, which uses an approximative representation of the result of all shooting paths.

This algorithm consists of two phases. In the first phase a lot of shooting walks are generated and the photon hits of these walks are stored in an appropriate data structure. Then, in the second phase rays are traced from the eye and the radiance of the visible points are approximated from the photon hits nearby this point.

Foton térkép gyűjtés



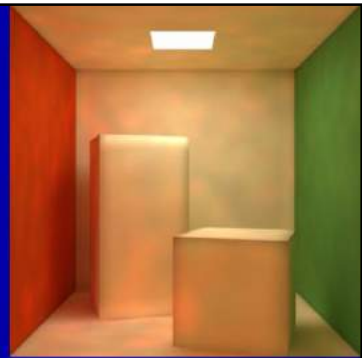
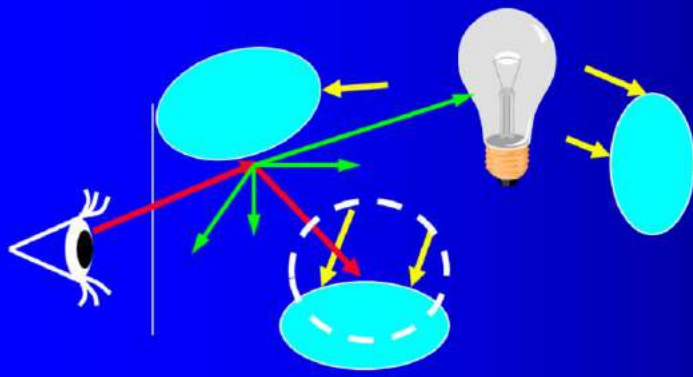
$$L = \sum \Delta\Phi_i / \Delta A f_r$$

$$\Delta A = r^2 \pi$$

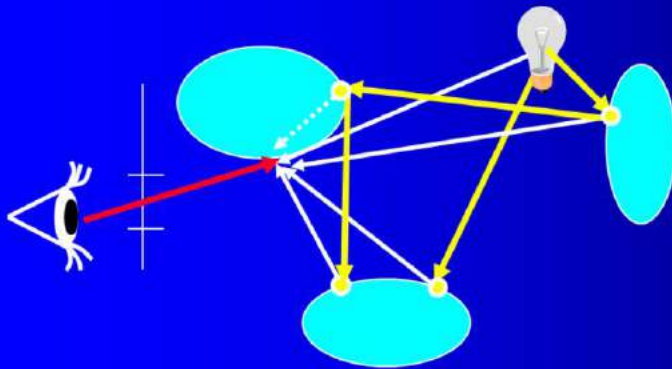
Direkt megjelenítés



Final gathering



Virtuális fényforrások



$$L = \Phi_{\text{in}} \frac{f_r(y) \cos \theta_v f_r(x) \cos \theta_x}{|x - y|^2} v(x, y)$$

The photon map algorithm uses those photons that are close to a point of interest, which limits the number of light paths obtained in a single step.

In instant radiosity, on the other hand, all photons are utilized when the illumination of a point is computed.

This method also consists of two phases. In the first phase a relatively few shooting paths are generated and the photon hits are stored.

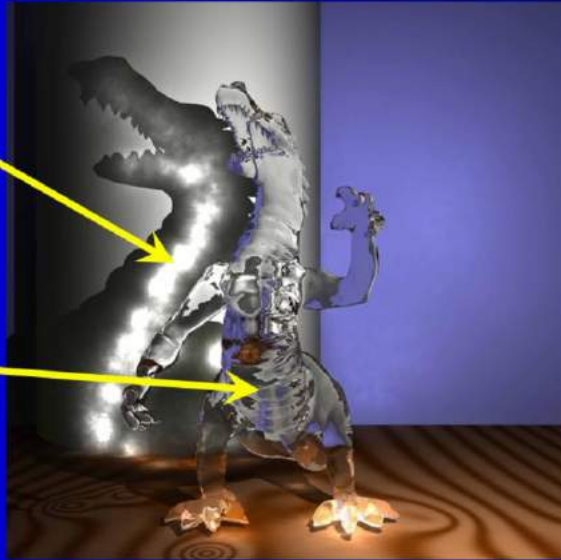
Then in the second phase, these photons act as virtual light sources whose illumination is responsible for indirect illumination.

The original version of this algorithm proposed by Keller, which is known as instant radiosity, the surfaces were assumed to be diffuse, thus the virtual light sources are also diffuse. The illumination of diffuse lights together with depth buffer shadows could be computed by the graphics hardware even at that time, which made this method really fast.

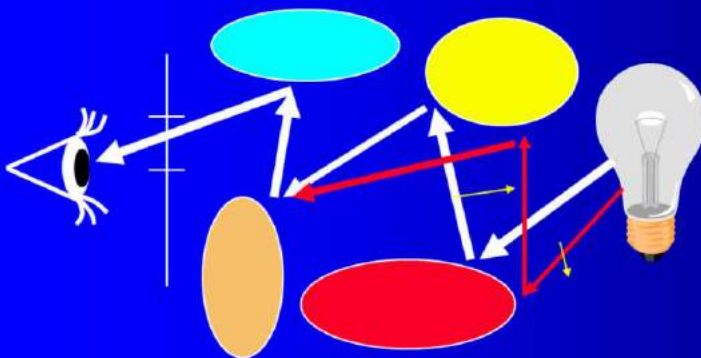
Virtuális fényforrások kiterjesztés

Caustics lövésből

Ideális tükröző, törő
felületek: path tracing



Metropolis light transport



Mutációk: irány, lépésszám váltás

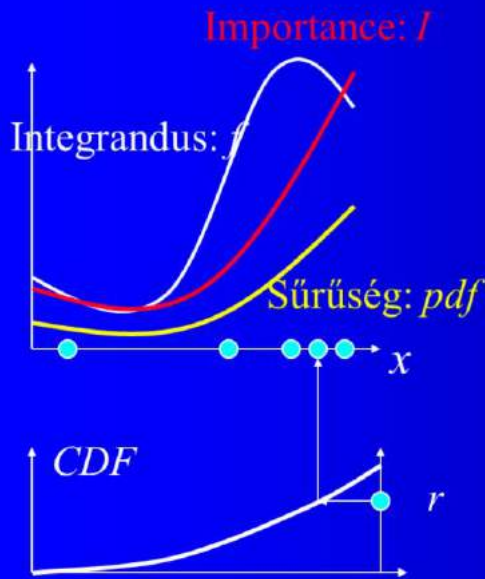
When it comes to the global illumination problem a sample is a light path connecting the light source to the eye, thus mutating the sample means a perturbation of this paths. Such perturbations should change all properties of the path with positive probability, as for example, the directions, the origins and the length as well.

Although, if this requirement is met, then the algorithm will converge to the correct result no matter what kind of mutations are used, the applied perturbation strategy significantly affects the speed of convergence.

For example, on specular surfaces it would be worth gradually refining the perturbation size, which cannot be made by the original approach, and consequently the original method is really efficient just on very difficult scenes. Moreover, if the paths are mutated, then the mutations will not be symmetric, which makes the formulae and the implementation more complicated.

Considering this, the efficient implementation of this seemingly brilliant and simple idea is not at all trivial, and the method has not become as popular as expected.

Klasszikus fontosság szerinti mintavétel



1. I ami közelíti f -t
2. I normalizálása

$$pdf = I / \int I \, dx$$
2. Valószínűségeloszlás

$$CDF(y) = \int^y pdf \, dx$$
3. Mintavétel:
 Egyenletes eloszlás
 transzformálása
 r in $[0,1]$:

$$x = CDF^{-1}(r)$$

The process of constructing the sampling distribution and generating the samples consists of the following steps:

First we find a scalar importance function that mimics the original, usually non-scalar integrand.

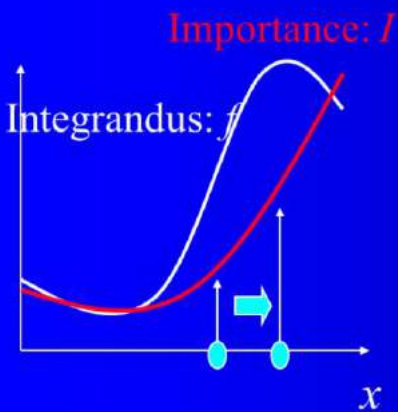
The importance is normalized to obtain a probability density, which is integrated to establish the cumulative probability distribution.

Finally the samples are generated from uniformly distributed pseudo or quasi random numbers by transforming them with the inverse of the cumulative distribution function.

Unfortunately, this process imposes severe requirements on the importance function, namely, we should know and analytically integrate it, and its integral should be invertible.

These requirements can only be met if the importance function is rather simple, which makes it impossible to mimic the integrand properly.

Metropolis mintavételezés



1. I ami hasonló f
2. Normalizáló konstans:
$$b = \int I \, dx$$
3. Mintavétel:
Mutáció/Elfogadás
Folyamat, amely x -et
 $I(x)/b$ valószínűségi sűrűséggel
mintavételezi

Metropolis sampling, on the other hand, when carries out importance sampling, assumes much less about the importance function.

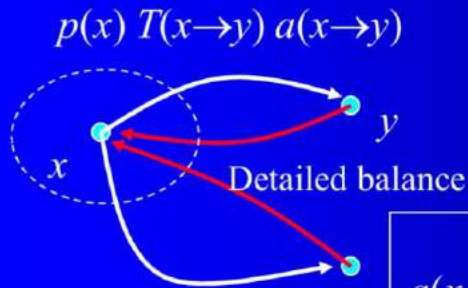
Instead of transforming uniformly distributed numbers, Metropolis sampling randomly mutates the previous sample to obtain a new tentative sample, and decides randomly on the acceptance or rejection of the perturbed sample.

If the acceptance of the tentative sample is proportional to the importance degradation, then the probability of obtaining a sample in the stationary limiting case will be proportional to the importance function, with almost all mutation strategies.

Note that Metropolis sampling does not even need the analytic form of the importance function, it is enough if we can point sample it, thus the importance sampling strategy can be much more effective, resulting in accurate images with just a few samples.

Elfogadási valószínűség

- “Bármilyen” mutáció $T(x \rightarrow y)$
- Elfogadási valószínűség $a(x \rightarrow y)$ úgy, hogy a határeloszlás a fontossággal arányos legyen: $p(x) \propto I(x)$



$$\frac{a(x \rightarrow y)}{a(y \rightarrow x)} = \frac{I(y) T(y \rightarrow x)}{I(x) T(x \rightarrow y)}$$

Maximális konvergencia:

$$a(x \rightarrow y) = \min \left\{ \frac{I(y) \cdot T(y \rightarrow x)}{I(x) \cdot T(x \rightarrow y)}, 1 \right\}$$

Metropolis algoritmus



FOR $i=1$ TO M DO

Using z_i choose another random, tentative point z_t

$a(z_i \Rightarrow z_t) = (I(z_t) T(z_t \Rightarrow z_i)) / (I(z_i) T(z_i \Rightarrow z_t))$

// accept with probability $a(z_i \Rightarrow z_t)$

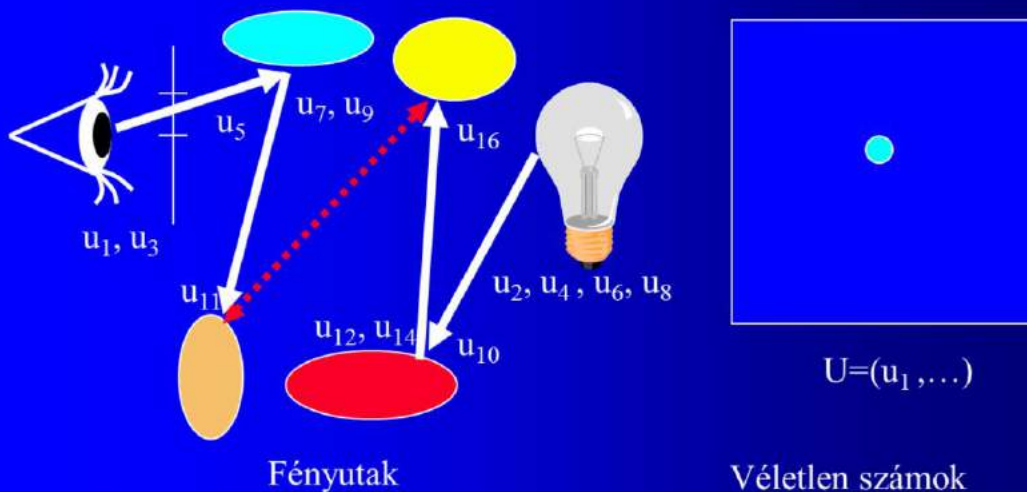
Generate random number r in $[0,1]$

IF $r < a(z_i \Rightarrow z_t)$ THEN $z_{i+1} = z_t$ ELSE $z_{i+1} = z_i$

Use z_{i+1} in the integral quadrature

ENDFOR

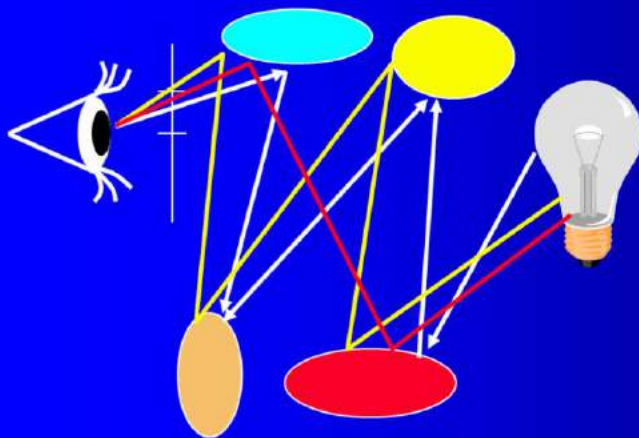
Mutációk az elsődleges mintavételi térben



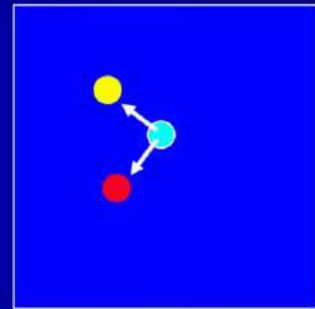
A sequence of pseudo random numbers unambiguously defines the light path for a given random walk algorithm. Let us suppose, for example, that the simplest version of bi-directional path tracing is used. Here, with two random numbers the pixel is sampled, then on the visible surface another random value is needed whether or not the walk should be terminated according to Russian roulette. If not, two new random values determine the direction of the reflection direction and the next one is responsible for the termination.

If the eyepath is terminated, four new random values are taken to find a light source point and direction that initiate a shooting path. Having terminated the shooting path, the eye and shooting subpaths are connected, and thus a complete path is established, which is defined by sixteen pseudo random numbers or a point in a sixteen dimensional unit cube.

Mutációk az elsődleges mintavételi térben



Path space

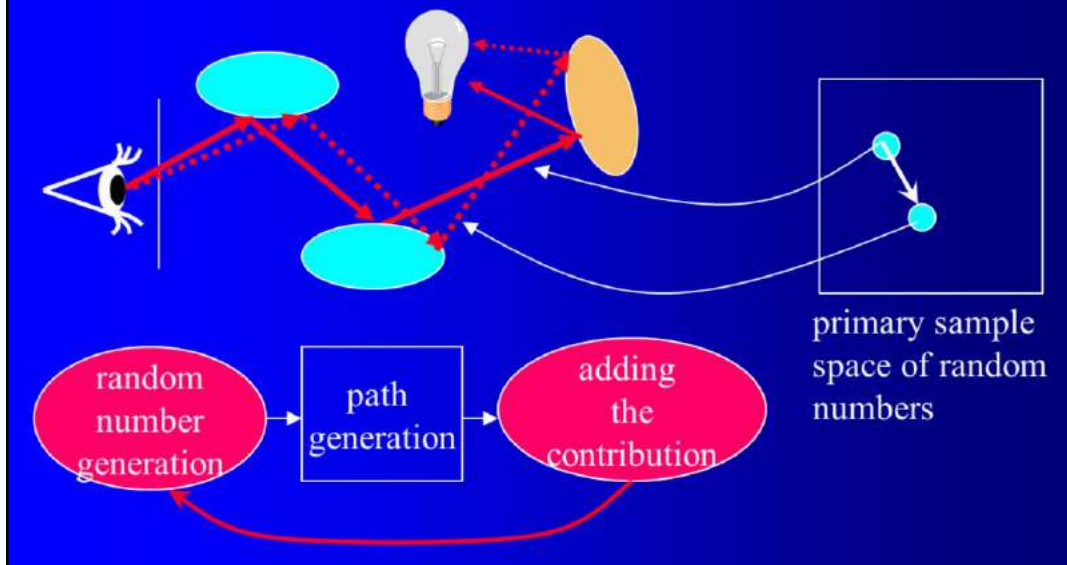


$$U=(u_1, \dots)$$

Primary sample space

Mutating in the primary sample space means that we change the point in the unit cube, and regenerate the path with the given random walk approach. Note that this can handle all types of path mutations. For example, if the mutation is smaller than allowed by the albedo limits used in Russian roulette, then the structure of the path is not altered, only the directions and the lightsource point are modified. However, when the mutation exceeds the albedo limits, steps are deleted or new steps might be introduced.

Mutációk az elsődleges mintavételi térben



In order to attack these problems, we proposed the perturbations to be realized in the so called primary sample space from where normal random walks obtain the uniformly distributed random numbers. Since normal importance sampling transforms these points in a way that a given mutation will correspond to a small path change for high contribution paths and a larger change for low contribution paths, this strategy adapts to the general properties of the scene and will be efficient not only for difficult but also for moderately difficult lighting conditions.

Moreover, this is very simple to implement if we already have an arbitrary random walk implementations, such as path tracing or bi-directional path tracing. The main module of a random walk algorithm is responsible for path generation. It calls the random number generator to get uniformly distributed random numbers and adds the contribution of the paths to the affected pixel.

In order to implement our Metropolis Sampler, the random number generation should be slightly changed. The new generator will perturb the previous numbers instead of obtaining a completely new one. On the other hand, the contribution should be computed differently, since we have to remember the previous sample, decide on the acceptance randomly, and restore the original sample if the new sample is rejected. This restoration also affects the random number generator.

