

Introduction to Computer Graphics

Claude Fuhrer

Bern University of Applied Sciences
School of Engineering and Information Technology

Spring 2016



Outline

Introduction



Part I

Introduction



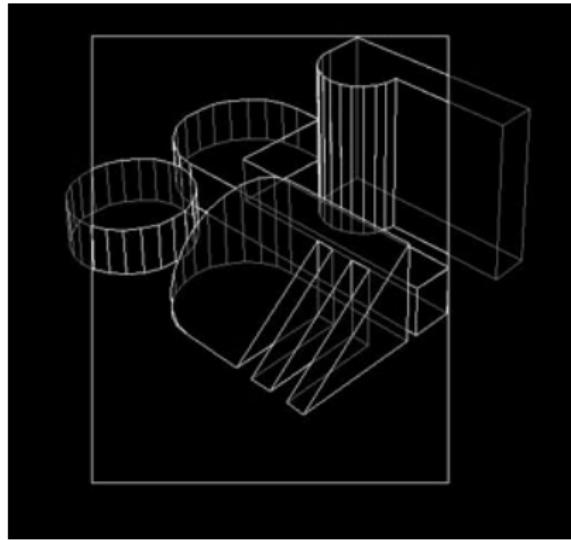
What is computer graphics ?

- ▶ The term computer graphics includes almost everything on computers that is not text or sound.
- ▶ Here in our lab, we think of computer graphics as drawing pictures on computers, also called rendering.
- ▶ We will also see some basics of animation



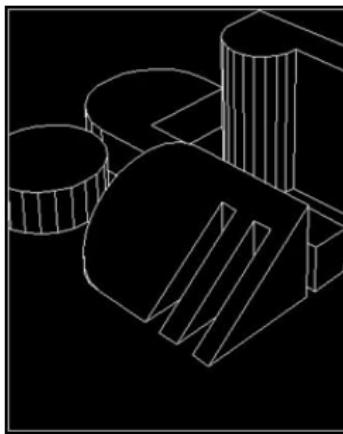
What is computer graphics ? (cont'd)

A model of the object is created to store the locations, or coordinates, of corner points.



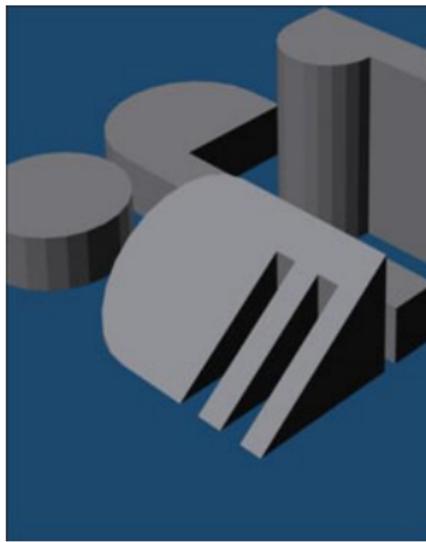
What is computer graphics ? (cont'd)

By a process called hidden line removal, only the portions of the edges visible to the viewer are drawn. The effect simplifies understanding complex scenes by eliminating overlapping lines, and makes objects look solid.



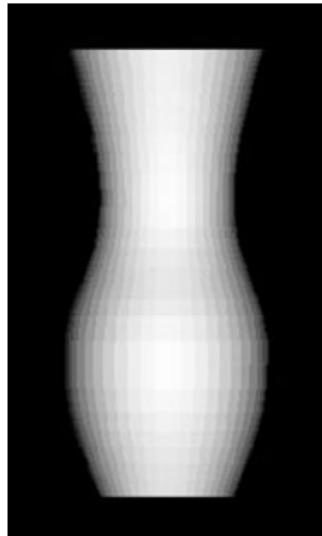
What is computer graphics ? (cont'd)

Using shading on the object surfaces visible to the viewer further improves our ability to interpret their shapes and positions.



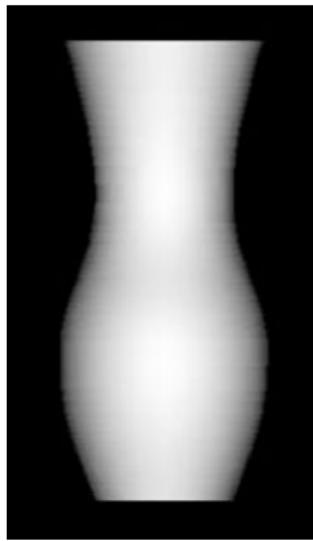
Object Rendering and Shading

This vase has been modeled as a symmetrical pattern of vertically-oriented surfaces - tiny flat patches which approximate the round shape of the vase.



Object Rendering and Shading (cont'd)

By introducing a technique called Gouraud shading, we can smooth out the appearance of the vase and hide the individual surfaces from view.



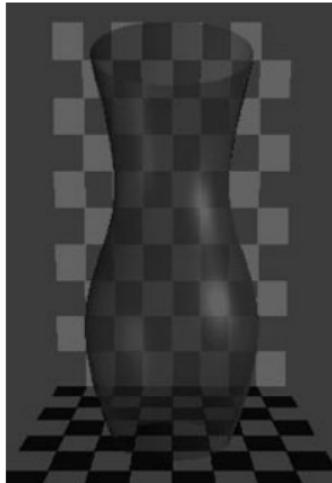
Object Rendering and Shading (cont'd)

Phong shading improves the apparent realism of the rendering still further by introducing highlights. The way light reflects from real surfaces depends on how shiny the surface is and on the angle you are looking from. Most surfaces are not shiny, but have a more dull or "matte" or "diffuse" appearance.



Object Rendering and Shading (cont'd)

In this last image the shading technique has been extended to let some light pass through the vase - for transparency. Note that the reflection highlights are still there, even if they are less visible.



Colors in computer graphics

Computers typically display color in three components - red, green, and blue. When combined, these three colors make the full-color image seen in the upper left of this image.



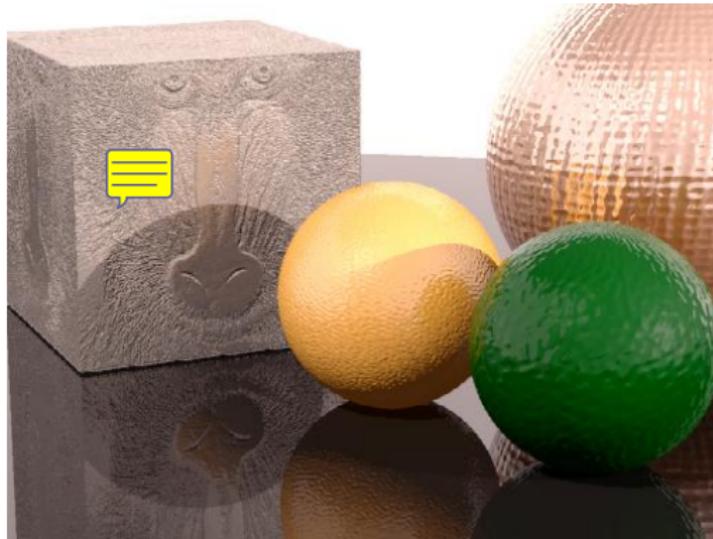
Colors in computer graphics (cont'd)

By controlling color display, we can simulate on the computer different kinds of color blindness.

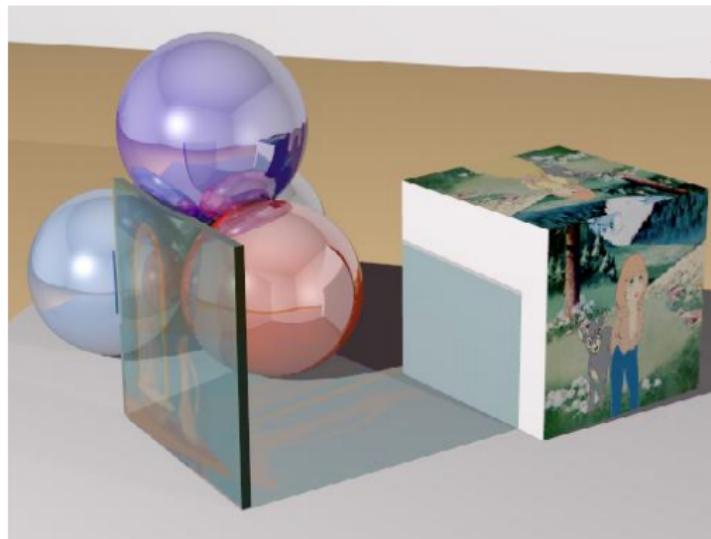


Reflection and Transparency

These two images illustrate the extent to which light reflection can be modeled by tracing the paths of light rays entering your eye.



Reflection and Transparency (cont'd)



Part II

Linear algebra



Outline

Vector spaces

Operations on vectors

Matrices



Vector space

Let \mathcal{F} be a field (such as the \mathbb{R} or \mathbb{C}), whose elements will be called *scalars*. Then a vector space over the field \mathcal{F} is a set V together with two operations:

1. vector addition $V \times V \rightarrow V$, denoted $\vec{v} + \vec{w}$ where $\vec{v}, \vec{w} \in V$;
2. scalar multiplication $\mathcal{F} \times V \rightarrow V$ denoted $\alpha\vec{v}$ where $\alpha \in \mathcal{F}$ and $\vec{v} \in V$



Vector space (cont'd) I

Moreover, to be a vector space, V must satisfy the following axioms:

- ▶ Vector addition is associative, i.e. for all $\vec{u}, \vec{v}, \vec{w} \in V$, we have $\vec{u} + (\vec{v} + \vec{w}) = (\vec{u} + \vec{v}) + \vec{w}$.
- ▶ Vector addition is commutative, i.e. for all $\vec{v}, \vec{w} \in V$ we have $\vec{v} + \vec{w} = \vec{w} + \vec{v}$.
- ▶ Vector addition has an identity element, i.e. there exists an element $\vec{0} \in V$, called the zero vector, such that $\vec{v} + \vec{0} = \vec{v}$ for all $\vec{v} \in V$.
- ▶ Vector addition has inverse element, i.e. for all $\vec{v} \in V$, there exists an element $\vec{w} \in V$, called the additive inverse of \vec{v} , such that $\vec{v} + \vec{w} = \vec{0}$.



Vector space (cont'd) II

- ▶ Distributivity holds for scalar multiplication over vector addition, i.e. for all $\alpha \in \mathcal{F}$ and $\vec{v}, \vec{w} \in V$, we have $\alpha(\vec{v} + \vec{w}) = \alpha\vec{v} + \alpha\vec{w}$.



Vector space (cont'd) I

- ▶ Distributivity holds for scalar multiplication over field addition, i.e. for all $\alpha, \beta \in \mathcal{F}$ and $\vec{v} \in V$, we have $(\alpha + \beta)\vec{v} = \alpha\vec{v} + \beta\vec{v}$
- ▶ Scalar multiplication is associative, i.e. for all $\alpha, \beta \in \mathcal{F}$ and $\vec{v} \in V$, we have $\alpha(\beta\vec{v}) = (\alpha\beta)\vec{v}$.
- ▶ Scalar multiplication has an identity element, i.e. for all $\vec{v} \in V$, we have $1\vec{v} = \vec{v}$, where 1 denotes the multiplicative identity in \mathcal{F} .



Vector space (cont'd)

Note that some sources may choose to also include two axioms of closure:

- ▶ Vector addition is closed, i.e. if $\vec{u}, \vec{v} \in V$, then $\vec{u} + \vec{v} \in V$.
- ▶ Scalar multiplication is closed, i.e. if $\alpha \in \mathcal{F}$ and $\vec{v} \in V$, then $\alpha\vec{v} \in V$.



Elementary properties of vector spaces

There are a number of properties that follow easily from the vector space axioms.

- ▶ The zero vector $\vec{0} \in V$ is unique, i.e. if $\vec{0}_1$ and $\vec{0}_2$ are zero vectors in V , such that $\vec{0}_1 + \vec{v} = \vec{v}$ and $\vec{0}_2 + \vec{v} = \vec{v}$ for all $\vec{v} \in V$, then $\vec{0}_1 = \vec{0}_2 = \vec{0}$.
- ▶ Scalar multiplication with the zero vector yields the zero vector, i.e. for all $\alpha \in \mathcal{F}$, we have $\alpha\vec{0} = \vec{0}$.
- ▶ Scalar multiplication by zero yields the zero vector, i.e. for all $\vec{v} \in V$, we have $0\vec{v} = \vec{0}$, where 0 denotes the additive identity in \mathcal{F} .
- ▶ No other scalar multiplication yields the zero vector, i.e. we have $\alpha\vec{v} = \vec{0}$ if and only if $\alpha = 0$ or $\vec{v} = \vec{0}$.



Elementary properties of vector spaces (cont'd)

- ▶ The additive inverse $-\vec{v}$ of a vector \vec{v} is unique, i.e. if \vec{w}_1 and \vec{w}_2 are additive inverses of $\vec{v} \in V$, such that $\vec{v} + \vec{w}_1 = \vec{0}$ and $\vec{v} + \vec{w}_2 = \vec{0}$, then $\vec{w}_1 = \vec{w}_2$. We call the inverse $-\vec{v}$ and define $\vec{w} - \vec{v} \equiv \vec{w} + (-\vec{v})$.
- ▶ Scalar multiplication by negative unity yields the additive inverse of the vector, i.e. for all $\vec{v} \in V$, we have $(-1)\vec{v} = -\vec{v}$, where 1 denotes the multiplicative identity in \mathcal{F} .
- ▶ Negation commutes freely, i.e. for all $\alpha \in \mathcal{F}$ and $\vec{v} \in V$, we have $(-\alpha)\vec{v} = \alpha(-\vec{v}) = -(\alpha\vec{v})$



Outline

Vector spaces

Operations on vectors

Matrices



Operations on vectors

For the elements of a vector space, one can define the following operations:

- ▶ The addition of two vectors \vec{u} and \vec{v} is defined as:
$$\vec{w} = \vec{u} + \vec{v} \Rightarrow w_i = u_i + v_i$$
- ▶ The subtraction of two vectors \vec{u} and \vec{v} is defined as:
$$\vec{w} = \vec{u} - \vec{v} \Rightarrow w_i = u_i - v_i$$
- ▶ The multiplication of a vector \vec{v} by a scalar α is defined as:
$$\vec{w} = \alpha \vec{v} \Rightarrow w_i = \alpha v_i$$



Operations on vectors (cont'd)

On a vector space V , one can define an inner product (sometimes called scalar product) $\langle \cdot, \cdot \rangle$ which have the following properties :

- ▶ $\langle u + w, v \rangle = \langle u, v \rangle + \langle w, v \rangle$
- ▶ $\langle \alpha v, w \rangle = \alpha \langle v, w \rangle$
- ▶ $\langle v, w \rangle = \langle w, v \rangle$
- ▶ $\langle v, v \rangle \geq 0$ and $\langle v, v \rangle = 0 \Leftrightarrow v = 0$



Operations on vectors (cont'd)

In an euclidian space, the inner product of two vectors \vec{v} and \vec{w} may be seen as:

$$\vec{v} \cdot \vec{w} = \|\vec{v}\| \|\vec{w}\| \cos \theta$$

where θ is the angle between the two vectors

Moreover, from this definition one can easily find that:

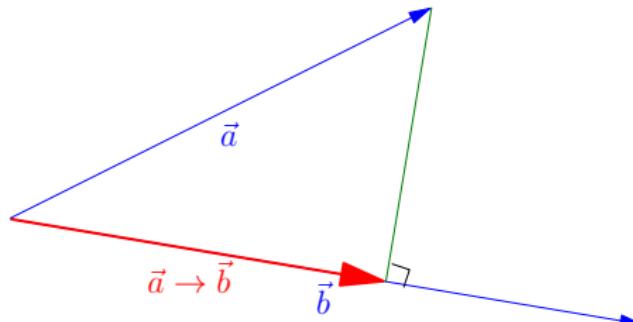
$$\|\vec{v}\|^2 = \vec{v} \cdot \vec{v}$$



Operations on vectors (cont'd)

The dot product can be used to find the projection of one vector onto another. This is the length $\vec{a} \rightarrow \vec{b}$ of a vector \vec{a} that is projected onto a vector \vec{b}

$$\vec{a} \rightarrow \vec{b} = \|\vec{a}\| \cos \varphi = \frac{\vec{a} \cdot \vec{b}}{\|\vec{b}\|}$$



Operations on vectors (cont'd)

On \mathbb{R}^3 one can define another product, called *vector product* or *cross product*. This product is defined as:

$$\vec{v} \times \vec{w} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix}$$
$$= \vec{i}(v_y w_z - v_z w_y) - \vec{j}(v_x w_z - v_z w_x) + \vec{k}(v_x w_y - v_y w_x)$$

The result vector is perpendicular either to \vec{v} and to \vec{w} . The norm of the product is equivalent to the surface of the parallelogram defined by the two vectors \vec{v} and \vec{w} , i.e. $\|\vec{v} \times \vec{w}\| = \|\vec{v}\| \|\vec{w}\| \sin \beta$ where β is the angle between \vec{v} and \vec{w} .



Outline

Vector spaces

Operations on vectors

Matrices



Matrices

A matrix may be viewed as a rectangular table of numbers (i.e. scalar of the field \mathcal{F}). A matrix is often characterized by its dimension, i.e. the number of lines and columns it have.

We often write $\mathbf{A} = (a_{i,j})_{m \times n}$

A matrix defines a linear application in $\mathbb{R}^{m \times n}$



Properties and operations on matrices

One can define the addition of two matrices, by the addition of each term of the matrices (**the size must be the same**). If $\mathbf{A} = (a_{i,j})_{m \times n}$ and $\mathbf{B} = (B_{i,j})_{m \times n}$ then $\mathbf{C} = \mathbf{A} + \mathbf{B}$ is defined as:

$$\mathbf{C} = (c_{i,j})_{m \times n} = a_{i,j} + b_{i,j}$$

The substraction of two matrices is symmetrically defined.



Properties of the matrices

1. $(A + B) + C = A + (B + C);$
2. $A + B = B + A;$
3. $0 + A = A;$
4. $A + (-A) = 0;$
5. $(s + t)A = sA + tA, (s - t)A = sA - tA;$
6. $t(A + B) = tA + tB, t(A - B) = tA - tB;$
7. $s(tA) = (st)A;$
8. $1A = A, 0A = 0, (-1)A = -A;$
9. $(tA = 0) \Rightarrow (t = 0 \text{ or } A = 0).$



Matrix product

Let \mathbf{A} be a matrix of size $m \times n$ and \mathbf{B} a matrix of size $n \times p$ (that is the number of columns of \mathbf{A} equals the number of rows of \mathbf{B}).

The matrix $\mathbf{C} = \mathbf{AB}$ is a matrix of size $m \times p$ whose (i, k) -th element is defined as:

$$c_{ik} = \sum_{j=1}^{j=n} a_{ij} b_{jk} = a_{i1} b_{1k} + \cdots + a_{in} b_{nk}$$

Matrix product is not commutative



Properties of the matrix product

1. $(AB)C = A(BC)$ if $A; B; C$ are $m \times n; n \times p; p \times q$, respectively;
2. $t(AB) = (tA)B = A(tB);$
3. $A(-B) = (-A)B = -(AB);$
4. $(A + B)C = AC + BC$ if A and B are $m \times n$ and C is $n \times p$;
5. $D(A + B) = DA + DB$ if A and B are $m \times n$ and D is $p \times m$.



Special matrices

1. Identity matrix \mathbf{I} (or $\mathbf{1}$) is a **square matrix** (that is a matrix of size $n \times n$) defined as:

$$\mathbf{I} = I_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

2. Diagonal matrix \mathbf{D} is a matrix of size $n \times n$ defined as:

$$\mathbf{D} = D_{ij} = \begin{cases} \neq 0 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$



Square matrices

A square matrix \mathbf{A} is called *non-singular* or *invertible* if there exist a matrix \mathbf{B} such that

$$\mathbf{AB} = \mathbf{I} = \mathbf{BA}$$

Any matrix \mathbf{B} with the property above is called *inverse* of \mathbf{A} and is often written \mathbf{A}^{-1} .

Remark : if the inverse matrix \mathbf{A}^{-1} of \mathbf{A} exist, it is unique.

The inverse of the product of two matrices is the product of the inverses in reverse order, that is:

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$



The transpose of a matrix

Let \mathbf{A} be an $m \times n$ matrix. Then ${}^t\mathbf{A}$ the *transpose* of \mathbf{A} is obtained by interchanging the rows and columns of \mathbf{A} that is if $\mathbf{A} = (a_{ij})$ then $({}^t\mathbf{A})_{ji} = a_{ij}$

The transpose operation has the following properties:

1. ${}^t({}^t\mathbf{A}) = \mathbf{A};$
2. ${}^t(\mathbf{A} \pm \mathbf{B}) = {}^t\mathbf{A} \pm {}^t\mathbf{B}$ if \mathbf{A} and \mathbf{B} are $m \times n$;
3. ${}^t(\alpha\mathbf{A}) = \alpha{}^t\mathbf{A}$ is α is scalar;
4. ${}^t(\mathbf{AB}) = {}^t\mathbf{B}{}^t\mathbf{A}$ if \mathbf{A} is $m \times n$ and \mathbf{B} is $n \times p$;
5. if \mathbf{A} is non-singular, then ${}^t\mathbf{A}$ is also non-singular and we have
$$({}^t\mathbf{A})^{-1} = {}^t(\mathbf{A}^{-1})$$



Symmetric matrices and skew-symmetric matrices

A matrix \mathbf{A} is called *symmetric* if $\mathbf{A} = {}^t\mathbf{A}$ that is $a_{ij} = a_{ji}$

A matrix \mathbf{A} is called *skew-symmetric* if $\mathbf{A} = -{}^t\mathbf{A}$ that is $a_{ij} = -a_{ji}$

Remark 1 : If \mathbf{A} is skew-symmetric imply that $a_{ii} = 0$ for all i .

Remark 2 : If \mathbf{A} is a square matrix, the product ${}^t\mathbf{A}\mathbf{A}$ is always symmetric.



Determinant of matrices

Every square matrix has a number associated with it called *determinant* and denoted by $|\mathbf{A}|$ or $\det \mathbf{A}$. This number describes the effect that a linear transformation has on areas and volumes of objects for example.

For a two-by-two matrix \mathbf{A} , the determinant is simply the difference of the products of the number of the diagonal, that is:

$$|\mathbf{A}| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$



Determinant of matrices (cont'd)

If \mathbf{A} is a three-by-three matrix, the determinant has the form:

$$|\mathbf{A}| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

Remark : if the matrix \mathbf{A} is singular, its determinant will be 0.



Matrix cofactor

Each element a_{ij} of a square matrix \mathbf{A} has a corresponding *cofactor* A_{ij} that is defined as $(-1)^{i+j}$ times the determinant of the matrix formed by deleting the i -th row and the j -th column from \mathbf{A} . The

inverse $\mathbf{B} = \mathbf{A}^{-1}$ of a non singular matrix \mathbf{A} is defined by:

$$b_{ij} = \frac{A_{ij}}{|\mathbf{A}|}$$

however, this a computationnally very bad method to compute the determinant of a matrix.



Part III

Geometrical transformations in space and plane



Outline

Transformations in plane

Linear transformations

Homogeneous transformations

3D space



Coordinates and homogeneous transformations

The machinery of linear algebra can be used to express many operations required to arrange objects in a 3D scene, view them with camera and get them onto screen



Linear transformations

A 2×2 matrix can be used to transform a 2D vector.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{bmatrix}$$

This kind of matrix describe a *linear transformation*



Scaling

Suppose one need to scale an object by a factor k , that is, each of its coordinates should be multiplied by a factor k . One has:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} kx \\ ky \end{bmatrix}$$

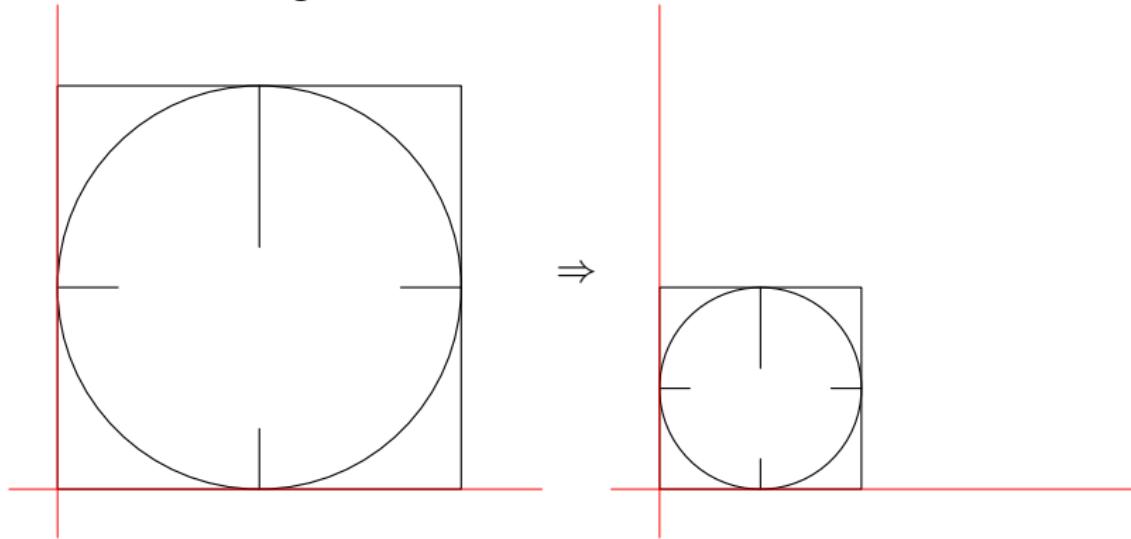
More generally, the scaling factors may be different in the two direction and one should write:

$$\text{scaling}(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$



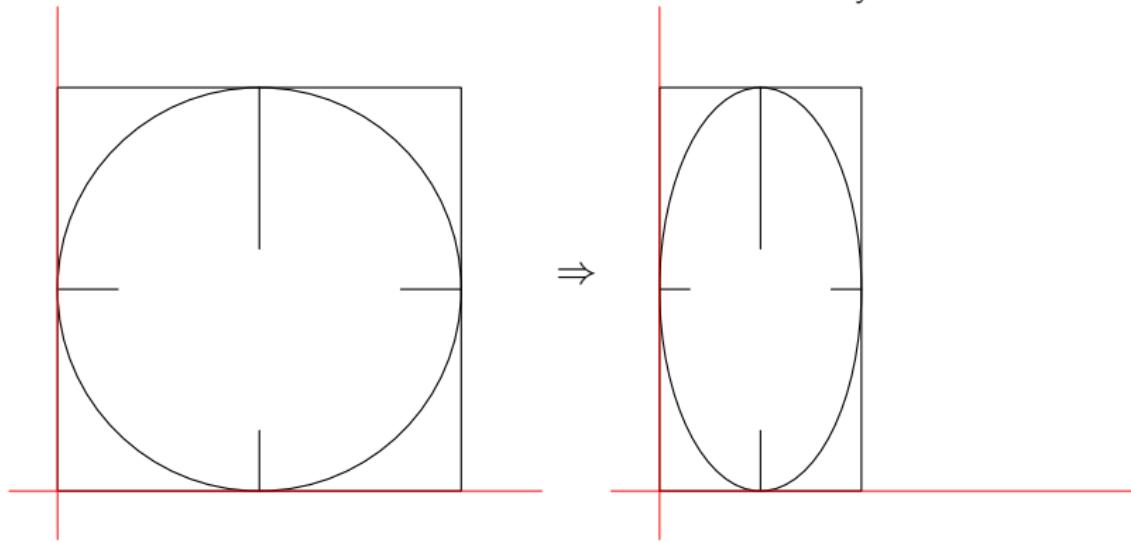
Scaling

For example, a uniform scaling (both factors have the same value) with factors 0.5 gives:



Scaling

For example, a scaling with values $s_x = 0.5$ and $s_y = 1.0$ gives:



Shearing

A shear is a transformation that "pushes things sideways". The horizontal shear matrix is:

$$\text{shear_x}(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

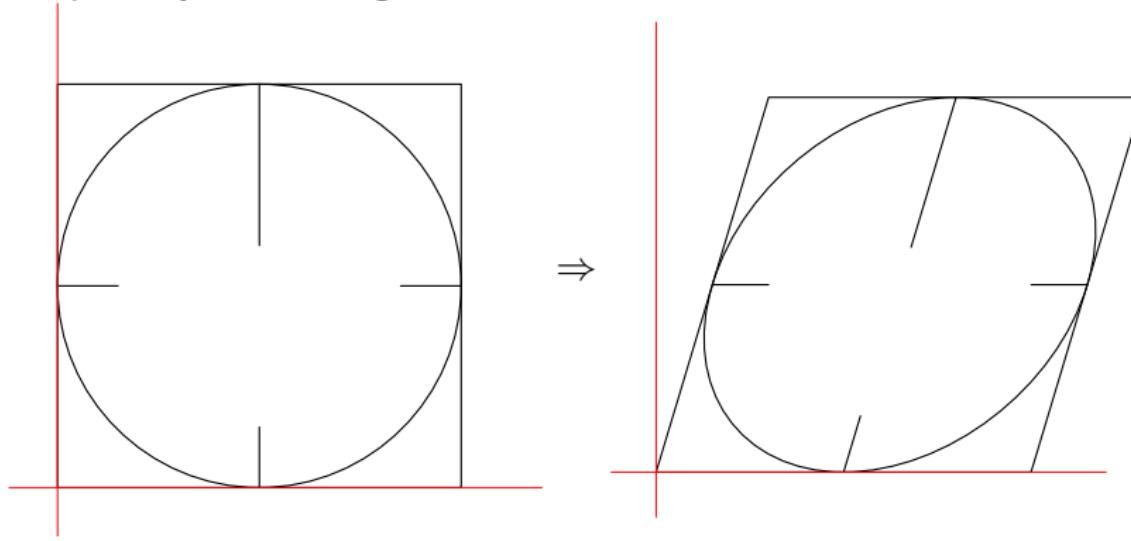
Applying this matrix to a vector gives:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + sy \\ y \end{bmatrix}$$



Shearing

Graphically, a shearing looks like:

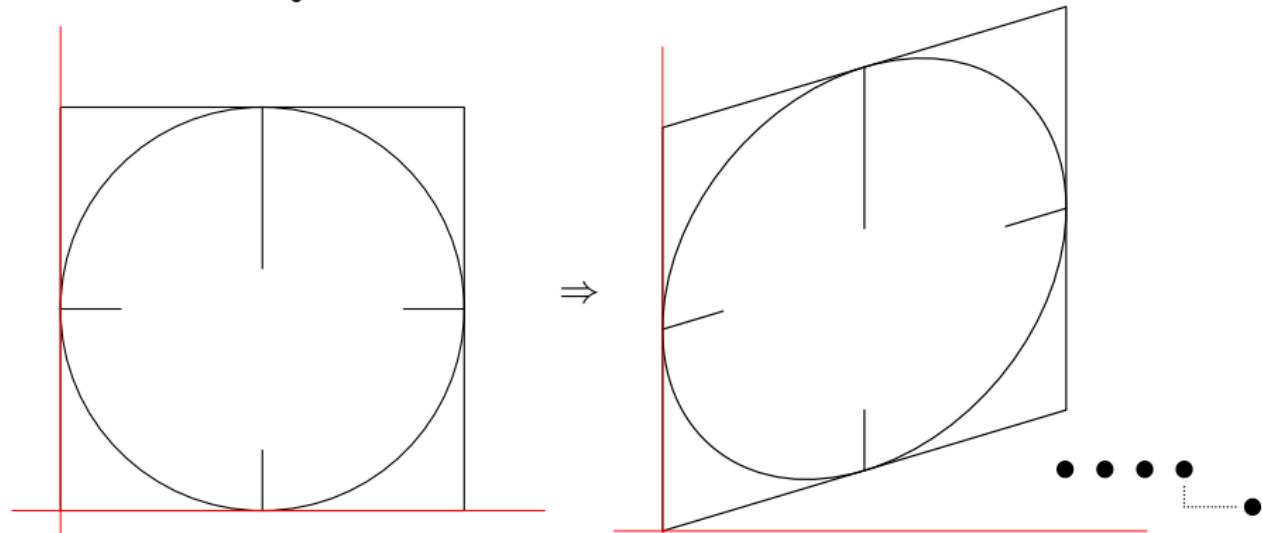


Shearing

A shearing may also be in the other direction, using the shearing matrix:

$$\text{shearing}_y(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

The modified object looks like:



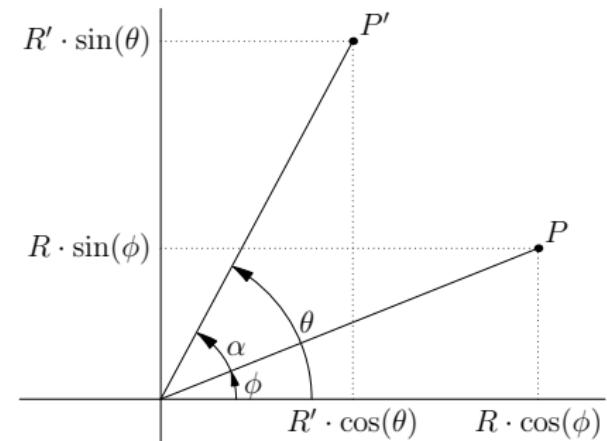
Rotation about the origin

Let P be a point given by

$P = (x; y)$. One need to rotate this point around the origin with an angle of α to obtain a point $P' = (x'; y')$.

Let R (resp. R') be the distance between the origin of the coordinates system and the point P (resp P').

One can see that $\theta = \phi + \alpha$.



Rotation about the origin

Some standard computing gives that:

$$x' = R \cos(\phi + \alpha)$$

$$y' = R \sin(\phi + \alpha)$$

Using standard trigonometric calculation, this is equivalent to:

$$x' = \underbrace{R \cos(\phi)}_x \cos(\alpha) - \underbrace{R \sin(\phi)}_y \sin(\alpha)$$

$$y' = \underbrace{R \sin(\phi)}_y \cos(\alpha) + \underbrace{R \cos(\phi)}_x \sin(\alpha)$$

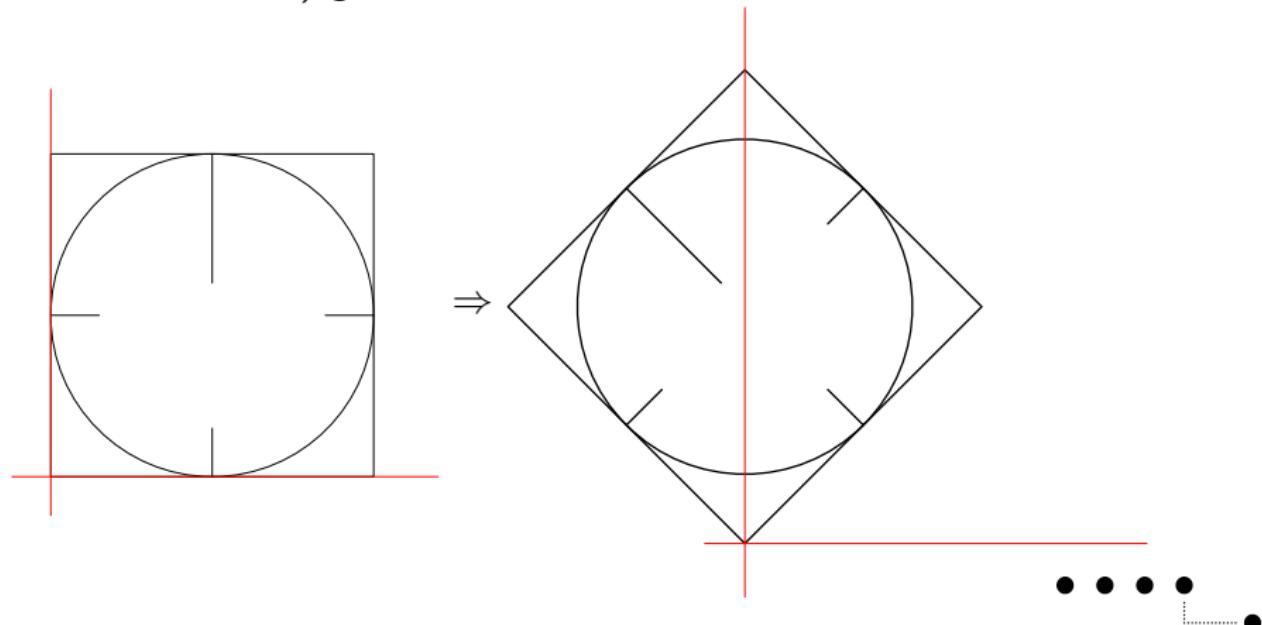
This can be written as a matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$



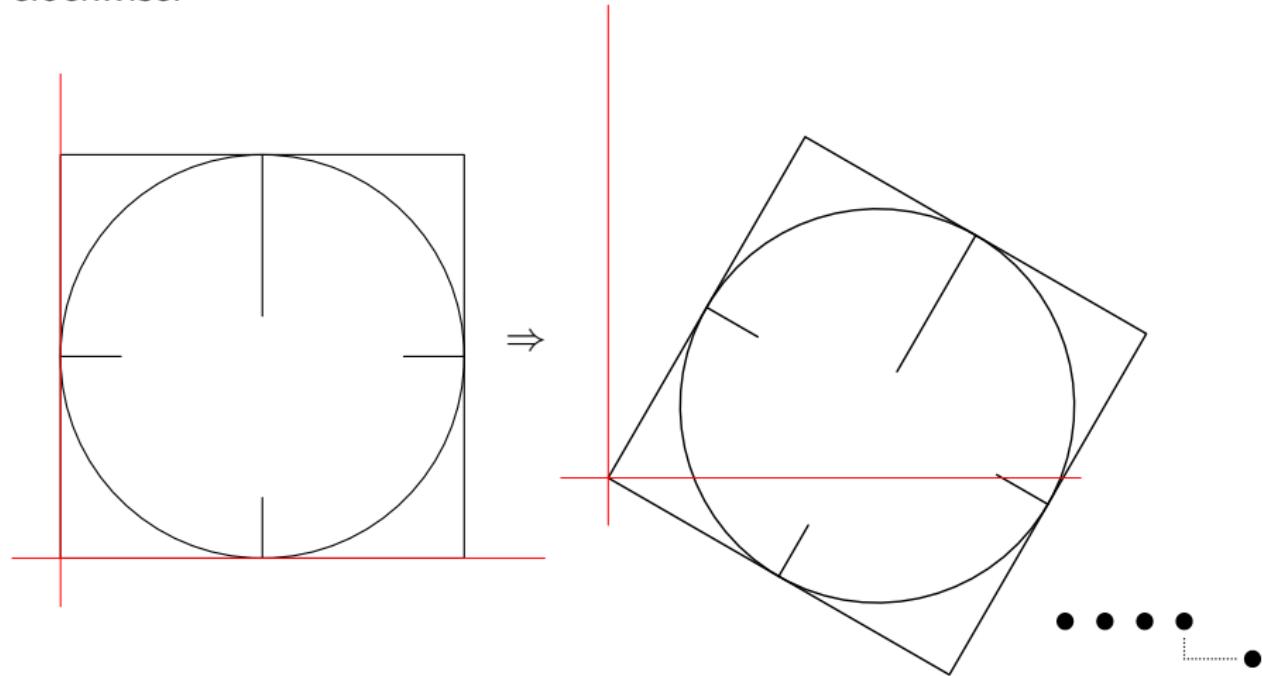
Rotation around the origin

Rotating an object by an angle of $\pi/4$ (positive angle means counterclockwise) gives:



Rotation around the origin

Giving a negative angle (e.g. $-pi/6$) allow to turn the object clockwise.



Reflection along an axis

A reflection along the x -axis or the y -axis may be defined by a negative scaling. For example

$$\text{reflect_y}() = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

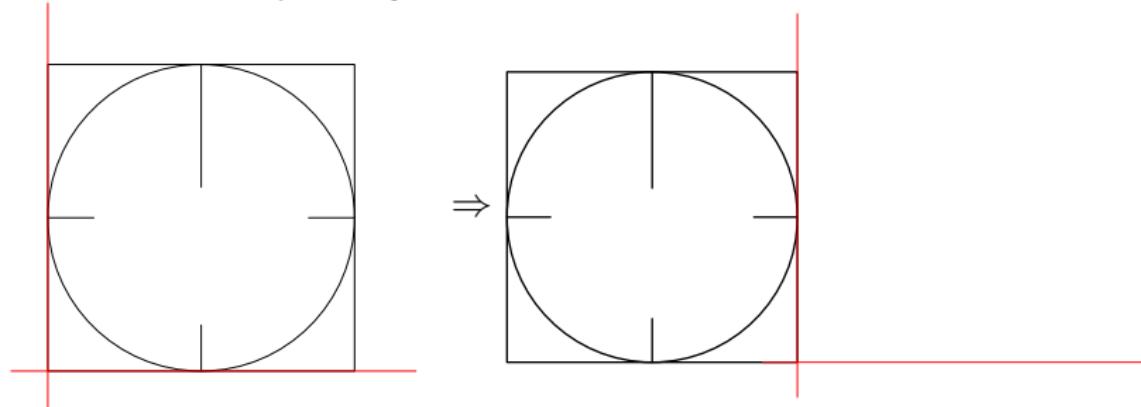
and

$$\text{reflect_x}() = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$



Reflection along an axis

For example, to do a reflection along the y -axis, all x coordinates should be multiplied by -1 .



Reflection along an arbitrary line

Should be studied later !



Composition of transformations

It is common for graphics programs to apply more than one transformation to one object.

For example, we want to first apply a scale S and then a rotation R . One can do this in two steps:

$$\mathbf{a}' = S(\mathbf{a})$$

$$\mathbf{a}'' = R(\mathbf{a}')$$

Functionnaly, these two steps can be combined as:

$$\mathbf{a}'' = R(S(\mathbf{a}))$$

Using the associativity of matrix multiplication, this transformation can be computed as:

$$\mathbf{a}'' = (RS)(\mathbf{a})$$



Composition of transformations

To combine two or more transformations,
one can simply multiply the corresponding
matrices.

The multiplication is done at the left of the
preceding matrix.

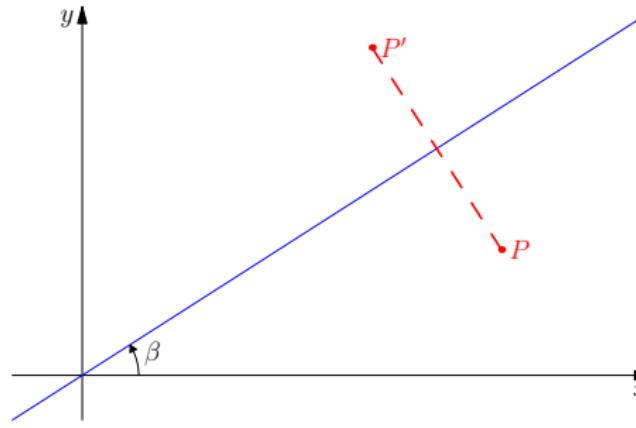




Example of composition of transformations

Suppose one want to compute a reflection along a line which is not an axis (but go through the origin).

First, one should apply a rotation to move the reflection line onto the x -axis. Then one can apply the reflection and finally rotate the whole system back in place.



Example of composition of transformations

The final transformation is then:

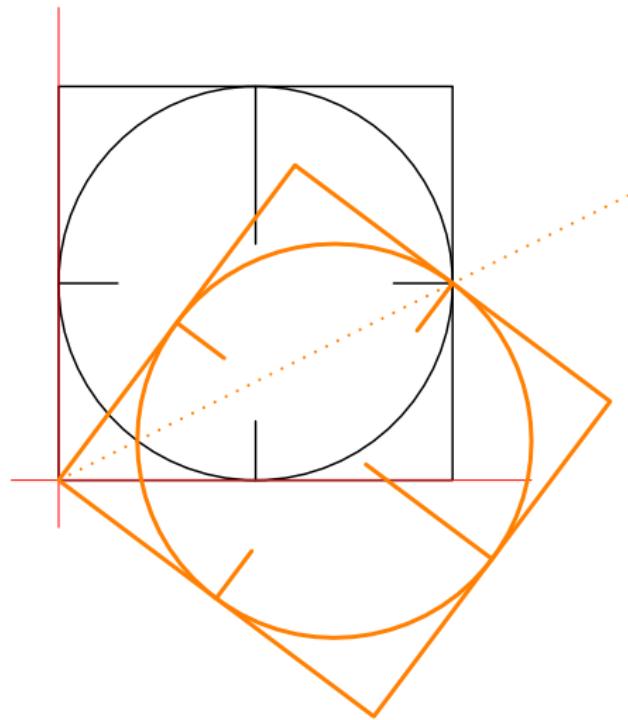
$$\begin{aligned} T &= \begin{bmatrix} \cos \beta & -\sin \beta \\ \sin \beta & \cos \beta \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos -\beta & -\sin -\beta \\ \sin -\beta & \cos -\beta \end{bmatrix} \\ &= \begin{bmatrix} \cos^2 \beta - \sin^2 \beta & 2 \cos \beta \sin \beta \\ 2 \cos \beta \sin \beta & \sin^2 \beta - \cos^2 \beta \end{bmatrix} \end{aligned}$$

One can see the the inverse of a rotation is a rotation with angle $-\beta$.



Example of composition of transformations

Applying this transformation to our clock gives:



Example 2 of composition of transformations

It is very important to understand that applying many transformation, one after another is not a commutative operation. So is also the multiplication of matrices representing these transformations.

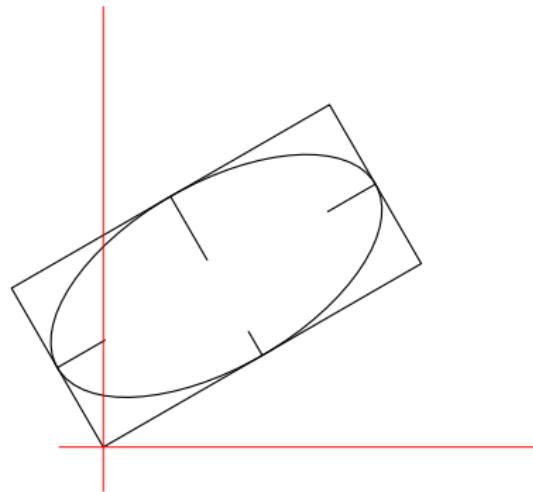
Consider the two cases:

1. One apply first a scaling $S(1; 0.5)$ and then a rotation $R(\gamma)$.
2. One apply first a rotation $R(\gamma)$ and then a scaling $S(1; 0.5)$.

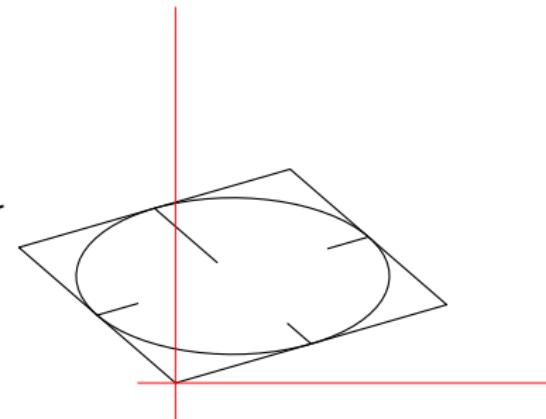


Example 2 of composition of transformations

The result of these two different transformations is:



or



Outline

Transformations in plane

Linear transformations

Homogeneous transformations

3D space



Modelling translations

- ▶ The mechanism described above does not allow to describe a translation using a 2×2 matrix.
- ▶ A translation of a point p by a vector \vec{v} is given by:

$$p'_x = p_x + v_x$$

$$p'_y = p_y + v_y$$

- ▶ One solution could be to consider (and compute) translations separately from other transformation
- ▶ An other possible solution could be to use a tool of projective geometry : *homogeneous coordinates*



Homogeneous coordinates

- ▶ To be able to process translation as any other transformation in plane, one should use the homogeneous coordinates.
- ▶ Every point in the plane should be transformed from "normal" to homogeneous
- ▶ An homogeneous representant of the point $P = (x; y)$ is the point $\tilde{P} = (wx; wy; w)$ where the coordinates w can take any value different than 0.
- ▶ To find the "standard" point P from its homogeneous representant, one need to compute:

$$P = (x; y) = \left(\frac{\tilde{x}}{\tilde{w}}; \frac{\tilde{y}}{\tilde{w}} \right)$$

for $\tilde{w} \neq 0$

Many homogeneous coordinates may represent the same "standard" point



Homogeneous coordinates

- ▶ To be able to process vectors in homogeneous coordinates, one should also modify their representation.
- ▶ 2D homogeneous vectors have three coordinates, where the third coordinate is set to 0.
- ▶ If one consider a vector given by two points at its extremities, one have:

$$\tilde{v} = \tilde{P}_2 - \tilde{P}_1 = \begin{pmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ 1 \end{pmatrix} - \begin{pmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ 1 \end{pmatrix} = \begin{pmatrix} \Delta x \\ \Delta y \\ 0 \end{pmatrix}$$



2D transformation in homogeneous coordinates

Transformation matrices, for 2D-transformation should also be modified:

- ▶ A rotation of angle θ is now

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

- ▶ A scaling is given by:

$$S_{x,y} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Translation in 2D space

A translation in plane may now be written using a matrix operator:

$$T(\vec{v}) = \begin{pmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{pmatrix}$$



Outline

Transformations in plane

Linear transformations

Homogeneous transformations

3D space



Transformation in 3D spaces

- ▶ Similarly to 2D space, one may use matrices to model transformation in 3D space.
- ▶ As for transformation in plane, one uses homogeneous coordinates to be able to represent translation with a matrix.
- ▶ 3D points in homogeneous coordinates are represented by 4 coordinates:

$$\tilde{P} = (wx, wy, wz, w)$$

where the component w is per default 1

- ▶ As for 2D point, a "standard" 3D point may have multiple representation in homogeneous coordinates.



3D translation in homogeneous coordinates

- ▶ A translation $T = (t_x, t_y, t_z)$ is modeled by the matrix:

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



3D scaling in homogeneous coordinates

- ▶ A scaling $S = (s_x, s_y, s_z)$ is modeled by the matrix:

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

- ▶ For a uniform scaling, the three factors s_x , s_y and s_z should be the same.



Rotations in \mathbb{R}^3

- ▶ A rotation of angle θ about the z-axis is modeled by the matrix :

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



Rotations in \mathbb{R}^3 (cont'd)

- ▶ A rotation of angle θ about the x -axis is modeled by the matrix :

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



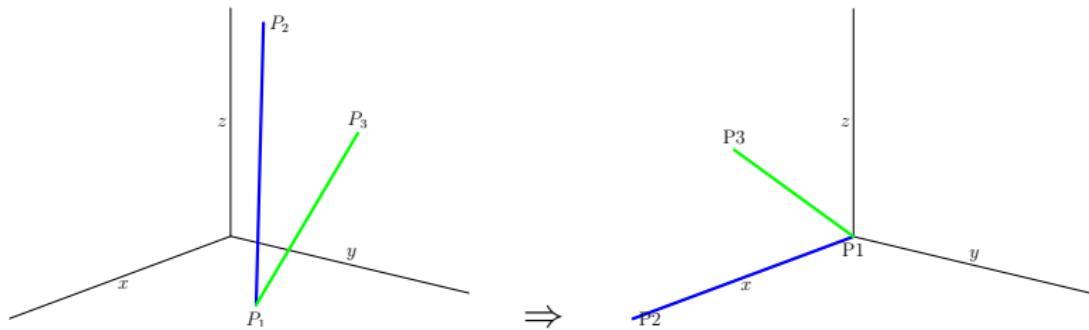
Rotations in \mathbb{R}^3 (cont'd)

- ▶ A rotation of angle θ about the y -axis is modeled by the matrix :

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



Composition of 3D transformations



Composition of 3D transformations (cont'd)

This complete transformation should be done in four steps:

1. Translate P_1 to the origin
2. Rotate about the z axis such that P_1P_2 lies in the (xy) -plane
3. Rotate about the y-axis such that P_1P_2 lies on the x-axis
4. Rotate about the x-axis such that P_1P_3 lies in the (xz) -plane



Quaternions

- ▶ To model rotation in the plane, one can use complex numbers
- ▶ To model rotation in a 3D-space, one should use an extension of complex numbers called *Quaternions*
- ▶ Quaternions are numbers written under the form

$$q = w + x i + y j + z k$$

- ▶ Quaternions can be seen as a generalization of the complex numbers.
- ▶ w is said to be the scalar part of q ;
- ▶ $x i + y j + z k$ is said to be the vector part of q



Quaternion (cont'd)

Let $q_1 = w_1 + x_1 i + y_1 j + z_1 k$ and $q_2 = w_2 + x_2 i + y_2 j + z_2 k$ be two quaternion.

- ▶ The sum $q_3 = q_1 + q_2$ is a quaternion defined by:

$$q_3 = (w_1 + w_2) + (x_1 + x_2) i + (y_1 + y_2) j + (z_1 + z_2) k$$

- ▶ The addition is associative $(q_1 + (q_2 + q_3)) = (q_1 + q_2) + q_3$) and commutative $(q_1 + q_2 = q_2 + q_1)$;
- ▶ One can define a zero-quaternion (with all elements equals to 0) such that $0 + q = q + 0 = q$;
- ▶ One can define the conjugate quaternion q_1^* as:

$$q_1^* = w_1 - x_1 i - y_1 j - z_1 k$$



Quaternion (cont'd)

- ▶ For the multiplication of quaternion, one have to consider the following relationship:
 1. $i^2 = j^2 = k^2 = -1$
 2. $ij = -ji = k$
 3. $jk = -kj = i$
 4. $ki = -ik = j$

The multiplication of quaternion is **not** commutative, i.e, if q_1 and q_2 are two quaternion $q_1 \cdot q_2 \neq q_2 \cdot q_1$



Quaternion (cont'd)

- ▶ Consider the multiplication of a quaternion and its conjugate

$$q \cdot q^* = w^2 + x^2 + y^2 + z^2$$

which gives the square of the norm of the quaternion $\|q\|^2$

- ▶ it follows that we can define the inverse q^{-1} of a quaternion q as:

$$q^{-1} = \frac{q^*}{\|q\|^2}$$



Axis-Angle to quaternion

Let $\hat{a} = (x_0, y_0, z_0)$ be a unit-length axis of rotation (i.e. $\|\hat{a}\| = 1$) and θ the angle of rotation. The quaternion q represent that rotation if q satisfy the following:

1. $q = w + x i + y j + z k$
2. $w = \cos(\theta/2)$
3. $x = x_0 \sin(\theta/2)$
4. $y = y_0 \sin(\theta/2)$
5. $z = z_0 \sin(\theta/2)$

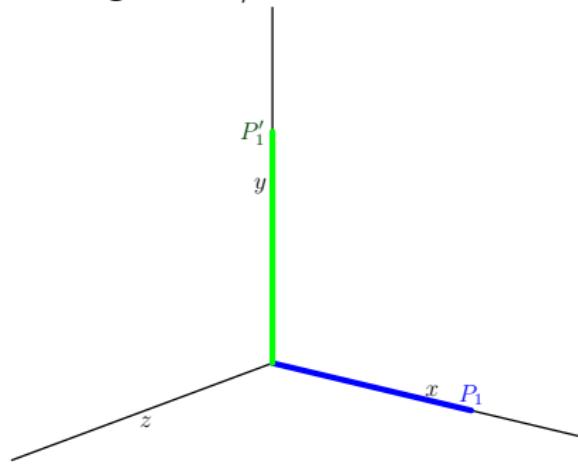
If a vector $\vec{v} = (v_0, v_1, v_2)$ is represented by the pure vectorial quaternion $v = v_0 i + v_1 j + v_2 k$, then to rotate the vector \vec{v} around \hat{a} with angle θ , one has to compute

$$\vec{v}' = q \cdot v \cdot q^*$$



Axis-Angle to quaternion : example

One want to rotate the vector $\vec{v} = (1, 0, 0)$ around the z-axis with an angle of $\pi/2$.



Quaternion to Axis-Angle

Let $q = w + xi + yj + zk$ be a unit quaternion (i.e. $\|q\| = 1$). To find the pair axis-angle described by this quaternion, one can do the following:

1. If $\|w\| = 1$ then the angle θ is 0 and v is any unit length vector (there is no rotation)
2. if $\|w\| < 1$ then $\theta = 2 \arccos(w)$ and the axis is computed by
 $\hat{u} = (x, y, z) / \sqrt{(1 - w^2)}$



Quaternion to Matrix

A rotation matrix \mathbf{R} that describes the same rotation as a given unit-length quaternion $q = w + xi + yj + zk$ is defined by:

$$\mathbf{R} = \begin{pmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{pmatrix}$$



Matrix to Quaternion

The mathematical justification of the transformation of a rotation matrix \mathbf{R} to quaternion q can be found in the book *Geometric Tools for Computer Graphics*.



Quaternion : performances issues

If one consider the memory usage:

Representation	# of floats	Comments
Rotation matrix	9	
Axis-angle	4	without precomputing of $\sin(\theta)$ and $(1 - \cos(\theta))$
Axis-angle	6	with precomputing of $\sin(\theta)$ and $(1 - \cos(\theta))$
Quaternion	4	





Using quaternion with Java3D

- ▶ The multiple rotation must be performed, the use of quaternion may seriously enhance the speed of the computations.
- ▶ The API Java3D provides two classes to uses quaternion in 3D applications. These classes are:
 - ▶ `javax.vecmath.Quat4f` for computation with float
 - ▶ `javax.vecmath.Quat4d` for computation with double

Part IV

Geometrical Modelling



Outline

Base equations

Intersection of two lines

Intersection of a line and a plane

Intersection of a line and a sphere

Intersection of a line and a plane



Equation of a line

A line may be defined by two manner

- ▶ Giving its direction vector \vec{v} and a point p onto the line;

$$\mathbf{d} = p + \vec{v} \cdot t$$

- ▶ Giving two points onto the line p_1 and p_2

$$\mathbf{d} = \overline{p_1; p_2}$$



Equation of a plane

A plane may be defined by four manner

- ▶ Giving its general equation

$$\Pi := \alpha x + \beta y + \gamma z + \delta = 0$$

- ▶ Giving a vector \vec{n} normal to the plane and a point onto the plane;
- ▶ Giving three points onto the plane;
- ▶ Giving two linearly independant vectors belonging to the plane, and a point onto the plane. This would define the parametric equation of the plane

$$\Pi := P_0 + s\vec{v} + t\vec{w}$$

The conversion between the four definition may be easily computed.



Equation of a sphere

- ▶ The analytical equation of a sphere centered in $C = (C_x; C_y; C_z)$ and radius r is given by:

$$S : (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

- ▶ A sphere is a 2-dimensionnal object in the 3D space.
Therefore, to define a point onto a sphere, it should be sufficient to give only two values. This would define the parametrical form of a sphere

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

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

$$z(u, v) = r \cos(v)$$

The parameters u and v are sometime called *longitude* resp *latitude*. The variable u takes values between 0 and 2π and the variable v takes values between $-\pi/2$ and $\pi/2$



Outline

Base equations

Intersection of two lines

Intersection of a line and a plane

Intersection of a line and a sphere

Intersection of a line and a plane



Intersection between 2 line segments

- ▶ AB and CD two line segment
- ▶ Parametric representation of supporting lines :

$$AB(t) = A + \mathbf{b} \cdot t \quad \text{where } \mathbf{b} = B - A \text{ and } 0 \leq t \leq 1$$

and

$$CD(t) = C + \mathbf{d} \cdot u \quad \text{where } \mathbf{d} = D - C \text{ and } 0 \leq u \leq 1$$

- ▶ Line segment AB and CD intersect if there exist two values for t and u such that

$$A + \mathbf{b} \cdot t = C + \mathbf{d} \cdot u$$

Hint: compute the intersection for two coordinates and check with the third coordinate if the lines really crosses.



Intersection between two line segments

- ▶ Let be s_1 a line segment given by two points P_1 and P_2 and s_2 a line segment given by Q_1 and Q_2 .
- ▶ As direction vector for the two supporting line, one can use $\vec{v}_1 = \vec{P_1 P_2}$ and $\vec{v}_2 = \vec{Q_1 Q_2}$
- ▶ If the supporting lines intersect and the values of the parameter u and t are both comprised within 0 and 1, the line segments crosses



Outline

Base equations

Intersection of two lines

Intersection of a line and a plane

Intersection of a line and a sphere

Intersection of a line and a plane



Intersection between a line and a plane

- ▶ Parametric equation of the line:

$$\mathbf{d} = p + t \cdot \vec{v}$$

or, in matrix form:

$$\begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} = \begin{pmatrix} p_x + t \cdot v_x \\ p_y + t \cdot v_y \\ p_z + t \cdot v_z \end{pmatrix}$$

- ▶ Equation of the plane:

$$\Pi : \alpha x + \beta y + \gamma z + \delta = 0$$

- ▶ To compute the intersection plane between the line and the plane one just have to insert the values of d_x (resp. d_y , d_z) for x (resp. y and z) in the equation of the plane.



Intersection between a line and a plane(cont'd)

- We obtain the equation:

$$\Pi : \alpha(p_x + v_x t) + \beta(p_y + v_y t) * \gamma(p_z + v_z t) + \delta = 0$$

Solving this equation for t gives:

$$t = \frac{\alpha p_x + \beta p_y + \gamma p_z + \delta}{\alpha v_x + \beta v_y + \gamma v_z}$$

- If the denominator of this fraction is 0, this imply that the line does not cross the plane.



Outline

Base equations

Intersection of two lines

Intersection of a line and a plane

Intersection of a line and a sphere

Intersection of a line and a plane



Intersection between a line and a sphere

- ▶ The equation defining a sphere is:

$$S : \|X - C\|^2 = r^2$$

which may be also written as:

$$S : (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$$

- ▶ Introducing the coordinates d_x (resp. d_y and d_z) for x (resp. for y and z) into this equation would define an equation of second degree for t .
- ▶ The value of t gives the coordinates of intersections points.



Intersection between a line and a sphere (cont'd)

- ▶ Before one can compute the roots of the intersection equation, one have to consider the discriminant Δ :
 1. if $\Delta > 0$, the intersection equation has 2 roots \Rightarrow the line fully intersect the sphere;
 2. if $\Delta == 0$, the intersection equation has only 1 root \Rightarrow the line is tangent the sphere;
 3. if $\Delta < 0$, the intersection equation hasn't real solution \Rightarrow the line and the sphere does not cross;



Outline

Base equations

Intersection of two lines

Intersection of a line and a plane

Intersection of a line and a sphere

Intersection of a line and a plane



Intersection between a sphere and a plane

- ▶ Let S a sphere :
$$S : (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r^2 = 0$$
 centered in
(C_x, C_y, C_z) and with radius r .
- ▶ Let a plane Π given by : $\Pi(t_u, t_v) = P + t_u \vec{u} + t_v \vec{v}$
The coordinates of the points of the plane are:

$$x = P_x + t_u \cdot u_x + t_v \cdot v_x$$

$$y = P_y + t_u \cdot u_y + t_v \cdot v_y$$

$$z = P_z + t_u \cdot u_z + t_v \cdot v_z$$



Intersection between a sphere and a plan (cont'd)

Inserting these coordinates into the equation of the spheres gives:

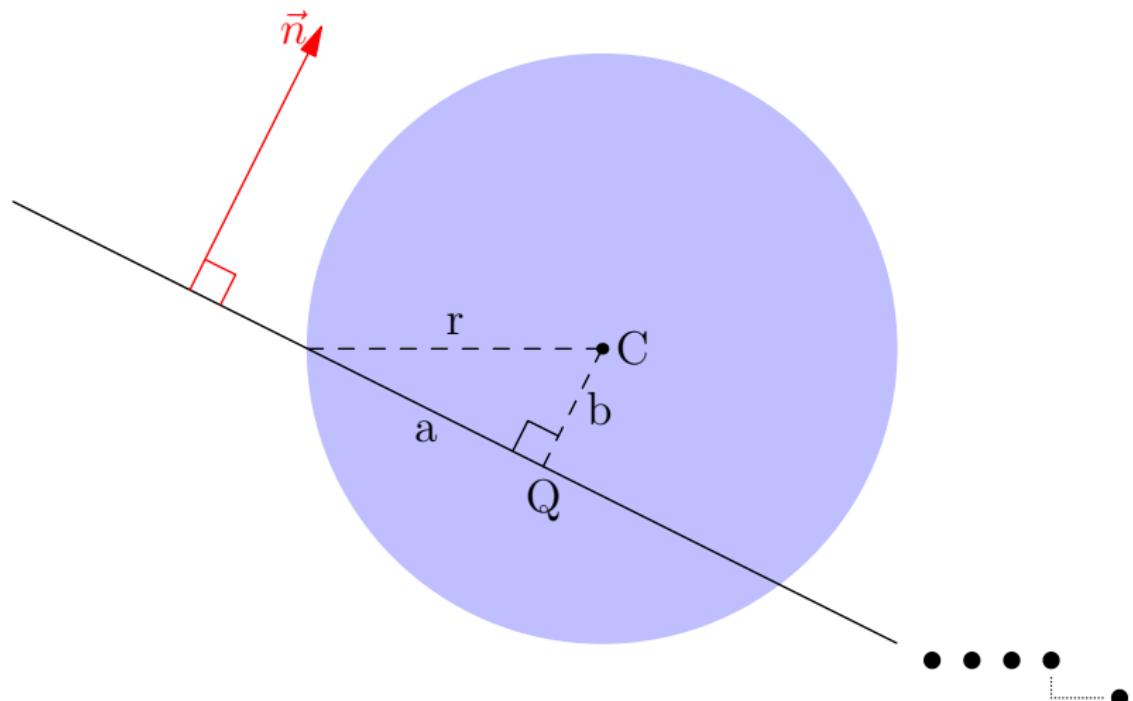
$$S = (P_x + t_u \cdot u_x + t_v \cdot v_x - C_x)^2 + (P_y + t_u \cdot u_y + t_v \cdot v_y - C_y)^2 + (P_z + t_u \cdot u_z + t_v \cdot v_z - C_z)^2 - r^2$$

Which is a polynom of degree 2 with 2 unknown that looks like:

$$a t_u^2 + b t_u t_v + c t_v^2 + d t_u + e t_v + f = 0$$



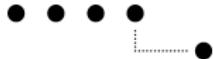
Intersection between a sphere and a plan (cont'd)



Intersection plane - sphere

```
Vect3D  $\vec{v}$  := sphere.center -  $\Pi$ .pointOnPlane  
float  $b$  :=  $|\Pi \cdot \vec{n}| \{ \|\vec{n}\| = 1 \}$   
if  $b < \text{sphere.radius}$  then  
    Point3D Q := sphere.center -  $b * \Pi \cdot \vec{n}$   
    float circle.radius :=  $\sqrt{\text{sphere.radius}^2 + b^2}$  sollte  $-b^2$  sein  
    if radius <  $\varepsilon$  then  
        intersection type := point  
        intersection point := Q  
    else  
        intersection type := circle  
        intersection center := Q  
        intersection radius := radius  
    end if  
    RETURN an intersection occurs  
end if  
RETURN no intersection occurs
```





Part V

3D Modelling



Outline

Polymeshes and polytopes

How to describe a mesh ?



Polymeshes, polyhedra and polytopes

- ▶ The description of 3D complex objects mainly based on vertices, faces and edges.
(=eckpunkte)
- ▶ vertices are single points in space.
- ▶ edges as line segment whose endpoints are vertices;
- ▶ faces are **convex** polygons that live in 3D.

Many applications support only triangular faces because of their simplicity. Non convex polygons are sometimes allowed although this complicates the implementation and the manipulation of objects.

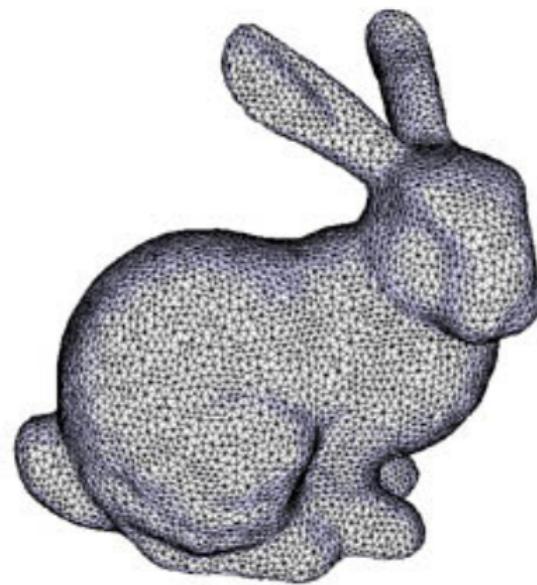


Polymeshes

- ▶ A finite collection of vertices, edges and faces is called *polygonal mesh* or *polymesh*.
- ▶ A polymesh must satisfy the following conditions:
 1. Each vertex must be shared by at least one edge
 2. Each edge must be shared by at least one face
 3. If two faces intersect, the vertex or edge of intersection must be a component of the mesh

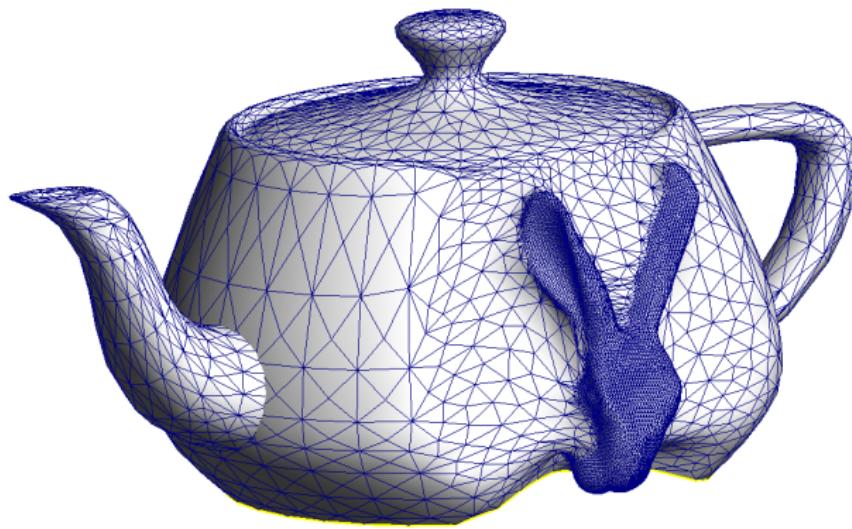


Example of meshes



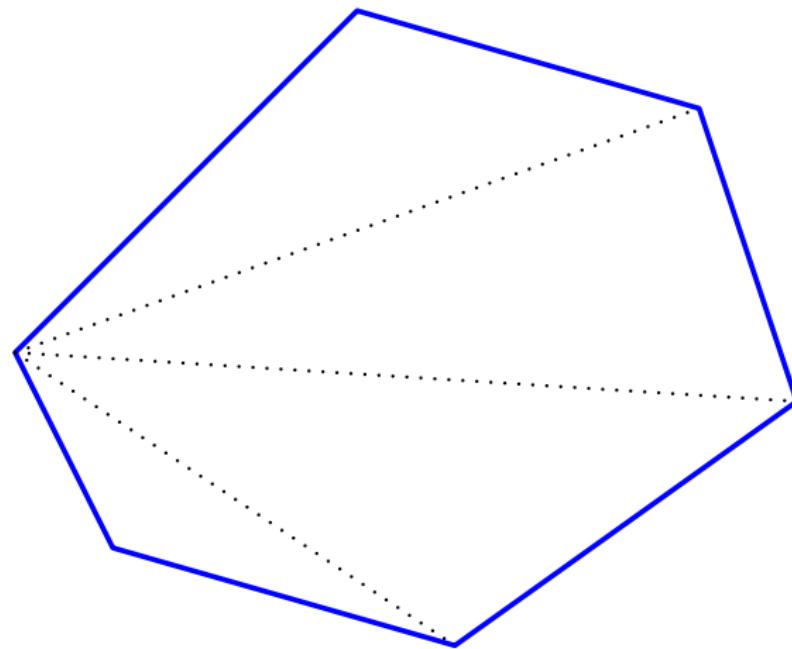
Example of meshes

bei ruhigen regionen -> es kÖnnen grÖssere
faces verwendet werden

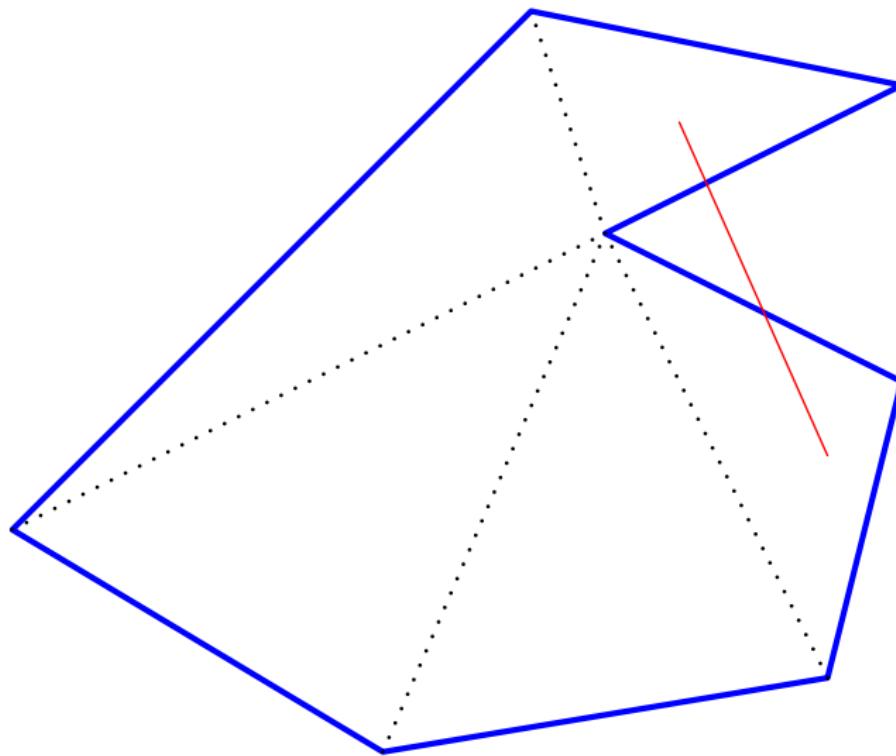


Convex polygon

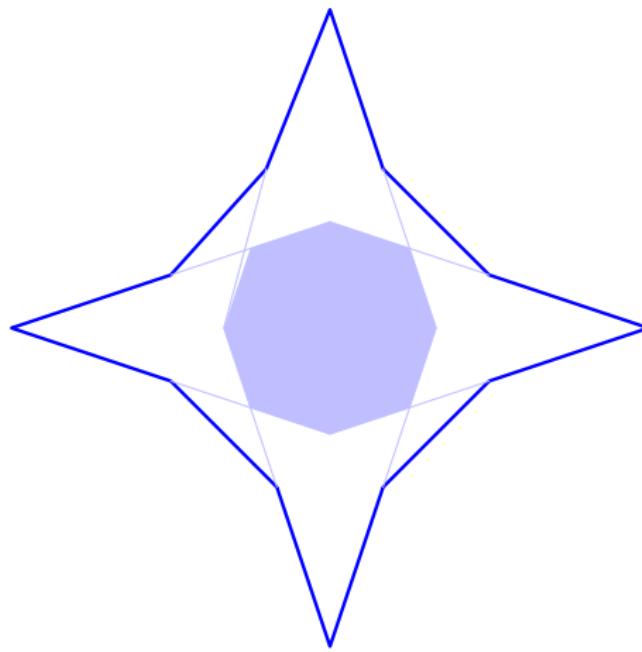
For any pair of points P_0 and P_1 on the border of the polygon, the line segment joining P_0 and P_1 is totally inside the polygon



Non convex polygon



Star shaped polygon



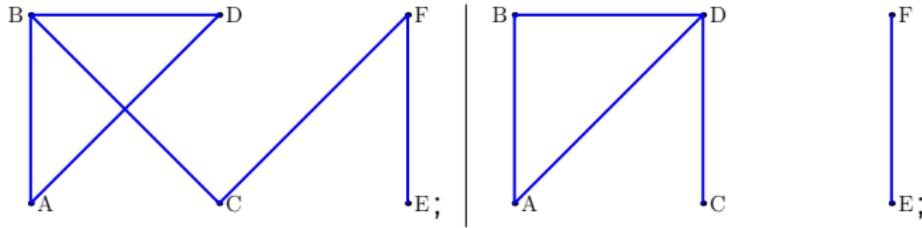
Some properties of meshes

- ▶ **Solidity** : a mesh represent a solid object if its faces together enclose a positive and finite amount of space
- ▶ **Connectedness** : a mesh is connected if every face share at least one edge with some other face.
- ▶ **Simplicity** : a mesh is simple if the object it represent is solid and has no holes through it. It can be deformed into a sphere without tearing
- ▶ **Planarity** : a mesh is planar if every face is a planar polygon.
- ▶ **Convexity** : a mesh is convex if it represent a convex object (i.e. the line between every pair of point inside the object is totally inside the object)



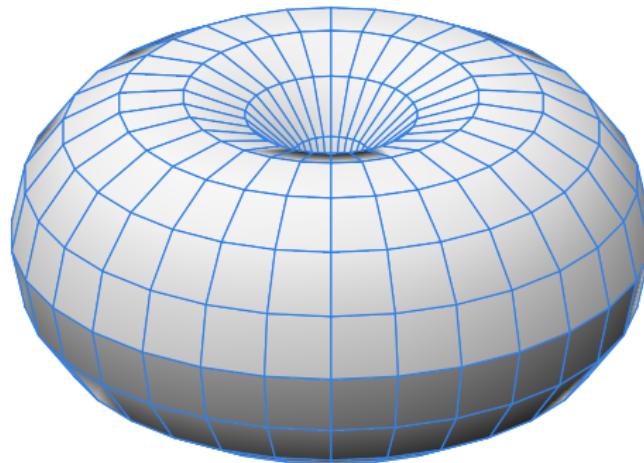
Polyhedron I

- ▶ A polyhedron is a polymesh that has additional constraints
 1. The mesh is connected. When viewed as a graph whose nodes are the vertices and the arcs are the edges, it should be possible to reach every node from every source by following a path on the graph



Polyhedron II

2. Each edge is shared by exactly 2 faces. This condition force the mesh to be a closed and bounded surface.



The Euler formula

The Euler formula provide a fundamental relationship between the number of faces, edges and vertices (F , E and V respectively) of simple polyhedron. This formula state that, for a simple polyhedra:

$$V + F - E = 2 \tag{1}$$

See http://en.wikipedia.org/wiki/Euler_characteristic for some examples or the document EulerCharacteristics.pdf in the ressource folder for a proof.



The Euler formula : some examples

1. For a *tetrahedron*, one have $F = 4$, $E = 6$ and $V = 4$. Applying the Euler formula gives : $4 + 4 - 6 = 2$
2. For a *cube*, one have $F = 6$, $E = 12$ and $V = 8$. Applying the Euler formula gives : $8 + 6 - 12 = 2$
3. For an *icosahedron*, one have $F = 20$, $E = 30$ and $V = 12$. Applying the Euler formula gives $12 + 20 - 30 = 2$

With these examples, one can see that every object that has an Euler characteristic of 2, one can find a homeomorphism to continuously transform this object into another with the same characteristic.



Outline

Polymeshes and polytopes

How to describe a mesh ?



The Indexed Face Set Model

- ▶ One list of points for the vertices of the polyhedra
- ▶ One list of faces with the indexes of the vertices, CCW sorted
- ▶ Easy to implement
- ▶ Queries are difficult

```
Point3D vert[] =  
{  
    new Point3D(p0x, p0y, p0z),  
    new Point3D(p1x, p1y, p1z),  
    new Point3D(p2x, p2y, p2z),  
} ...  
  
int edges[][] = {  
    {1,3,4,5},  
    {2,5,7,1,3},  
    ...};
```



The Vertex-Edge-Face Tables

Another representation of a polymesh is the *vertex-edge-face table* given by the following grammar:

VertexIndex := 0 through N-1

VertexIndexList := EMPTY or
 {VertexIndex v, VertexIndexList vList}

EdgeList := EMPTY or
 {Edge e; EdgeList eList}

FaceList := EMPTY or
 {Face f; FaceList fList}

Vertex := {VertexIndex v; EdgeList eList; FaceList fList}

Edge := {VertexIndex v[2]; FaceList fList }

Face := {VertexIndexList vList}



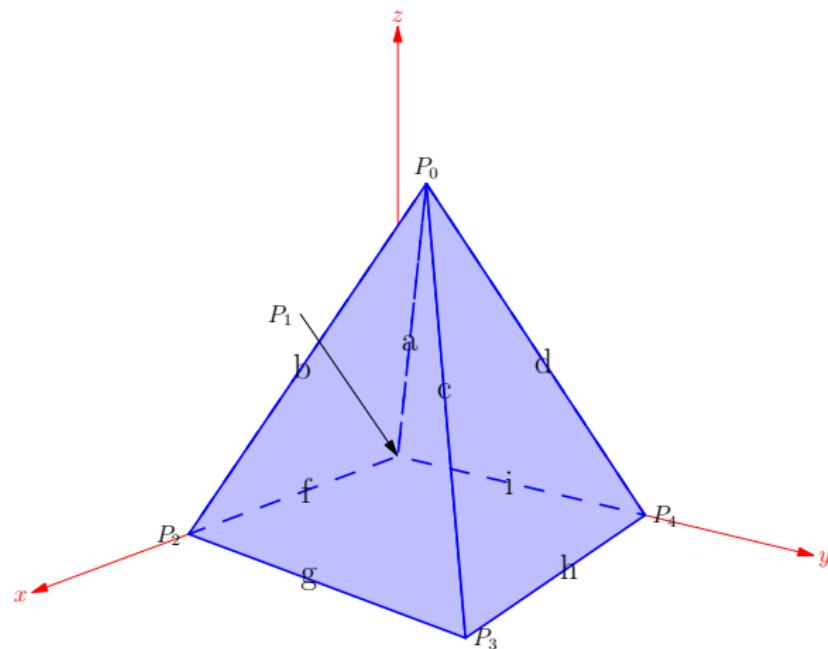
Closed Meshes

A connected mesh is said to be a *closed mesh* if:

1. each of its edges are shared by exactly 2 faces;
2. it is non self-intersecting



A closed mesh : the pyramid



The vertex-edge table for the pyramid

Vertex	Coordinates	Edge	Vertices
P_0	(1.0; 1.0; 2.0)	$E_0 = a$	P_0, P_1
P_1	(0.0; 0.0; 0.0)	$E_1 = b$	P_0, P_2
P_2	(2.0; 0.0; 0.0)	$E_2 = c$	P_0, P_3
P_3	(2.0; 2.0; 0.0)	$E_3 = d$	P_0, P_4
P_4	(0.0; 2.0; 0.0)	$E_4 = f$	P_1, P_2
		$E_5 = g$	P_2, P_3
		$E_6 = h$	P_3, P_4
		$E_7 = i$	P_4, P_1



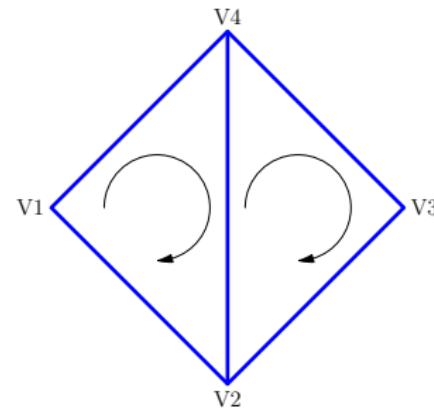
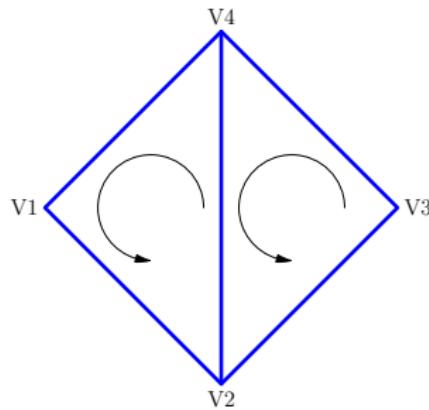
The indexed face-set model for the pyramid

Vertex	Coordinates	Face	Edge List
P_0	(1.0; 1.0; 2.0)	F_0	$P_0 - P_1 - P_2;$
P_1	(0.0; 0.0; 0.0)	F_1	$P_0 - P_2 - P_3;$
P_2	(2.0; 0.0; 0.0)	F_2	$P_0 - P_3 - P_4;$
P_3	(2.0; 2.0; 0.0)	F_3	$P_0 - P_4 - P_1;$
P_4	(0.0; 2.0; 0.0)	F_4	$P_1 - P_4 - P_3 - P_2$



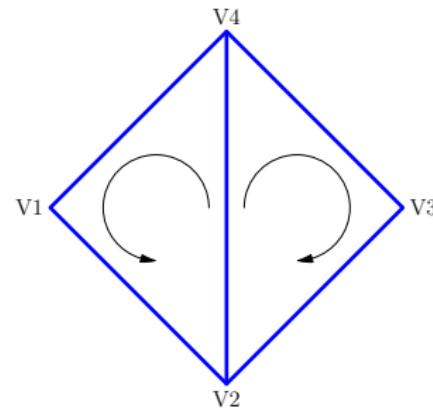
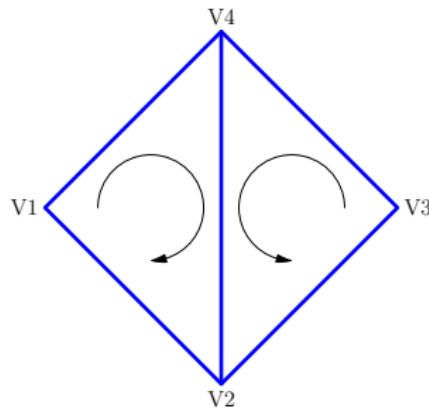
Consistent ordering of meshes

A manifold mesh has two consistent ordering of its faces :



Consistent ordering of meshes (cont'd)

Symetrically there is two inconsistent ordering of faces:



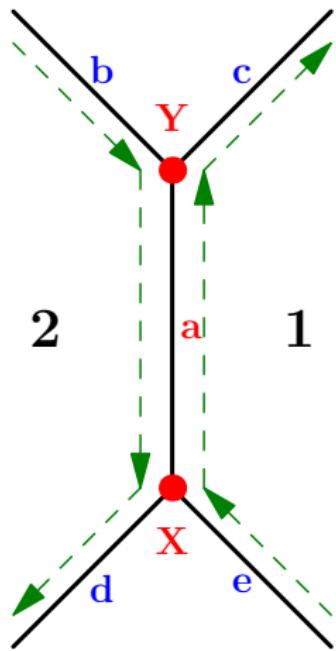
The Winged-Edge model

The simple model does not allow to retrieve easily the informations needed for the standard polygon processing. In particular, one would answer to the following questions:

1. for any face, find all the edges, traversed in CW- or CCW-order;
2. for any face, traverse all the vertices;
3. for any vertex, find all the faces that meet that vertex;
4. for any vertex, find the edges that meet that vertex;
5. for any edge, find its two vertices;
6. for any edge, find its two faces;
7. for any edge, find the next edge on a face in a certain order (CW or CCW)



The Winged-Edge model (cont'd)



Every edge contain:

1. Next edge CW
2. Next edge CCW
3. Previous edge
4. Previous edge CCW
5. Next face
6. Previous face
7. Next vertex
8. Previous vertex

The Winged-Edge model for the pyramid

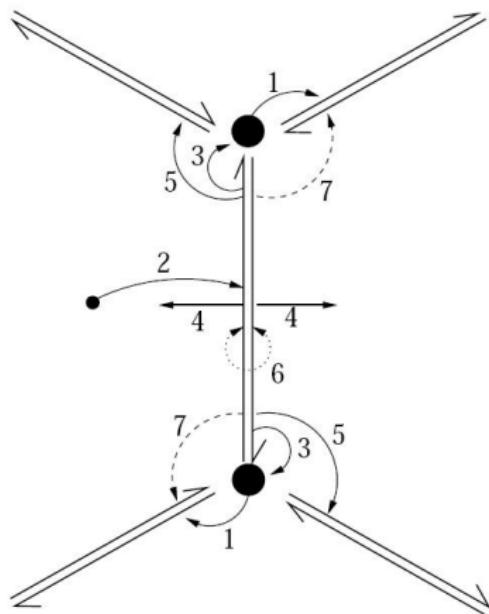
Edge	V_1	V_2	faceL	faceR	predL	predR	succL	succR
a	V_0	V_1	F_0	F_3	d	b	i	f
b								
c								
d								
f								
g								
h								
i								



Open Mesh

- ▶ Openmesh is an open source library (written in C++) which provide an efficient and generic way to defines meshes.
- ▶ It was designed to be flexible and efficient (both in time and space).
- ▶ Openmesh provides a *edge-based* data structure to describe a mesh, using the *halfedge* description model.
- ▶ Openmesh URL : <http://www.openmesh.org/>

Open Mesh (cont'd)



1. Vertex → one outgoing halfedge
2. Face → one halfedge
3. Halfedge → target vertex
4. Halfedge → its faces
5. Halfedge → next halfedge
6. Halfedge → opposite halfedge
(implicit)
7. Halfedge → previous halfedge
(optional)



Part VI

2D Clipping



Line segment clipping

The Cohen-Sutherland algorithm

The Liang-Barsky algorithm

Polygon clipping

The Sutherland-Hodgman algorithm

The Weiler-Atherton algorithm

Boolean operations on polygons



Outline

Line segment clipping

The Cohen-Sutherland algorithm

The Liang-Barsky algorithm

Polygon clipping

The Sutherland-Hodgman algorithm

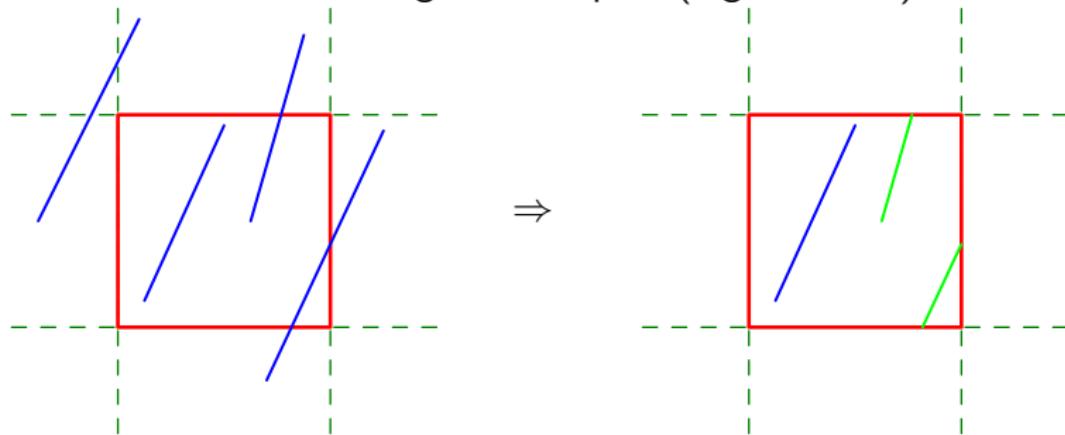
The Weiler-Atherton algorithm

Boolean operations on polygons



Clipping

The goal of clipping is mainly to eliminate parts of the scene which are not visible within a given viewport (e.g. window).



Clipping (cont'd)

If the boundaries of the clipping rectangle are at (x_{min}, y_{min}) ; (x_{max}, y_{max}) the the four inequalities must be satisfied for a point (x, y) to be inside the clipping region:

1. $x_{min} \leq x$
2. $x \leq x_{max}$
3. $y_{min} \leq y$
4. $y \leq y_{max}$



The Cohen-Sutherland algorithm

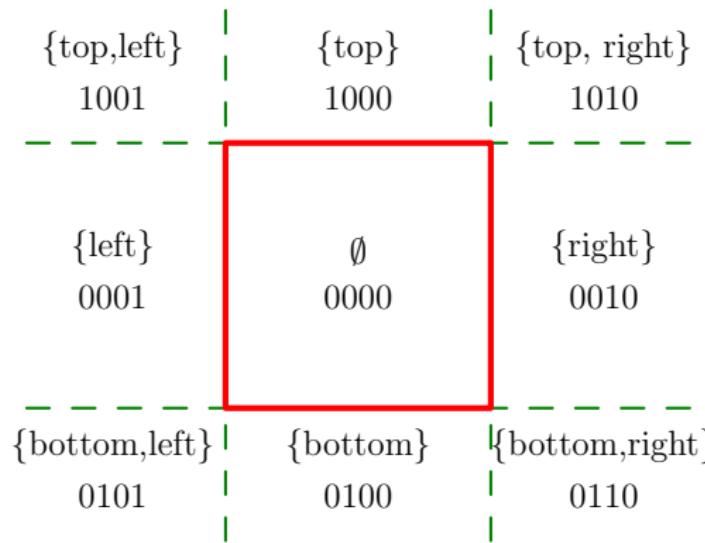
This algorithm performs some tests on lines to determine whether clipping computation can be avoided. These tests say that:

1. If the line segment is *totally inside* the clipping region ⇒ **do nothing and accept the line segment**
2. If the line segment is *totally outside* the clipping region ⇒ **do nothing and reject the line segment**
3. If the line segment is *partially inside* the clipping region ⇒ **clip the line segment**

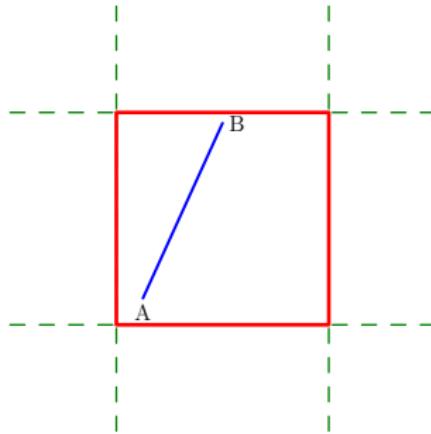


The Cohen-Sutherland algorithm (cont'd)

For these tests, the space is divided into 9 regions



The Cohen-Sutherland algorithm : example 1



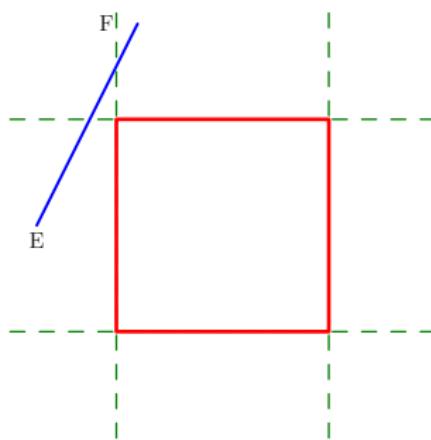
$$A = \emptyset$$

$$B = \emptyset$$

$$A \cup B = \emptyset$$

⇒ accept

The Cohen-Sutherland algorithm : example 2



$$E = \{\text{left}\}$$

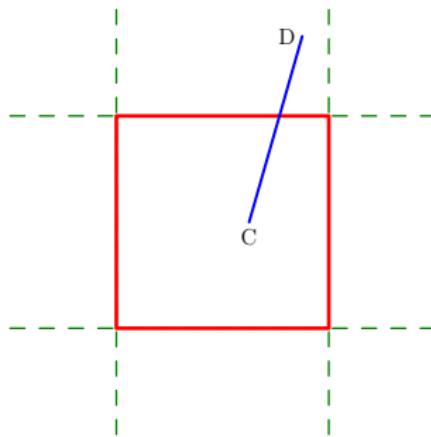
$$F = \{\text{top}\}$$

$$E \cup F \neq \emptyset$$

$$E \cap F = \emptyset$$

⇒ clip

The Cohen-Sutherland algorithm : example 3



$$C = \emptyset$$

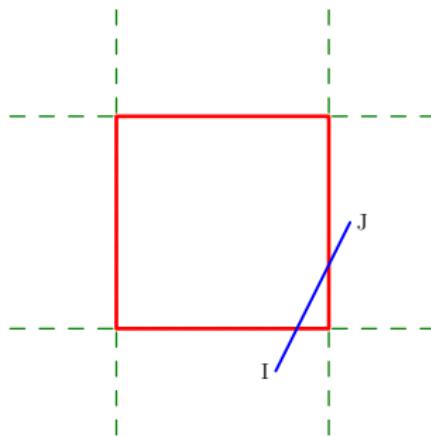
$$D = \{top\}$$

$$C \cup D \neq \emptyset$$

$$C \cap D = \emptyset$$

⇒ **clip**

The Cohen-Sutherland algorithm : example 4



$I = \{bottom\}$

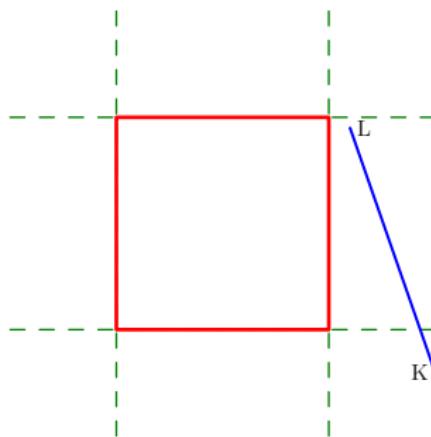
$J = \{right\}$

$I \cup J \neq \emptyset$

$I \cap J = \emptyset$

⇒ clip

The Cohen-Sutherland algorithm : example 5



$$K = \{bottom, right\}$$

$$L = \{right\}$$

$$K \cup L \neq \emptyset$$

$$K \cap L \neq \emptyset$$

⇒ **reject**

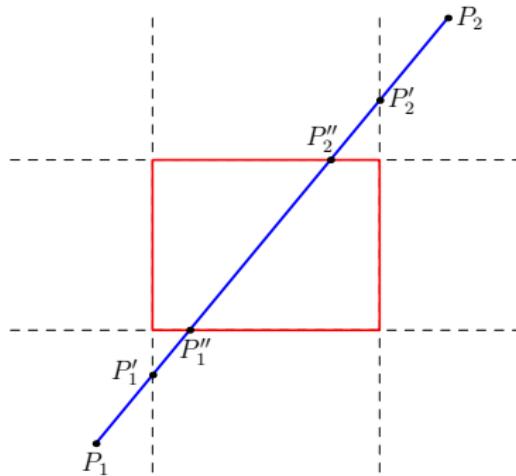
The Cohen-Sutherland algorithm (cont'd)

The rules for accepting (or rejecting) a line segment into the clipping algorithm are:

1. If $P_1 \cup P_2 = \emptyset$ then **accept**. The line segment is completely inside the clipping region.
2. If $P_1 \cap P_2 \neq \emptyset$ then **reject**. The line segment is completely outside the clipping region.
3. If $P_1 \cap P_2 = \emptyset$ then **clip**. The line segment should be clipped.



The Cohen-Sutherland algorithm (cont'd)



$$P_1 = \{\text{bottom, left}\}$$

$$P_2 = \{\text{top, right}\}$$

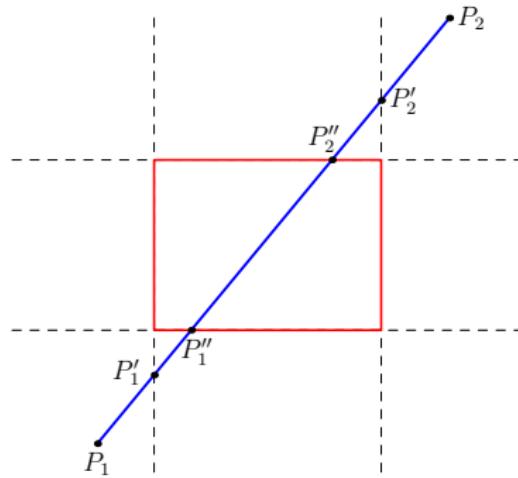
$$P_1 \cup P_2 = \{\text{bottom, top, left, right}\}$$

$$P_1 \cap P_2 = \{\}$$

⇒ do clipping

$$m := \frac{P_2.y - P_1.y}{P_2.x - P_1.x}$$

The Cohen-Sutherland algorithm (cont'd)



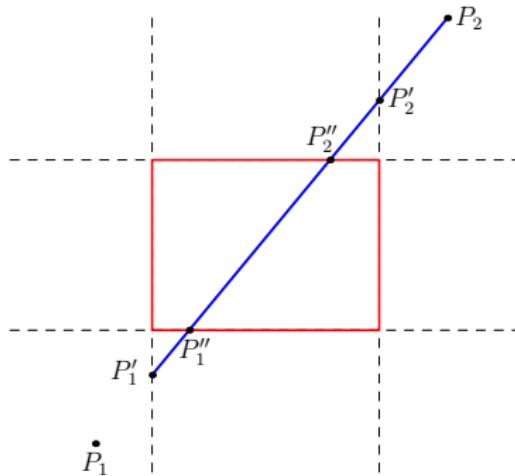
Is {left} subset of P_1 ?

Yes \Rightarrow

$$P'_1.y = (winLeft - P_1.x) \cdot m + P_1.y$$

$$P'_1.x = winLeft$$

The Cohen-Sutherland algorithm (cont'd)



Is {bottom} subset of P_1 ?

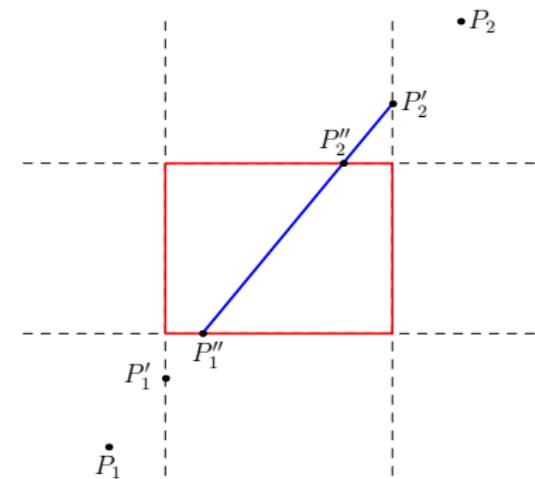
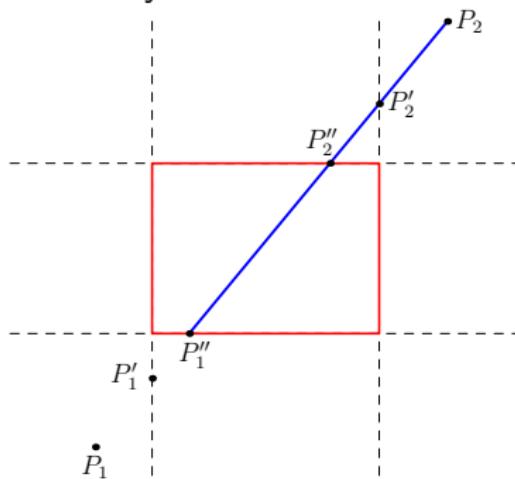
Yes \Rightarrow

$$\begin{aligned} P''_1.x &:= \\ &(\text{winBottom} - P'_1.y)/m + P'_1.x \end{aligned}$$

$$P''_1.y = \text{winBottom}$$

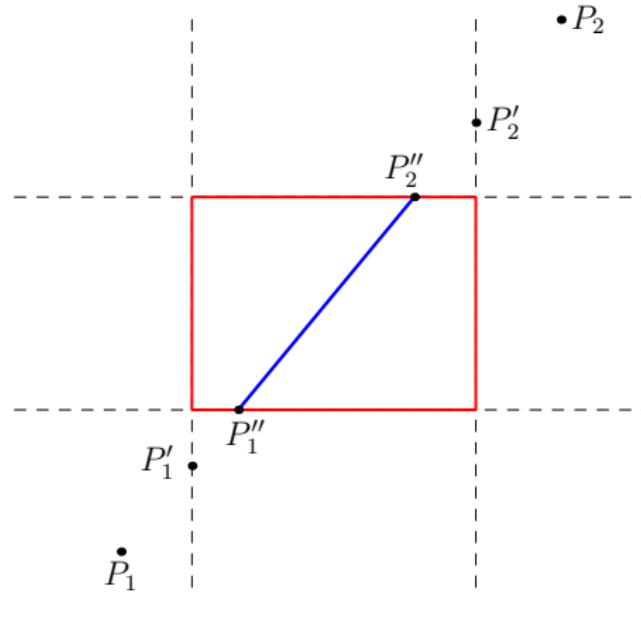
The Cohen-Sutherland algorithm (cont'd)

Similarly



The Cohen-Sutherland algorithm (cont'd)

And finally:



Pseudo-code for the Cohen-Sutherland algorithm

Require: a rectangular axis aligned clipping window
Require: a segment given by two distinct points
Ensure: a clipped segment that is totally inside the clipping window

```
if trivial accept then
    return the segment unmodified
end if
if trivial reject then
    return an empty segment
end if
for all point ∈ segmentEnd do
    if point in the BOTTOM area then
        clip point against BOTTOM boundary
    end if
    if point in the TOP area then
        clip point against TOP boundary
    end if
    if point in the LEFT area then
        clip point against LEFT boundary
    end if
    if point in the RIGHT area then
        clip point against RIGHT boundary
    end if
end for
return segment
```



2D Liang-Barsky Clipping algorithm

The main idea of the Liang-Barsky clipping algorithm is to do as much testing as possible before computing line intersection.

Let s_1 be line segment defined by its extremities $(x_0; y_0)$ and $(x_1; y_1)$. The parametric form for this line segment is given by:

$$x = x_0 + \lambda(x_1 - x_0) = x_0 + \lambda \Delta x$$

$$y = y_0 + \lambda(y_1 - y_0) = y_0 + \lambda \Delta y$$

with $0 < \lambda \leq 1$.



2D Liang-Barsky Clipping algorithm (cont'd)

A point is in the clip window if:

$$x_{min} \leq x_0 + \lambda \Delta x \leq x_{max}$$

$$y_{min} \leq y_0 + \lambda \Delta y \leq y_{max}$$

which can be expressed by the 4 inequalities :

$$\lambda p_k \leq q_k \quad k = 1, 2, 3, 4$$

where

$$p_1 = -\Delta x \quad q_1 = x_0 - x_{min}$$

$$p_2 = \Delta x \quad q_2 = x_{max} - x_0$$

$$p_3 = -\Delta y \quad q_3 = y_0 - y_{min}$$

$$p_4 = \Delta y \quad q_4 = y_{max} - y_0$$

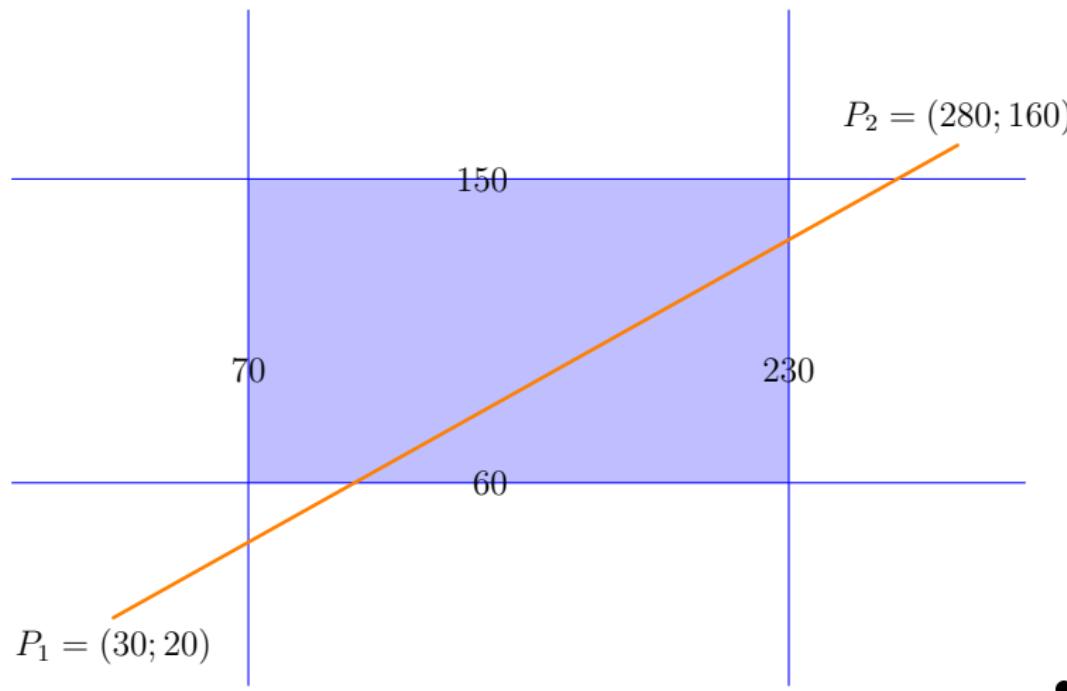


2D Liang-Barsky Clipping algorithm (cont'd)

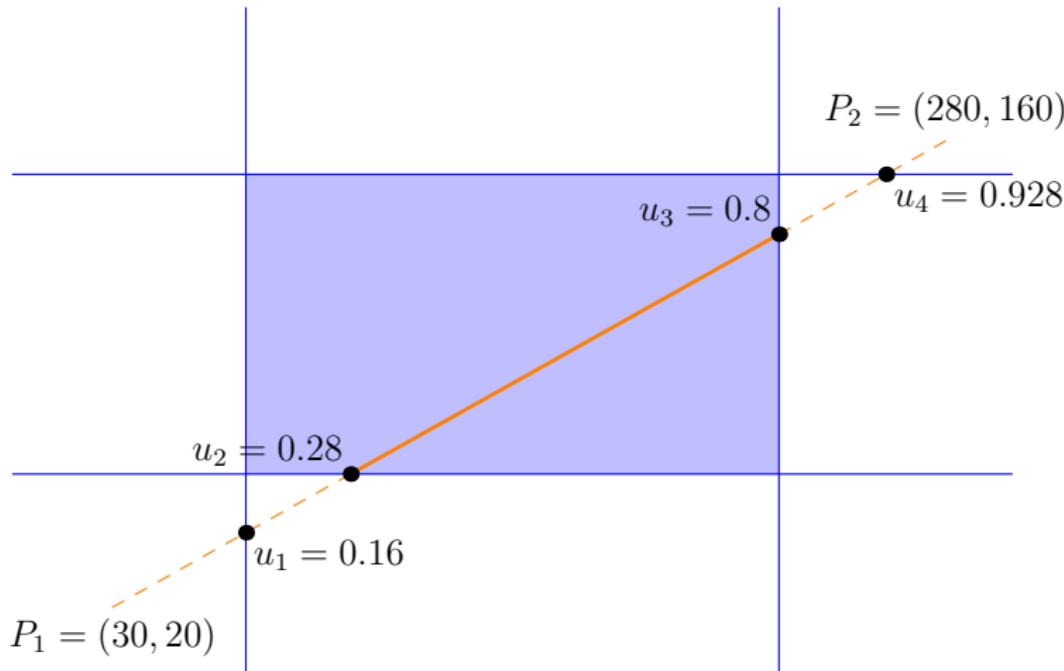
1. A line parallel to a clipping window edge has $p_k = 0$ for that boundary.
2. If for the *same value* of k , $q_k < 0$, the line is completely outside and can be eliminated.
3. When $p_k < 0$ the line proceeds outside to inside the clip window and when $p_k > 0$ the line proceeds inside to outside.
4. For a non-zero p_k , the value $\lambda_k = q_k/p_k$ gives the intersection point with the window boundaries
5. For each line segment calculate λ_1 and λ_2 . For λ_1 look at the boundaries for which $p_k < 0$ and set $\lambda_1 = \max(0, q_k/p_k)$. For λ_2 look at boundaries for which $p_k > 0$ and set $\lambda_1 = \min(1, q_k/p_k)$. If $\lambda_1 > \lambda_2$ the line is outside and therefore rejected.



2D Liang-Barsky Clipping : an example



2D Liang-Barsky Clipping : an example



Polygon clipping

- ▶ Most of the graphics package support only fill area that are polygons
- ▶ Sometimes only convex polygons are accepted
- ▶ A standard line clipping (e.g. Cohen-Sutherland) will produce a set of disjoint lines
- ▶ One need an algorithm which returns a closed polygon or a set of closed polygons.



Outline

Line segment clipping

The Cohen-Sutherland algorithm

The Liang-Barsky algorithm

Polygon clipping

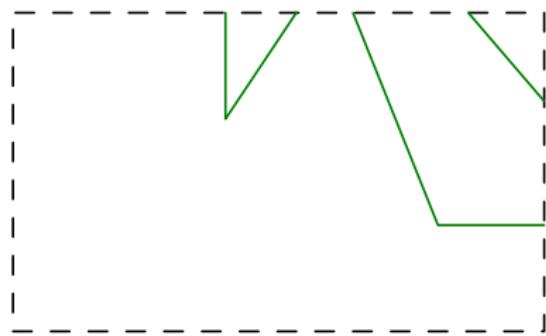
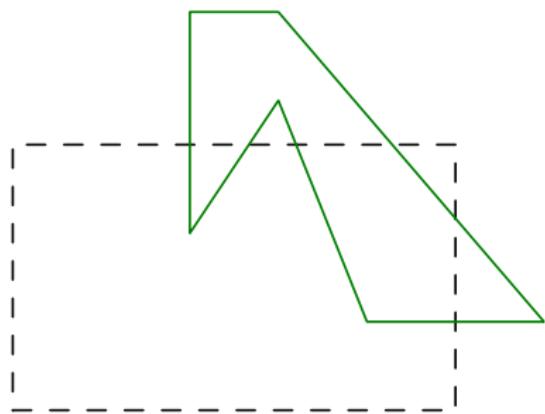
The Sutherland-Hodgman algorithm

The Weiler-Atherton algorithm

Boolean operations on polygons

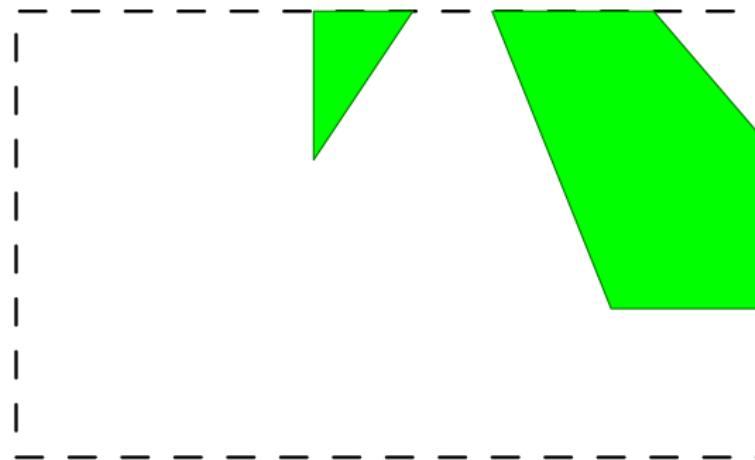


Polygon clipping using line clipping algorithm



Polygon clipping

The correct solution should be:

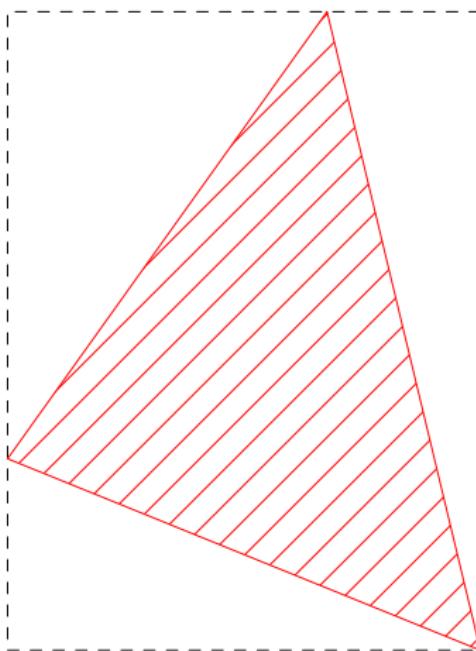


Polygon clipping

- ▶ Basically one can process using the same approach as in a line clipping algorithm
- ▶ First one can check if a polygon can be totally saved or totally rejected by testing if its coordinates extends (bounding box).
- ▶ The simplest bounding box is the Axis Aligned Bounding Box (AABB) which can be computed finding the minimal x coordinate of all vertices of the polygon, the maximal x coordinate, the minimal y coordinate and the maximal y coordinate.

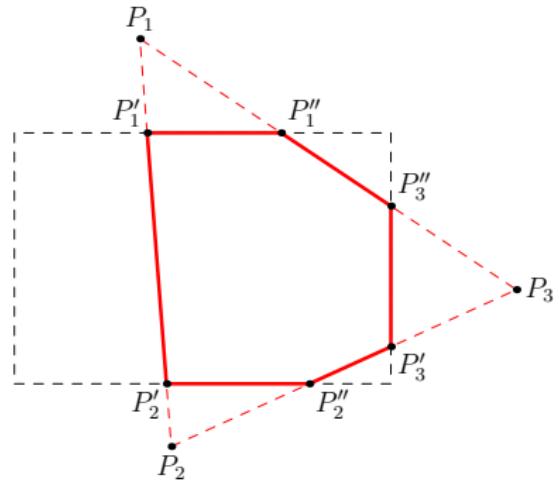
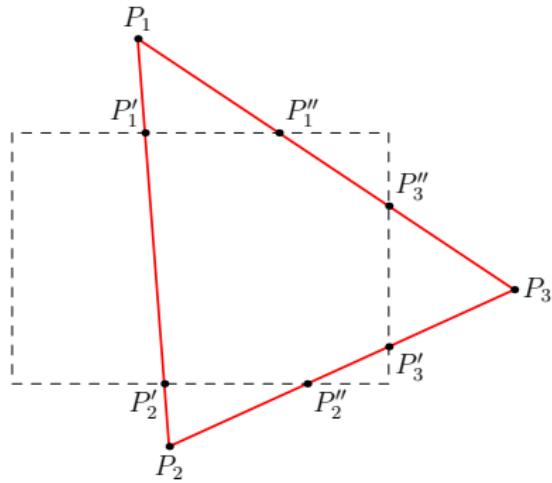


Polygon clipping



Polygon clipping

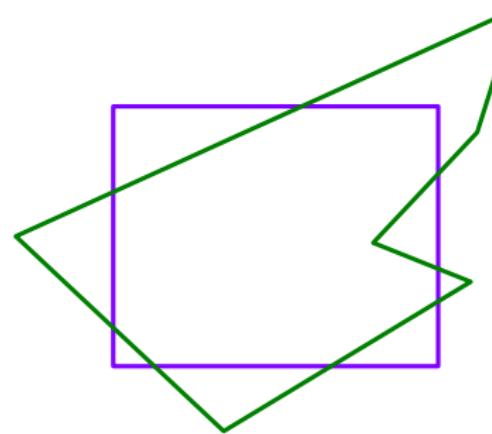
If the fill area is not completely inside or completely outside, one need to locate the polygon intersection with the clipping boundaries.



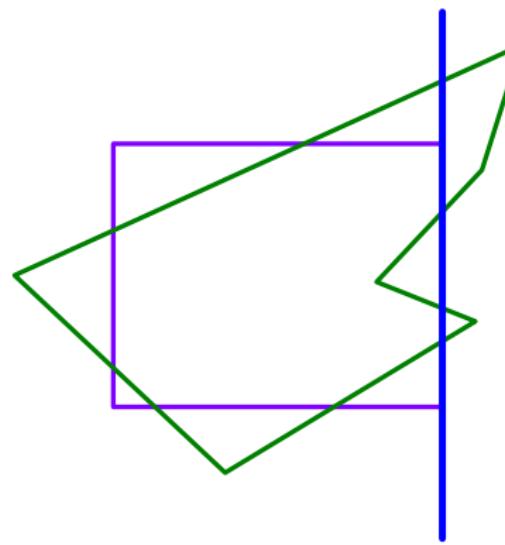
The Sutherland-Hodgman algorithm

The basic idea of the algorithm is :

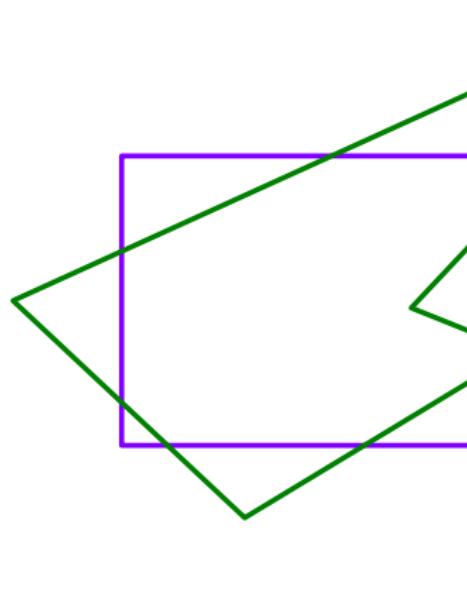
- ▶ Consider each edge of the viewport individually
- ▶ Clip the polygon against the edge equation
- ▶ After doing all planes, the polygon is fully clipped



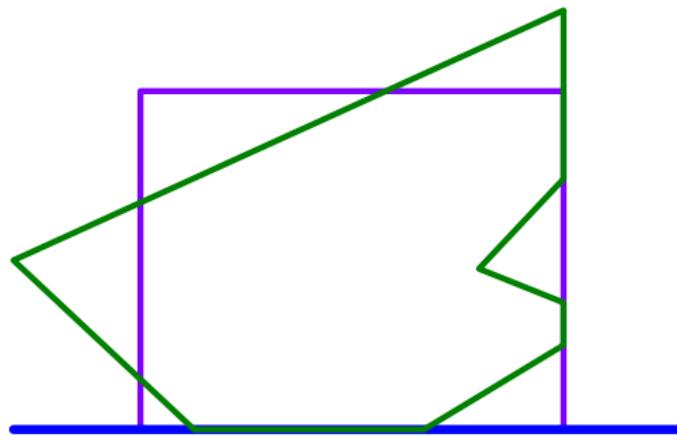
The Sutherland-Hodgman algorithm (cont'd)



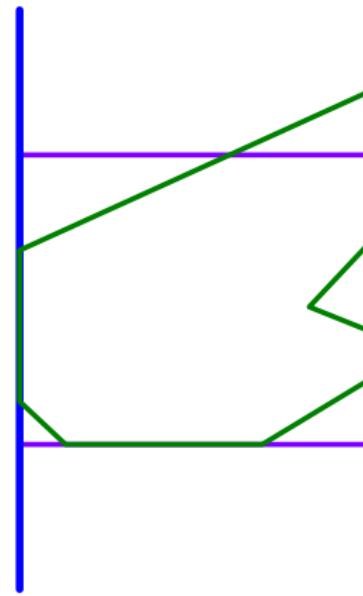
The Sutherland-Hodgman algorithm (cont'd)



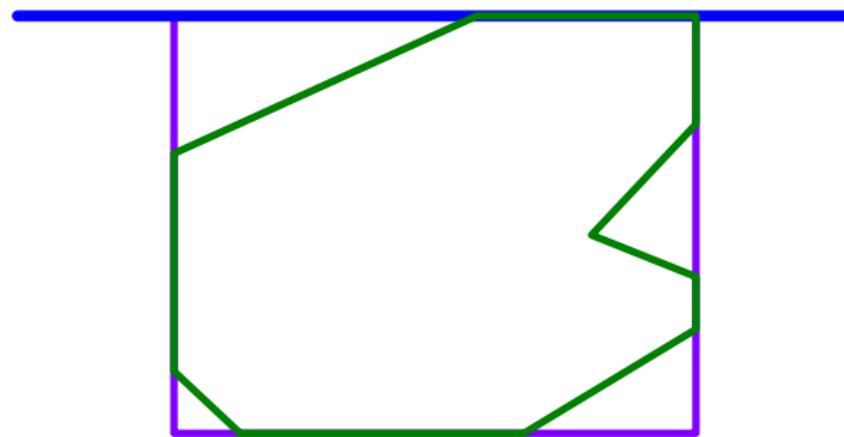
The Sutherland-Hodgman algorithm (cont'd)



The Sutherland-Hodgman algorithm (cont'd)



The Sutherland-Hodgman algorithm (cont'd)



The Sutherland-Hodgman algorithm (cont'd)

- ▶ The Sutherland-Hodgman algorithm is an efficient algorithm to clip convex polygon
- ▶ For the non convex polygon, the result may be wrong since the algorithm always return a single polygon
- ▶ The general strategies is to sent the pair of endpoints for each successive polygon edges through the series of clipper (left, right, bottom, top).
- ▶ The different clipper may work in parallel.



The Sutherland-Hodgman algorithm (cont'd)

Each clipper should consider 4 situations:

1. Both the start point and the end point of the edge as inside the clipping region
2. The start point is inside the clipping region and the end point is outside
3. Both vertices are outside the clipping region
4. The start point is outside the clipping region and the end point is inside



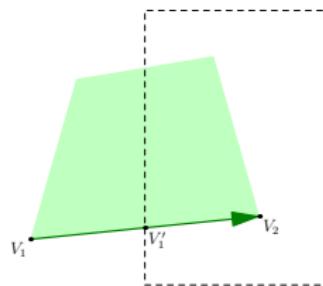
The Sutherland-Hodgman algorithm (cont'd)

Each of the successive clipper generates an output to the next clipper according to:

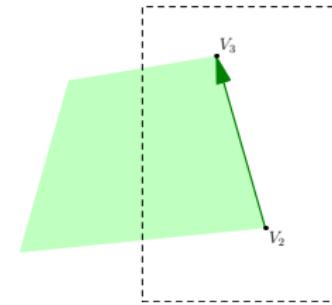
1. If start point is outside and end point inside
⇒ output = {intersection with the border, end point}
2. If both points are inside
⇒ output = {end point}
3. If start point is inside and end point outside
⇒ output = {intersection with the border}
4. If both points are outside
⇒ output = {}



The Sutherland-Hodgman algorithm (cont'd)

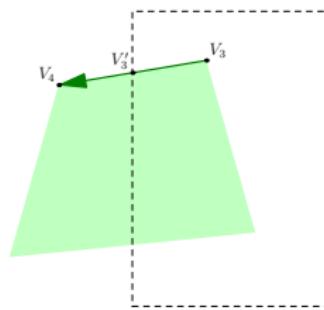


Input : {outside, inside}
Output : $\{V_1', V_2\}$

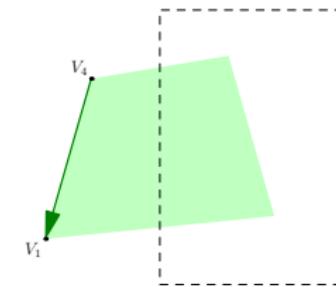


Input : {inside, inside}
Output : $\{V_3\}$

The Sutherland-Hodgman algorithm (cont'd)

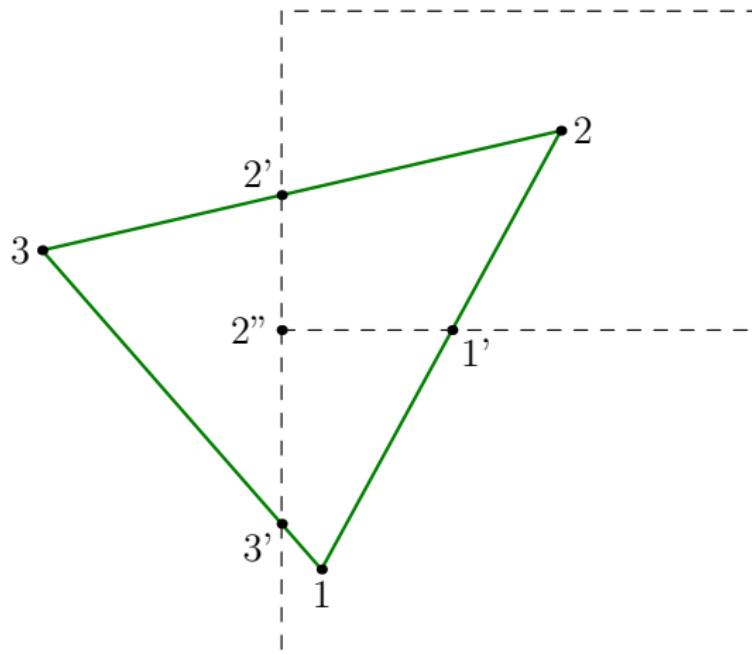


Input : {inside, outside}
Output : { V'_3 }

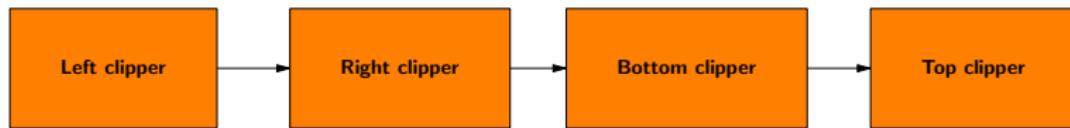


Input : {outside, outside}
Output : {}

The Sutherland-Hodgman algorithm (cont'd)

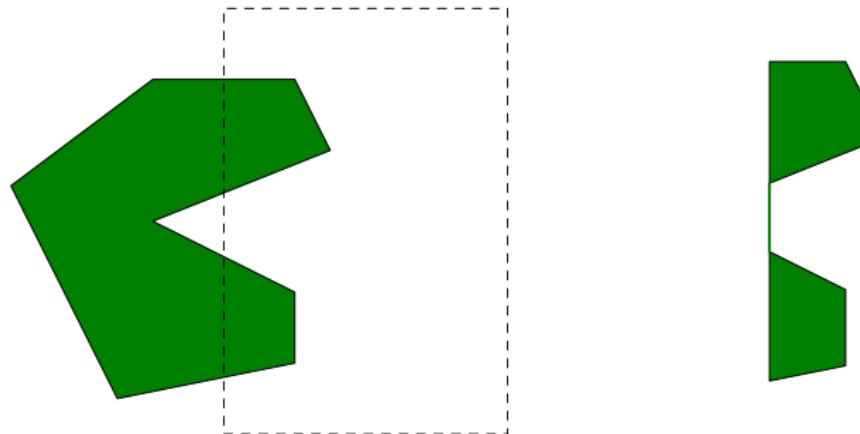


The Sutherland-Hodgman algorithm (cont'd)

 $\{1,2\} \rightarrow \{2\}$ $\{2,3\} \rightarrow \{2'\}$ $\{3,1\} \rightarrow \{3',1\}$ $\{2,2'\} \rightarrow \{2'\}$ $\{2',3'\} \rightarrow \{3'\}$ $\{3',1\} \rightarrow \{1\}$ $\{1,2\} \rightarrow \{2\}$ Bottom clipper Top clipper $\{2',3'\} \rightarrow \{2''\}$ $\{3',1\} \rightarrow \{\}$ $\{1,2\} \rightarrow \{1',2\}$ $\{2,2'\} \rightarrow \{2'\}$ $\{2'',1'\} \rightarrow \{1'\}$ $\{1',2\} \rightarrow \{2\}$ $\{2,2'\} \rightarrow \{2'\}$ $\{2',2''\} \rightarrow \{2''\}$ 

The Sutherland-Hodgman algorithm (cont'd)

For non-convex polygon, the Sutherland-Hodgman may produce wrong results:



Pseudo-code for Sutherland-Hodgman

```
List outputList = subjectPolygon;
for (Edge clipEdge in clipPolygon) do
    List inputList = outputList;
    outputList.clear();
    Point S = inputList.last;
    for (Point E in inputList) do
        if (E inside clipEdge) then
            if (S not inside clipEdge) then
                outputList.add(ComputeIntersection(S,E,clipEdge));
            end if
            outputList.add(E);
        else if (S inside clipEdge) then
            outputList.add(ComputeIntersection(S,E,clipEdge));
        end if
        S = E;
    end for
end for
```



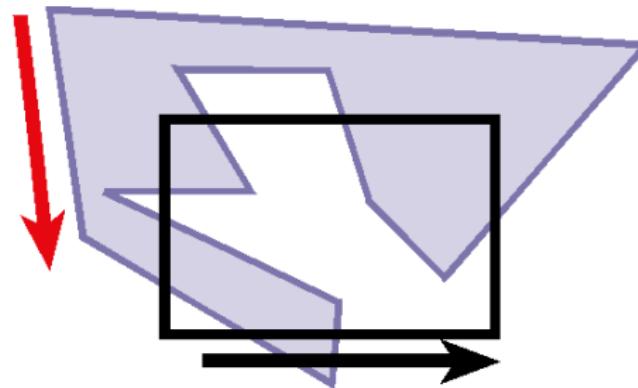
The Weiler-Atherton algorithm

1. It is a general clipping algorithm which can clip non convex polygon against arbitrary clipping window.
2. This algorithm is also used in 3D for hidden surface removal.



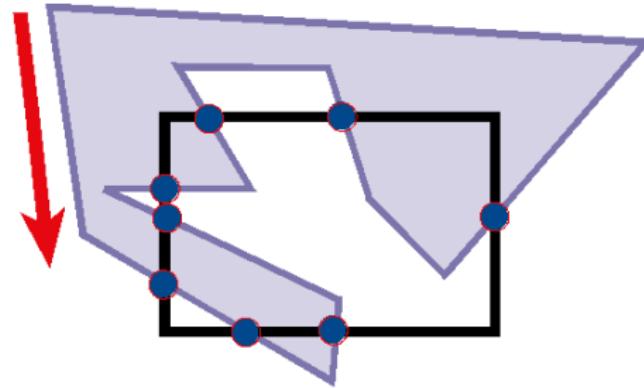
The Weiler-Atherton algorithm (cont'd)

Main algorithm strategy : walk along the polygon and window edges counterclockwise to find the clipped regions



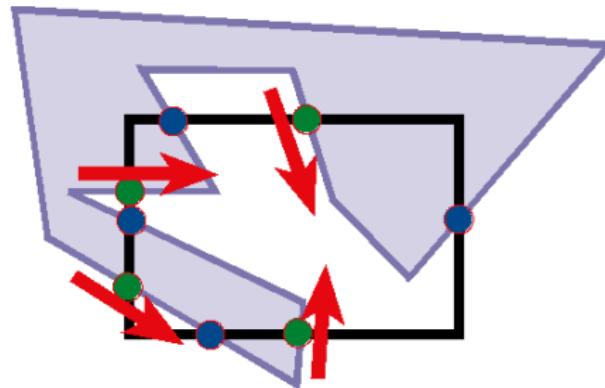
The Weiler-Atherton algorithm (cont'd)

First find all intersection points between the edges of polygon and clipping window (clipping window may be non-rectangular).



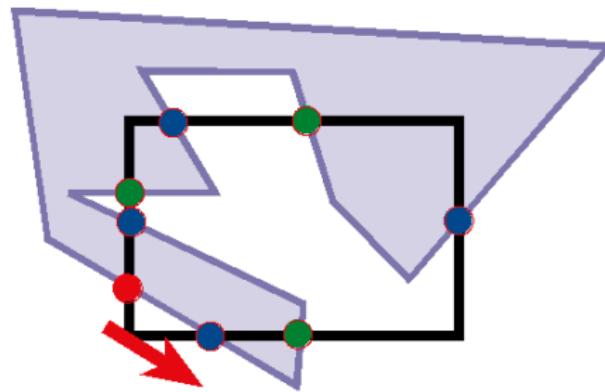
The Weiler-Atherton algorithm (cont'd)

Mark points where polygon enters clipping window (green here) as the walk is taken along the polygon



The Weiler-Atherton algorithm (cont'd)

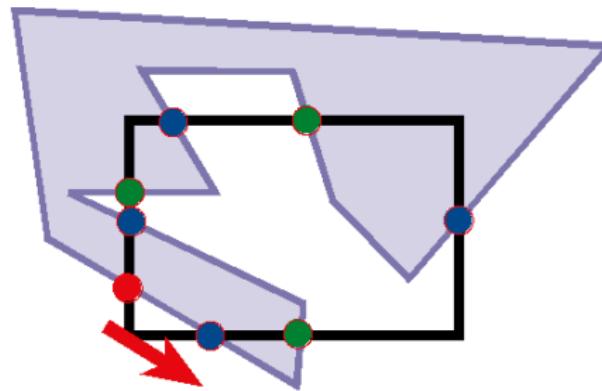
While there is still an unprocessed entering intersection : Walk"
polygon/window boundary



The Weiler-Atherton algorithm (cont'd)

If intersection point is *out* → *in*:

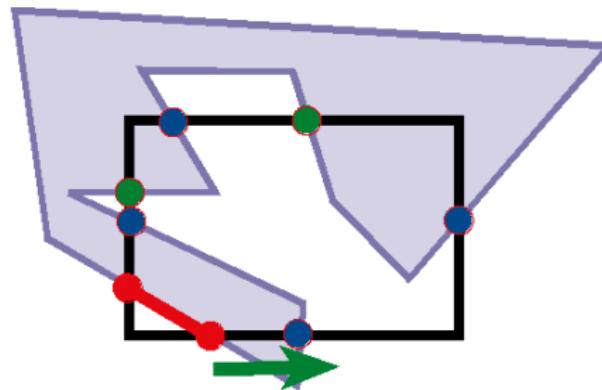
- ▶ Record clipped point
- ▶ Follow polygon boundary



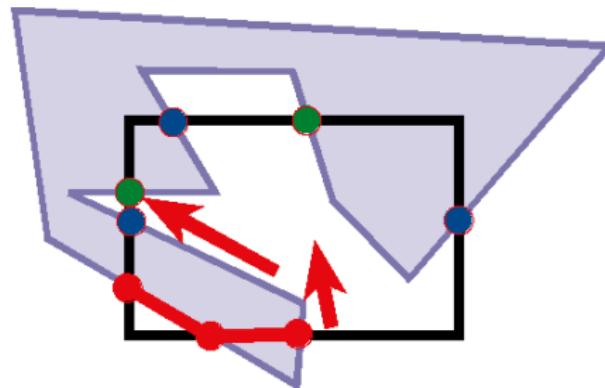
The Weiler-Atherton algorithm (cont'd)

If intersection point is *in* → *out*:

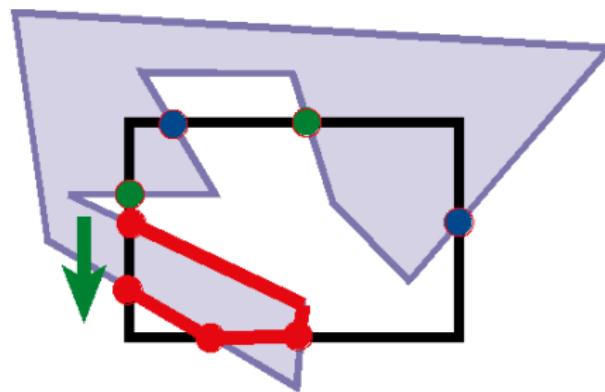
- ▶ Record clipped point
- ▶ Follow window boundary



The Weiler-Atherton algorithm (cont'd)

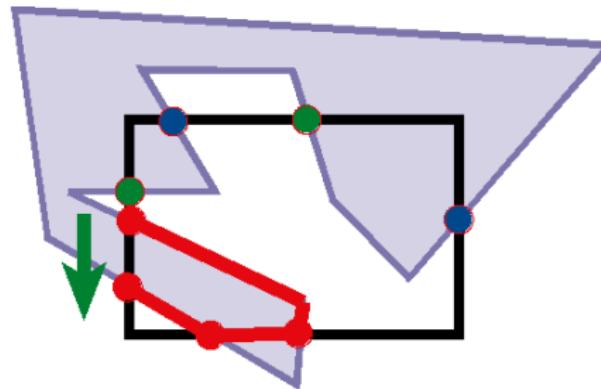


The Weiler-Atherton algorithm (cont'd)



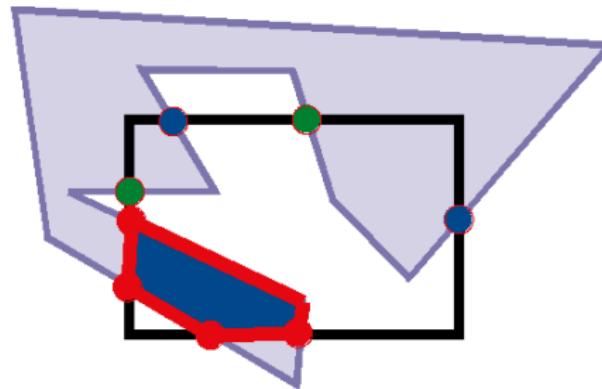
The Weiler-Atherton algorithm (cont'd)

Repeat until the clipped-polygon is closed



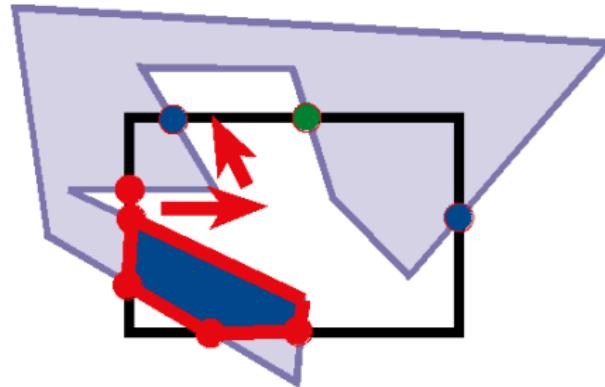
The Weiler-Atherton algorithm (cont'd)

Find the next unprocessed entering point

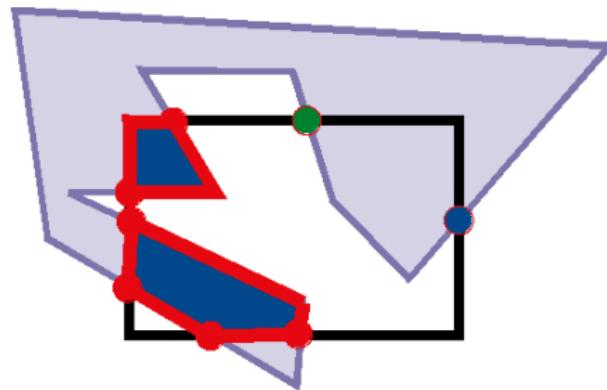


The Weiler-Atherton algorithm (cont'd)

Compute the next clipped subpolygon

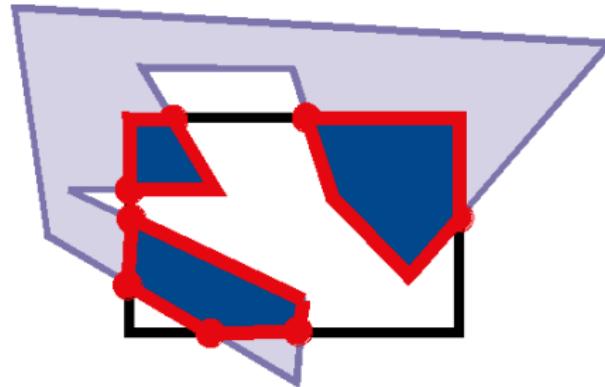


The Weiler-Atherton algorithm (cont'd)



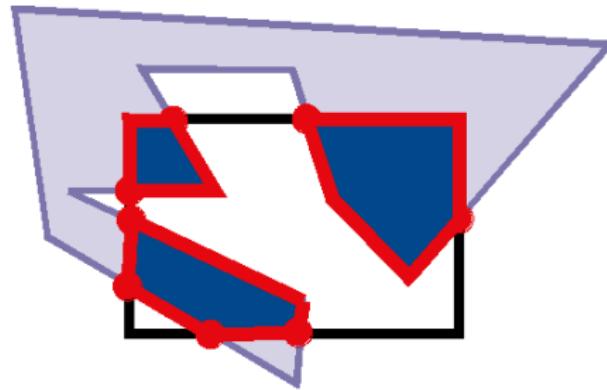
The Weiler-Atherton algorithm (cont'd)

Repeat until there is no more unprocessed entering point



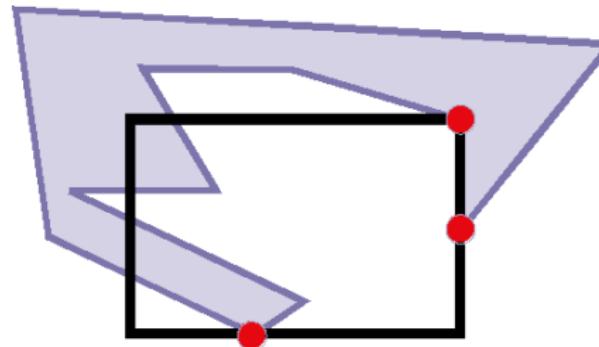
The Weiler-Atherton algorithm (cont'd)

Importance of good adjacency data structure (Here we can simply list the oriented edges)



The Weiler-Atherton algorithm (cont'd)

- ▶ What if a vertex is on the boundary?
- ▶ What happens if it is "almost" on the boundary?
 - ▶ Problem with floating point precision
- ▶ **Welcome to the real world of geometry!**



Outline

Line segment clipping

The Cohen-Sutherland algorithm

The Liang-Barsky algorithm

Polygon clipping

The Sutherland-Hodgman algorithm

The Weiler-Atherton algorithm

Boolean operations on polygons



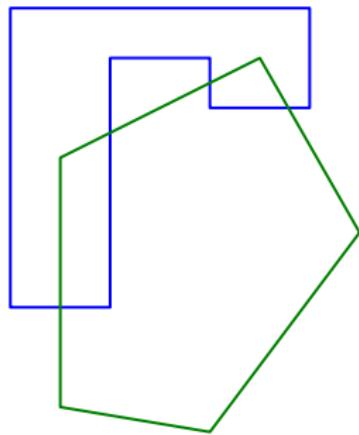
Boolean operations on polygons

- ▶ Sometimes one need to perform some boolean operation on polygon (e.g. CAD systems).
- ▶ The meaning of these operations is the same as for the operations on sets
- ▶ The methods and techniques to choose which part of the polygon should be considered are the same as those used for polygon clipping.
- ▶ These algorithms expect that the input polygons are simple (or Jordan) which mean that the edges does not cross outside the polygon vertices.



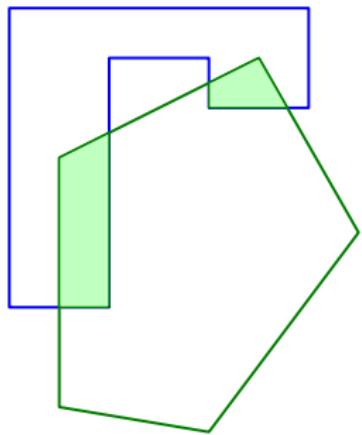
Boolean operations on polygons (cont'd)

Let P_1 and P_2 be the two polygons:



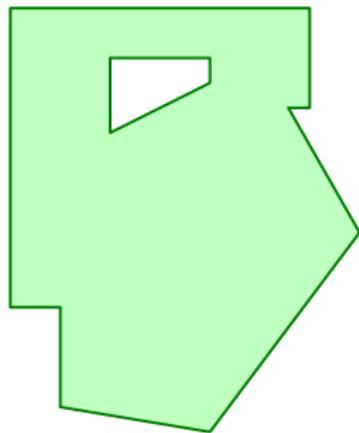
Boolean operations on polygons (cont'd)

The *intersection* of these two polygons is defined as:



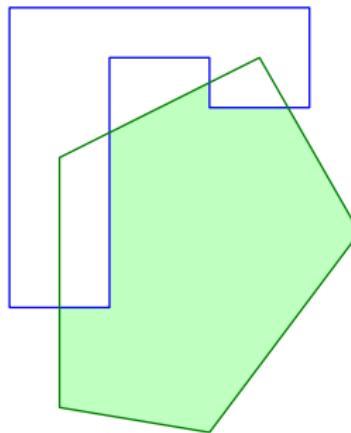
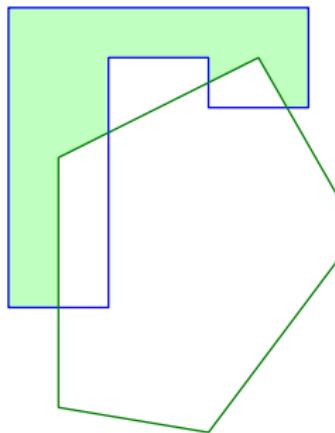
Boolean operations on polygons (cont'd)

The *union* of these two polygons is defined as:



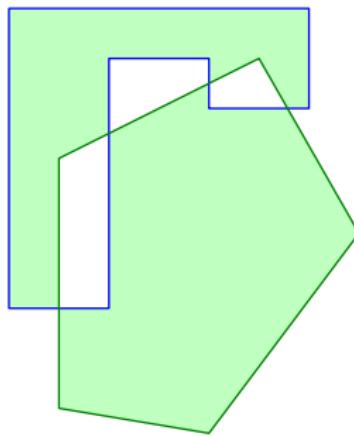
Boolean operations on polygons (cont'd)

The *difference* of these two polygons is defined as:



Boolean operations on polygons (cont'd)

The *symmetrical difference (XOR)* of these two polygons is defined as:



Part VII

The colors model and artificial light



Outline

The perception of light and color

Color models

Transformations between color models

Color Look-up Table

Illumination models

Diffuse reflection

Specular reflection

Shading models

Flat shading

Gouraud Shading

Phong Shading

Full Illumination equation

The Cook-Torrance model

BRDF

BSDF

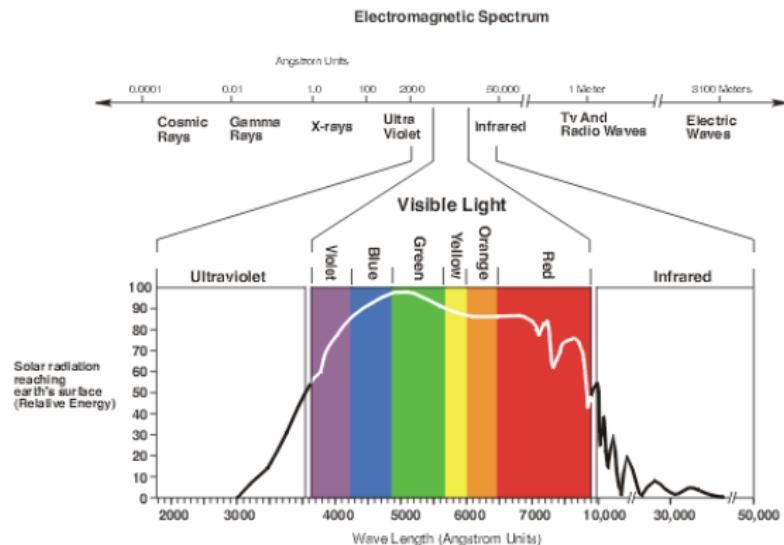


The Kajiya equation

Light and electromagnetical waves

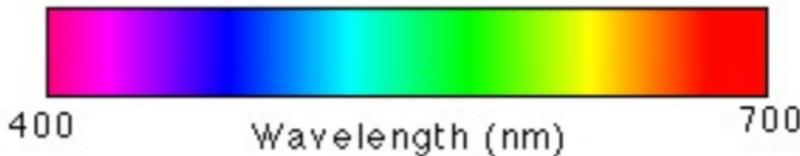
Visible light, ultraviolet light, x-rays, TV and radio waves, etc are all forms of electromagnetic energy which travels in waves. The wavelength of these waves is measured in a tiny unit called the Angstrom, equal to 1 ten billionth of a meter. Another unit sometimes used to measure wavelength of light waves is nanometers (nm) which are equal to 1 billionth of a meter.

Light and electromagnetical waves



Visible Light

There is a narrow range of this electromagnetic energy from the sun and other light sources which creates energy of wavelengths visible to humans. Each of these wavelengths, from approximately 4000 Angstroms to 7000 Angstroms, is associated with a particular color response. For example, the wavelengths near 4000 Angstroms (400 nm) are violet in color while those near 7000 (700 nm) are red.



Outline

The perception of light and color

Color models

 Transformations between color models

Color Look-up Table

Illumination models

 Diffuse reflection

 Specular reflection

Shading models

 Flat shading

 Gouraud Shading

 Phong Shading

Full Illumination equation

The Cook-Torrance model

 BRDF

 BSDF

The Kajiya equation

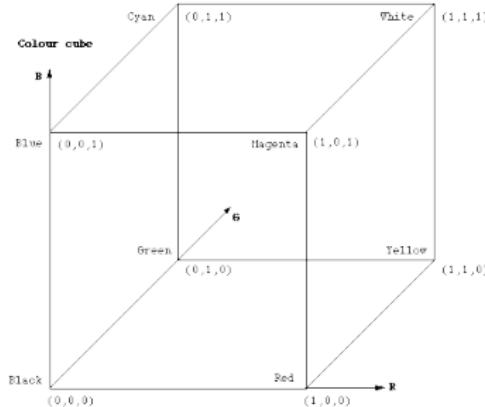


The RGB color model

Computer graphics stated that every colour in the visible spectrum is the result of a combination of different bases colors.

Screen (LCD and CRT) are based on the RGB color model which state that the three base colour are **red**, **green** and **blue**.

The set of all colours one can generates with thee 3 bases builds the RGB Color Cube.

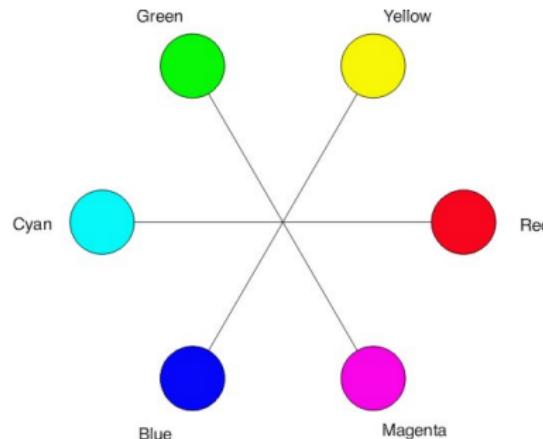


The CMY color model

Printer (for example ink jet printers) use another color model called *subtractive color model* like the CMY model. On a sheet of paper, when for a given pixel every color has its maximal intensity, the printed color is **black**. The three base color for the CMY model are **cyan**, **magenta** and **yellow**.

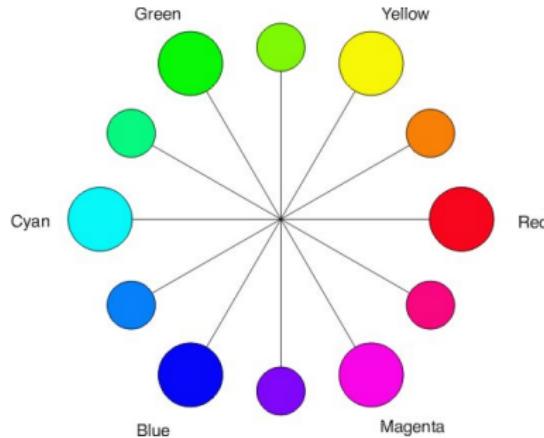
The RGB and the CMY color model

As one can see on the RGB color cube, these two model are complementary, i.e. every combination of two base colour in one model gives one base color in the other model.



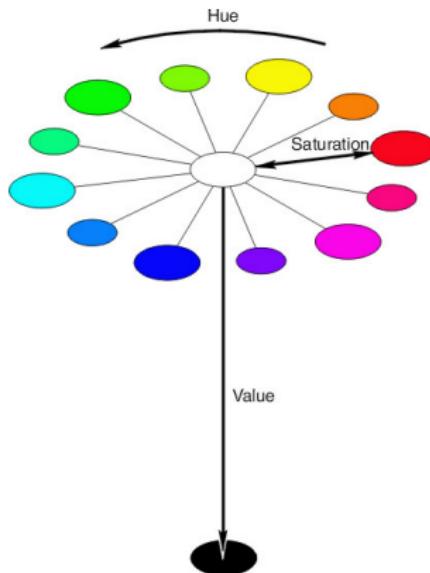
The HSV color model

One can refine the model by adding more combination of the different colors.

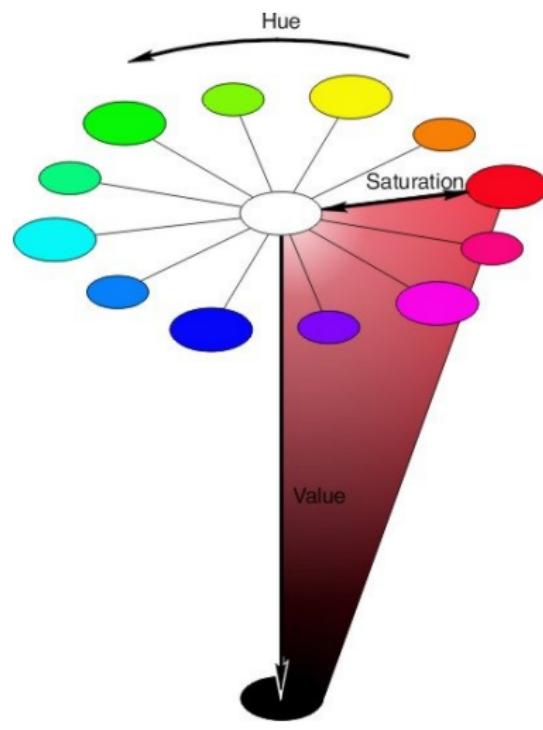


The HSV color mode (cont'd)

The HSV (Hue, Saturation, Value) is another color model used in computer graphics. The main advantage of this model is that if one need to display a gradation of color, one just has to modify one coordinate of the color.

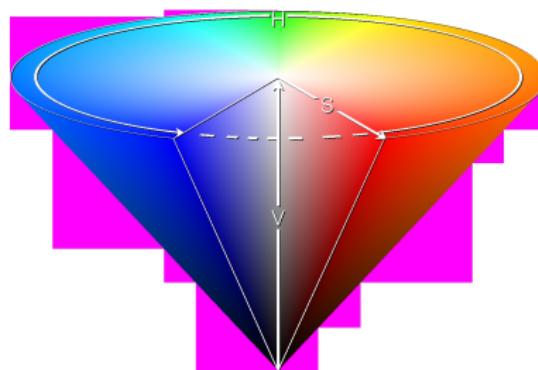


The HSV color mode (cont'd)



The HSV color mode (cont'd)

Another representation of the same cone:



Transformation RGB → HSV

$$H = \begin{cases} \text{undefined} & \text{if } \max = \min \\ 60 \cdot \frac{G - B}{\max - \min} & \text{if } \max = R \text{ and } G \geq B \\ 60 \cdot \frac{G - B}{\max - \min} + 360 & \text{if } \max = R \text{ and } G < B \\ 60 \cdot \frac{B - R}{\max - \min} + 120 & \text{if } \max = G \\ 60 \cdot \frac{R - G}{\max - \min} + 240 & \text{if } \max = B \end{cases}$$

and

$$S = \begin{cases} 0 & \text{if } \max = 0 \\ 1 - \frac{\min}{\max} & \text{otherwise} \end{cases}$$

and $V = \max$ where \max (resp \min) is the max value (resp min)



Transformation HSV → RGB

$$H' = \lfloor H/60 \rfloor \bmod 6 \quad f = (H/60) - H'$$

$$p = V(1 - S) \quad q = V(1 - fS) \quad t = V(1 - (1 - f)S)$$

and then

$$\text{if } H' == 0 \quad R = V \quad G = t \quad B = p$$

$$\text{if } H' == 1 \quad R = q \quad G = V \quad B = p$$

$$\text{if } H' == 2 \quad R = p \quad G = q \quad B = t$$

$$\text{if } H' == 3 \quad R = p \quad G = q \quad B = V$$

$$\text{if } H' == 4 \quad R = t \quad G = p \quad B = V$$

$$\text{if } H' == 5 \quad R = V \quad G = p \quad B = q$$



Outline

The perception of light and color

Color models

Transformations between color models

Color Look-up Table

Illumination models

Diffuse reflection

Specular reflection

Shading models

Flat shading

Gouraud Shading

Phong Shading

Full Illumination equation

The Cook-Torrance model

BRDF

BSDF

The Kajiya equation



Color Look-up Table (CLUT)

Modern PCs are generally capable of dealing with 24-bits color. In brief, that means they can handle 16,777,216 different shades.

A color management system that lets a computer handle a large number of colors without devoting an excessive amount of memory to keeping track of each possible shade.

CLUTs minimize the memory requirement by working on the assumption that while more than 16 million colors may be possible, only a very limited number of those colors are likely to be in use at one time.

Color Look-up Table (CLUT) (cont'd)

Binary code	Lookup Table			Color
000	00	00	00	Black
001	11	00	00	Red
010	11	01	00	Orange
011	11	11	00	Yellow
100	00	11	11	Cyan
101	00	10	01	Green-Blue
110	11	00	11	Magenta
111	11	11	11	White



Outline

The perception of light and color

Color models

Transformations between color models

Color Look-up Table

Illumination models

Diffuse reflection

Specular reflection

Shading models

Flat shading

Gouraud Shading

Phong Shading

Full Illumination equation

The Cook-Torrance model

BRDF

BSDF



The Kajiya equation

Illumination models

- ▶ The color of an object (or more exactly the color of a pixel) is given by the combination of the color of object itself, but also by the color of the light that fall onto this object.
- ▶ To be realistic, an image should take care of different parts of the illumination model like:
 1. ambient light
 2. light emitted by the object
 3. diffuse reflection of the light by the object
 4. specular reflection of the light by the object
- ▶ Some more complicated models add to these the luminescence et the fluorescence characteristic of the object and of the air.



Ambient light

- ▶ This is the simplest model of illumination
- ▶ Ambient light is equal in all direction and in all point of the virtual world.
- ▶ It only has a color and an intensity
- ▶ Java3D has a class `javax.media.j3d.AmbientLight`

Directionnal light

- ▶ Some scene, to be realistics, require more complex model of lights than the simple ambient light
 - ▶ Directionnal lights simulates light sources with non constant behaviour. One can have:
 1. Point light
 2. Cylindrical light
 3. Spot light
 - ▶ Not all API provides all these light models and some defines even more.
 - ▶ Java3D defines `javax.media.j3d.PointLight`, and `javax.media.j3d.SpotLight`. The cylindrical light model is not directly provided but may sometimes be approximated with the `javax.media.j3d.DirectionalLight` class.
- 

Class PointLight of Java3D

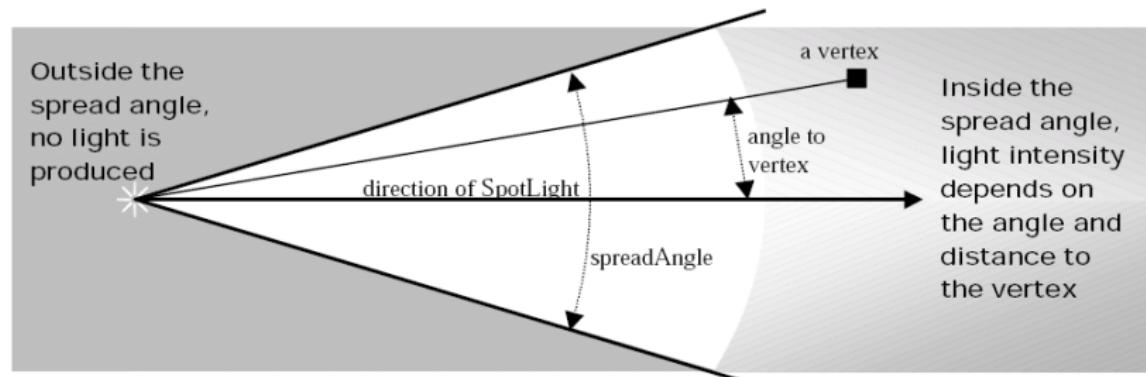
- ▶ The PointLight object specifies an attenuated light source at a fixed point in space that radiates light equally in all directions away from the light source
 - ▶ A point light contributes to diffuse and specular reflections, which in turn depend on the orientation and position of a surface. A point light does not contribute to ambient reflections.
 - ▶ A PointLight is attenuated by multiplying the contribution of the light by an attenuation factor. The attenuation factor causes the the PointLight's brightness to decrease as distance from the light source increase. A PointLight's attenuation factor contains three values:
 1. Constant attenuation
 2. Linear attenuation
 3. Quadratic attenuation
- 
- A set of small, dark blue navigation icons located in the bottom right corner of the slide. From left to right, they include: a left arrow, a right arrow, a double left arrow, a double right arrow, a magnifying glass, and a document icon.

Class SpotLight of Java3D

- ▶ The class `SpotLight` is a subclass of `PointLight`.
- ▶ The `SpotLight` object specifies an attenuated light source at a fixed point in space that radiates light in a specified direction from the light source. A `SpotLight` has the same attributes as a `PointLight` node, with the addition of the following:
 1. *Direction* - The axis of the cone of light. The default direction is $(0.0, 0.0, -1.0)$. The spot light direction is significant only when the spread angle is not π radians (which it is by default).
 2. *Spread angle* - The angle in radians between the direction axis and a ray along the edge of the cone
 3. *Concentration* - Specifies how quickly the light intensity attenuates as a function of the angle of radiation as measured from the direction of radiation.



Class SpotLight (cont'd)



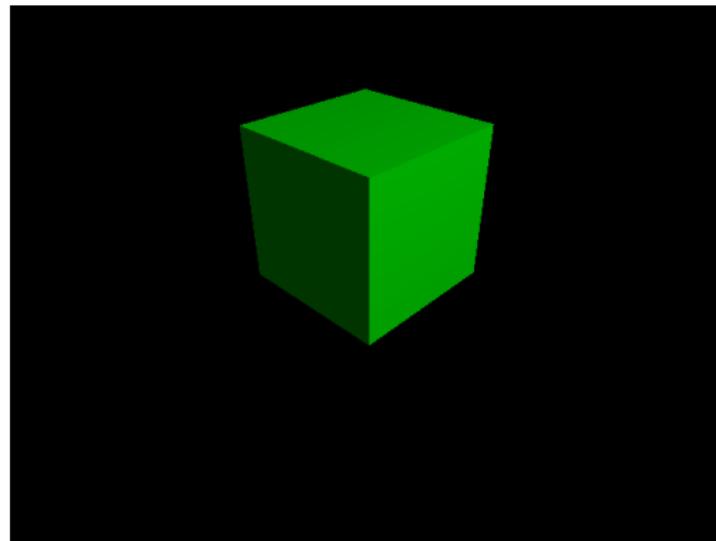
Object visibility

- ▶ An object is only visible if it can reflect a part of the light it receives.
- ▶ When one define the *color* of an object, one define the color of the incident light that object will reflect. A real black object does not reflect anything, a real white object reflect all the light it receives and a colored object reflect only the component of that object with this color.



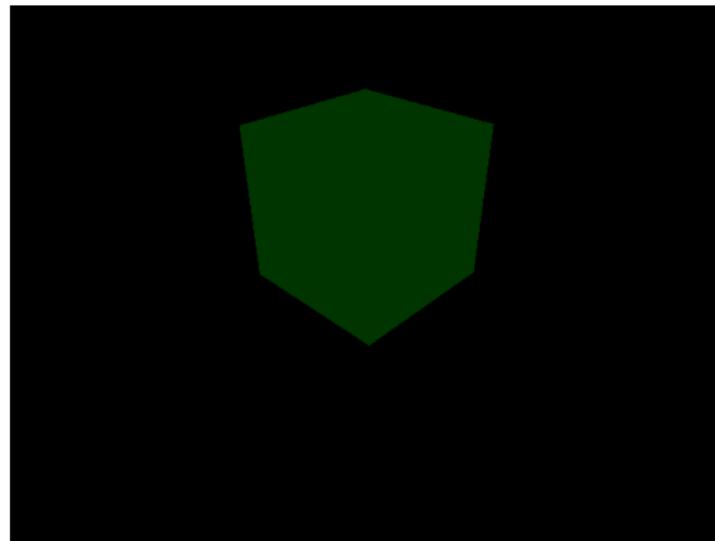
Object visibility : example 1

This example shows a green cube ($0.0; 1.0; 0.0$) illuminated by a white point light and some white ambient light



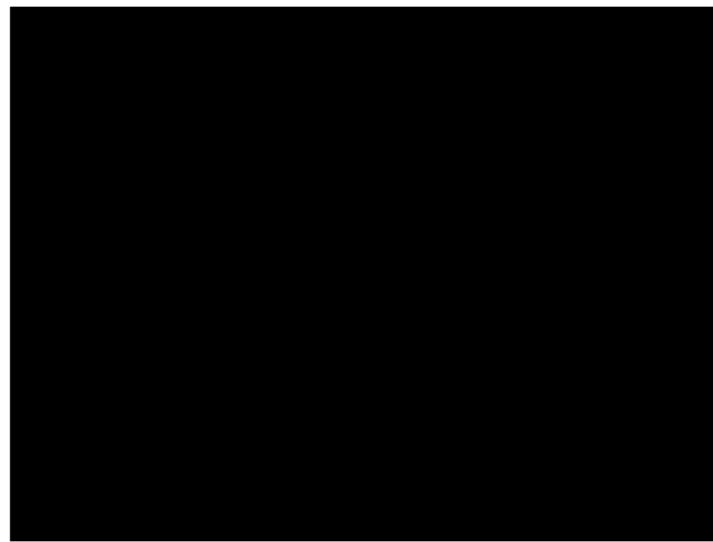
Object visibility : example 2

If one removes the green component of the light source, the object only reflects the ambient light.



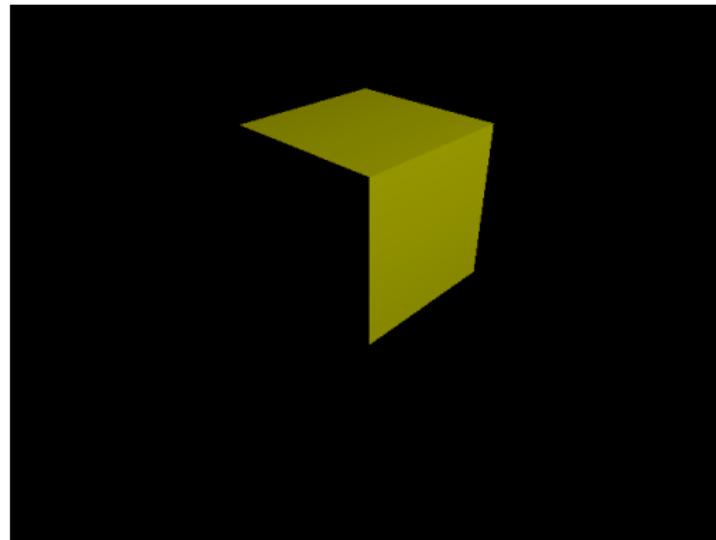
Object visibility : example 3

If one removes the ambient light, the scene appear black.



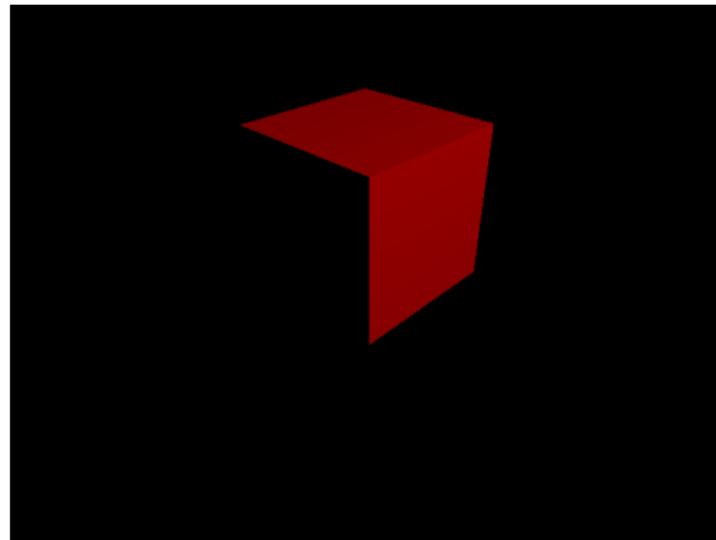
Object visibility : example 4

For this example, there is no ambient light. The color of the object is yellow (1.0; 1.0; 0.0) and the color of the light is white:



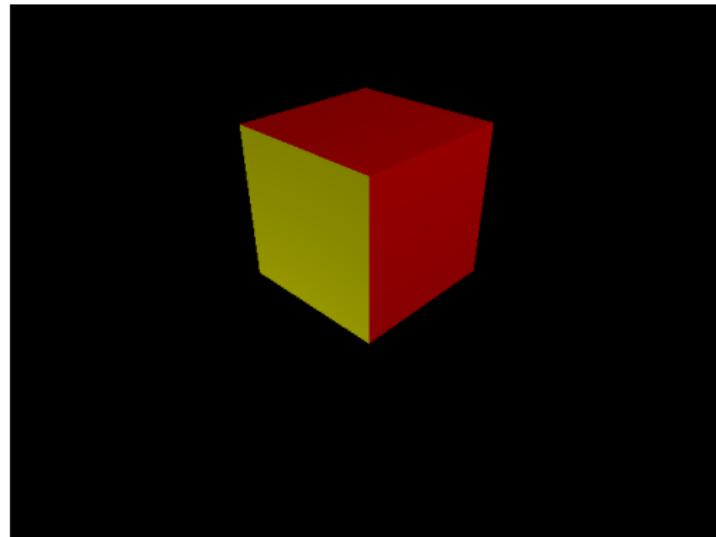
Object visibility : example 5

The object is the same as in example 4, but now, the light is pure red (1.0; 0.0; 0.0);



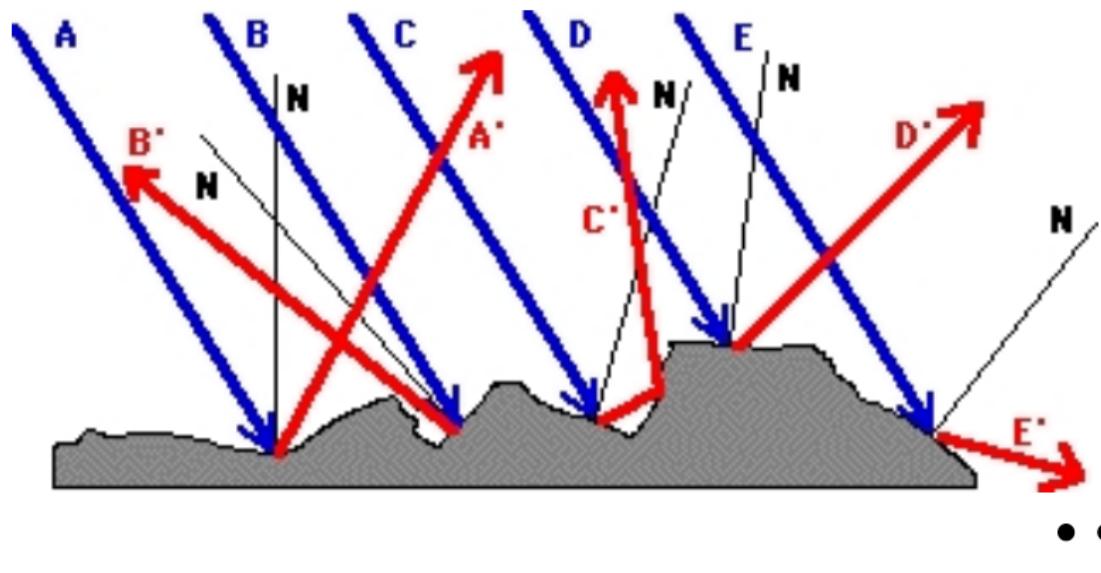
Object visibility : example 6

If one adds a supplementary light source (white) to the previous object, one can simulate different colors on the cube, even if the "original" color is still yellow



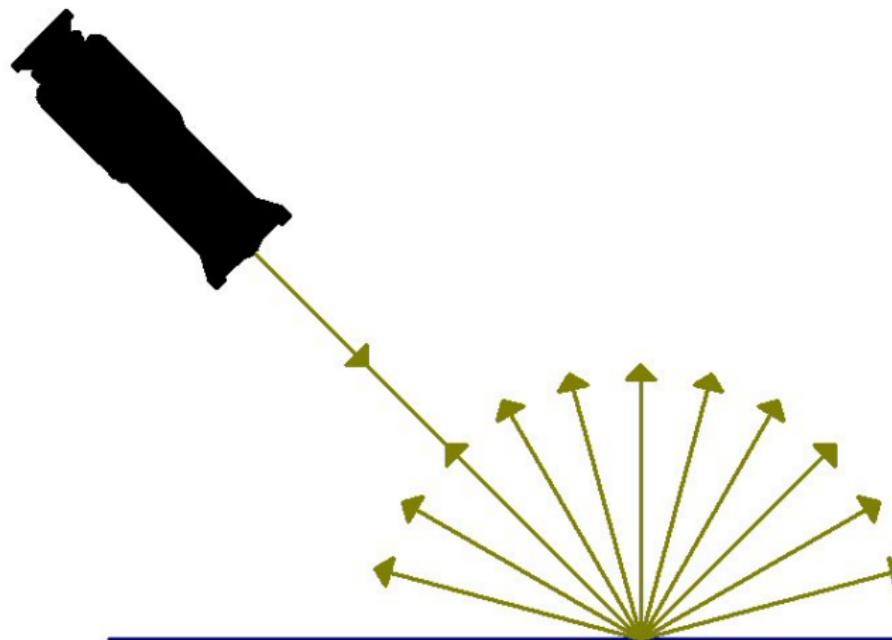
Diffuse reflection

An object which reflects any light source equally in all directions is said to be a *diffuse reflector*.



Diffuse reflection (cont'd)

One can see the diffuse reflection property as something like:



Diffuse reflection (cont'd)

The incident light is equally reflected in all the half sphere. The formula is then:

$$I_d = L_d k_d (\vec{l} \cdot \vec{n})$$

where :

- ▶ I_d is the intensity of the diffuse reflection
- ▶ L_d is the incident light intensity
- ▶ k_d is the diffuse reflection factor ($0 \leq k_d \leq 1$)
- ▶ \vec{l} is the normalized incident light direction
- ▶ \vec{n} is the normalized normal vector at the incident point

This equation should be solved for each color red, green and blue



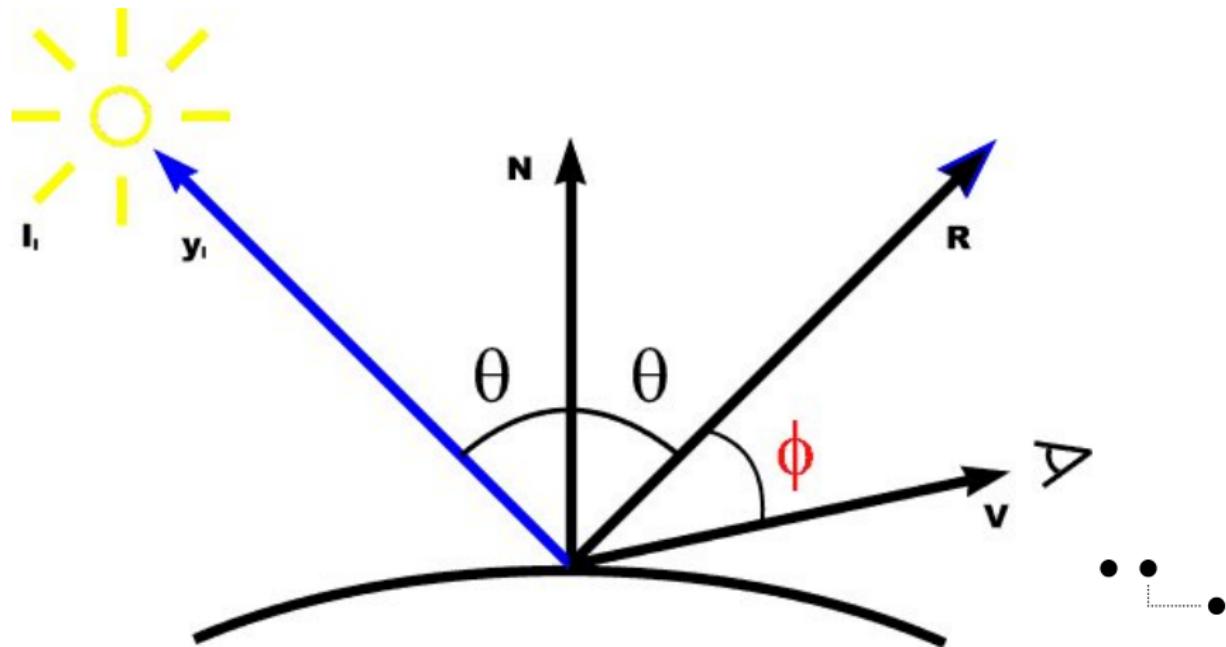
Specular reflection

The specular reflection modelize the mirrors.



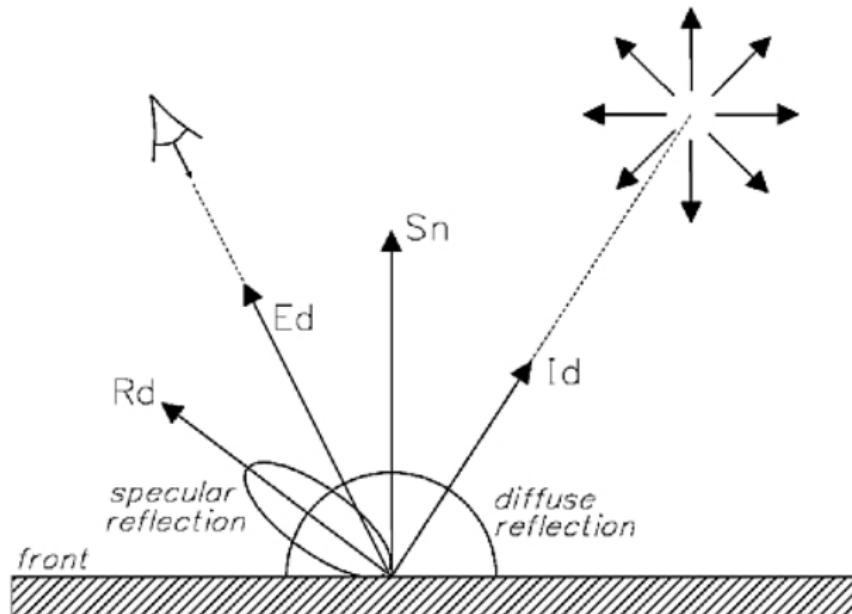
Specular reflection

The pure specular reflection is a *mirror like* reflection that is an incident light ray is totally reflected into a given direction.



Specular reflection (cont'd)

A surface is never a perfect specular reflector. It may only reflect much more light in a direction around the theoretical reflection direction.



Specular reflection (cont'd)

The formula describing the specular reflection may be:

$$I_s = L_s k_s (\vec{R} \cdot \vec{V})^\alpha$$

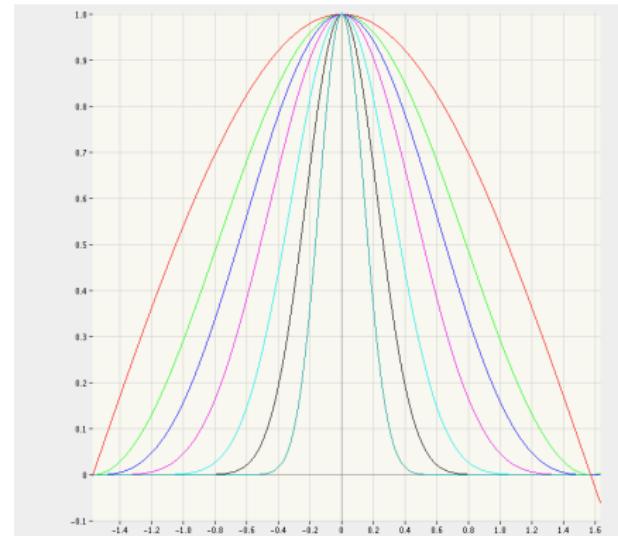
where:

- ▶ L_s intensity of the light source
- ▶ k_s specular reflection coefficient
- ▶ \vec{R} normalised vector for the ideal reflection direction
- ▶ \vec{V} normalised vector toward the observator (or the camera)
- ▶ α shininess

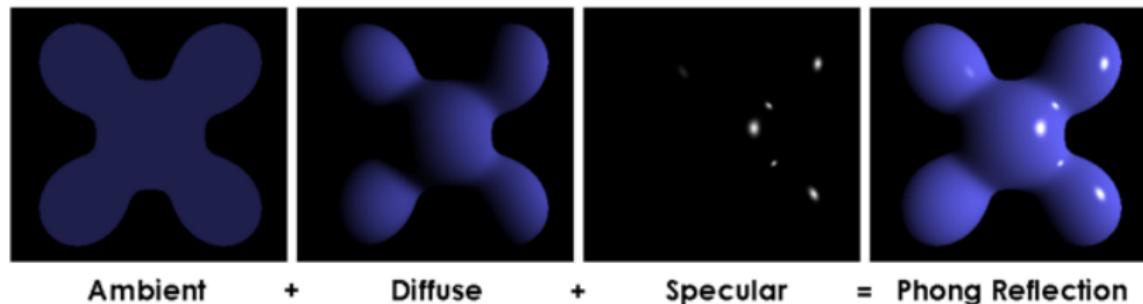


Specular reflection (cont'd)

If the two vectors \vec{R} and \vec{V} are normalised, the dot product returns the cosine of the angle between these two vectors. To simulate a specular reflection, one exponentiate this cosine to a coefficient called shininess. For different values of α , one can have:



Combining all kind of reflection



Combining all kind of reflection

The big formula is now:

$$I_p = k_a(\lambda)I_a(\lambda) + \sum_{\text{light sources}} I_i(\lambda) \left\{ k_d(\lambda)(\vec{L} \cdot \vec{N}) + k_s(\lambda)(\vec{R} \cdot \vec{V})^\alpha \right\}$$



Complete illumination model

When only considering the ambient light:



Complete illumination model

Adding some diffuse reflection:



Complete illumination model

And the with the specular reflection:



Outline

The perception of light and color

Color models

 Transformations between color models

Color Look-up Table

Illumination models

 Diffuse reflection

 Specular reflection

Shading models

 Flat shading

 Gouraud Shading

 Phong Shading

Full Illumination equation

The Cook-Torrance model

 BRDF

 BSDF

The Kajiya equation



Shading

In computer graphics, the word *shading* describes different algorithms used to describe the surface rendering, i.e. the light-reflection behaviour of surfaces.

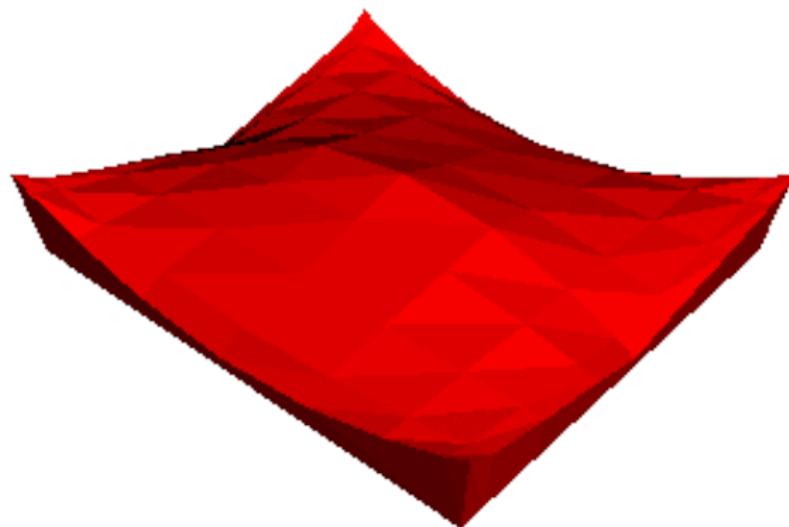
This behaviour is based on the illumination equation seen before and some interpolation techniques to simulate the "smoothness" of surfaces.



Flat shading

- ▶ Flat shading is the simplest and the fastest method to illuminate a polyhedra.
- ▶ Each face is drawn with a single color (computed using the normal of the face)
- ▶ This shading model may produce *Mach band*, i.e. color discontinuities along the edges
- ▶ This shading model is declared by the attribute `SHADE_FLAT` defined in the class `ColoringAttributes` in Java3D.
- ▶ This shading is declared by `glShadeModel(GL_FLAT)` in OpenGL

Flat shading : an example



Gouraud Shading I

- ▶ For the Gouraud shading model, one use the normal of all faces that meet at a given vertex.
- ▶ One compute an "average normal vector" at a given vertex, which is defined as:

$$\vec{n}_{v_i} = \frac{\sum_{k=1}^N \vec{n}_k}{\|\sum_{k=1}^N \vec{n}_k\|}$$

If all normal vectors are normalised, this equation can be rewritten as:

$$\vec{n}_{v_i} = \frac{1}{N} \sum_{k=1}^N \vec{n}_k$$

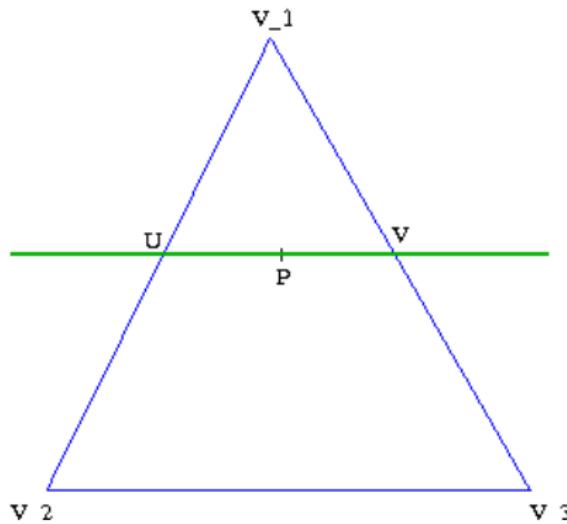


Gouraud Shading II

- ▶ Using this average normal vector, one can compute the "illumination intensity" (i.e. the color) of every vertex of a given face.
- ▶ The color of the pixels composing the faces are computed using a bilinear interpolation of the vertices color.



Gouraud Shading : example



$$I_U = \alpha I_{V_1} + (1 - \alpha) I_{V_2}$$

$$I_V = \beta I_{V_1} + (1 - \beta) I_{V_3}$$

$$I_P = \gamma I_U + (1 - \gamma) I_V$$

with

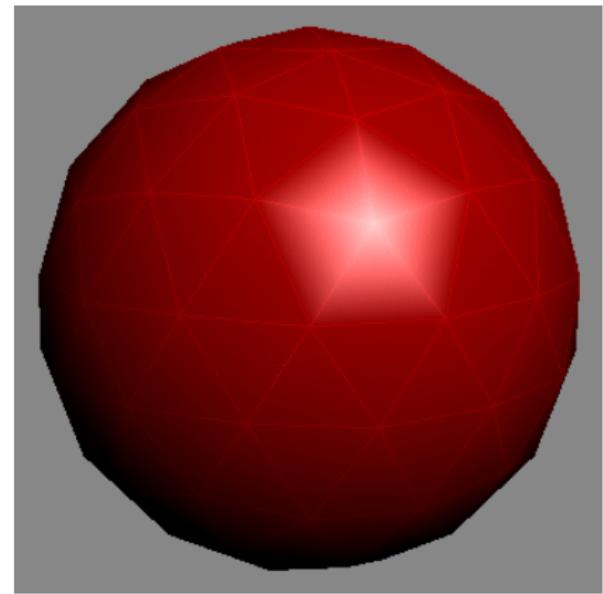
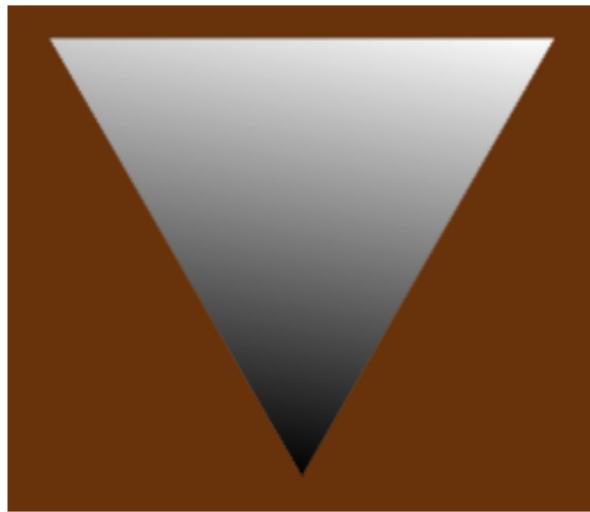
$$\alpha = \frac{|\overline{UV_2}|}{|\overline{V_1V_2}|}$$

$$\beta = \frac{|\overline{UV_3}|}{|\overline{V_1V_3}|}$$

$$\gamma = \frac{|\overline{UP}|}{|\overline{UV}|}$$



Gouraud Shading : example

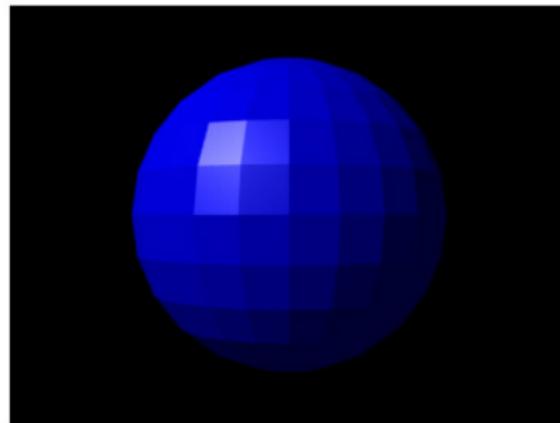


Phong Shading

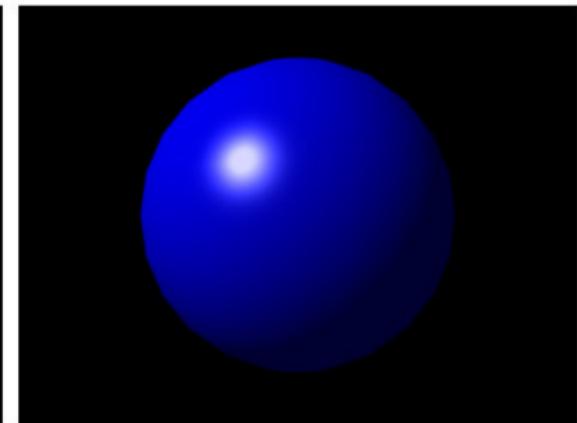
- ▶ As for Gouraud shading, Phong shading starts to compute an average normal vector for each vertex of the face.
- ▶ For each pixel in the face, one should bilinearly interpolate these normal vector to find a "pseudo-normal" vector. Computing these interpolated vector for *each pixel* means that one should work over the projected area of the face.
- ▶ For each pixel along a scan line, one should apply the illumination model to calculate the pixel intensity.



Phong Shading : example



FLAT SHADING



PHONG SHADING

Phong Shading (cont'd)

The normal vectors are the same as those used for the intensities values in the Gouraud method.

Given the normal vectors for two vertices, the interpolated vector may be computed as:

$$\vec{n}_p = \frac{V - V_2}{V_1 - V_2} \vec{n}_1 + \frac{V_1 - V}{V_1 - V_2} \vec{n}_2$$



Outline

The perception of light and color

Color models

 Transformations between color models

Color Look-up Table

Illumination models

 Diffuse reflection

 Specular reflection

Shading models

 Flat shading

 Gouraud Shading

 Phong Shading

Full Illumination equation

The Cook-Torrance model

 BRDF

 BSDF

The Kajiya equation



Full illumination equation

$$\begin{aligned}
 L(\vec{r}, \vec{\omega}, \lambda, \vec{e}, t) = & \mu(\vec{r}, \vec{s}) \left[L^e(\vec{s}, \vec{\omega}, t, \lambda) \right. \\
 & + m_p(\vec{\omega}) \int_{-\infty}^t d(t - \tau) P_p(\vec{s}, \lambda) \int_{\Theta_i^i} L(s, \vec{\omega}', \lambda, \vec{e}, \tau) \cos \theta' d\vec{\omega}' d\tau \\
 & + \int_{\Theta_i^i} f(\vec{s}, \lambda, \vec{\omega}' \rightarrow \vec{\omega}) \int_{\mathcal{R}_\nu} P_f(\vec{s}, \lambda' \rightarrow \lambda) L(\vec{s}, \vec{\omega}', \lambda', \vec{e}, t) d\lambda' \cos \theta' \\
 & + \int_0^{h(\vec{r}, \vec{\omega})} \mu(\vec{r}, \vec{a}) \left[L^e(\vec{a}, \vec{\omega}, t, \lambda) \right. \\
 & + m_p(\vec{\omega}) \int_{-\infty}^t d(t - \tau) P_p(\vec{a}, \lambda) \int_{\Theta_i^i} L(s, \vec{\omega}', \lambda, \vec{e}, \tau) \cos \theta' d\vec{\omega}' d\tau \\
 & \left. \left. + \int f(\vec{a}, \lambda, \vec{\omega}' \rightarrow \vec{\omega}) \int P_f(\vec{a}, \lambda' \rightarrow \lambda) L(\vec{a}, \vec{\omega}', \lambda', \vec{e}, t) d\lambda' \right] \right. \\
 & \left. \left. \cdots \cdots \cdots \right] \right]
 \end{aligned}$$

Outline

The perception of light and color

Color models

 Transformations between color models

Color Look-up Table

Illumination models

 Diffuse reflection

 Specular reflection

Shading models

 Flat shading

 Gouraud Shading

 Phong Shading

Full Illumination equation

The Cook-Torrance model

 BRDF

 BSDF

The Kajiya equation



The BSDF

- ▶ The full illumination equation is far too complex to be used in most simulation.
- ▶ Several simplifications of this model have been explored to enhance the rendering given by the Phong model, but still having an acceptable complexity (in time as in memory).
- ▶ One of these model is the BSDF (Bidirectional Scattering Distribution Function)
- ▶ The BSDF is a superset (and a generalisation) the the BRDF and the BTDF.

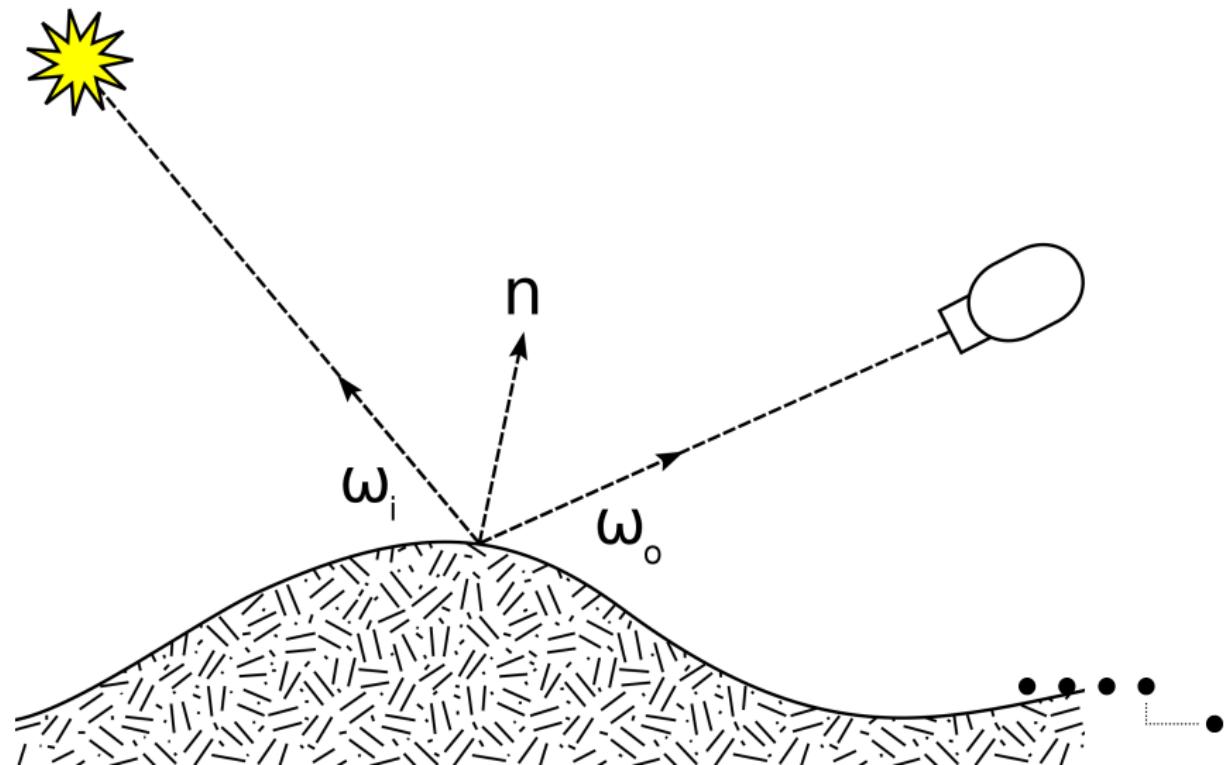


The Bidirectional Reflection Distribution Function

- ▶ The bidirectional reflectance distribution function is a four-dimensional function that defines how light is reflected at an opaque surface.
- ▶ The function takes an incoming light direction, ω_i , and outgoing direction, ω_o , both defined with respect to the surface normal \vec{n} , and returns the ratio of reflected radiance exiting along ω_o to the irradiance incident on the surface from direction ω_i .
- ▶ Each direction ω is itself parameterized by azimuth angle ϕ and zenith angle θ , therefore the BRDF as a whole is 4-dimensional



The Bidirectional Reflection Distribution Function



The Bidirectional Reflection Distribution Function

The BRDF was first defined by Fred Nicodemus around 1965. The modern definition is:

$$f_r(\omega_i, \omega_o) = \frac{dL_r(\omega_o)}{dE_i(\omega_i)} = \frac{dL_r(\omega_o)}{L_i(\omega_i) \cos \theta_i d\omega_i}$$

where L is the radiance, E the irradiance and θ_i is the angle between the incoming ray ω_i and the normal n .

- ▶ The radiance is the radiometric measures that describe the amount of radiation such as light or radiant heat that passes through or is emitted from a particular area, and falls within a given solid angle in a specified direction.
 - ▶ The irradiance is the power of electromagnetic radiation per unit area (radiative flux) incident on a surface. Radiant emittance or radiant exitance is the power per unit area radiated by a surface.
- 

Physically based BRDFs

- ▶ Physically based BRDFs have additional properties:
 1. positivity: $f_r(\omega_i, \omega_o) \geq 0$
 2. obeying Helmholtz reciprocity: $f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i)$.
 3. conserving energy:

$$\forall \omega_i, \int_{\Omega} f_r(\omega_i, \omega_o) \cos \theta_o d\omega_o \leq 1$$

- ▶ In all these cases, the dependence on wavelength has been ignored and binned into RGB channels. In reality, the BRDF is wavelength dependent, and to account for effects such as iridescence or luminescence the dependence on wavelength must be made explicit: $f_r(\lambda_i, \omega_i, \lambda_o, \omega_o)$

Models

BRDFs can be measured directly from real objects using calibrated cameras and lightsources; however, many phenomenological and analytic models have been proposed including the Lambertian reflectance model frequently assumed in computer graphics. Some useful features of recent models include:

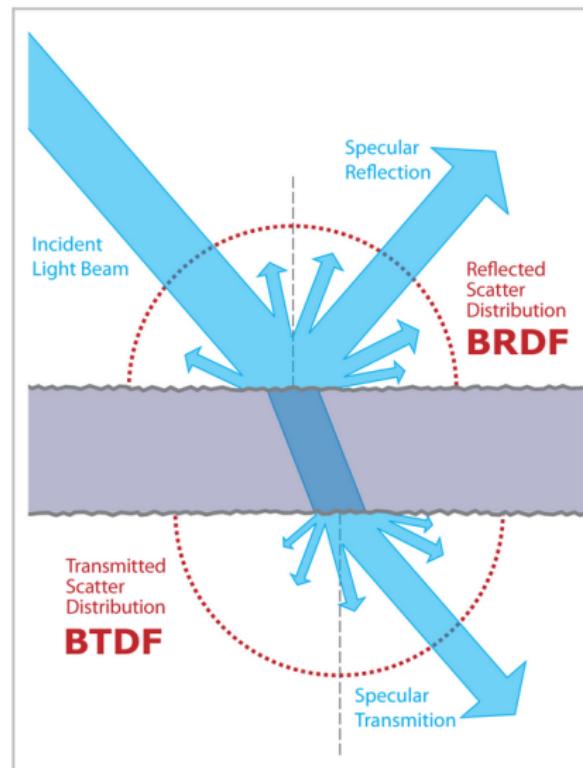
1. accommodating anisotropic reflection
2. editable using a small number of intuitive parameters
3. accounting for Fresnel effects at grazing angles
4. being well-suited to Monte Carlo methods.



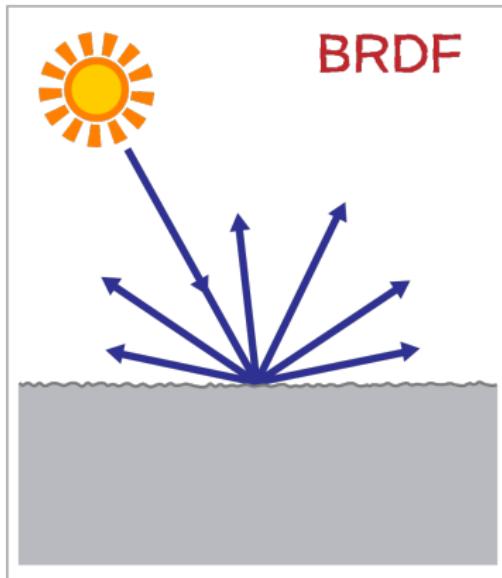
The Bidirectional Scattering Distribution Function

The definition of the BSDF (Bidirectional scattering distribution function) is not well standardized. The term was probably introduced in 1991 by Paul Heckbert[1]. Most often it is used to name the general mathematical function which describes the way in which the light is scattered by a surface. However in practice this phenomenon is usually split into the reflected and transmitted components, which are then treated separately as BRDF (Bidirectional reflectance distribution function) and BTDF (Bidirectional transmittance distribution function).

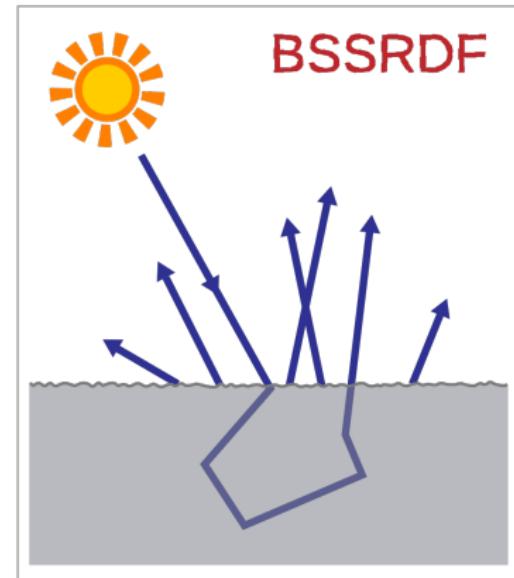
The Bidirectional Scattering Distribution Function



Subsurface scattering



BRDF



BSSRDF

Outline

The perception of light and color

Color models

 Transformations between color models

Color Look-up Table

Illumination models

 Diffuse reflection

 Specular reflection

Shading models

 Flat shading

 Gouraud Shading

 Phong Shading

Full Illumination equation

The Cook-Torrance model

 BRDF

 BSDF



The Kajiya equation

The Kajiya equation

James Kajiya proposed, in 1986, a simplified version of the *Global Illumination Equation*. This equation is:

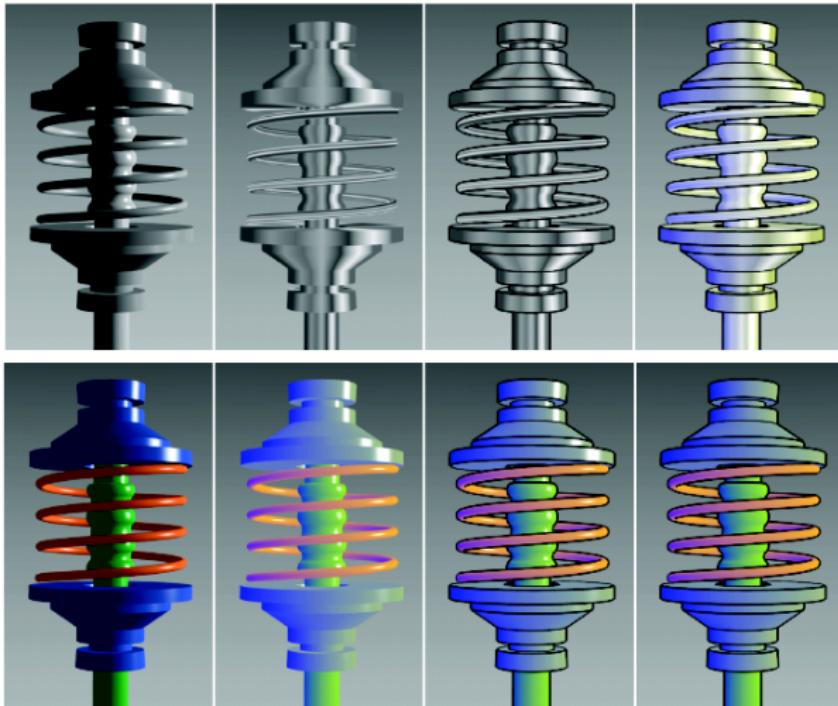
$$L_o(x, \omega, \lambda, t) = L_e(x, \omega, \lambda, t) + \int_{\Omega} f_r(x, \omega', \omega, \lambda, t) L_i(x, \omega', \lambda, t) (-\omega' \cdot \mathbf{n}) d\omega'$$

The physical basis for the rendering equation is the law of conservation of energy. Assuming that L denotes radiance, we have that at each particular position and direction, the outgoing light (L_o) is the sum of the emitted light (L_e) and the reflected light. The reflected light itself is the sum of the incoming light (L_i) from all directions, multiplied by the surface reflection and cosine of the incident angle.



Non photorealistic rendering

Sometimes, photorealism is not the best way to display information



Part VIII

The virtual camera model



Outline

The camera model

Parallel projection

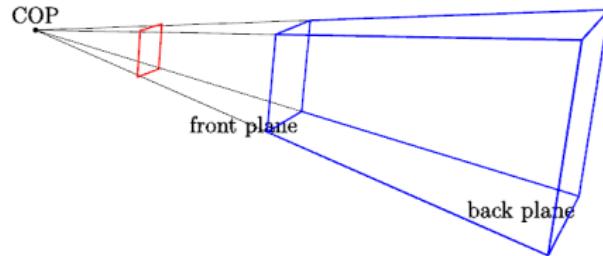
Perspective projection

Anamorphosis



The camera model

Computer graphics use the "pin hole" camera model. This camera does not have optical parts (no lenses).



The camera model

A virtual camera is basically defined by 4 parameters:

1. Its position in space (vector 3D)
2. Its "lookAt" direction (vector 3D)
3. Its orientation (vector 3D)
4. Its view angle (real number, normally only half angle, or specified as a viewport)
5. A near and far plane to apply transformations only on objects that can be viewed.
6. Near and far plane, together with the viewport build the frustum



The camera model with OpenGL

OpenGL defines the camera using the `glLookAt()` function, which have the prototype:

```
void gluLookAt( GLdouble eyeX,  
                GLdouble eyeY,  
                GLdouble eyeZ,  
                GLdouble centerX,  
                GLdouble centerY,  
                GLdouble centerZ,  
                GLdouble upX,  
                GLdouble upY,  
                GLdouble upZ
```



The camera model with Open GL (cont'd)

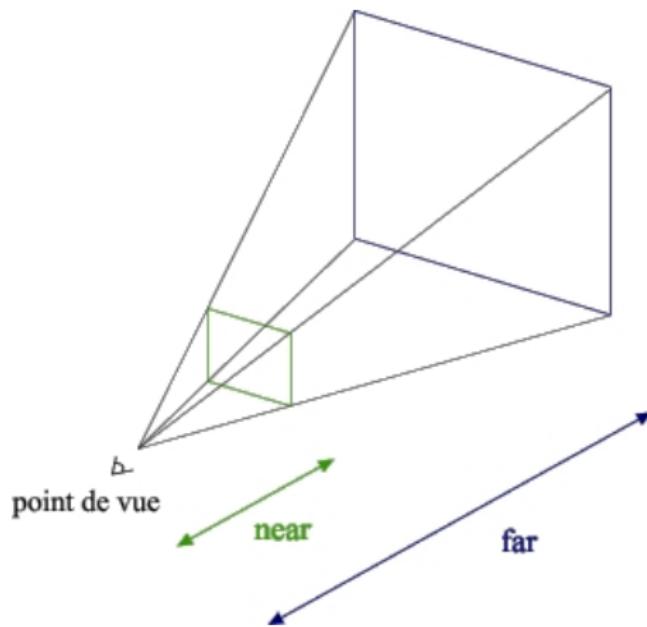
The angle of vision is defined using the function `glViewPort()`:

```
void glViewport( GLint x,  
                  GLint y,  
                  GLsizei width,  
                  GLsizei height )
```

where `x` and `y` specify the lower left corner of the viewport rectangle, in pixels (the initial value is `(0,0)`) and `width` and `height` specify the width and height of the viewport.



The camera model with OpenGL (cont'd)



Three dimensionnal viewing

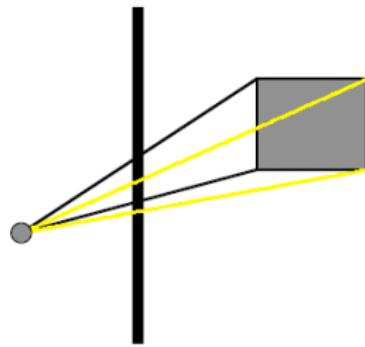
Unlike real camera picture, the virtual camera allows to choose different methods for projecting a scene onto a view plane.

parrallel projection : the object points are projected along parrallel lines, perpendicular to the projection plane

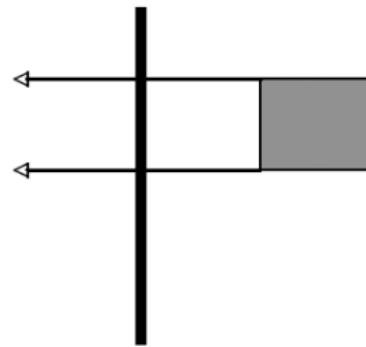
perspective projection Projects the points on the view plane along converging lines.



Three dimensionnal viewing (cont'd)

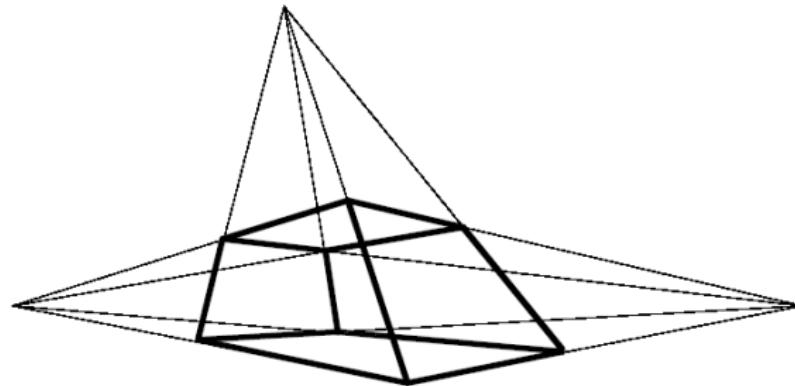
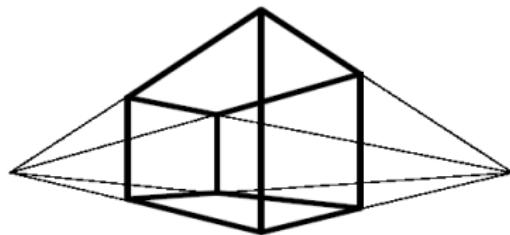
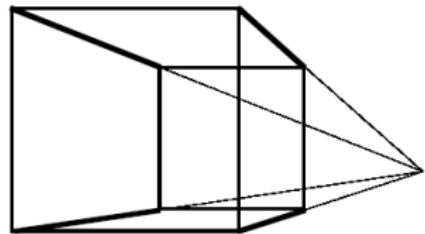


Perspective projection



Parallel projection

Perspective View



Perspective View (cont'd)



Parrallel projection

All the perspective lines are parrallel but not necessarily perpendicular to the projection plane

One define three types of parrallel projections:

1. orthographic
2. axonometric
3. oblique



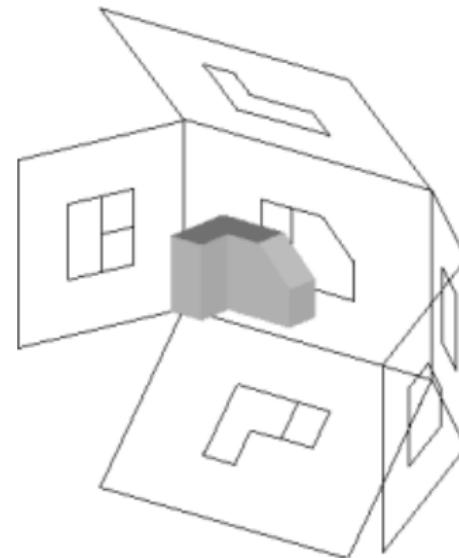
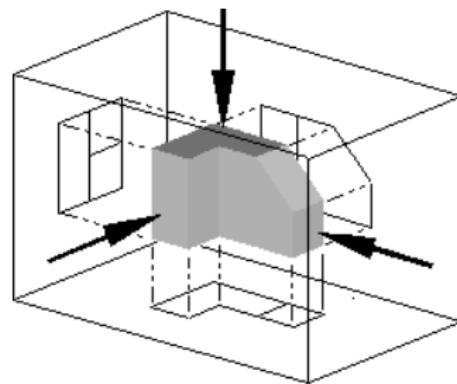
Orthographic projection

From the greek *ορθός* which mean "correct" and *γραφος* which mean "that write".

The principle is to imagine a box around the object to be projected and to project this object "flat" on each of the six sides of the box



Orthographic projection (cont'd)



Orthographic projection (cont'd)

If one side of the box is the xy -plane, then the point $P = (x, y, z)$ is projected on this side by removing the z coordinate, i.e.

$P^* = \mathbf{T}_z \cdot P$ where:

$$\mathbf{T}_z = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

respectively

$$\mathbf{T}_x = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \mathbf{T}_y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



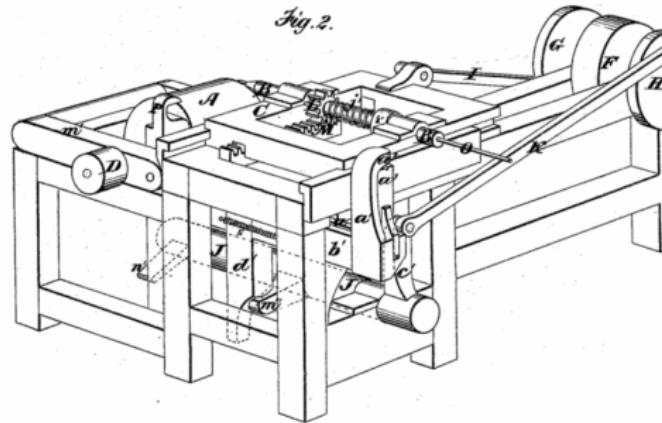
Axonometric projections

Orthographic projection of an object shows the details of only one face of this object. Interpreting orthographic projection require experience and may be confused

Axonometric projections show more to the object in each projection, but at the price of having wrong dimensions and angle. An axonometric projection typically shows three faces (or more) of the object, but it shrinks some of the dimensions



Axonometric projections (cont'd)

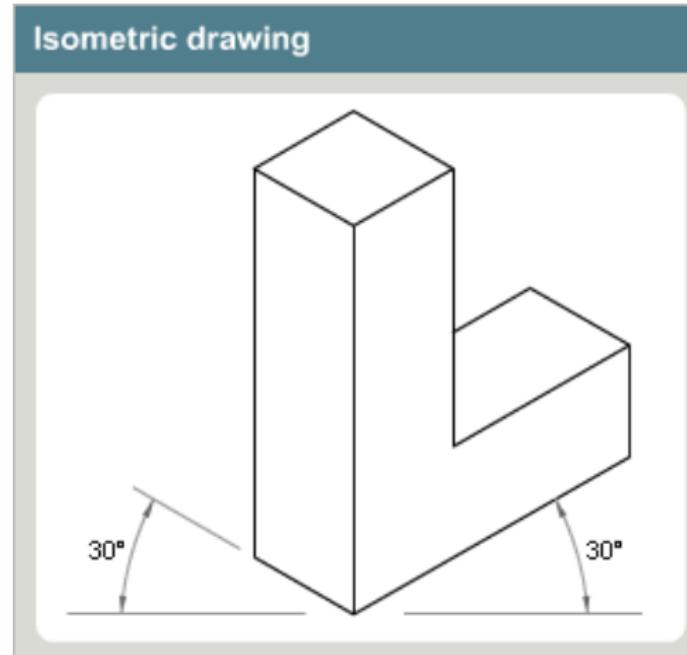


Properties of the axonometric projection

- ▶ Axonometric projections are parallel, so a group of parallel lines appear parallel in the projection
- ▶ There are no vanishing point. For a wide image, at every point the viewer has the same perspective
- ▶ Distant objects retains their size regardless of their distance from the observer

Isometric projection

Isometric projection is one of the multiple kinds of axonometric projection.



Outline

The camera model

Parallel projection

Perspective projection

Anamorphosis



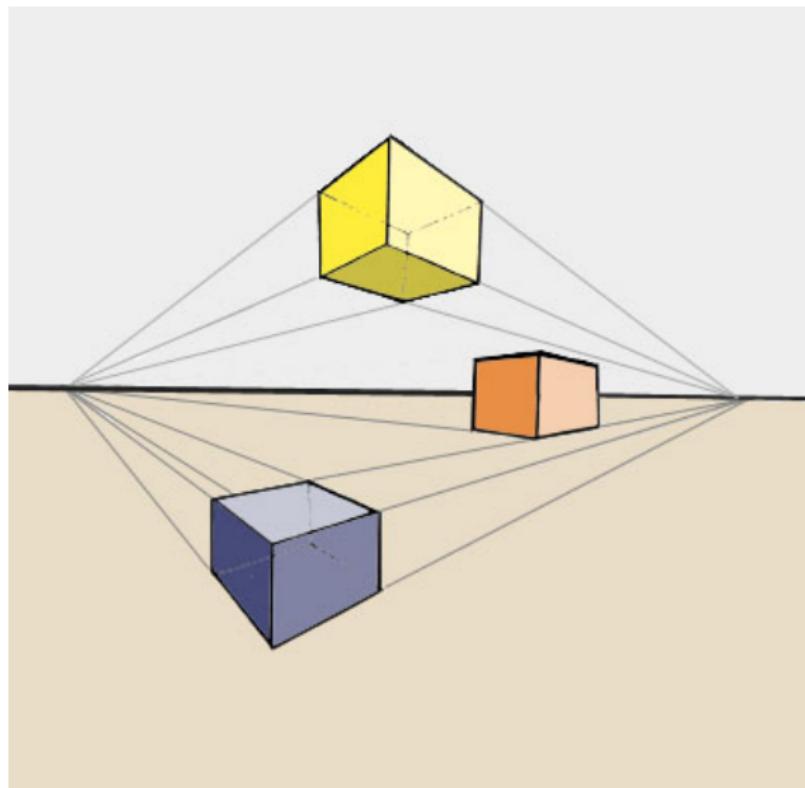
Perspective projection

The term of *perspective* refer to several techniques to create the illusion of depth.

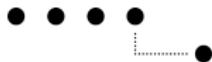
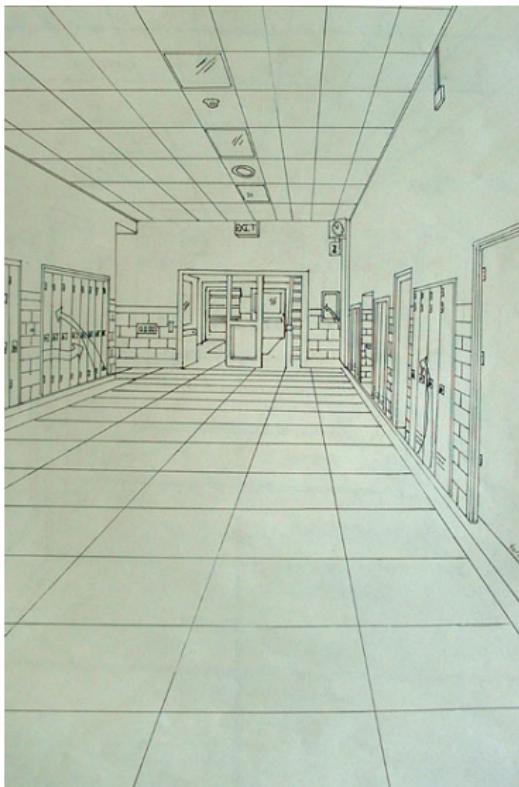
The main tool employed by linear perspective is *vanishing points*, i.e. the illusion of depth is made by *converging lines* which parallel lines in the real model no more parallel in the drawing.



Perspective projection (cont'd)



Perspective projection (cont'd)



Perspective projection step by step

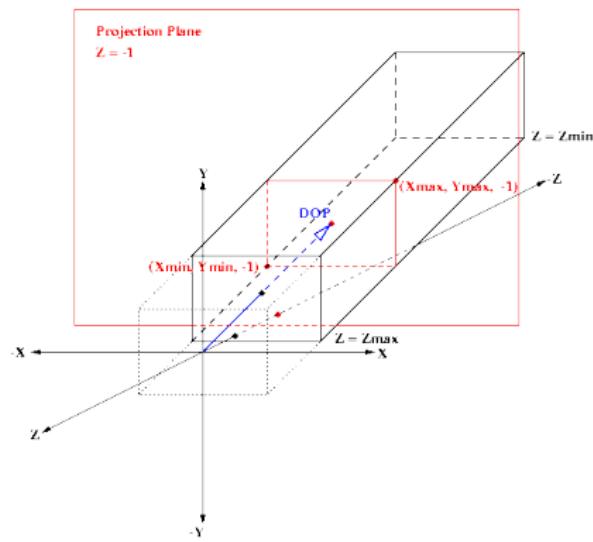
1. Translate the View Reference Point (VRP) to the origin.
2. Rotate the View Reference Coordinate (VRC) such as the View Plane Normal (VPN) becomes the z -axis, the u -axis become the x -axis and the v -axis becomes the y -axis
3. Translate such that the center of projection (COP) given by the Projection Reference Point (PRP) is at the origin
4. Shear such that the centerline of the view volume becomes the z -axis
5. Scale such that the view volume becomes the canonical perspective view volume, i.e. the truncated pyramid.



Perspective projection (cont'd)

For parallel projection, the view frustum is a box

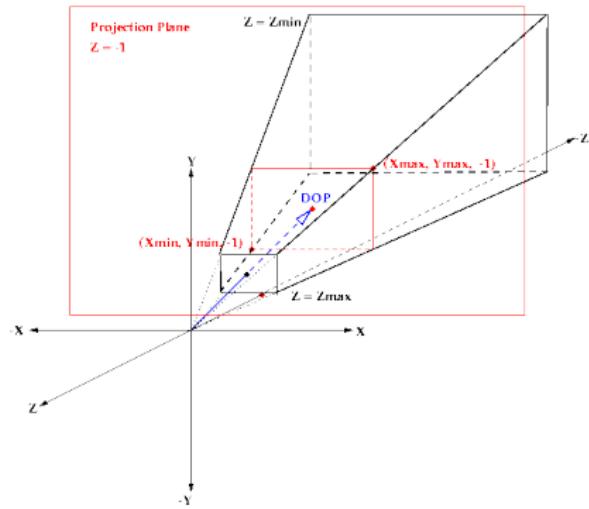
Parallel Projection Frustum



Perspective projection (cont'd)

For perspective projection, the view frustum is a truncated pyramid

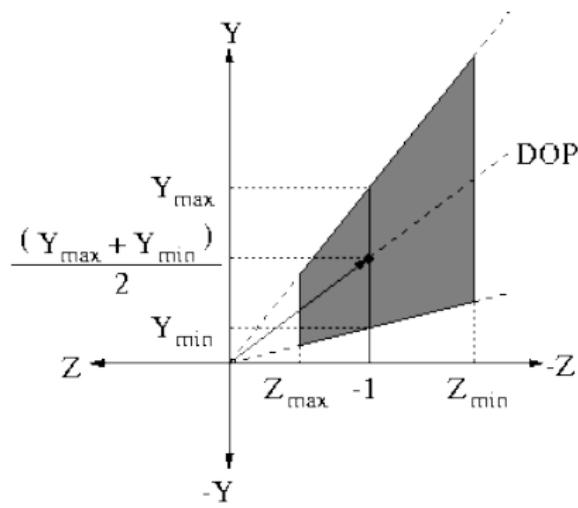
Perspective Projection Frustum



Perspective projection (cont'd)

Steps 1 to 3 set the perspective frustum coordinates into the system coordinates

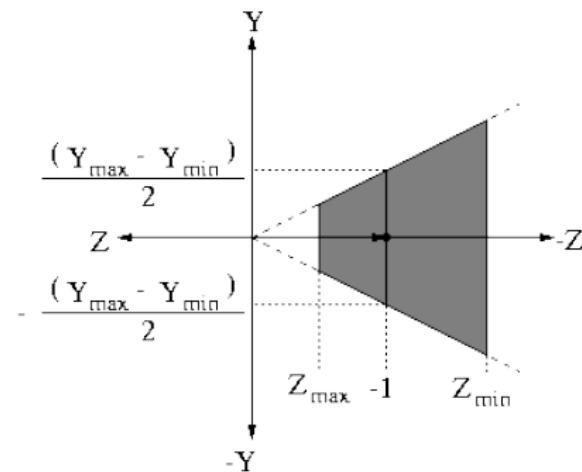
Perspective Frustum in VRC



Perspective projection (cont'd)

Step 4 sets the Direction Of Projection (DOP) to the z -axis

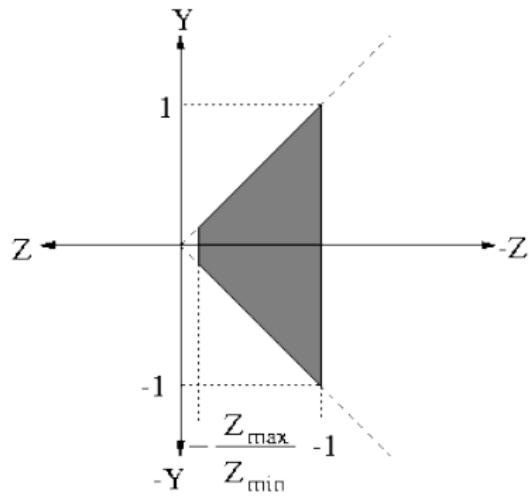
Shear the DOP to the $-Z$ Axis



Perspective projection (cont'd)

Step 5 scales the view frustum to normalize it

Normalized Frustum



Outline

The camera model

Parallel projection

Perspective projection

Anamorphosis



Anamorphosis



Anamorphosis



Anamorphosis



Anamorphosis



Anamorphosis



More examples on

<http://users.skynet.be/J.Bever/pave.htm>

Part IX

Hidden Surface Removal



Outline

Hidden Surface Removal

Generality

Classification of algorithms

Backface culling

Painter algorithm's

The Z-Buffer

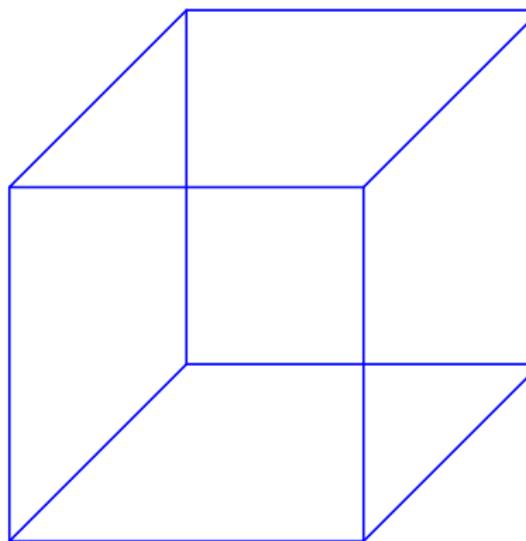
The Warnock Algorithm

BSP trees

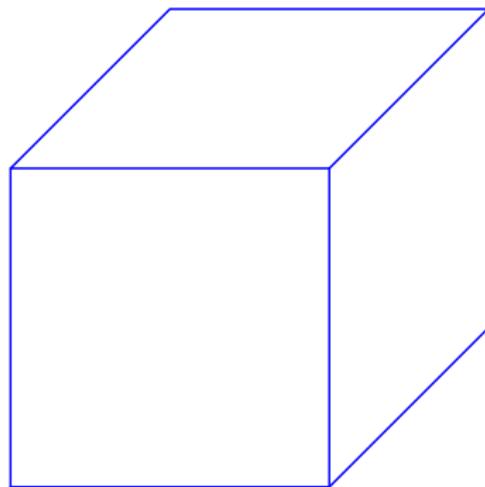


How to draw a cube

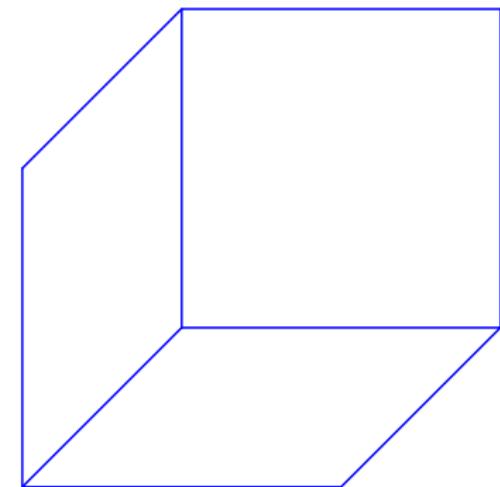
We have a simple cube, given by its vertices-faces description.



Interpretation ?



OR



Hidden Surface Removal

In 3D computer graphics, hidden surface determination is the process used to determine which surfaces and parts of surfaces are not visible from a certain viewpoint. A hidden surface determination algorithm is a solution to the visibility problem, which was one of the first major problems in the field of 3D computer graphics. The process of hidden surface determination is sometimes called hiding, and such an algorithm is sometimes called a hider. The analogue for line rendering is hidden line removal.



Classification of Hidden Surface Removal

The literature generally classifies the algorithms for hidden surfaces removal into three categories:

- ▶ Object precision algorithms (that is algorithms working mainly into the geometrical space)
- ▶ Image precision algorithms (that is algorithms working mainly into the image space)
- ▶ List priority algorithms (different algorithms, some working in the two spaces)



Object precision algorithms

Roughly, one may say that object precision algorithms only use the geometry of the objects and the position of the camera to do their jobs. One may sketch them as:

```
for all object  $O$  in the world do
    Find part  $A$  of  $O$  that is visible
    Display  $A$  appropriately
end for
```



Image precision algorithms

Image precision algorithms use the rasterization of the image to find visible and hidden surfaces. One may sketch these algorithms as:

for all pixel on the screen **do**

Determine the first visible object that is pierced by the ray from the viewer determined by the pixel

if there is such object O **then**

display the pixel with the color of the object O

else

display the pixel in the background color

end if

end for



List priority algorithms

List priority algorithms fall somewhere in between object and image precision algorithms. They differ from pure image precision algorithms in that they precompute, in object space, a visibility ordering **before** scan converting objects to image space in a simple back to front order.

Hidden Surface Removal (cont'd)

One can define four types of hidden surfaces:

1. Backfaces
2. Outside the view frustum
3. Occluded surfaces, either by the object itself or by another object
4. Insufficient contribution (i.e. surfaces that are too far away)



Where to search hidden faces

- ▶ Hidden faces may be searched into the object space (the virtual world) as in the image space (pixels).
- ▶ If working into the object space, one should be able to find hidden faces knowing only the position of the viewer
- ▶ If working into the image space, one should decide of the visibility at every pixel position



Object space hidden faces

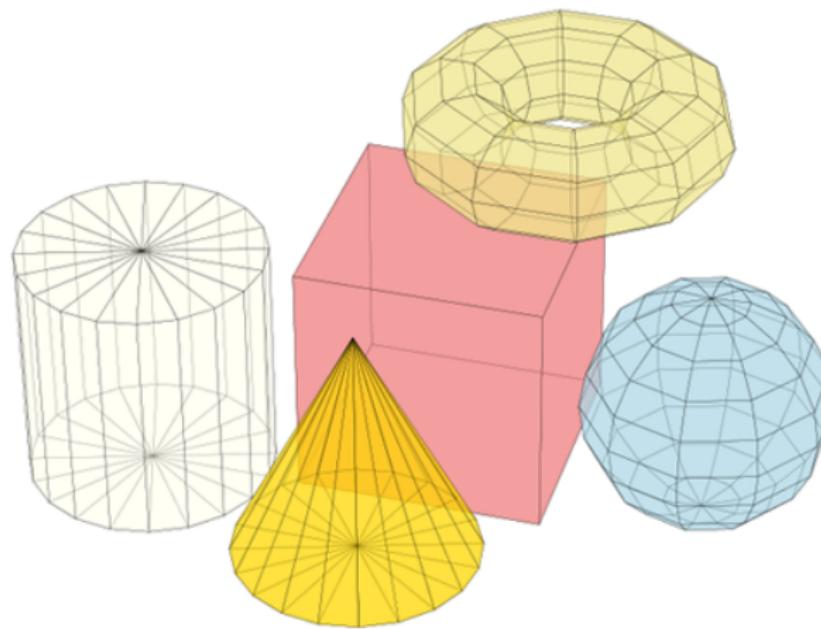
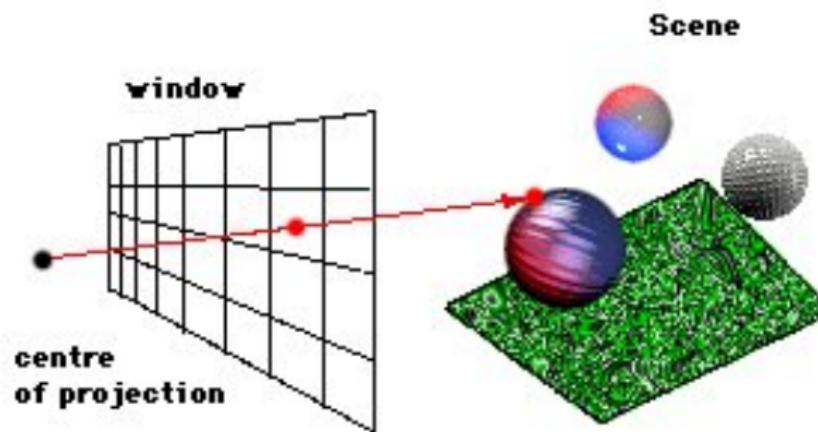


Image space hidden faces



Complexity of visible surface determination

- ▶ Image space algorithms
 - ▶ Complexity $O(\text{pixels} \times \text{objects})$.
 - ▶ For example : screen resolution 1286×1024 and 1 million polygons.
 - ▶ Complexity = $O(1.3 \cdot 10^{12})$
- ▶ Object space algorithms
 - ▶ Worstcase : compare n object with $n - 1$ objects
 - ▶ Complexity : $O(n^2)$.
 - ▶ For example : 1 million polygon require 10^{12} comparisons



Reducing the complexity

The computation complexity may be reduced if one exploit the coherence of the image.

- ▶ Object coherence : if the objects are well separated, compare the objects, not the faces
- ▶ Face coherence : if faces vary smoothly, one can modify the face incrementally
- ▶ Edge coherence : edge visibility only change if it crosses another edge or face
- ▶ Scan-line coherence : Set of visible object down not vary much between scan lines

Backface culling

- ▶ Backface culling is the simplest of the four cases.
- ▶ A face is a backface if the angle between its normal \vec{n} and the viewing direction \vec{V} is "positive" i.e. between 0 and π .
- ▶ Another formulation, for \vec{n} and \vec{V} being normalized, is:

$$\vec{n} \cdot \vec{V} = \cos \alpha \geq 0 \Rightarrow \text{face is a backface}$$



Occlusion culling

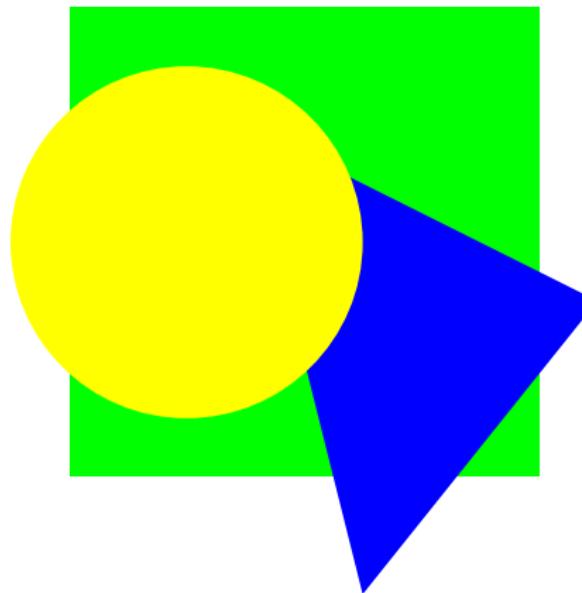
There are different algorithms to find occluded surfaces:

1. The painter
2. Z-Buffer algorithm
3. Warnock algorithm



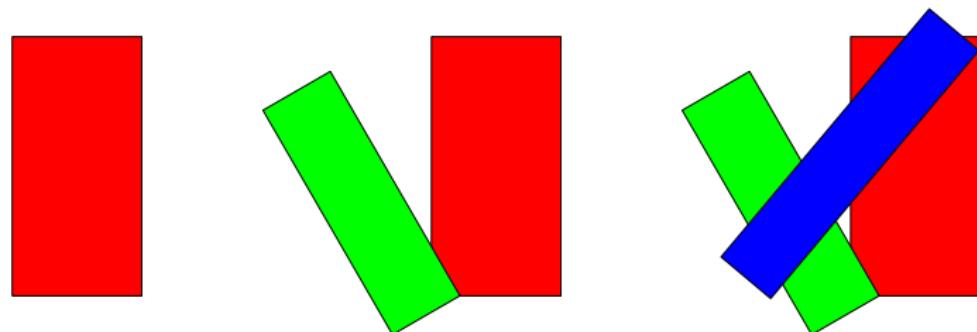
The painter algorithm's

If the object may be ordered along the view axis, one have only to draw them, for the farthest to the nearest



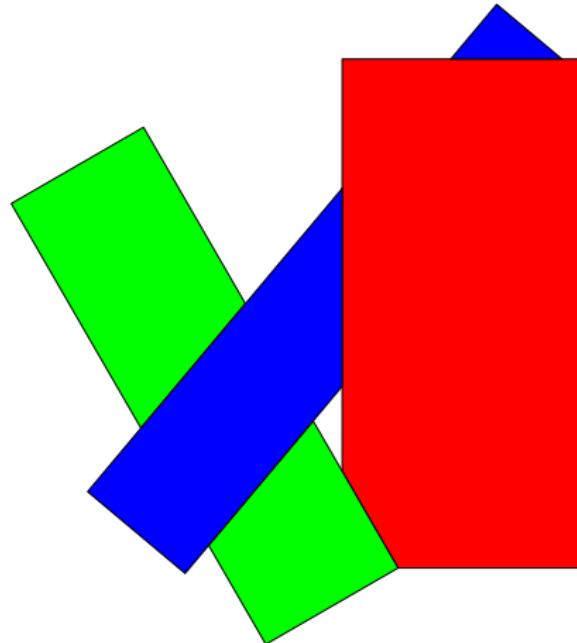
The Painter algorithm (cont'd)

Objects are drawn in reverse view axis order (the view axis of the camera)

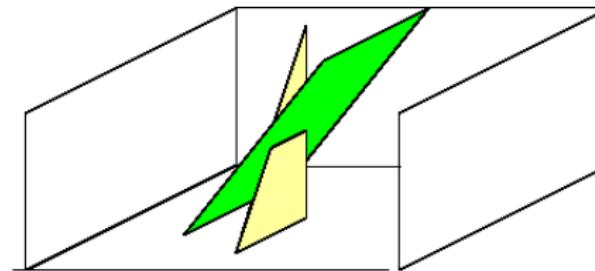


The Painter algorithm (cont'd)

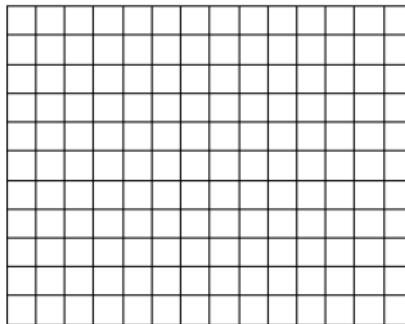
Objects may not always be fully ordered along the view axis



The Z-Buffer

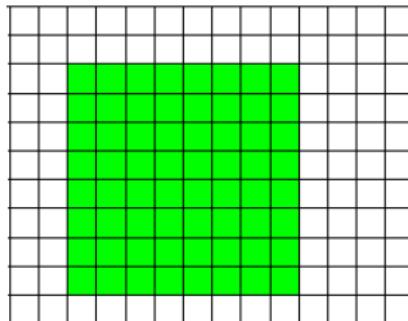


The Z-Buffer (cont'd)



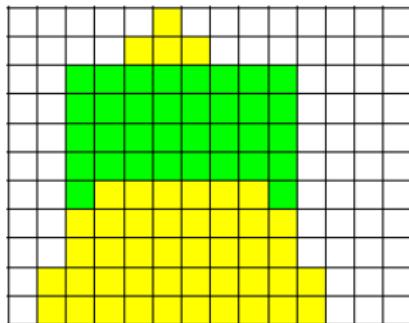
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The Z-Buffer (cont'd)



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	1	1	1	1	1	1	1	1
1	1	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	1	1	1	1	1	1	1	1
1	1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	1	1	1	1	1	1	1	1
1	1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	1	1	1	1	1	1	1	1
1	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1	1	1	1
1	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1	1	1	1
1	1	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	1	1	1	1	1	1	1	1
1	1	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The Z-Buffer (cont'd)



1	1	1	1	1	1	0.5	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	0.5	0.5	0.5	1	1	1	1	1	1	1	1	1
1	1	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	1	1	1	1	1
1	1	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	1	1	1	1	1
1	1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	1	1	1	1	1
1	1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	1	1	1	1	1
1	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1
1	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1
1	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1
1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1
1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1

The Z-Buffer (cont'd)

```
for all position(x,y) in the screen
    frame(x,y) = background color
    depth(x,y) = max_distance
endfor

for each polygon in the mesh
    for each point(x,y) in the polygon fill algo
        compute z // distance from COP
        if (depth(x,y) > z)
            depth(x,y) = z
            frame(x,y) = I(p) // shading
        endif
    endfor
endfor
```

Computing the Z value

Suppose that the objects of the scene are made of planar polygons. Normally, to calculate the z value, one have to solve the plane equation $Ax + By + Cz + D = 0$ for z , i.e.

$$z = \frac{-D - Ax - By}{C}$$

To accelerate the computation, one can use the property of **depth coherence**, that is, if for a point at (x, y) the depth is z_1 , then at $(x + \Delta x, y)$ the value z is

$$z_2 = z_1 - \frac{A}{C} (\Delta x)$$



Z-Buffer Resolution

To enhance the performance of the computation of the Z-Buffer algorithm, hardware implementation use integer arithmetic.

Each value in the Z-Buffer is stored onto 8 bits (for very old or very cheap cards), 16 bits, 24 bits or 32 bits.

The amount of memory needed for the Z-Buffer is given by:

$$(\text{graphics resolution}) \times (\# \text{ of byte per pixel})$$

but memory *is not expensive !!*



The Warnock Algorithm

The Warnock algorithm (sometimes called area-subdivision method) works essentially as an image-space methods.

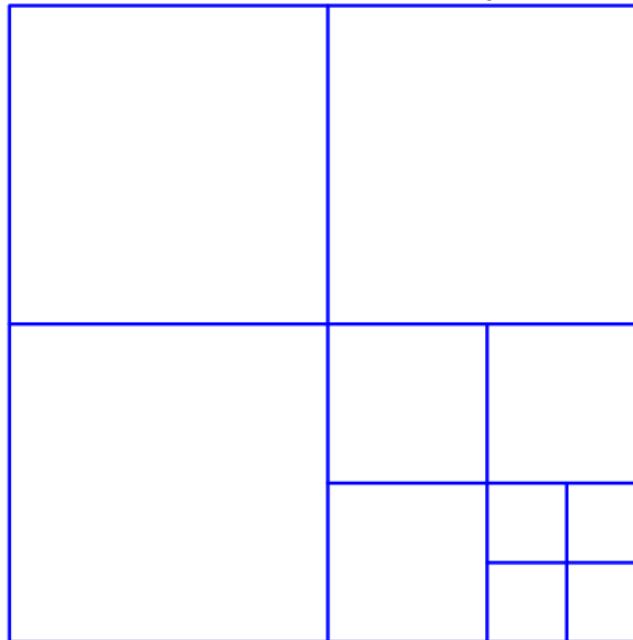
It takes advantage of the area coherence in a scene by locating those projection areas that represent part of a single surface.

The main idea is to successively divide the view plane into smaller and smaller rectangles until each rectangular area contains the projection of a part of a single visible surface, or the area has been reduced to the size of a pixel.



The viewport subdivision

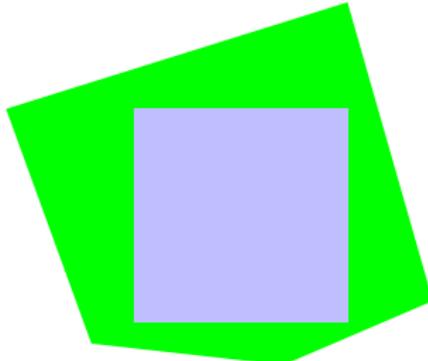
For an viewport of a resolution of 1024×1024 , one need at most 10 subdivision to reach the pixel size



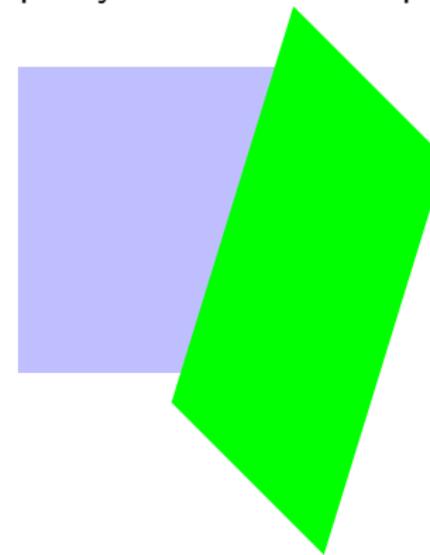
Possible relationship between surface and viewport

One can have 4 different relationship between a surface element and the viewport :

The surface completely enclose the viewport

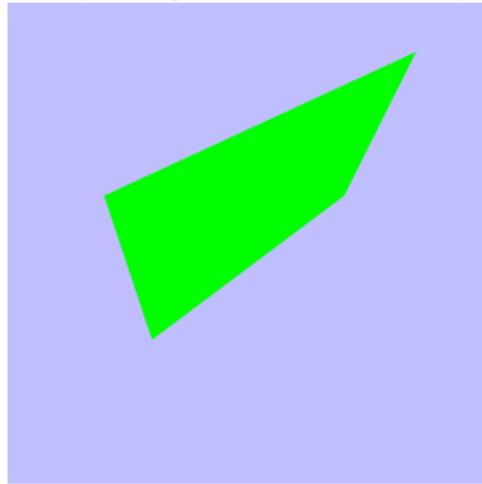


The surface is partly inside and partly outside the viewport

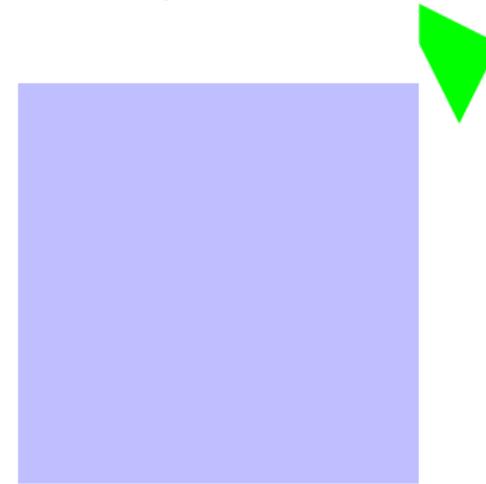


Possible relationship between surface and viewport (cont'd)

The surface completely is
completely inside the viewport



The surface is completely outside
the viewport



Possible relationship between surface and viewport (cont'd)

1. All background polygons are disjoint from the area. The background color can be displayed in this area
2. There is only one intersecting or one contained polygon. The area is first filled with the background color. The part of the polygon contained inside is scan-converted.
3. There is a single surrounding polygon. The area is filled with the color of the polygon.



Possible relationship between surface and viewport (cont'd)

The next case needs a little more attention.

- ▶ If there is a single polygon that completely surround the area and is in front of all other polygons contained in or intersecting that area then process as for step 3
- ▶ If it is not possible to find a single polygon that hide all other, split the viewport more finely.
- ▶ If the viewport size is one pixel, display the color of the first polygon (ordered by their z value) that cover that pixel.



The Warnock Algorithm

```
Warnock(PolygonList PL, ViewPort VP)
Begin
    If ( PL simple in VP) then
        Draw PL in VP
    else
        Split VP vertically and horizontally
            into VP1,VP2,VP3,VP4
        Warnock(PL in VP1, VP1)
        Warnock(PL in VP2, VP2)
        Warnock(PL in VP3, VP3)
        Warnock(PL in VP4, VP4)
    endIf
End
```



BSP Trees

- ▶ If we are making many image of the same geometry from different viewpoints, another method is to use the BSP tree
- ▶ The key aspect of BSP tree is to preprocess the scene to create a data structure that is usefull for any viewpoint.



Overview of the BSP tree algorithm

The BSP tree algorithm is a generalisation of the painter's algorithm. It can be implemented as:

sort object back to front relative to viewpoint

for all objects **do**

 draw object on screen

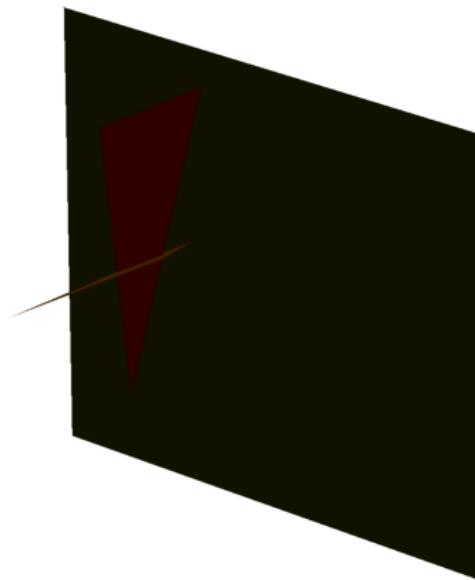
end for

The BSP tree algorithm works on any scene composed of polygons where no polygons crosses the planes defined by any other polygons



Overview of the BSP tree algorithm (cont'd)

The basic idea of the BSP tree may be illustrated with 2 triangles T_1 and T_2 .



Overview of the BSP tree algorithm (cont'd)

- ▶ The plane defined by T_1 is given by the implicit function:

$$T_1 : f_1(p) = 0$$

With this implicit function, if $f_1(q) < 0$ this means that the point q is one one side of the plane and if $f_1(q) > 0$, then q is on the other side.

- ▶ The triangle T_2 is totally on one side on the plane (for example on the negative side).
- ▶ The viewpoint is given by the variable e .



Overview of the BSP tree algorithm (cont'd)

The sketch of the BSP algorithm can be:

```
if  $f_1(e) < 0$  then
    draw  $T_1$  { $T_2$  is in front of  $T_1$ }
    draw  $T_2$ 
else { $T_1$  is in front of  $T_2$ }
    draw  $T_2$ 
    draw  $T_1$ 
end if
```



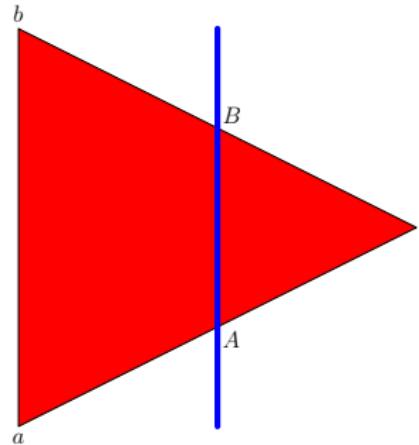
Building the BSP tree

- ▶ If all vertices of a polygon (for example a triangle) are in the same side of a given plane, the construction of the tree is easy. One can simply add the triangle on the "positive" or "negative" subtree of the node defined by that plane
- ▶ If the polygon crosses the plane, one should find the intersection points of the polygon with that plane and redefine sub-polygons such as all of these are either totally on the negative side or totally on the positive side.



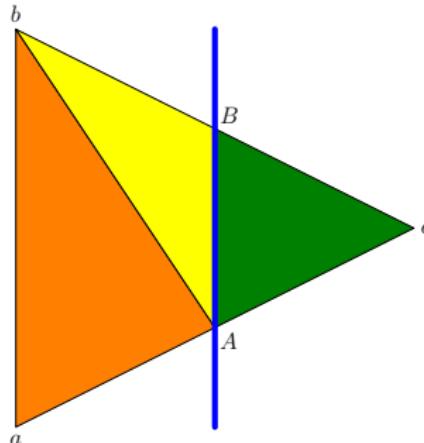
Building the BSP tree (cont'd)

Let's say T is a triangle which crosses the plane defined by another triangle.



Building the BSP tree (cont'd)

One can see that the intersection of the red triangle with the blue plane are the points A and B . Therefore the triangle should be decomposed into 3 subtriangles:



$$T_1 = (a, b, A)$$

$$T_2 = (b, B, A)$$

$$T_3 = (A, B, c)$$

The new subtriangle should respect the orientation of the original triangle

Cutting the triangles

To compute the intersection points, one could simply use the method seen above to find the intersection point of a line with a plane.

The parametric form of the line segment between a and c is given by the equation:

$$p(t) = a + t(c - a)$$

The intersection with the plane given by $n \cdot p + D = 0$ is found by plugging $p(t)$ into the plane equation:

$$n \cdot (1 + t(c - a)) + D = 0$$

Solving this equation for t gives:

$$t = \frac{n \cdot a + D}{n \cdot (c - a)}$$



Cutting the triangles (cont'd)

Calling this solution t_A , one can write the expression for A :

$$A = a + t_A(c - a)$$

A similar computation will give B

The names of the vertices a , b and c have been choosed randomly.

Therefore one can always swap theses names to have the two vertices a and b on one side of the plane and c on the other side.



Introduction to Computer Graphics

└ Hidden Surface Removal

 └ BSP trees



Part X

Textures and Bump Mapping



Outline

Main idea

Procedural textures

 Procedural textures

 Perlin noise

2D Textures

 Mapping 2D to 2D

 Mapping 2D to 3D to 2D

 Color computing

Bump Mapping



Main idea of textures

Adding realistic details to a 3D object may be a huge work.
Therefore, these details may be simulated by different methods.
Texture mapping is a method for adding detail, surface texture, or colour to a computer-generated graphic or 3D model.
These textures may be either algorithmically generated or taken from digital images.



Outline

Main idea

Procedural textures

 Procedural textures

 Perlin noise

2D Textures

 Mapping 2D to 2D

 Mapping 2D to 3D to 2D

 Color computing

Bump Mapping



Procedural textures

A procedural texture is a computer generated image created using an algorithm intended to create a realistic representation of natural elements such as wood, marble, granite, metal, stone, and others. Usually, the natural look of the rendered result is achieved by the usage of fractal noise and turbulence functions. These functions are used as a numerical representation of the “randomness” found in nature.

3D Textures

- ▶ One of the simplest way to add texture to an object is simply by adding "noise" to the geometry of this object.
- ▶ The noise may be either randomly generated, computed from a formula, or a mix of both methods.
- ▶ The main difficulty of this method is to have a realistic noise, that is a noise which makes the object appear realistic.
- ▶ This kind of method modifies the geometry of the object.



Normal perturbation

- ▶ If one need to represent an orange, one would like to see the small bumps and hollows on its surface.
- ▶ A standard orange is almost a sphere
- ▶ If one need to geometrically describe this bumps and hollow, it would be a huge works since it would imply to defines thousands of points on the surface of the fruit.
- ▶ A simplest method would be to slightly modify the normal vector during the rendering process.
- ▶ This perturbation has a random component but is not totally random. The perturbation at one pixel is partly influenced by the perturbation at its neighbours.



Perlin noise

- ▶ Prof. Ken Perlin is a professor in the Department of Computer Science at New York University.
- ▶ His invention of Perlin noise in 1985 has become a standard that is used in both computer graphics and movement.
- ▶ Perlin noise is a texturing primitive you can use to create a very wide variety of natural looking textures. Combining noise into various mathematical expressions produces procedural texture.
- ▶ One can find more information at the url
<http://www.noisemachine.com/talk1/index.html>

Perlin noise : an example

Suppose one need to draw a mountain. As a first approximation, one can see the mountain as a simple triangle.



This mountain does not appear very realistic.



Perlin noise : an example (cont'd)

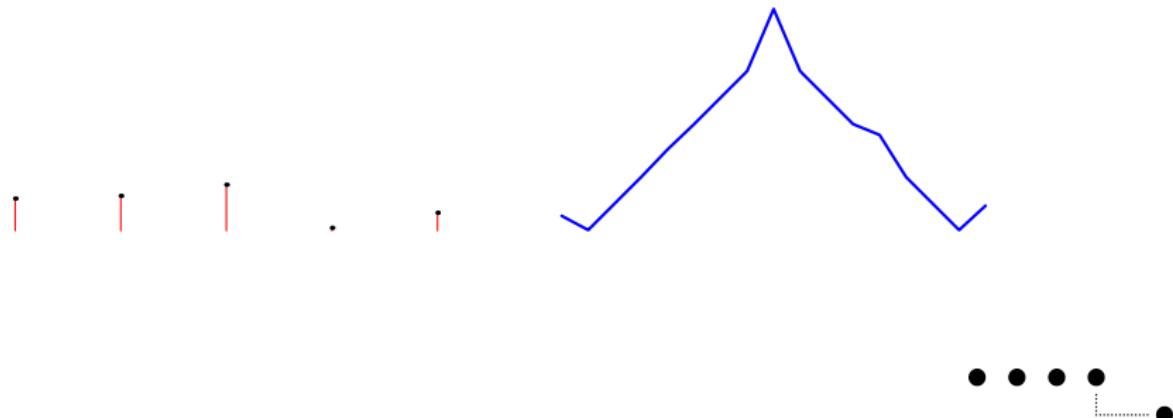
Would be a totally random sequence more realistic ?



Altough this "mountain" is less regular than the previous, it does not seems more realistic.

Perlin noise : an example (cont'd)

Combining the two techniques may help to compute a more realistic mountain. For that goal one shoud procede by steps. For the first step, only the "main points" of the first mountain are modified.



Perlin noise : an example (cont'd)

One can refine the form of the mountain using a second step, where more control will be modified, but, with a smaller amplitude.



Perlin noise : an example (cont'd)

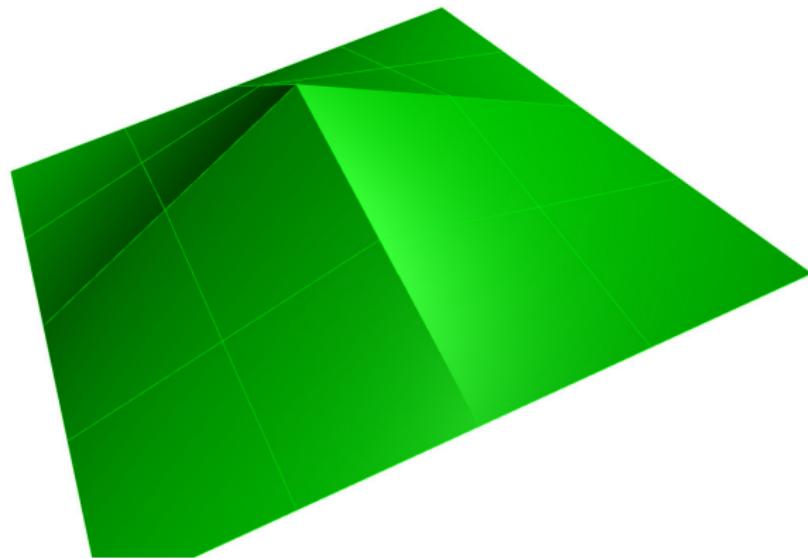
The number of processing steps that are necessary depends on different parameter like the kind of randomizing function being used.



Perlin noise for surfaces

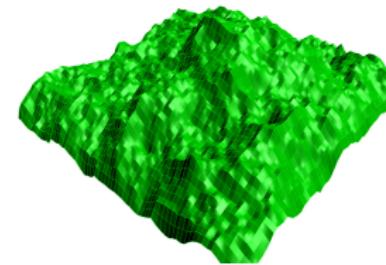
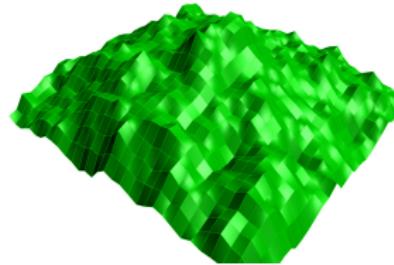
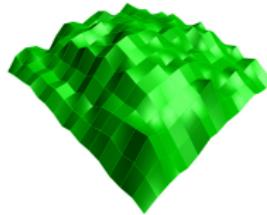
The technique for Perlin noise may be easily extended for surfaces.

The base surface is given by a mesh:



Perlin noise for surfaces

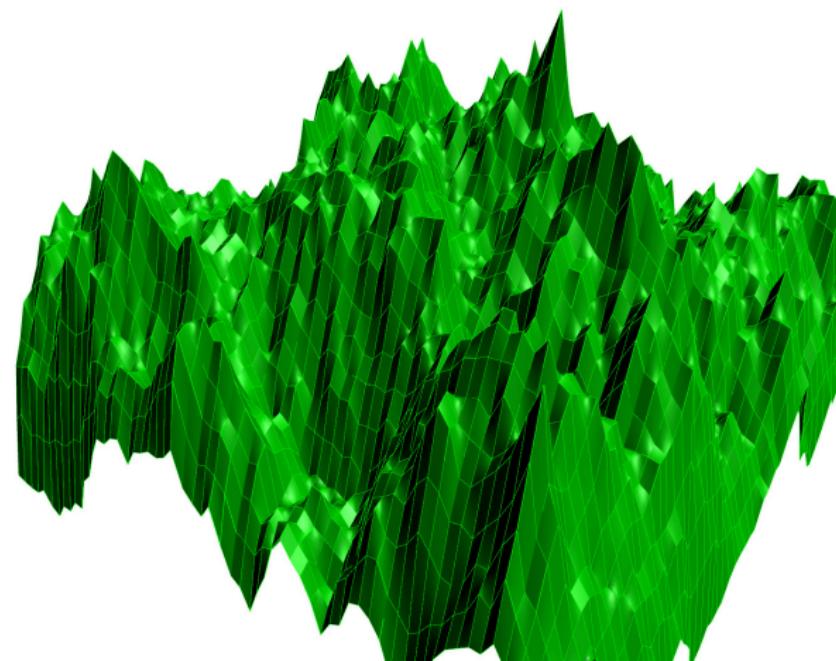
Applying the technique of Perlin noise to each parametric direction would give (after 1, 2 and 3 steps):



If you want to try it, see file `perlin6.asy`

Perlin noise for surfaces

The disturbance factor must be chosen carefully, otherwise the result will not be realistic.



Outline

Main idea

Procedural textures

 Procedural textures

 Perlin noise

2D Textures

 Mapping 2D to 2D

 Mapping 2D to 3D to 2D

 Color computing

Bump Mapping



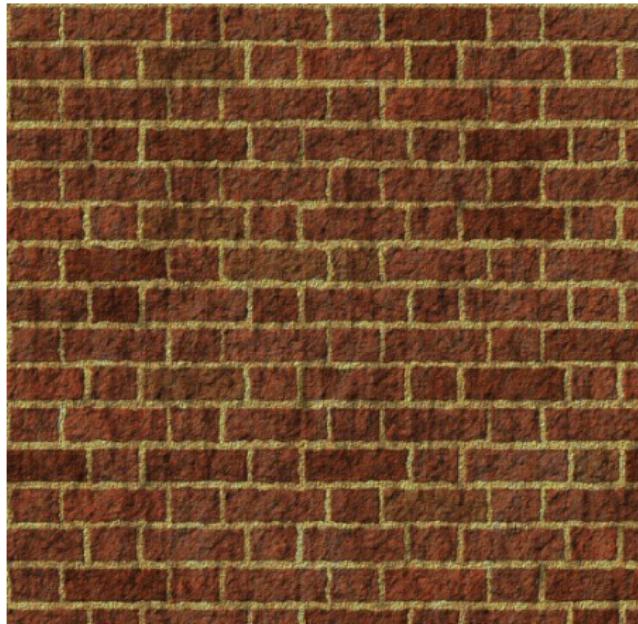
2D Textures

- ▶ As mentionned above, one may use a 2D digital picture as a base image for a texture.
- ▶ This would allow assign to an object the appearance of a real object which cannot be described by an algorithm or a random process (at least not easily).
- ▶ 2D textures modify the appearance of an object (its color), but not its geometry.



Textures

To draw an object with a appearance more complex than a simple color, one can use a *texture*. A texture is a bitmap file which is mapped onto an object as a wallpaper.



Textures

The main idea of texture is to map the picture space (which at which coordinate) onto a face. Each pixel in the texture file (called texel) map to a pixel on screen.

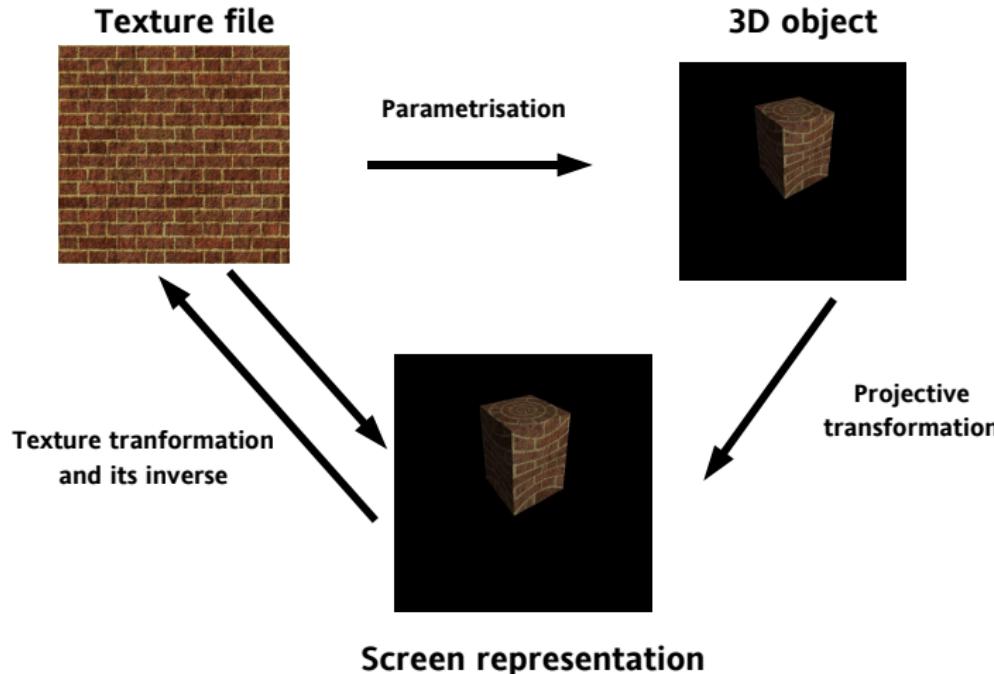


Mapping 2D to 2D

With the 2D to 2D mapping, one would be able to establish an application, which, for every pixel on the screen gives the coordinate of the texel which should appear there. This mean that one should also include in the application function the projective transformation.

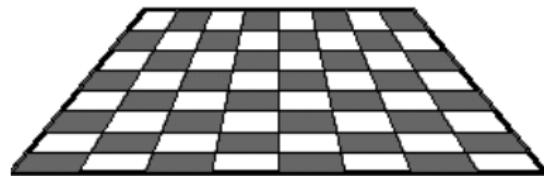


Mapping 2D to 2D

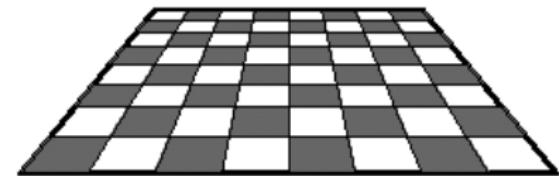


Texture deformation

This technique leads sometimes to texture deformations



Without hyperbolic interpolation



With hyperbolic interpolation

Mapping 2D to 3D to 2D

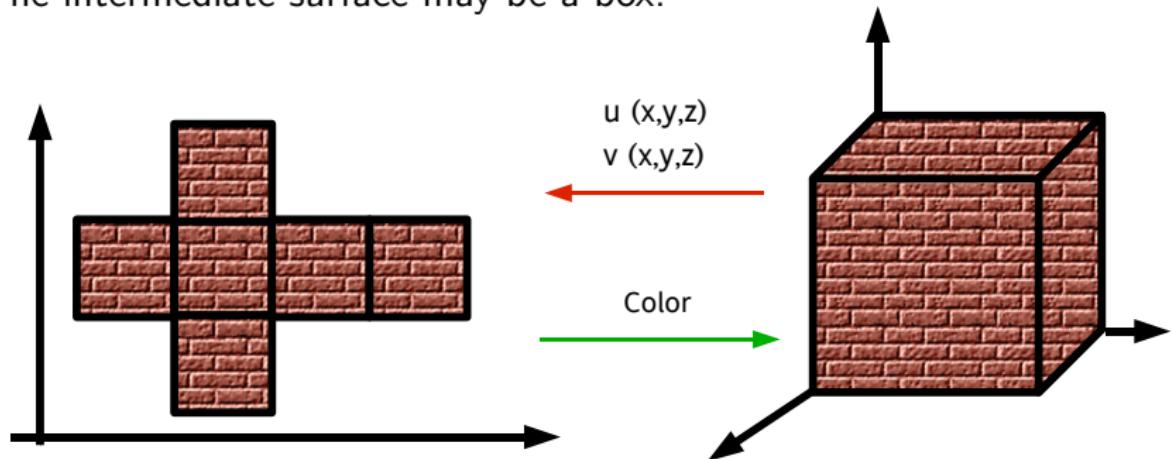
As it was shown above, the 2D to 2D technique may show some problem and "non natural" distortion of the texture. Therefore, in 1986, Bier & Sloan have proposed to make the mapping in two steps:

1. The texture is mapped onto an intermediate surface, which is in the same coordinate system as the object;
2. The texture is then transposed from this surface to the object using one of the four projection they proposed
3. Once the texture is on the surface of the object, one can apply the perspective transformation



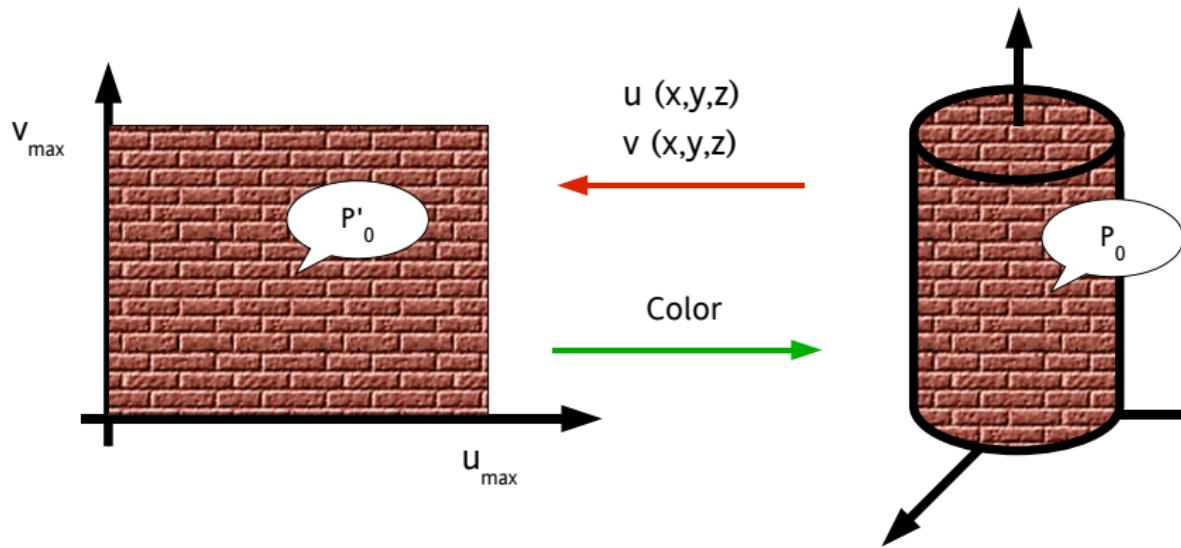
Mapping 2D to 3D to 2D (cont'd)

The intermediate surface may be a box:



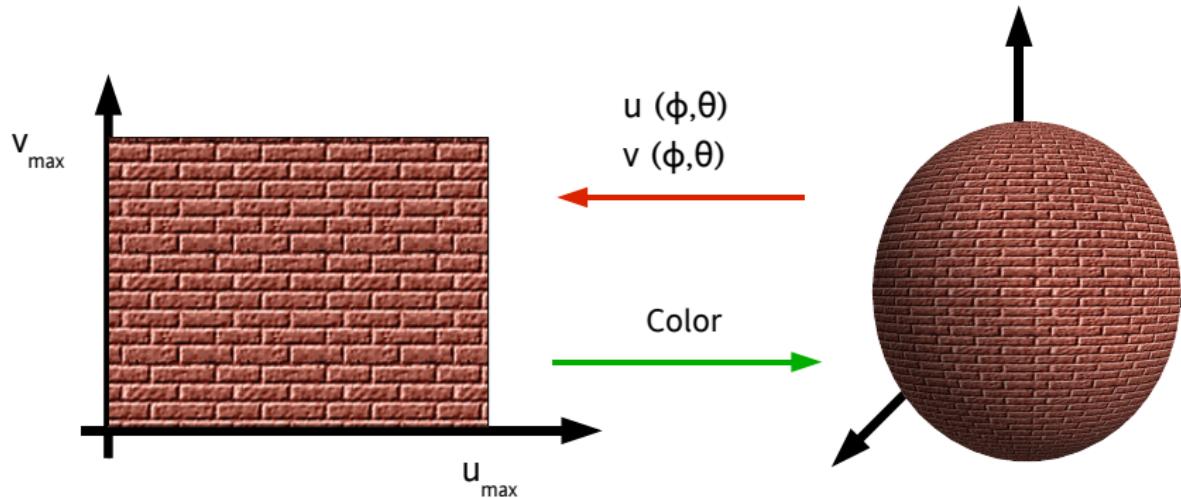
Mapping 2D to 3D to 2D (cont'd)

The intermediate surface may be a cylinder:



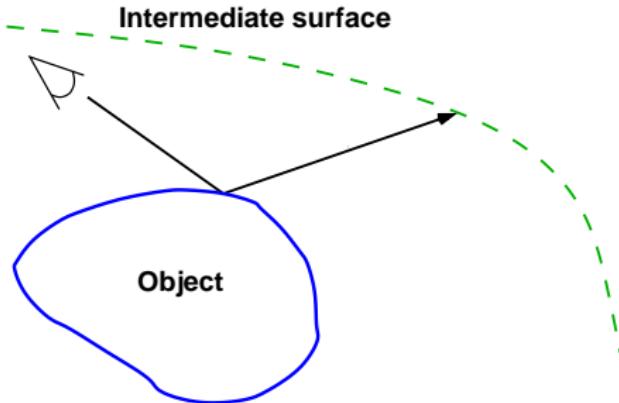
Mapping 2D to 3D to 2D (cont'd)

The intermediate surface may be a sphere:

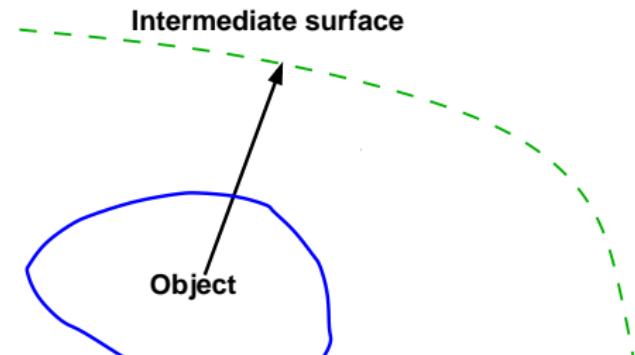


Mapping on the object

Direction of eye

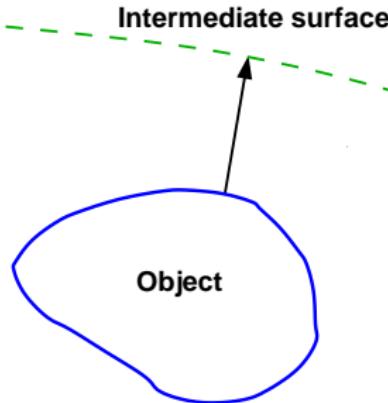


Centroid of the object

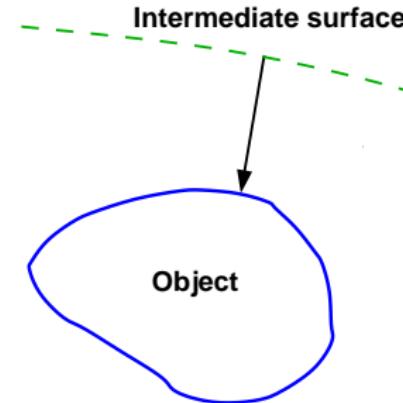


Mapping on the object

Normal to the object



Normal to the intermediate surface



Color computing I

Once one have found which texel should be applied onto which pixel of surface element of an object, one should also compute the "real" color of this pixel. For that goal, one can use different techniques:

- ▶ The color of the texel replaces the color of the object:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{final}} := \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{texture}}$$

- ▶ Use a linear interpolation between the color of the object and the color of the texel:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{final}} := \lambda \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{object}} + (1-\lambda) \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{texture}}$$

where $0 \leq \lambda \leq 1$



Color computing (cont'd)

- ▶ The modulation

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{final}} := \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{object}} * \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{texture}}$$

- ▶ The "mix"

$$\begin{pmatrix} R \\ G \\ B \\ \alpha \end{pmatrix}_{\text{final}} := (1 - \alpha_{\text{final}}) \begin{pmatrix} R \\ G \\ B \\ \alpha \end{pmatrix}_{\text{object}} + \alpha_{\text{final}} * \begin{pmatrix} R \\ G \\ B \\ \alpha \end{pmatrix}_{\text{texture}}$$

where $\alpha_{\text{final}} = \alpha_{\text{pixel}} + (1 - \alpha_{\text{pixel}})\alpha_{\text{texture}}$



Outline

Main idea

Procedural textures

 Procedural textures

 Perlin noise

2D Textures

 Mapping 2D to 2D

 Mapping 2D to 3D to 2D

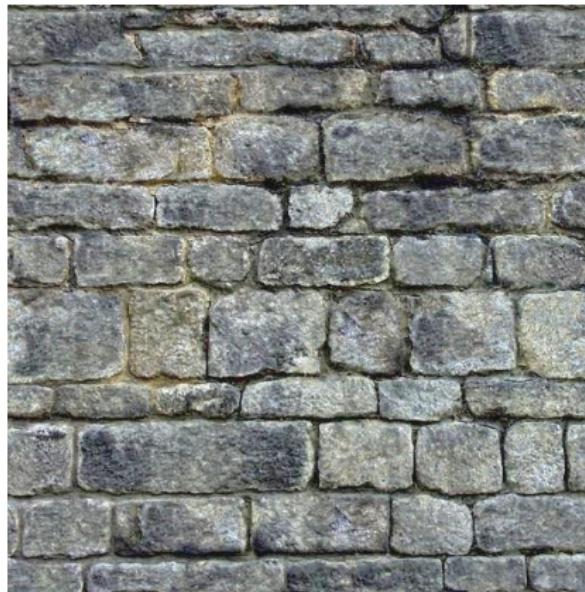
 Color computing

Bump Mapping



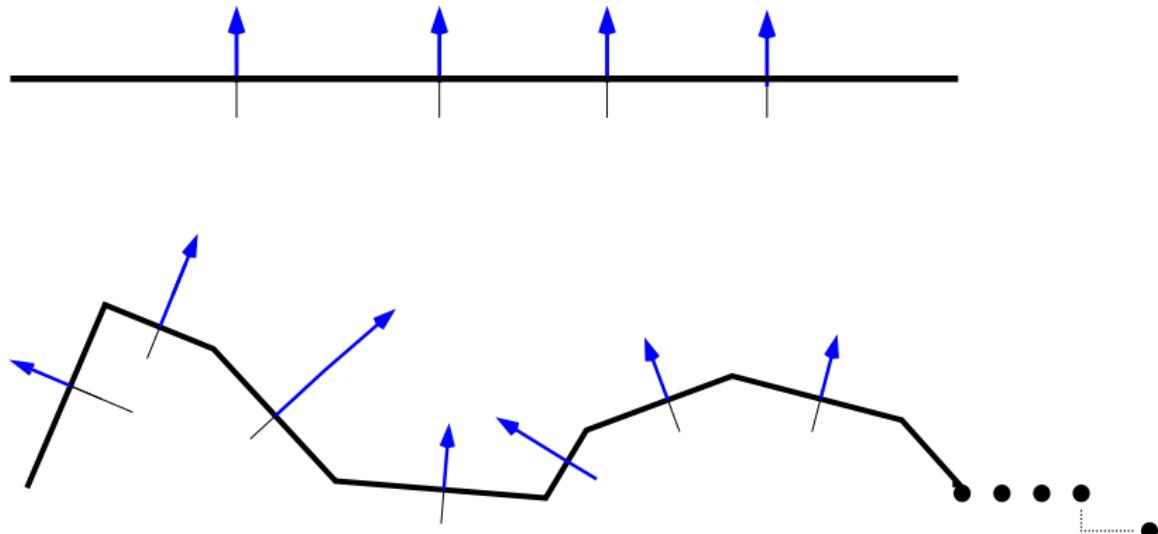
What is Bump Mapping ?

Basically, bump mapping is the art of making a 2D texture look as if it's in 3D as shown in the picture below:



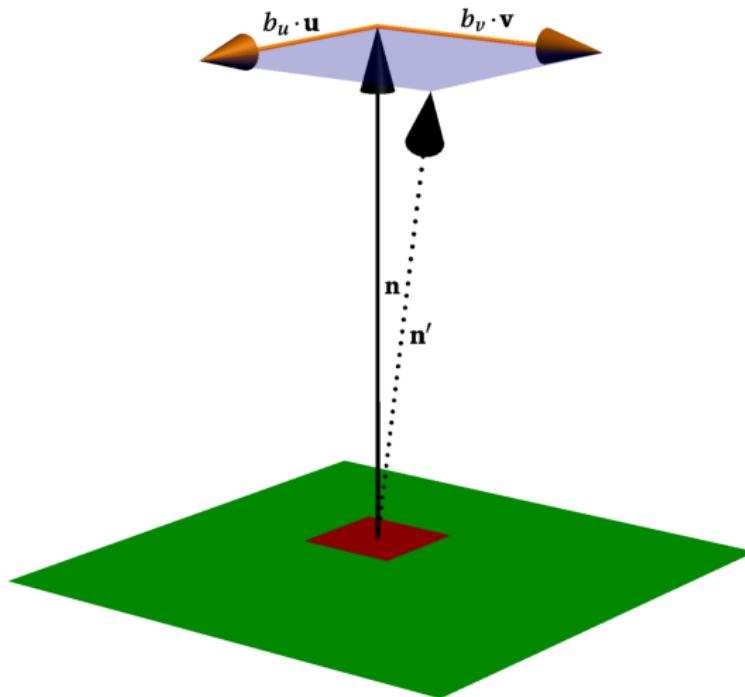
Why Bump Mapping ?

With ordinary textures, the surface where this texture is applied is plain flat and therefore the normals of the texture surface go straight up,



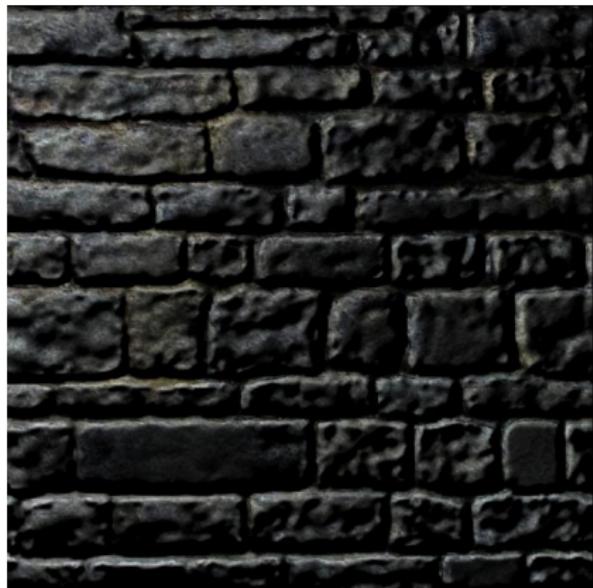
Why Bump Mapping

The Bump Mapping techniques modify the normal vector.



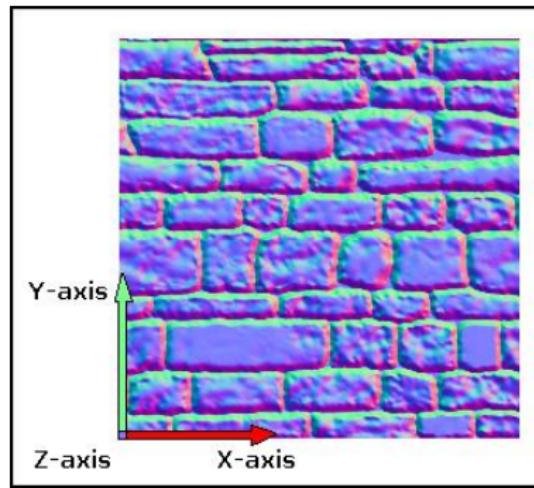
How to use a bump map

The normal perturbation is defined into a bitmap image where the color of each pixel is used to modify the normal vector of the surface of the object at a given texel.



How to use a bump map (cont'd)

One can also use an image to completely define the normal of the surface at a given point (a given texel). For that goal one can use an image where the colors of each pixel is "transformed" into vector coordinates:



Bump mapping algorithm

The interaction of the bump mapping with the object is computed before the shading stage of the rendering. One can define the bump mapping algorithm as:

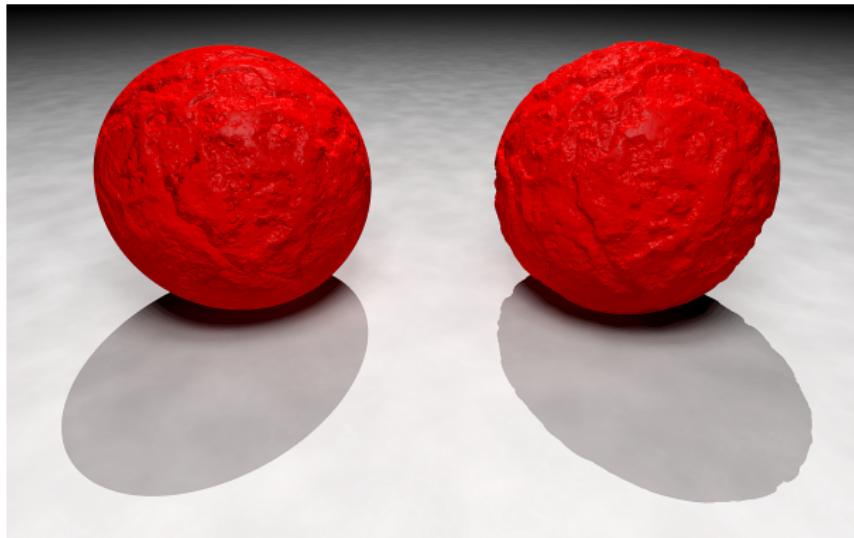
1. Compute the normal of the object at a certain point of the object.
2. Look up the position on the heightmap that corresponds to the position on the surface.
3. Calculate the surface normal of the heightmap.
4. Add the surface normal from step two to the geometric surface normal so that the normal points in a new direction.
5. Calculate the interaction of the new "bumpy" surface with lights in the scene using, for example, the Phong shading.

The result is a surface that appears to have real depth. The algorithm also ensures that the surface appearance changes as lights in the scene are moved around.



Bump mapping limitations

The main limitation of standard bump mapping technique is that it only modify the normal vector. The shape of the underlying object is not modified (see the shadow of the object).



Part XI

Floating point numbers



Introduction

Fixed point numbers

Floating points



Outline

Introduction

Fixed point numbers

Floating points



Floating points

At the beginning of computer science, computer only provided integer arithmetic. But, for scientific uses, it was needed to allow programmers to use "real numbers". The data types used to model \mathbb{R} , the real number of mathematicians, have special properties that every programmer should know. Most of programming languages (and computer hardware) now provide the IEEE 754 norm.



Outline

Introduction

Fixed point numbers

Floating points



Fixed point numbers

How to represent the decimal number 41.13 with fixed point arithmetic ? Remember that every number in base b can be written as:

$$x = \sum_{i=-\infty}^{\infty} d_i b^i$$

where the d_i are the digits of the numbers and b the base. So the number $(41.13)_{10}$ may be written as the sum:

$$41.13 = 4 * 10^1 + 1 * 10^0 + 1 * 10^{-1} + 3 * 10^{-2}$$



Binary fixed point numbers

As the number 41.13 exist independantly of its representation, it should be possible to write this number using the binary notation.

1. First code the integer part of the number in binary. The decimal value 41 is written $(101001)_2$.
2. Compute the decimal part of the number. The value 0.13 is written as $(0.001000010100011110\dots)_2$

Finally one have that:

$$(41.13)_{10} = (101001.001000010100011110\dots)_2$$



Some mathematical considerations

- ▶ A rational number may be written either with a finite decimal form or with a periodic form. The property "to be rational" is independant of the base used to write the number.
- ▶ A rational number may have a finite representation in one base and a periodic infinite representation in another base. For example, the number $(1/3)$ has an infinite periodic representation in base 10, but may be written as $(0.\overline{1})_3$ in base 3. The decimal number (0.1) has an infinite periodic representation in base 2. $(0.1)_{10} = (0.\overline{0011})_2$
- ▶ Irrational numbers have infinite and non-periodic form. These numbers does not have a finite representation in any base.



Outline

Introduction

Fixed point numbers

Floating points



Floating points

- ▶ The main drawback of fixed point numbers is that their size depends on their magnitude and their precision. But, in engineering, most of the time, one can reduce the precision as the magnitude augment. Engineers use mostly the concept of *significant digits*.
- ▶ This notation is usually known as "engineering notation" in pocket calculators. For example one can write :
 - ▶ $1.344 \cdot 10^4$ for the number 13440
 - ▶ $2.342 \cdot 10^{-5}$ for the number 0.00002342
 - ▶ $4.430 \cdot 10^0$ for the number 4.430

All theses numbers have 4 significant digits (the mantissa).

The "scaling" is done by the exponent of 10. The mantissa is always a number between 1.000 and 9.999.



IEEE 754 representation

The norm IEEE 754 used for binary representation of floating point numbers is based on the same idea.

- ▶ the mantissa is a number between 1.0 and 2.0 (smaller than 2.0). The number of significant digits is given by the type of floating point numbers used (`float`, `double`)
- ▶ The exponent is now an exponent of 2. It may take positive or negative values.
- ▶ The norm introduces special values (negative infinity, positive infinity, not a number, zero) that would be studied later in this chapter.
- ▶ The IEEE 754 norm introduce also rounding rules for numbers.

Floating numbers : a first example

For this example, the number is coded on 32 bits, which correspond to the float format of Java for example.

How would be the decimal number 0.5 represented ?

$$(0.5)_{10} = \textcolor{red}{0} \textcolor{blue}{\ 0111\ 1110\ 0000\ 0000\ 0000\ 0000\ 000}$$

- ▶ The first bit (red) is the sign bit. Its value is 0 for positive number and 1 for negative numbers.
- ▶ The second group (blue) is the exponent. Here one have the value -1. The exponent is coded as an unsigned integer with a bias. The value of the exponent is computed by the formula `valueOfTheField - 127`.
- ▶ The third field contains the mantissa which here is 0 (there is a hidden bit!)



The exponent

- ▶ As explained above, the exponent is coded as an unsigned integer (no two-complement) with a bias which correspond to the half of the magnitude of the exponent. For example if the exponent is coded on 8 bits, its magnitude is $2^8 = 256$ and the bias would be $2^{8-1} - 1 = 127$
- ▶ Some values of the exponent are reserved to represent special values of numbers. These values are $(00)_{16}$ and $(FF)_{16}$.



The mantissa

- ▶ As explained before the mantissa contain a value into the interval $1 \leq \text{mantissa} < 2$. Remark that the value 2 does not belong the the intervall.
- ▶ On this interval, one may see that the first bit is always 1 and therefore is not represented explicitly (it is called the [hidden bit](#)). The norm IEEE 754 define a special format (called denormalized numbers) to change this behavior.
This hidden bit cause the value of the mantissa to 0 on the example above.
- ▶ The bits of the mantissa represent the sum of negative power of 2.



Exercise

Compute the biggest value in magnitude (different of ∞) which can be represented by the format described above.



Solution

- ▶ The biggest value is represented by the biggest exponent and the biggest mantissa.
- ▶ Since the value $(FF)_{16}$ is not authorized for the mantissa, one should choose the value $(FE)_{16} = 254$.
- ▶ The biggest mantissa is the mantissa where all bits are sets to one. The mantissa represent the number:

$$\begin{aligned}\text{mantissa} &= \sum_{i=1}^{i=23} \frac{1}{2^i} = \frac{2^{23} - 1}{2^{23}} \\ &= \frac{8388607}{8388608} = 0.9999998807907104\end{aligned}$$

- ▶ The biggest number is then:

$$(1 + 0.9999998807907104) * 2^{127} = 3.402834 * 10^{38} \bullet \bullet \bullet$$

Exercise

- ▶ Compute the smallest positive number different of 0 with the representation above
- ▶ Compute the second smallest positive number with the same representation.



Solution 1

The smallest positive number has the smallest possible exponent and the smallest possible mantissa. That is:

- ▶ the exponent can be $(01)_{16}$ for the value 2^{-126} (remember that the value 0 is reserved)
- ▶ the smallest possible mantissa has all bits cleared.

The number is then:

$$\varepsilon = 1.0 \cdot 2^{-126} = 1.17549435 \cdot 10^{-38}$$



Solution 2

The second smallest positive number differ from ε from only one bit in the mantissa. To find it, just set the least significant bit of the mantissa. Then this value is:

$$\varepsilon_2 = (1 + 2^{-23}) \cdot 2^{-126} = 1.1754945 \cdot 10^{-38}$$

The difference between these two values is :



Special values

As it was mentionned above, the norm IEEE 754 introduced some special values.

Exponent	Mantissa	Value	Description
00_{16}	$= 0$	0	Zero
00_{16}	$\neq 0$	$\pm 0.m \cdot 2^{-126}$	Denormalized
01_{16} to FE_{16}	any	$\pm 1.m \cdot 2^{e-127}$	Normalized
FF_{16}	$= 0$	$\pm\infty$	\pm Infinity
FF_{16}	$\neq 0$	NaN	Not a Number

Remark : There is two possible representation for the value 0 (+0 and -0)



Comments on special values

- ▶ The value zero need a special representation as it is not possible to represent it using the standard model (see example above)
- ▶ Denormalized values are not provided by standard programming language. Be extremely careful if you need them (perfomance issues)
- ▶ Infinity represent numbers whose magnitude may not represented into the model. Arithmetical operations where one operand is infinity are well defined by IEEE norm (see next slide).
- ▶ Operation where an operand has the value NaN cause an error.



Operation with infinity

Operation	Result
$x / \pm\infty$	0
$\pm\infty \cdot \pm\infty$	$\pm\infty$
$\pm \text{non zero} / 0$	$\pm\infty$
$\infty + \infty$	∞
$\pm 0 / \pm 0$	NaN
$\infty - \infty$	NaN
$\pm\infty / \pm\infty$	NaN
$\pm\infty \cdot 0$	NaN



Different type of floating points

The norm IEEE 754 defines two types of floating points : single precision and double precision. They differ only by the number of bits used to store them

	Sign	Exponent	Mantissa	Bias
Single precision	1	8	23	127
Double precision	1	11	52	1023



Rounding

IEEE standard has four different rounding modes. The first is the default, the others are called *directed rounding*.

- ▶ **Round to nearest** : rounds to the nearest value. If the number fall in the midway it is rounded to the nearest value with an even (zero) least significant bit.
- ▶ **Round toward 0** - directed rounding towards 0
- ▶ **Round toward $+\infty$**
- ▶ **Round toward $-\infty$**



Example of rounding

Rounding mode	+11.5	+12.5	-11.5	-12.5
to nearest, ties to even	+12.0	+12.0	-12.0	-12.0
to nearest, ties away from zero	+12.0	+13.0	-12.0	-13.0
toward 0	+11.0	+12.0	-11.0	-12.0
toward $+\infty$	+12.0	+13.0	-11.0	-12.0
toward $-\infty$	+11.0	+12.0	-12.0	-13.0



Exercises

- ▶ Write a C program to display the value (bitwise) of 14.34, 0.0, Nan and $+\infty$ stored as float.
- ▶ Write a Java program to display the same values also stored as float.

