Computer Graphics

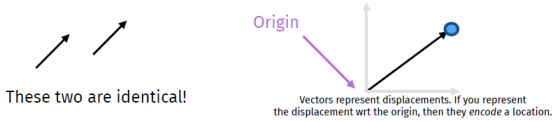
Riccardo Caprile

March 2022

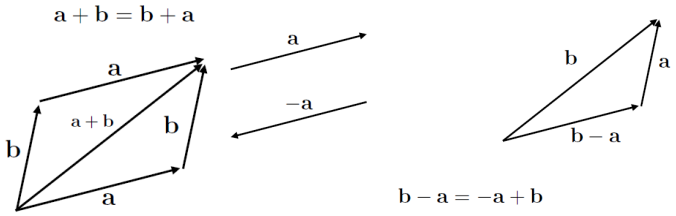
1 Linear Algebra

1.1 Vectors

A vector describes a **direction** and a **length**. When you encode them in your program, they will both require 2 or 3 numbers to be represented, but they are not the same object. The C++ command is Eigen::VectorXd

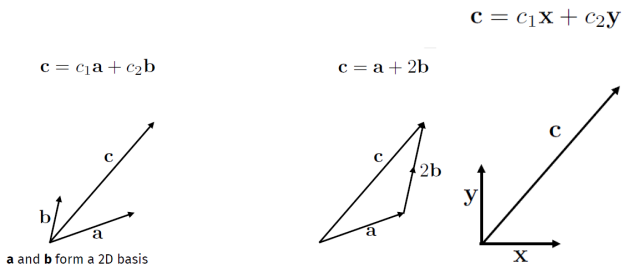


1.2 Sum and Difference



The Operator in C++ for the sum is + and for the difference -

1.3 Coordinates and Cartesian Coordinates



The operator is [] and x and y form a canonical, Cartesian basis

1.4 Length

The length of a vector is denoted as $||\mathbf{a}||$.

If the vector is represented in cartesian coordinates, then it the L2 norm of the vector : $||a|| =$

$$\sqrt{a_1^2 + a_2^2}$$

A vector can be **normalized**, to change its length to 1 , without affecting the direction : b =

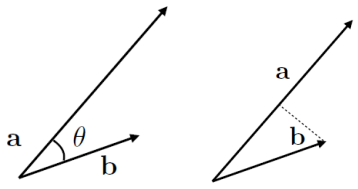
$$\frac{a}{||a||}$$

The command in C++ are a.norm(), b.normalize() in place, b.normalized() returns the normalized vector

1.5 Dot Product and Projection

$a \cdot b = ||a|| \cos\theta$ The dot product is related to the length of vectors and to the angle between them. If both vectors are normalized, it is the cosine of the angle between them. a.dot(b), a.transpose()*b The length of the projection of b onto a can be computed using the dot product : b $\rightarrow$a = $||b|| \cos\theta =$

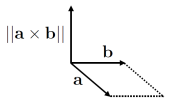
$$\frac{b \cdot a}{||a||}$$



1.6 Cross Product

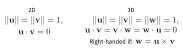
Defined only for 3D vectors. The resulting vector is perpendicular to both a and b, the direction depends on the right hand rule. The magnitude is equal to the area of the parallelogram formed by a and b. Eigen::Vector3d v(1,2,3) Eigen::Vector3d w(4,5,6) v.cross(w)

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



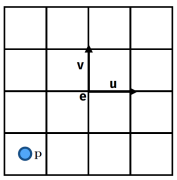
1.7 Coordinate Systems

You will often need to manipulate coordinate systems. You will always use **orthonormal bases**, which are formed by pairwise orthogonal unit vectors :



1.8 Coordinate Frame

- **e** is the origin of the reference system
- **p** is the center of the pixel
- **u,v,w** are the coordinates of p



$\mathbf{p} = \mathbf{e} + u\mathbf{u} + v\mathbf{v} + w\mathbf{w}$

If you have a vector a expressed in global coordinates, and you want to convert it into a vector expressed in a local orthonormal u-v-w coordinate system , you can do it projections of a onto u,v,w : a^C = (a · u , a · v, a · w)

1.9 Matrices

Matrices will allow us to conveniently represent and ally transformations on vectors, such as translation , scaling , and rotation.

A matrix is an array of numeric elements. Eigen::MatrixXd A(2,2)

Sum $\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} + \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} = \begin{bmatrix} x_{11}+y_{11} & x_{12}+y_{12} \\ x_{21}+y_{21} & x_{22}+y_{22} \end{bmatrix}$

A.array() + B.array()

Scalar Product $y * \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} yx_{11} & yx_{12} \\ yx_{21} & yx_{22} \end{bmatrix}$

A.array() * y

Transpose

The transpose of a matrix is new matrix whose entries are reflected over the diagonal .

$$\begin{bmatrix} 1 & 2 \end{bmatrix}^T = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

The transpose of a product is the product of the transposed, in reverse order $(\mathrm{A}B^T) = B^T\ A^T$

Matrix Product

The entry i,j is given by multiplying the entries on the i-th row of A with the entries of the j-th column of B and summing up the results and it is not commutative AB =/ BA. A*B

$$\begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} -\mathbf{r_1}- \\ -\mathbf{r_2}- \\ -\mathbf{r_3}- \end{bmatrix} \begin{bmatrix} | \\ \mathbf{x} \\ | \end{bmatrix} \qquad \begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \mathbf{c_1} & \mathbf{c_2} & \mathbf{c_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$y_i = \mathbf{r_i} \cdot \mathbf{x}$

Dot product on each row

$\mathbf{y} = x_1\mathbf{c_1} + x_2\mathbf{c_2} + x_3\mathbf{c_3}$

Weighted sum of the columns

Inverse Matrix

The inverse of a matrix A is the matrix A^-1 such that $\mathrm{A}A^-1 = \mathrm{I}$ where I is the identity matrix. $(AB)^-1 = B^-1\ A^-1$ A.inverse()

Diagonal Matrices

They are zero everywhere except the diagonal. A = v.asDiagonal()

Orthogonal Matrices An orthogonal matrix is a matrix where each column

is a vector of length 1 and each column is orthogonal to all the others. A useful property of orthogonal matrices that their inverse corresponds to their transpose $:(R^T\mathrm{R}) = \mathrm{I} = (\mathrm{R}R^T)$

1.10 Linear Systems

We will often encounter in this class linear systems with n linear equationd that depend on n variables :

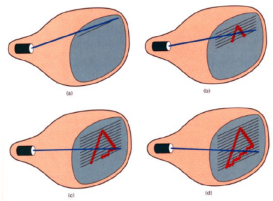
- For example: $\begin{aligned} 5x + 3y - 7z &= 4 \\ -3x + 5y + 12z &= 9 \\ 9x - 2y - 2z &= -3 \end{aligned}$ $\qquad \begin{bmatrix} 5 & 3 & -7 \\ -3 & 5 & 12 \\ 9 & -2 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \\ -3 \end{bmatrix}$
- To find *x,y,z* you have to "solve" the linear system. Do not use an inverse, but rely on a direct solver:

```
Matrix3f A;
Vector3f b;
A << 5,3,-7,  -3,5,12,  9,-2,-2;
b << 4, 9, -3;
cout << "Here is the matrix A:\n" << A << endl;
cout << "Here is the vector b:\n" << b << endl;
Vector3f x = A.colPivHouseholderQr().solve(b);
cout << "The solution is:\n" << x << endl;
```

2 Images

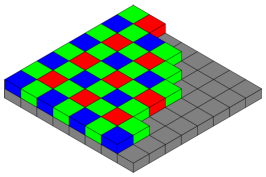
2.1 Vector devices and Transition to raster

The Cathode Ray Tube is a vector devices but the we have passed to raster. Scan pattern fixed in display hardware. Intensity modulated to produce image. Originally for TV (c continuous along signal), whereas for computer : discretized in the horizontal direction (matrix of pixels)



Output Raster devices : 2D (Display LCD , LED) , 1D (Hardcopy) Input Raster devices : 2D array(digital camera) , 1D array(scanner)

Bayesian Color-Filter



Pixel Values 1-bit greyscale - text , 8-bit RGB(24 bits) - web and email , 8 -bit RGBA (32 bits) - alpha channel, 16/24/32 bits - high accuracy for photography and HDR.

Monitors Intensity , Gamma Correction

What is the minimal and maximal light intensity?? THe intermediate intensities are different for each person , and it is non linear. Monitors needs to be calibrated for a certain viewer, using a procedure called **Gamme correction**. The rule is simple : displayed intensity = (max intensity) * a^γ . a^γ is a pixel value.

We have to find the neutral gray : $0.5 = a^\gamma$. Compute : $\gamma =$

$$\frac{ln0.5}{lna}$$

.

The colors will not be uniform on normal screens , one of the major factor affecting the cost of screens is their ability to be consistent on all pixels.

2.2 Alpha Compositing

A way to represent transparency. The pixels of an image are blendend linearly with the image below. c = $\alpha c_n ew$ + (1 - α)$c_o ld$

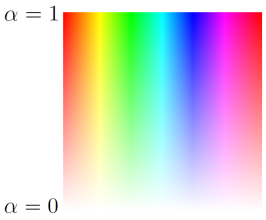
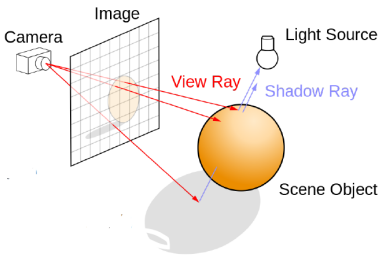


Image formats can be lossy as **jpeg** or lossless such as **png** which is common for web applications.

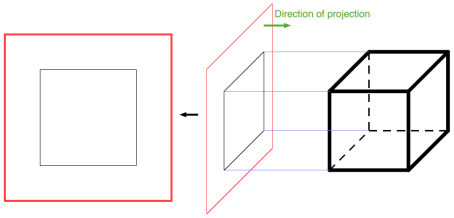
3 Ray Tracing

3.1 Basic RayTracing

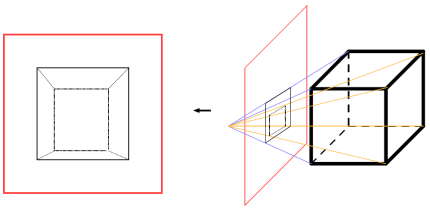
1. Generation of rays (one per pixel)
2. Intersection with objects in the scene
3. Shading (computation of the color of the pixel)



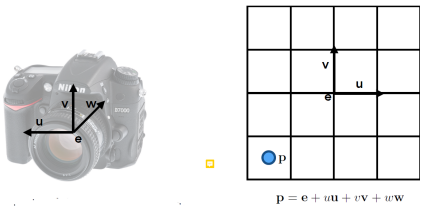
Projection - Parallel
Commonly used in modeling tools.



Perspective Projection
Each ray has a different direction.

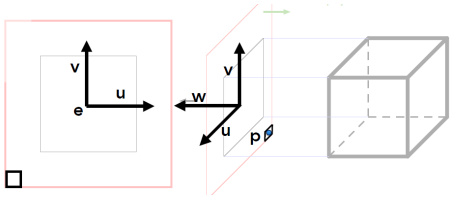


3.2 First Step :Compute Rays



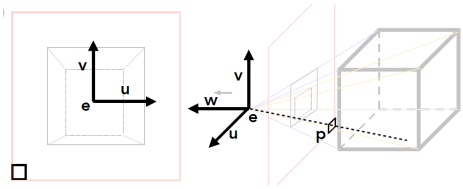
e is the origin of the reference system and p is the center of the pixel. u,v,w in italic are the position of the pixel p

Ortographic



For the ray assigned to pixel p is the Origin p and -w the Direction.

Perspective



For the ray assigned to pixel p : e is the Origin and p - e the Direction. (Ray in every direction)

3.3 Second Step : Intersections

This is an expensive operation and we will study two useful cases :

- Spheres
- Triangles (by combining triangles you can approximate complex surfaces)

Ray-Sphere Intersection
we have a ray in explicit form : $p(t) = e + td$.

And we have a sphere of Radius r and Center c in implicit form : $f(p) = (p-c)\cdot(p-c) - R^2 = 0$ This equation just gives us a condition that the point belong to the sphere $||p-c||^2 = R^2$. To find the intersection we need to find the solutions of $f(p(t)) = 0$

- Plug first equation into second: $(\mathbf{e}+t\mathbf{d}-\mathbf{c})\cdot(\mathbf{e}+t\mathbf{d}-\mathbf{c}) = R^2$

$$\underbrace{(\mathbf{d}\cdot\mathbf{d})}_{a}t^2 + \underbrace{2\mathbf{d}\cdot(\mathbf{e}-\mathbf{c})}_{b}t + \underbrace{(\mathbf{e}-\mathbf{c})\cdot(\mathbf{e}-\mathbf{c}) - R^2}_{c} = 0$$

$$t = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

$b^2-4ac<0$ no intersection
$b^2-4ac=0$ 1 intersection (tangent)
$b^2-4ac>0$ 2 intersections

Normal at p :

$$\frac{p-c}{||p-c||}$$

Ray-Triangle Intersection

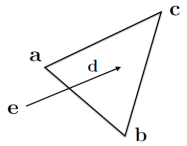
Explicit parametrization of a triangle with vertices a,b,c : $f(u,v) = a + u(b-a) + v(c-a)$

Explicit ray : $p(t) = e + td$

The ray intersects the triangle if a t,u,v exist : $f(u,v) = p(t)$ with $t > 0$ $0 \le u, v$ $u+v \le 1$

- Plug equations (1) and (2) into (3)

$$\mathbf{e}+\mathbf{d}t = \mathbf{a} + u(\mathbf{b}-\mathbf{a}) + v(\mathbf{c}-\mathbf{a})$$
$$(\mathbf{a}-\mathbf{b})u + (\mathbf{a}-\mathbf{c})v + \mathbf{d}t = \mathbf{a}-\mathbf{e}$$



- 3 equations with 3 variables: unique solution unless degenerate

$$\begin{bmatrix} (\mathbf{a}-\mathbf{b})_x & (\mathbf{a}-\mathbf{c})_x & \mathbf{d}_x \\ (\mathbf{a}-\mathbf{b})_y & (\mathbf{a}-\mathbf{c})_y & \mathbf{d}_y \\ (\mathbf{a}-\mathbf{b})_z & (\mathbf{a}-\mathbf{c})_z & \mathbf{d}_z \end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \begin{bmatrix} (\mathbf{a}-\mathbf{e})_x \\ (\mathbf{a}-\mathbf{e})_y \\ (\mathbf{a}-\mathbf{e})_z \end{bmatrix}$$

The solution of the system will give : Position - Intersection exists iff : $t > 0$ and $v \ge 0$ $u+v \le 1$ in the **Parallelogram** $u \le 1$ and $v \le 1$

The normal :

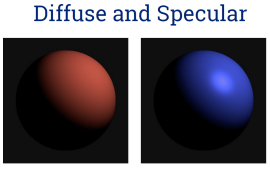
$$\frac{(b-a)x(b-a)}{(b-a)x(b-a)}$$

3.4 Third Step : Shading

Modeling accurately the behaviour of light is difficult and computationally expensive. We will use an approximation that is simple and efficient. It is divided in 3 parts :

- Diffuse Lambertian Shading
- Specular Shading
- Ambient Shading

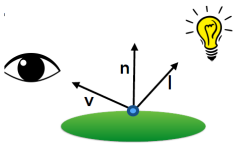
The three terms will be summed together to obtain the final color



Shading Variables

The shading depends on the entire scene, the light can bounce , reflect and be absorbed by anything that it encounters. We will simplify it so that it depends only on :

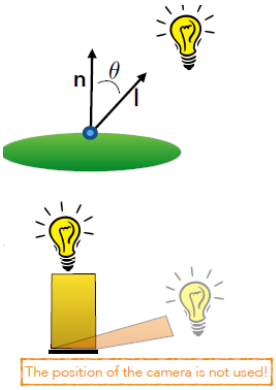
- The light Direction **l**(a unit vector pointing to the light source)
- The view direction **v** (a unit vector pointing toward the camera)
- The surface normal **n** (a vector perpendicular to the surface at the point of intersection)



Diffuse Shading

Lambert observed that the amount of energy from a light source that falls on an area of surface depends on the angle of the surface of the light. To model it , we make the amount of light proportional to the angle θ between the n and l. $L = k_d I max(0, n) \cdot l$, k_d is the diffuse coefficient and I is the intensity of the light.

Specular Shading

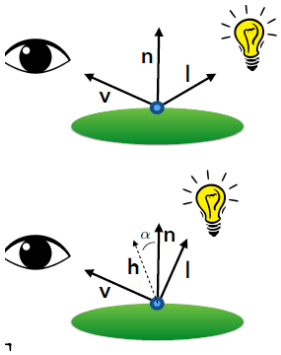


Specular highlights depend on the position of the viewer. The idea is to produce a reflection that is bright if v and l are symmetric wrt n. To measure the asymmetry we measure the angle between h (the bisector of v and l) and n.

h =

$$\frac{(v+l)}{v+l}$$

$L = k_d Imax(0, n) \cdot l + k_s Imax(0, n \cdot h)^p$, k_s is the specular coefficient and p is the Phong exponent. p = 100 Shiny , P = 1000 Glossy , p ¿ 10000 Mirror

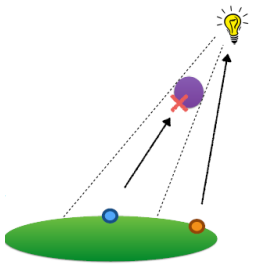


Final Shading Equation

L = $k_a I_a + k_d Imax(0, n \cdot l) + k_s Imax(0, n \cdot h)^p$ First part is ambient then diffuse and specular. If you have multiple light, simply sum them up all together. Note that the ambient light should be considered only once.

Shadows

The blue point does not receive light, while the orange one does. To check it , cast a ray from each point to the light , if you intersect (before reaching the light) then it is in a shadow area, and the light should not contribute to its color. These rays usually called **shadow rays**.



Ideal Reflections(Mirror Reflections) It is easy to add ideal reflections to your ray tracing program. $r = d - 2(d \cdot n)n$

4 Spatial Data Structures

Graphic applications often requires spatial queries.

- Find the k points closer to a specific point p
- Is object X intersection with object Y?
- What is the volume of the intersection between two objects

The data structure to use is application-specific.

We have two main ideas :

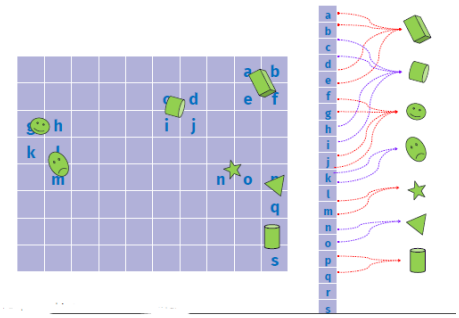
1. You can explicitly index the space itself (Spatial Index)
2. You can sort the primitives in the scene , which implicitly induces a partition of the space.

4.1 Spatial Indexing Structures

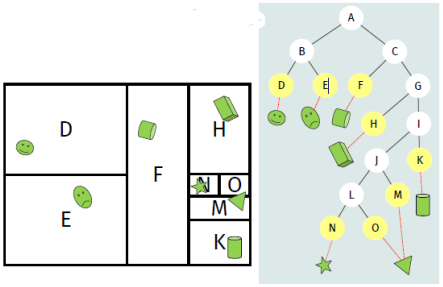
Data structures to accelerate queries of the kind : "I'm here. Which object is around me?". Tasks : 1) Construction / update, for static parts of the scene , a preprocessing. For moving parts of the scene , an update. 2) Access/usage

4.2 Regular Grid aka lattice

- Array 3D of cells : each cell contains a list of pointers to colliding objects.
- Indexing function : Point3D $\rightarrow$ cell index
- Construction $\rightarrow$ for each object B[i] find the cells C[j] which it touches and add a pointer in C[j] to B[i]
- Queries $\rightarrow$ Given a point to p, find cell C[j], test all objects linked to it
- Problem $\rightarrow$ Cell size , too small (memory occupancy too large) or too big(many objects in one cell)



4.3 kD-tree

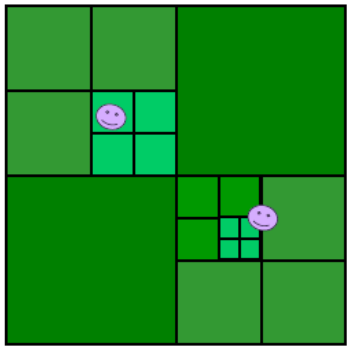


First we compute the enclosing space. Then we start partitioning by dividing it into two parts (or in the middle or to have the same number of objects in both parts). Now we keep splitting, we proceed this way until we have a maximum number of elements (in this case 1). Hierarchical structure : a tree. Each node

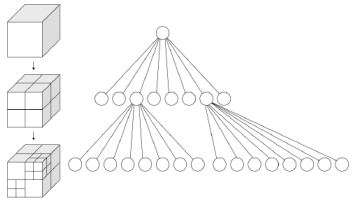
→ a subpart of the 3D space, root → all the world, child nodes → partitions of the father, objects linked to the leaves. kD-tree : binary tree. Each node →

split over one dimension

4.4 Quad-Tree (2D) and Oct-Tree(3D)



The space is subdivided in a regular way. Split into four parts in of a 2D and then for each quadrant that is not empty you split again (you will decide when you want to stop splitting



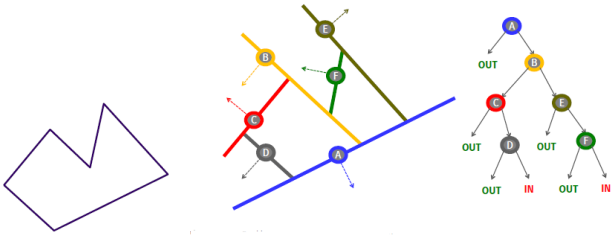
They are similar to kD-trees, but : tree → branching factor (4(2D) or 8(3D), each node →splits into all dimensions at once The construction : continue splitting until a end nodes has few enough objects.

4.5 Binary Spatial Partitioning tree

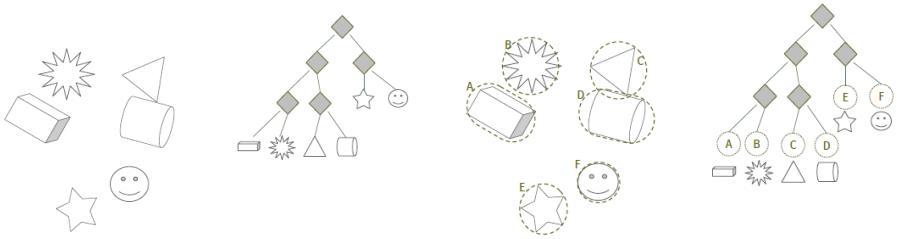
Equation of the plane : $f(x, y, z) = ax + by + cz = 0$. $P(x_p, y_p, z_p)f(p) = 0$ the point is on the plane if ¿0 it is in a part of the plane if ¡ 0 it is in the other part of the plane

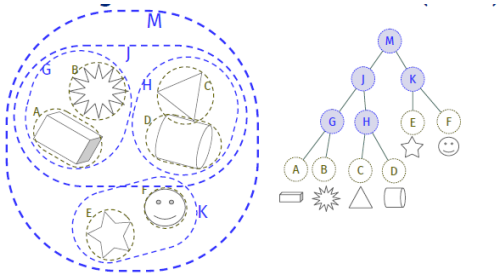
BSP-trees for the Concave Polyhedron proxy

A binary tree but each node is split by an arbitrary plane , plane is stored at node , as(nx,ny,nz,k). There is another use to test polyhedron proxy : note with planes defined in its object space, each leaf is inside or outside



4.6 Primitive Sorting Structures - Bounding Volume Hierarchies





The idea is to use the scene hierarchy by the scene graph (instead of a spatial derived one). Associate a bounding volume to each node (rule : a BV of a node bounds all objects in the subtree). The construction / update is fast (bottum-up : recursive)

4.7 Pro and Cons of Spatial Indexing Structures

- Regular Grid → The most parallelizable , constant time access, quadratic/cubic space
- kD-tree,Oct-tree,Quad-tree → compact,simple, non costant accessing time
- BSP-tree → optimazed splits, best performance when accessed, ideal for static parts of the scene, optimazed splits, more complex construction

- BVH → simplest construction, ideal for dynamic parts of the scene, non necessarily very efficient

5 Intersection Acceleration Data Structures

5.1 Collision Detection

It is easy to do , the challenge is to do it efficiently. Most pair of objects do not intersect each other in a scene because collisions are rare. Optimizing the intersections directly is important but not sufficient, we need to optimize the direction of non intersecting pairs.

5.2 Geometric Proxy

Extremely coarse approximation. Used as a Bounding volume : the entire object must be contained inside , exact result, you need to do more work if you detect a collision. Used as a Collision Object or Hit-Box : approximation of the object. No need to do anything else if an approximation is ok for your use case. Ex : Fighting Games

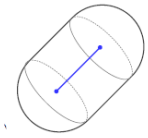
5.3 Geometry Proxies : Sphere

It's easy to compute and update , compact and very efficient collision test. But it can only be transformed rigidly and the quality of the approximation is low.

5.4 Geometry Proxies : Capsule

Def : Sphere == set of all points with dist from a point ¡ radius. Capsule == set of all points with dist from a segment ¡ radius .

Store with a segment (two end-points) and a radius (a scalar)

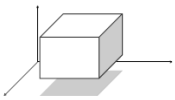


5.5 Geometry Proxies : Half Space

Trivial but useful, for example for a flat terrain or a wall. Storage : (nx,ny,nz,k), a normal (a distance from the origin)

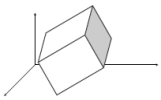
5.6 Geometry Proxies : Axis-Aligned Bounding Box (AABB)

It's easy to update , compact , and trivial to test but it can only be translated or scaled and rotations are not supported.



5.7 Geometry Proxies : Box

Similar to AABB, but not axis aligned. It's more expensive to compute and store and you need intervals and a rotation. Still not a great approximation

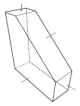


5.8 Geometry Proxies (in 2D): Convex Polygon

Intersection of half-planes each delimited by a line. Stored as : a collection of oriented lines. A point is inside iff it is in each half-plane, it has a moderate complexity

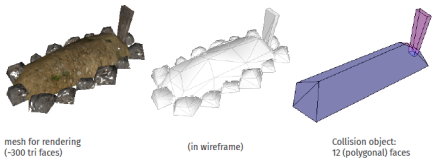
5.9 Geometry Proxies (in 3D): Convex Polyhedron

Intersection of half-spaces, it is similar to the previous but in 3D. Stored as a collection of planes. Each plane is a normal + distance from origin. Inside proxy iff inside each half-space



5.10 3D Meshes as Hit-Boxes

These are often NOT the mesh that you use for rendering because : much lower resolution , no attributes , closed , often convex only and can be polygonal



5.11 Geometry Proxies : Composite Hit-Boxes

Union of Hit-Boxes. Inside iff insied of any sub Hit-Box. Union of convex HIt-Boxes (Concave Hit.Box). Shape partially defined by a sphere, they are created typically by hand.

5.12 Collision Detection Strategies

Static Collision Detection. A posteriori, discrete, approximated, simple and quick

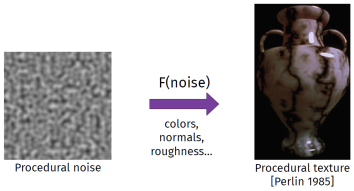
Dynamic Collision detection, A priori , accurate, demanding

6 Procedural Synthesis

6.1 Procedural Noise

6.1.1 We start with noise functions

Goal : create realistic "textures" at inexpensive costs.



6.1.2 What is a good noise function?

Randomly controlled primitive F(p) : smooth function + salt.

6.1.3 Noise Requirements

We want noise with **controllable appearance**. Other desirable noise properties : Compact , Continuous , Non-periodic , Fast.

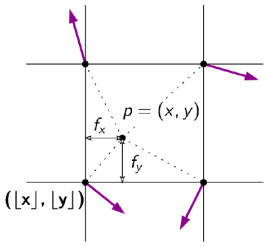
Applied to 3D objects : map to surface or 3D function

6.1.4 Perlin Noise

Based on a regular lattice , with a 2D random vector v defined on every corner.

Algorithm :

1. Given a point p in 2D find the 4 lattice corners c_1, c_2, c_3, c_4
2. Compute v(c_i) · (p-c_i)
3. Return the bilinear interpolation of the dot products evaluated at p



Source code at 6.9 and 6.10

6.1.5 Why are procedural textures popular?

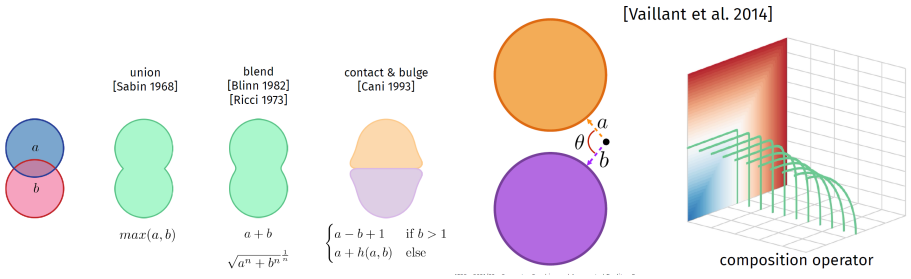
No needs to store them , Reduce GPU transfer for rasterization and Reduce scene size for ray tracing.

They can be evaluated at any point. Computation-bound instead of memory-bound. Infinite resolution.

6.2 Impicit Modeling

S = x $\in R^n$ — f(x) = d

Distance field = it tells how much we are distant from the point



6.2.1 Implicit Model

All points where f(p) = 0

f : $\mathbb{R}^3 \rightarrow \mathbb{R}$

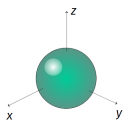
it directly defines an inside and outside :

- f(p) < 0 → p inside
- f(p) > 0 → p outside
- f(p) = 0 → p on the surface

By construction , it defines a closed watertight model.

6.2.2 Sphere

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = x^2 + y^2 + z^2 - R^2$$



6.2.3 Categories

- **Algebraic surfaces** : f() is polynomial
- **Quadratic surfaces** : f() degree is 2, simple equations have good expressive power
- **Cubic surfaces**: f() degree is 3
- Higher order

6.2.4 Implicit Modeling : Pros and Cons

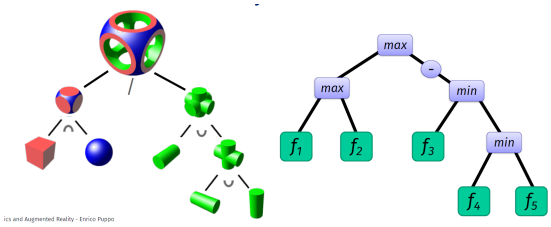
The pros are : compact , CSG , Good model for both fluids and solids and easy to render with ray tracing. Cons : difficult to render for rasterization-based pipelines

6.2.5 Implicit Solid Modeling

Let A and B be two solid objects described implicitly by implicit functions f_A and f_B. We can define :

- **Complement** : -f_A
- **Intersection** : max(f_A , f_B)
- **Union** : min (f_A,f_B)
- **Subtraction** : max(f_A,f_B)

6.2.6 Constructive Solid Geometry (CSG) and Geometric Solid Modelling



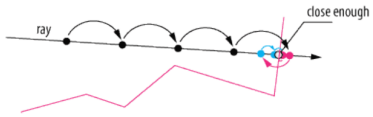
6.3 Rendering Implicits Ray Marching

6.3.1 Ray Marching

Similar to ray tracing. Since the implicit function might not be quadratic , we cannot find the intersection explicitly as we did for triangles and spheres.

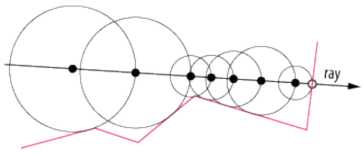
Algorithm 1

The simplest algorithm resembles gradient descent. You proceed on the ray by a fixed amount. You start to do bisection search if you get closer than a given to the surface.



Algorithm 2 : "Sphere Tracing"

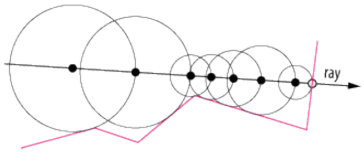
Instead of taking a constant step , you check the closest point on the surface , and you move by that amount. Can be done exactly with a distance field. It can be conservative estimate for more general cases.



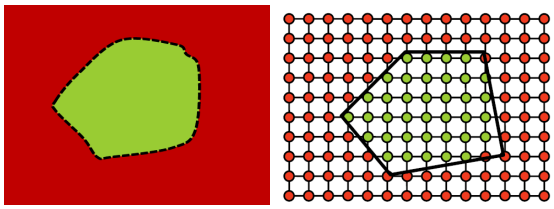
6.4 Rendering Implicits Explicit Meshing

6.4.1 Extracting the Surface

Wish to compute a manifold mesh of the level set

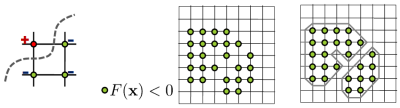


6.4.2 Sample the SDF



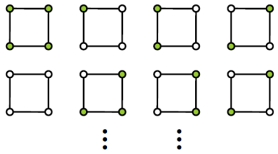
6.4.3 Tessellation

Want to approximate an implicit surface with a mesh. Can't explicitly compute all the roots (sampling the level set is difficult) Solution : find an approximate roots by trapping the implicits surface in a grid (lattice)



6.4.4 Marching Squares

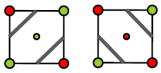
16 different configurations in 2D. 4 equivalence classes (up to rotational and reflection symmetry + complement)



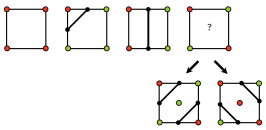
6.4.5 Tessellation in 2D

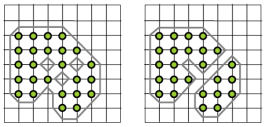
4 equivalence classes (up to rotational and reflection symmetry + complement)

- Case 4 is ambiguous:



- Always pick consistently to avoid problems with the resulting mesh

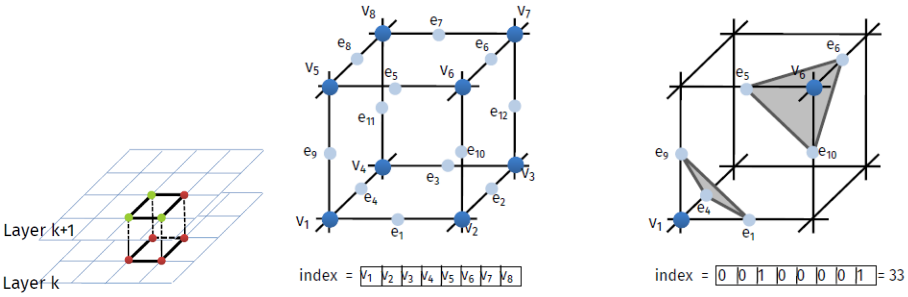




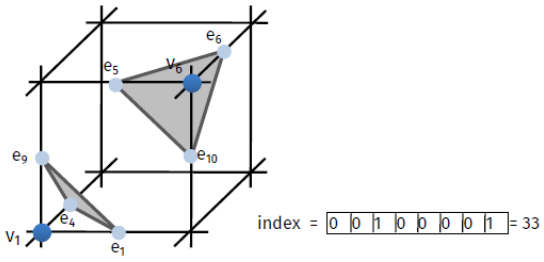
6.4.6 Marching Cubes

1. Load 4 layers of the grid into memory
2. Create a cube whose vertices lie on the two layers
3. Classify the vertices of the cube according to the implicit function (inside,outside or on the surface)

4. Compute case index . We have $2^8 = 256$ cases (0/1 for each of the eight vertices) - can store as 8 bit (1 byte) index

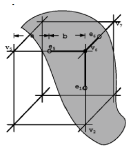


5. Using the case index , retrieve the connectivity in the look-up table. For example : the entry for index 33 in the look-up table indicates that the cut edges $e_1; e_4; e_5; e_6; e_9; e_10$; the output triangles are $(e_1; e_9; e_4)$ and $(e_5; e_10;_e 6)$



6. Compute the position of the cut vertices by linear interpolation

$$\mathbf{v}_s = t\mathbf{v}_a + (1-t)\mathbf{v}_b$$
$$t = \frac{F(\mathbf{v}_b)}{F(\mathbf{v}_b) - F(\mathbf{v}_a)}$$

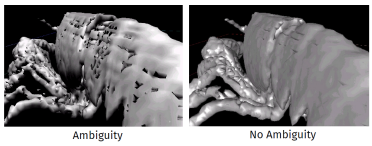


7. Move to the next cube

Problems

Have to make consistent choices for neighboring cubes - otherwise get holes. Resolving ambiguities.

Grid not adaptive. Many polygons required to represent small features.



7 Transformation

7.1 2D Linear Transformations

Each 2D linear map can be represented by a unique 2x2 matrix

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Concatenation of mappings corresponds to multiplication of matrices : $L_2(L_1(x)) = L_2 L_1 x$

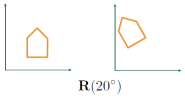
Linear transformations are very common in computer graphics

7.1.1 2D Scaling

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}}_{\mathbf{S}(s_x, s_y)} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

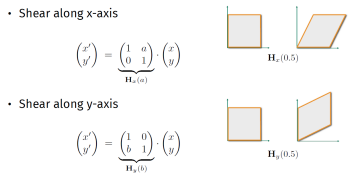
7.1.2 2D Rotation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}}_{\mathbf{R}(\alpha)} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

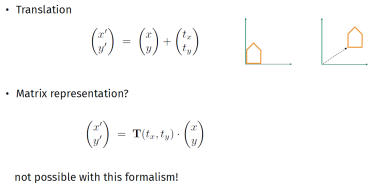


Special case: $\mathbf{R}(90) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$

7.1.3 2D Shearing



7.1.4 2D Translation



7.2 Affine Transformations

Translation is not a linear transformation , but is affine. Origin is no longer a fixed point.

Affine map = linear map + translation.

Is there a matrix representation for affine transformations?

We would like to handle all transformations in a unified framework and it is simpler to code and easier to optimize

7.2.1 The Affine Algebra

The constituents of the affine algebra belong to three classes :

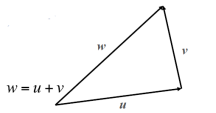
- Points define objects (characterized just from a position in space)
- Vectors define displacement (characterized from direction , orientation , norm)
- Scalars define the amount of displacement (characterized just from a value)

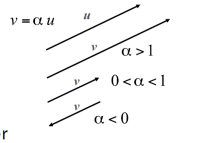
7.2.2 Line in the affine space

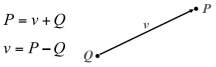
Parametric representation of a line : P(α) = P_0 + α d

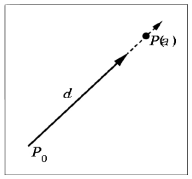
Given point P0 and vector d , points of the straight line parallel to d and passing through P0 are obtained varying scalar parameter α

- Operations:
  - Usual arithmetic between scalars
  - Sum vector + vector gives another vector
  - Product scalar · vector changes the norm (and the orientation, if the scalar is negative) of the vector
  - Sum point + vector (displacement) gives another point
  - Difference point - point gives a vector









7.2.3 Affine sum

In the affine space there exist neither sum between points nor product between a scalar and a point.

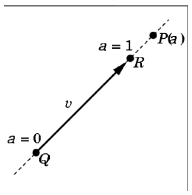
But there exists the affine sum :

v = R - Q

P = Q + av

P = Q + a(R - Q) = a R + (1 - a)Q

The affine sum allows us to linearly interpolate between points P and Q.



7.2.4 The affine space

Geometric interpolation : 2 planes in R^3

Vectors live in the plane through the origin. Points live in a parallel plane

7.2.5 Homogenous Coordinates

Add a third coordinate :

2D point $= (x, y, 1)^T$ 2D vector $= (x, y, 0)^T$

Now we can give a matrix representation of translations :

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

Valid operation if the resulting w-coordinate is 1 or 0

- vector + vector = vector
- point - point = vector
- point + vector = point
- point + point is undefined

Affine Transformations

- Affine map = linear map + translation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

- Using homogenous coordinates:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

2D Transformations

- Scale

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

- Rotation

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

- Translation

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

3D Transformations

Add one more row and column

- Scale

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

- Translation

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

3D Transformations

Rotations about coordinate axes

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

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

Generic rotations can be obtained as compositions of the above

7.2.6 Concatenation of Transformations

Sequence of affine maps A_1, A_2, A_3

Concatenation by matrix multiplication :
$A_n(...A_2(A_1(x))) = A_n \cdot A_2 \cdot A_1 \cdot (x, y, 1)^T$

7.3 Rotation and Translation

Matrix multiplication is not commutative!

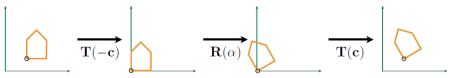
- First rotation, then translation

- First translation, then rotation

7.3.1 2D Rotation

How to rotate around a given point c?

1. Translate c to origin
2. Rotate
3. Translate back



Matrix representation?
T(c) · R(a) · T(-c)

7.3.2 3D rotation about the origin

- General form: orthogonal matrix

$$\mathbf{R} = \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} & 0 \\ r_{yx} & r_{yy} & r_{yz} & 0 \\ r_{zx} & r_{zy} & r_{zz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- where (r_{xx}, r_{xy}, r_{xz}) (r_{yx}, r_{yy}, r_{yz}) (r_{zx}, r_{zy}, r_{zz}) are all unit-length and orthogonal among them
- We have $\mathbf{R}^{-1} = \mathbf{R}^T$ and $|\mathbf{R}| = 1$

- Row vectors of matrix **R** transformed through **R** will coincide with the coordinate axes

$$\mathbf{R}\begin{bmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{R}\begin{bmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{R}\begin{bmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

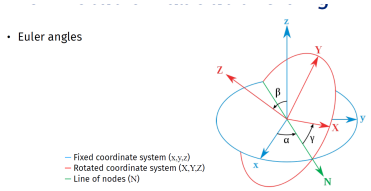
- If we know at least two orthogonal vectors that must be rotated to coincide with coordinate axes, we can compute matrix **R** explicitly

Any rotation can be expressed as the combination of three rotations about coordinate axes.

Therefore : given R about the origin , there always exists R_x, R_y, R_z rotation matrices about the three coordinate axes such that R $= R_z R_y R_x$

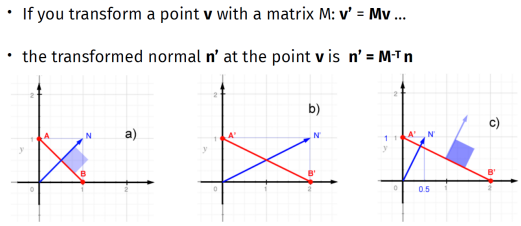
The order of the three rotations is not unique , but the resulting matrix is unique.

Problem : how can we find the three matrices? not easy

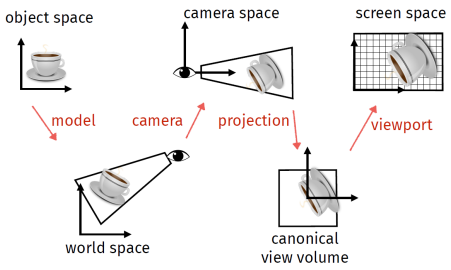


Euler's rotation theorem : any 3D rotation can be expressed as a single rotation about some axis

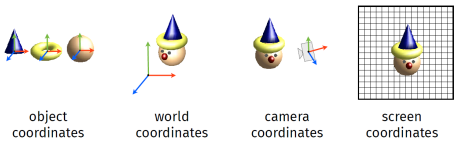
7.3.3 A note on transforming normals



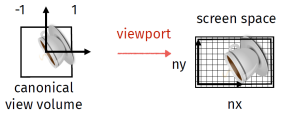
8 Viewing Transformations



8.1 Coordinate Systems



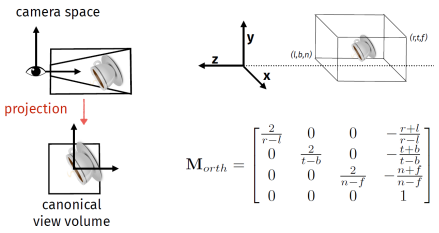
8.2 Viewport transformation



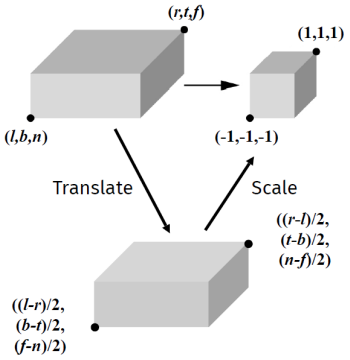
$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ 1 \end{bmatrix} = \begin{bmatrix} nx/2 & 0 & \frac{nx-1}{2} \\ 0 & ny/2 & \frac{ny-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{canonical} \\ y_{canonical} \\ 1 \end{bmatrix}$$

How does it look in 3D?

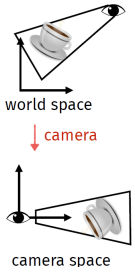
8.3 Ortographic Projection



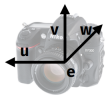
$$\mathrm{M}_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



8.4 Camera Transformation



1. Construct the camera reference system given:
   1. The eye position **e**
   2. The gaze direction **g**
   3. The view-up vector **t**

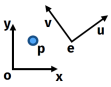


$$\mathbf{w} = -\frac{\mathbf{g}}{||\mathbf{g}||}$$
$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{||\mathbf{t} \times \mathbf{w}||}$$
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

2. Construct the unique transformations that converts world coordinates into camera coordinates

$$\mathbf{M}_{cam} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

8.5 Change of frame



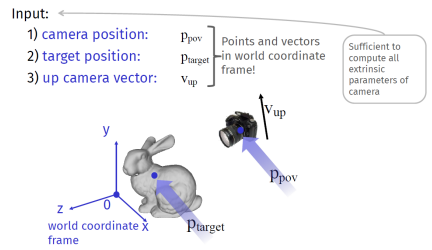
$$\mathbf{p} = (p_x, p_y) = \mathbf{o} + p_x\mathbf{x} + p_y\mathbf{y}$$
$$\mathbf{p} = (p_u, p_v) = \mathbf{e} + p_u\mathbf{u} + p_v\mathbf{v}$$

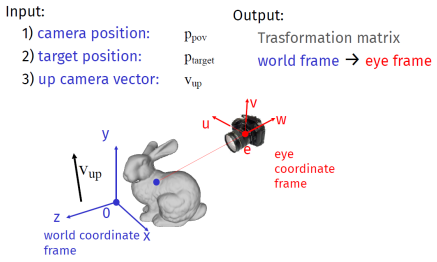
$$\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & e_x \\ 0 & 1 & e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & 0 \\ u_y & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & e_x \\ u_y & v_y & e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix}$$

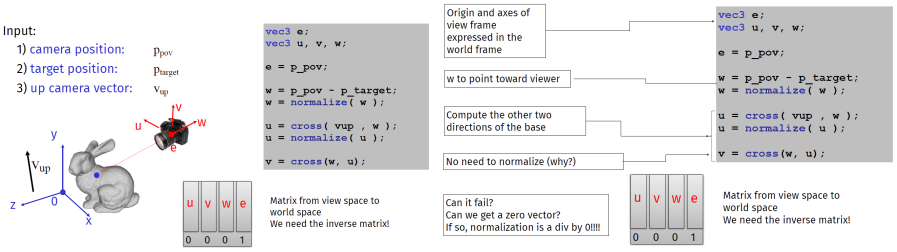
$$\mathbf{p}_{xy} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{uv} \qquad \mathbf{p}_{uv} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{p}_{xy}$$

Can you write it directly without the inverse?

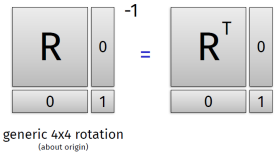
8.5.1 Camera transformation : typical example

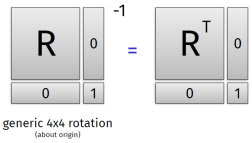






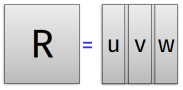
8.6 Reminder : inverting a rotation



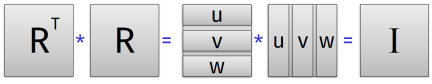


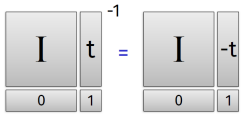
where:

R 3x3 rotation,
i.e., orthonormal special,
i.e., u, v, n:
- unit length
- mutually orthogonal

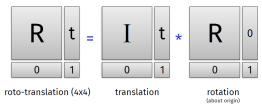


proof (sketch):

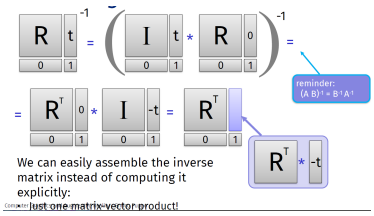




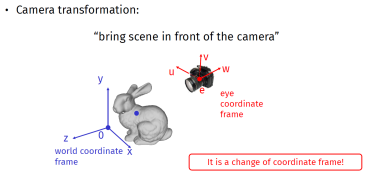
8.7 Roto-translation (isometry)



8.8 Inverting a roto-translation



8.9 View transformation : summary

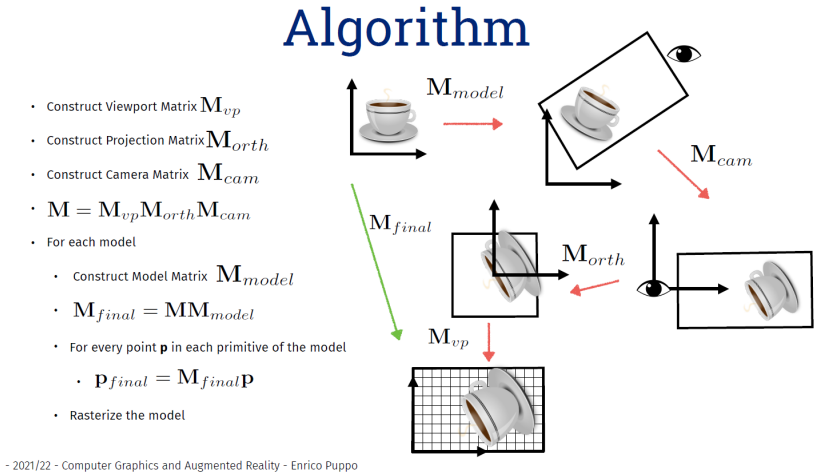


The view transformation needed to place the camera at a given position is given by the inverse matrix of the modeling transformation needed to place an object at the same position.

Two symmetrical ways to see placement : Operation that moves the camera with respect to the scene and operation that moves the world with respect to the camera.

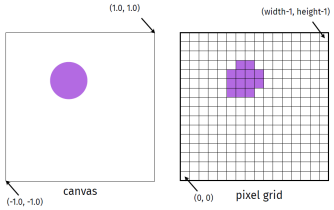
First way corresponds to a change of reference frame

Second way corresponds to a further modeling transformation of the whole scene (distinction between global modeling transformation and viewing transformations to place the camera can be blurred).



9 Rasterization - Theory

9.1 2D Canvas



9.2 Implicit Geometry Representation

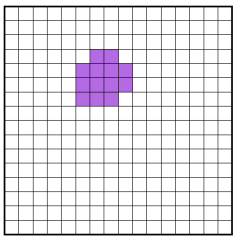
Define a curve as a zero set of 2D implicit function

- F(x,y) = 0 : on curve
- F(x,y) < 0 : inside curve
- F(x,y) > 0 : outside curve

Example : Circle with center (c_x, c_y) and radius r
$F(x,y) = (x - c_x)^2 + (y - c_y)^2 - r^2$

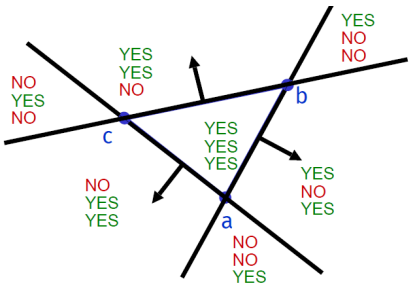
9.3 Implicit Rasterization

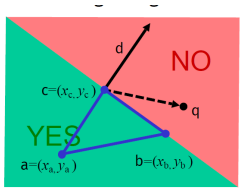
```
for all pixels (i,j)
    (x,y) = map_to_canvas (i,j)
if F(x,y) < 0
    set_pixel (i,j, color)
```



9.4 Point-in-triangle test

Triangle = intersection of 3 half-planes.





$d = (-(y_b - y_c), x_b - x_c)$

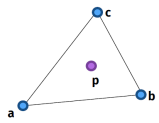
$f(q) = (q - c) \cdot d$

Only the sign of f is relevant!

EDGE FUNCTION

9.5 Barycentric Interpolation

- Barycentric coordinates:
- **p** = α**a** + β**b** + γ**c** with α + β + γ = 1
- Unique for non-collinear **a**,**b**,**c**



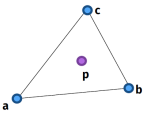
$$\begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

- Barycentric coordinates:
- **p** = α**a** + β**b** + γ**c** with α + β + γ = 1
- Unique for non-collinear **a**,**b**,**c**
- Ratio of triangle areas

$$\alpha(p) = \frac{\text{area}(p, b, c)}{\text{area}(a, b, c)}$$

$$\beta(p) = \frac{\text{area}(p, c, a)}{\text{area}(a, b, c)}$$

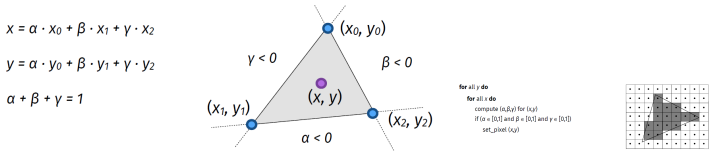
$$\gamma(p) = \frac{\text{area}(p, a, b)}{\text{area}(a, b, c)}$$



inside iff alpha,beta,gamma >= 0 and the area is = 1/2 (b-a) * (c-a)

9.6 Triangle Rasterization

Each triangle is represented as three 2D points (x_0, y_0),(x_1, y_1),(x_2, y_2).
Rasterization using barycentric coordinates



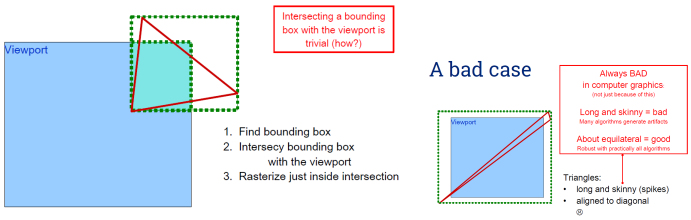
Barycentric interpolation uses barycentric coordinates to interpolate vertex normals (or other data like colors)

$n(P) = \alpha \cdot n(A) + \beta \cdot n(B) + \gamma \cdot n(C)$

9.7 Clipping

Start with a scene and a camera , we don't want to rasterize part of the object which is outside the canvas.

It's ok if you do it brute force. Care is required if you are explicitly tracing the boundaries.



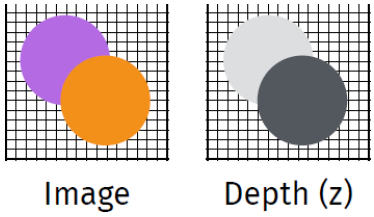
9.8 Objects Depth Sorting

Objects can occlude other objects behind them. To handle occlusion , you can sort all the objects in a scene by depth but this is not always possible!

9.8.1 z - buffering

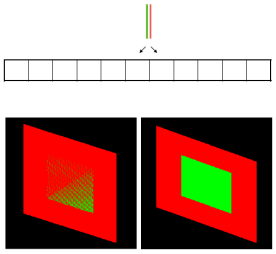
You render the image both in the image and in the depth buffer , where you store only the depth.

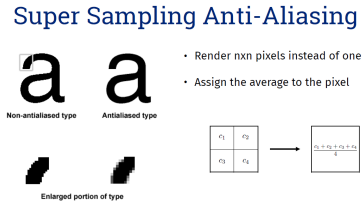
When a new fragment comes in , you draw it in the image only if it is closer. This always work and it is cheap to evaluate! It is the default in all graphics hardware



The z-buffer is quantized (the number of bits is heavily dependent on the hardware platform)

Two close object might be quantized differently , leading to strange artifacts , usually called "z-fighting".





10 Rasterization - Implementation

How to do it ? With specialized hardware (GPU). There are APIs to interact with the hardware (OpenGL,DirectX,Metal).

Before you can draw anything you need to : open a window and initialize the API and assign the available screen space to it. This is technical step and it is heavily dependent on the operating system and on the hardware.

There are many libraries that take care of this for you , hiding all the complexity and providing a cross-platform and cross-hardware interface. A window manager usually provides an event management system.

Writing code for the GPU makes debugging harder , since the standard debugging tools cannot be used.

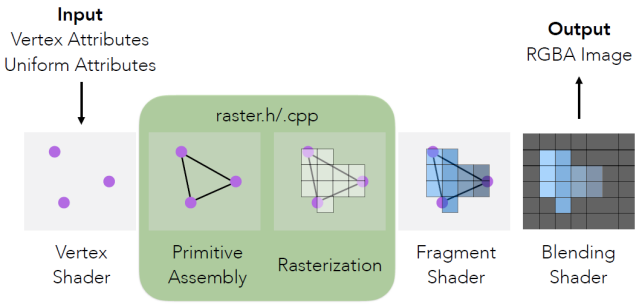
The code needs to be customized for the specific operating system you are developing for. The good news is that all modern APIs , at a high-level , offer the same concepts and the same features

10.1 Software Rasterization

10.1.1 raster.h/raster.cpp

The rasterizer mimics the API of modern OpenGL. It is minimalistic , supporting only rendering of triangles and lines.

10.1.2 Rasterization Pipeline



10.1.3 Vertex Input

You have to send the rasterizer a set of vertex attributes :

- Standardized Coordinates
- Color
- Normal

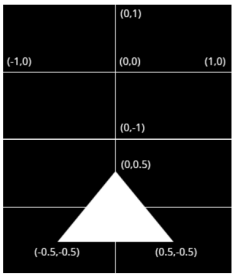
You can pass as many as you want , but remember that memory/bandwidth is precious , you want to send only what is required by the shaders.

- Only the triangles in the canonical cube will be rendered. The cube will be stretched to fill all available screen space.

```
vector<VertexAttributes> vertices;
vertices.push_back(VertexAttributes(-0.5,-0,5));
vertices.push_back(VertexAttributes(0.5,-0.5,0));
vertices.push_back(VertexAttributes(0,0.5,0));

rasterize_triangles(program,uniform,vertices,frameBuffer);

vector<uint8_t> image;
framebuffer_to_uint8(frameBuffer,image);
stbi_write_png("triangle.png", frameBuffer.rows(), frameBuffer.cols(), 4,
image.data(), frameBuffer.rows()*4);
```



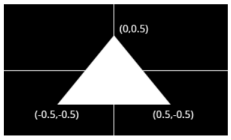
10.1.4 Shaders

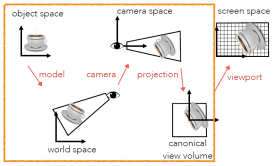
The name is historical - they were introduced to allow customization of the shading step ot the traditional graphics pipeline. Shaders are **general purpose functions** that will be executed in parallel on the GPU. They are usually written in a custom programming language , but in our case they will be standard C++ functions. They allow to customize the behavior of the rasterizer to achieve a variety of effects.

10.1.5 Vertex Shaders

The vertex shader is a function processing each vertex as they appear in the input. Its duty is to output the final vertex position in the canonical bi-unit cube and to output any data required by the fragment shader. All transformations from **world to device coordinates** happen here.

```
vector<VertexAttributes> vertices;
vertices.push_back(VertexAttributes(-0.5,-0,5));
vertices.push_back(VertexAttributes(0.5,-0.5,0));
vertices.push_back(VertexAttributes(0,0.5,0));
```



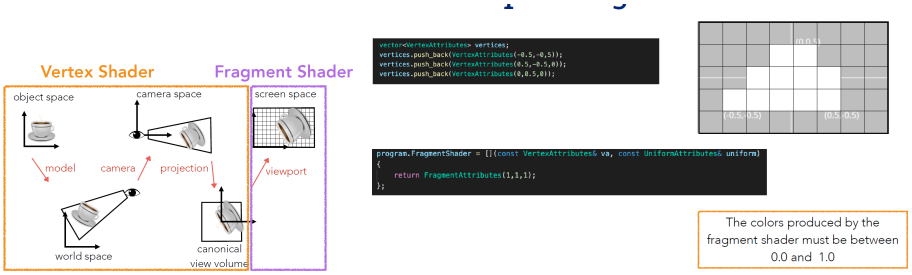



```
// The vertex shader is the identity
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform)
{
    return va;
};
```

The syntax [] defines a lambda function.

10.1.6 Fragment Shader

The output from the vertex shader is interpolated over all the pixels on the screen covered by a primitive. These pixels are called fragments and this is what the fragment shaders operates on. It has one mandatory ouptut , the final color of a fragment. It is up to you to write the code for computing this color from all the attributes that you attached to the vertices.



10.1.7 Blending Shader

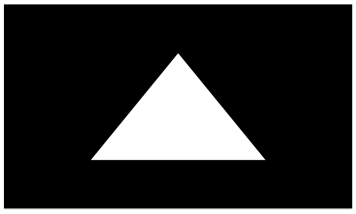
The blending shader decides how to blend a fragment with the colors/attributes already present on the corresponding pixel of the framebuffer. If the framebuffer uses bytes to represent the colors , the blending shader performs the conversion from float to byte.

After specifying the 3 shaders (vertex,fragment,blending) we can rasterize the vertices and the framebuffer can be saved as a png as we did for raytracing.

The rasterizer uses an additional file called attributes.h to define the struct used for vertex , fragment , and framebuffer attributes.

You will have to change this depending on what information you want your shader to have access to.

10.1.8 If everything was done correctly



Let's take a look at the complete source code.

Our current pipeline does not consider view transformations!

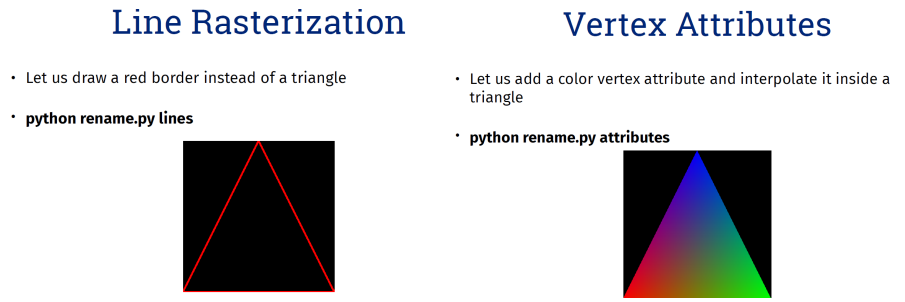
(Recommended Exercise: how would you change the current shaders to take the size of the window into account and always draw a equilateral triangle?)

10.1.9 Uniforms

Uniform are values that are constant for the entire scene , they are not attached to vertices.

They are essentially global variables within the shaders. All the vertices and all fragments will see the same value. For example , let's change the demo code to use a uniform to store the triangle color.

10.2 Software Rasterization Examples



10.2.1 View/Model Transformation

View/Model transformations are applied in the vertex shader. They are passed to the shader as uniform. To create transforation matrices you can do it by hand or use the geometry module of Eigen

10.2.2 How to prevent viewport distortion?

We need to adapt the view depending on the size of the framebuffer. We can then create a view transformation that maps a box with the same aspect-ratio of the viewport into the unit cube. Equivalently , we are using a camera that has the same aspect ratio as the window that we use for rendering.

In this way , the distortion introduced by the viewport transformation will cancel out.

10.2.3 Tests

The tests determine if a fragment will affect the color in the framebuffer or if it should be discarded : there are main uses cases.

Depth Test - Discards all fragments with a z coordinate bigger than the value in the depth buffer.

Stencil Test - Discards all fragments outside a user-specified mask

Depth Test

- Let's draw a couple of triangles with depth test active
- **python rename.py depth**

Blending

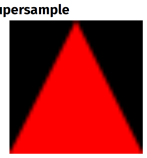
- Let's see how to render a transparent triangle
- **python rename.py depth**

Animation

- Let's see how to render a simple animation
- **python rename.py animation**

Supersampling

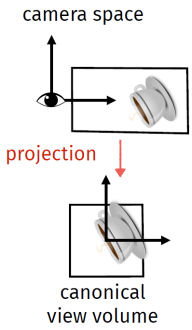
- We can render an image 4 times larger and then scale it down to avoid sharp edges
- **python rename.py supersample**

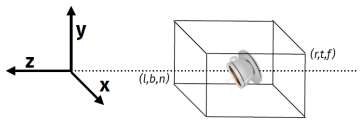


To interact with the scene , it is common to pick or select objects in the scene.

The most common way to do it is to cast a ray , starting from the point where the mouse is and going inside the screen.

The first object that is hit by the ray is going to be the selected object





$$\mathbf{M}_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

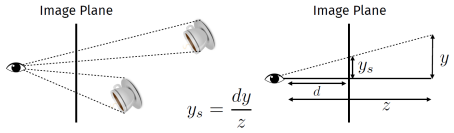
11 Perspective Projection

11.1 Projective Transformations

11.1.1 Ortographic Prjection

11.1.2 Perspective Projection

In Ortographic projection , the size of the objects does not change with distance.
In Perspective projection , the objects that are far away look smaller



11.1.3 Divisions in Matrix Form

We would like to reuse the matrix machinery that we built in the previous lectures.

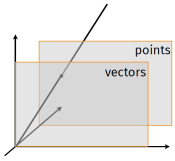
How do we encode divisions? $y_s = dy/z$

We extend homogeneous coordinates

11.1.4 The affine space

Geometric interpolation : 2 planes in R^3

Vectors live in the plane through the origin. Points live in all the rest (affine subspace) Points on a line through the origin are identified (projected to plane w = 1)



11.1.5 Affine transformations until now and Intuition

- What do we have left?

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_1x + b_1y + c_1 \\ a_2x + b_2y + c_2 \\ 1 \end{pmatrix}$$

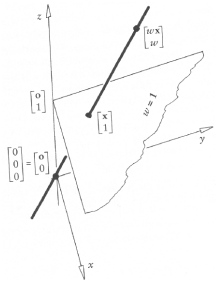
- We can use the last row of the transformation:

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ e & f & g \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_1x + b_1y + c_1 \\ a_2x + b_2y + c_2 \\ ex + fy + g \end{pmatrix} \sim \begin{pmatrix} \frac{a_1x+b_1y+c_1}{ex+fy+g} \\ \frac{a_2x+b_2y+c_2}{ex+fy+g} \\ 1 \end{pmatrix}$$

- Purely algebraic:

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} \sim \begin{pmatrix} x/w \\ y/w \\ 1 \end{pmatrix}$$

- Or as a projection, where each line is identified by a point on the plane z=1
- Note that in this case, you can think of it as a transformation in a space with one more dimension

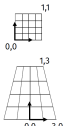


A transformation of this form is called a **projective transformation**. The points are represented in homogenous coordinates :

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ e & f & g \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_1x + b_1y + c_1 \\ a_2x + b_2y + c_2 \\ ex + fy + g \end{pmatrix} \sim \begin{pmatrix} \frac{a_1x+b_1y+c_1}{ex+fy+g} \\ \frac{a_2x+b_2y+c_2}{ex+fy+g} \\ 1 \end{pmatrix}$$

Example

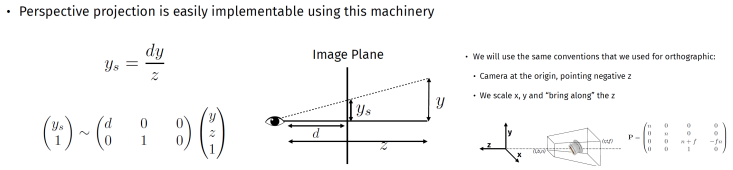
$$M = \begin{pmatrix} 2 & 0 & -1 \\ 0 & 3 & 0 \\ 0 & 2/3 & 1/3 \end{pmatrix}$$



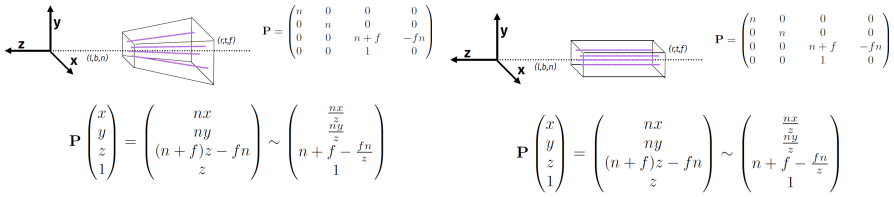
- It transforms a square into a quadrilateral — note that straight lines are preserved, but parallel lines are not!
- Note that you can use homogeneous coordinates for as many transformations as you want, only when you need the Cartesian representation you have to normalize

11.1.6 Perspective Projection

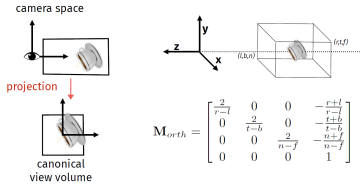
Perspective projection is easily implementable using this machinery :



11.1.7 Effect on the points

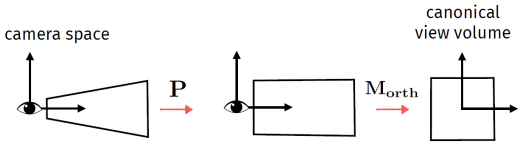


11.1.8 Ortographic Projection



11.1.9 Complete Perspective Transformation

$$\mathbf{P} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad \mathbf{M}_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



11.1.10 Parameters

How to set the parameters of the transformation?

If we look at the center of the window then the barycenter of the back plane should be at(0,0,f). If we want no distortion on the image we need to keep a fixed aspect ration : width / height = r/t .

There is only one degree of freedom left , the field of view angle θ. $tan\theta/2 = t/|n|$

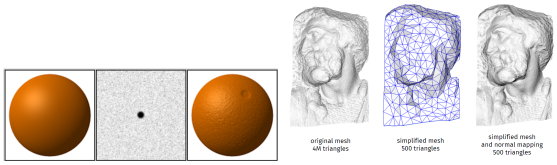
The parameters can thus by found by fixing n and θ. You can compute t and consequently all the other parameters needed to construct the transformation

It is important to convert to Cartesian coordinates before interpolating attributes such as positions.

12 Texture Mapping

12.0.1 Bump Mapping

Instead of encoding colors in a texture , you encode normals!



12.0.2 Texture Mapping

The idea is the same. Instead of encoding values at vertices of triangles, you encode them in images.

You gain all the advantages of images.

You can encode any property that you want , the most common are : colors (Texture Mapping) , Normals (Bump Mapping) , Displacements (Displacement Mapping)

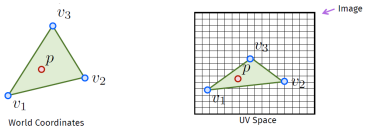
What do we need?

One additional per-vertex property , the UV coordinates.

An image uploaded to the GPU memory.

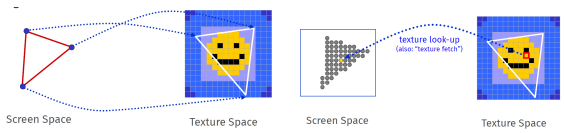
The UV coordinates are interpolated inside each triangle , and used to find the corresponding value in the texture.

The texture value is interpolated before it is used in the shader



12.0.3 UV Mapping

In practice we are defining a mapping between a 3D triangle and a texture triangle. Each **fragment** has its own coordinates u,v in texture space

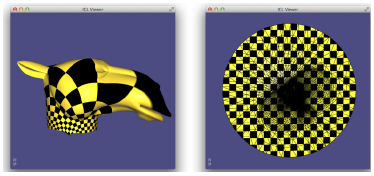


UV mapping is a difficult problem. In practice , we must :

- unfold the surface of the object onto the UV plane (texture space)
- assign(u,v) coordinates to each vertex of the unfolded mesh
- different triangles of the unfolded mesh cannot overlap

We may need to cut the mesh open to unfold it : seams. In general , it is not possible to unfold a mesh isometrically : distortion.

Checkboards are great to visualize a UV map



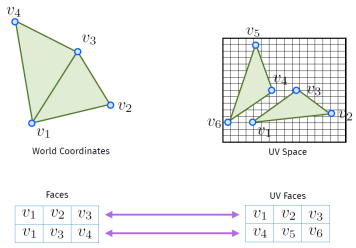
12.0.4 How are UV maps encoded?

2 versions of the mesh are stored , one for the triangles in 3D and one for those in 2D.

The faces of the 2 meshes are in one-to-one correspondence.

OpenGL does not support this you need to duplicate all the vertices on the seams and pass one single mesh. An easy way to do this is by duplicating all vertices and not using an element buffer.

12.0.5 A minimal example



12.0.6 Texture mapping as resampling

A 2D texture and the screen buffer are similar.

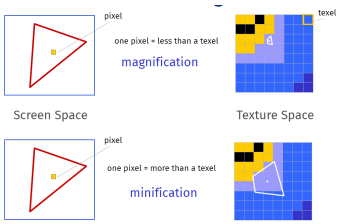
A. texture : a rasterized image B. screen buffer : a rasterized image.

Standard use of texture mapping can be seen as an operation of image resampling

12.0.7 Minification or Magnification?

Question : in which case are we?

1 texel : 1 pixel - each texel falls to a different pixel , and vice-versa 1 texel : N pixels - many pixels refer to the same texel, we are enlarging the texture (**Magnification**) N texels : 1 pixel - many texels fall in the same pixel , we are shrinking the texture (**Minification**



Answer depends on : uv-mapping of the mesh , current transformations, resolution of the screen and resolution of the texture . Answer can be different from pixel to pixel even inside the same triangle!

How to find it , for a specific access?

12.0.8 Derivatives in screen space

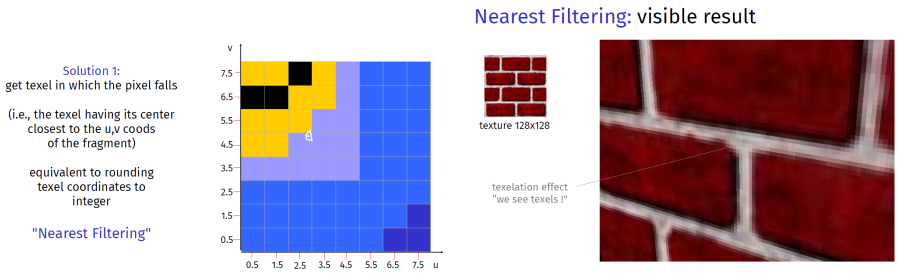
Idea : fragments are always processed in groups and in parallel. During fragment processing : given an expression , we can spy its values at adjacent fragments with a small overhead of synchronization / communication.

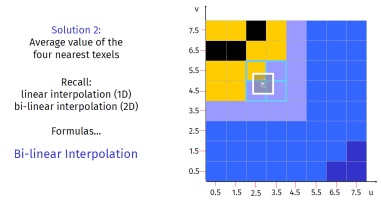
Difference : how much the value of the expression is varying in screen space.

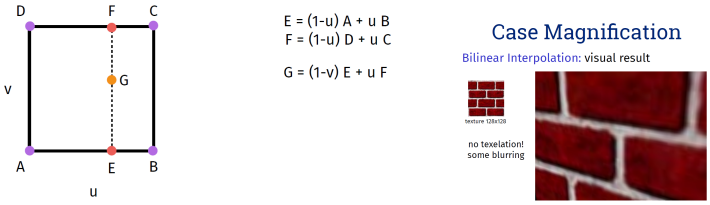
If a fragment is accessing to texture coordinates (u,v) then also the neighboring fragments are accessing to texture coordinates.

Difference between coordinates used by neighbors tell us if we are in case magnification or minification. Here , the derivative in screen space is used implicitly by the system.

12.0.9 Case Magnification







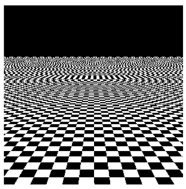
Mode nearest : "texelation" , ok if borders between texels are intentional , cheaper Mode interpolation : usually better quality , heavier and slower, it may be "blurred"

12.0.10 Case Minification

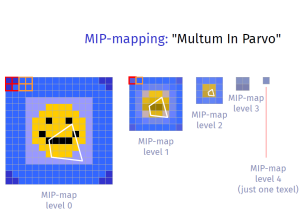
N texels $<--> 1$ pixel

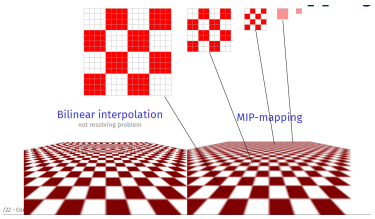
Further problem : texture subsampling , get one texel and skip N other texels. Bi-linear interpolation is not sufficient.

Moire pattern : Artifacts due to interference when texture resolution is higher than age resolution. Appear on digital photos too! How to get rid of them?



12.0.11 Case Minification : MIP-mapping





Define a scale factor , p = pixel / texel.

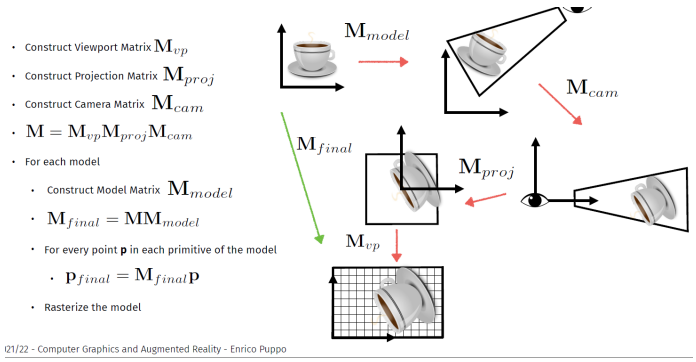
Obtained from derivatives in screen coordinates of s and t. It can change inside the same triangle.

The level of mipmap to be used is : log2 p. Level 0 = maximal resolution

13 View Details

13.1 Viewing Transformations Details

13.1.1 Transformations



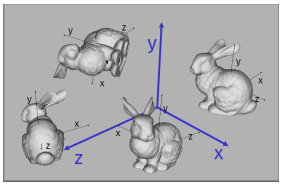
Performed in the vertex shader :

- M_{model} modeling matrix - depends on geometry of the scene (applied per single object
- M_{cam} camera matrix - depends on the viewer
- M_{proj} projection matrix - depends only on the camera

Performed during Setup in the rasterizer : M_vp viewport matrix - depends only on the rendering area

13.1.2 Modeling Transformation

Each object can be modeled in its own private reference frame : **Object Coordinates**. Different instances placed in the scene at different positions. Each transformation is expressed by an affine matrix. Objects can be moved or deformed



13.1.3 Object Coordinates to World coordinates

Each object is modeled in its provate Object Coordinates. During transform , first of all we bring the object to the common space containing the whole scene : from Object Coordinates to World Coordinates.

We can use the same model multiple times in the same scene. Each instance has the same Object Coordinates of vertices , but a different modeling transformation to get to different World Coordinates. For example : wheels of a car (4 instances of the same wheel), cars on the road , chairs in a room , pieces on a checkerboard , houses in village ecc.

13.1.4 Modeling Transformations

How do we proceed to create an instance of an object in a scene???

Assuming an object modeled about the origin :

1. Scale first , to obtain desired size and proportioned
2. Next, rotate to orient
3. Finally, translate to the desired position

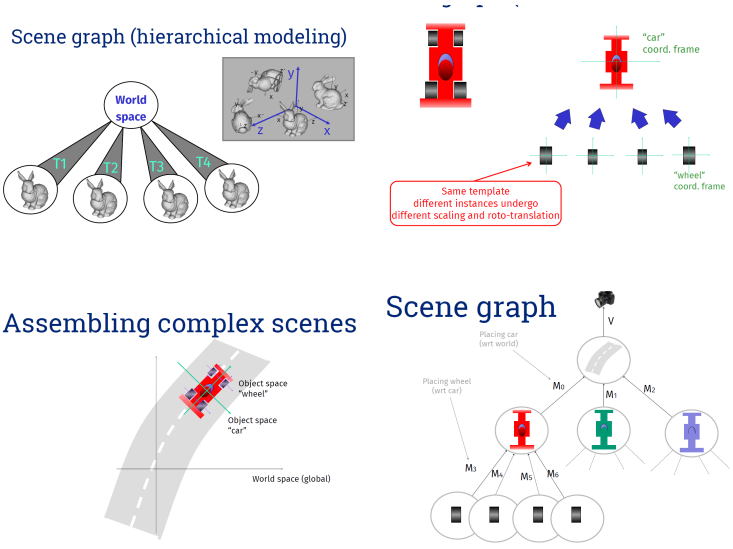
Modeling transformations are usually different from object to object. Modeling transformations are the last ones to be specified in the code , just before defining the object. Modeling transformations accumulate to the right on the previous ones , unless we reset the model-view matrix. Therefore : If we position object A first and object B next we do not want the transformations specified just for A to affect also B. We must reset the modeling matrix to its value preceding positioning of A, before we position B. On the other hand , there can be transformations that affect both A and B. We do not want to lose them when we reset the modeling matrix.

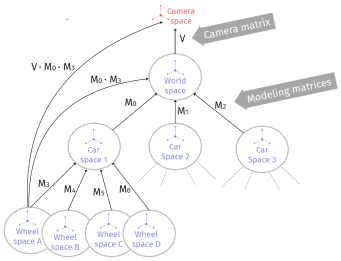
Given objects A,B,C... that compose the scene , we cannot write code as follows : Transformation for A , Draw A , Transformations for B , Draw B. Otherwise transformation for B would accumulate on the ones for A and so on...

On the other hand , we cannot just reset the matrix as follows : Reset matrix to Identity , Transformations for A , Draw A , Reset matrix to Identity , Transformations for B , Draw B. Because the reset operation cancels all transformations preceding it...

Some transformation should affect all objects. General Scheme : View transformations, Global modeling transformations , Transformations for A , Draw A , Transformations for B , Draw B.

How can we get rid of transformations for A without losing view and global modeling transformations?

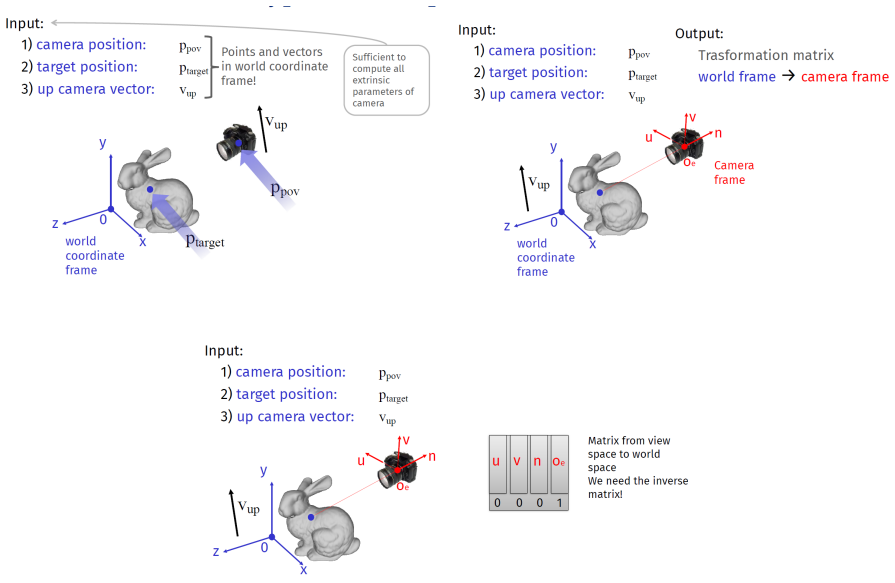




13.1.5 Setting the model-view matrix

Idea : depth first navigation of scene graph. Init : model-view = view Follow links from parents to children (accumulate modeling matrices) When backtracking to parent : Recover its matrix keeping a stack of matrices.

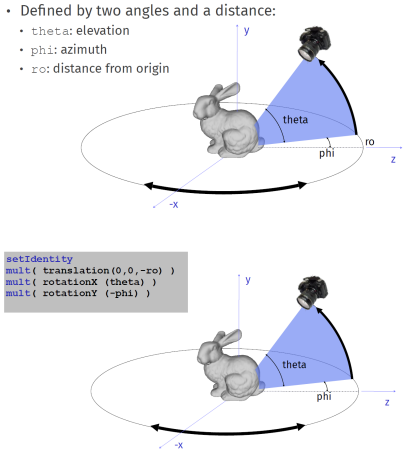
13.1.6 View Transformation : typical example



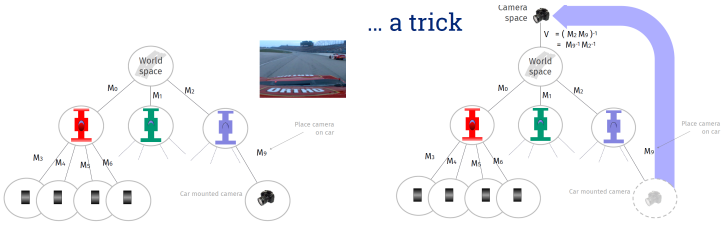
13.1.7 Example : Orbiting Camera

Basic user interface to select a point of view. Two angles (phi,theta) + distnace(ro)

Useful to examine a single object. We can just orbit viewpoint about it and we are always looking towards the center.



13.1.8 Camera on a car : a trick



The view transformation needed to place the camera at a given position and it is given by the inverse matrix of the modeling transformation needed to place an object at the same position.

13.1.9 Object Viewer

We wish to pan/dolly/rotate an object with respect to the eye reference frame :

- Pan is a translation of the scene in the uv plane
- Dolly is a translation of the scene along the n axis
- Rotation occurs about a point placed at the center of the view window and at a given distance from it.

Transformations must affect the object from its current position. They should be accumulated after the transformation used for current rendering and they should be specified before the current transformations in the code.

Pan and Dolly

Translations are easy to combine with current transformation : While rendering , save current ModelView in matrix M; at the next frame , apply M first;

then translate along z to zoom or along x and y to pan ; code must be written in reverse order

Rotate

Problem : the current matrix M contains both rotation and translation. Object is translated of -d along the n axis. New rotation must occur about the origin Thus , we must bring back the object to the origin prior to rotate it. We should perform in order : 1. Previous rotation (in current matrix M) , 2. New rotation (specified by angles about u and v axes), 3. translation (in current matrix M)

How can we insert a rotation between two transformations that are incorporated in matrix M?

Current view matrix is of type TR, where T corresponds to a translation of -d along the n axis. Thus $T^{-}1M$ is a pure rotation.

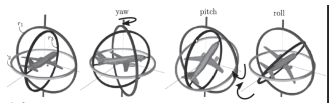
Thus we may : apply M (rotation and translation); translate with $T^{-}1$ to bring object back to the origin; apply new rotation R'; translate with T to push object back.

Composite matrix will be : TR' $T^{-}1M$

To drive viewer with the mouse , angle alpha and beta are proportional to mouse motion in the vertical and horizontal directions , respectively. Distance d must be stored , and it is updated upon dolly ops. Order of rotations is relevant (risk of gimbal lock) Order of rotations can be disregarded with input from key pressure or mouse drag , if incremental angles are always very small.

13.1.10 Euler Angles

Generic rotation expressed as a sequence of three rotations about axes : Yaw about y axis , Pitch about x axis , Roll about z axis

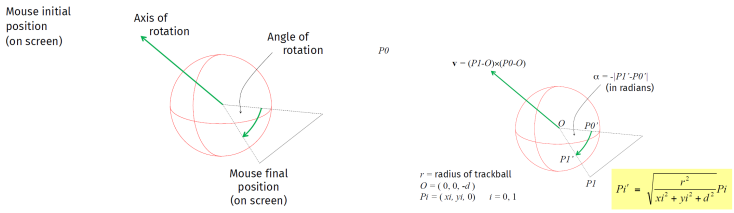


13.1.11 Gimbal Lock

Problem with order of rotations. Result depends on order : rotation about an axis influences later rotations about different axes. For small angles , there are little differences. With large pitch angles , roll and yaw occur on about the same axis.

13.1.12 Trackball

A more elegant solution : Put a sphere around the object. Get the intersections between the sphere and rays cast from consecutive mouse positions. Compute the angle and axis of rotation explicitly. Combine with previous rotation.



An even more elegant solution : Store the status of the trackball (center , radius , current rotation). Encode rotations with quaternions.

13.1.13 Quaternions

What is a quaternion?

An extension of complex numbers , q = w + xi + yj + zk where $i \cdot i = j \cdot j = k \cdot k = i \cdot j \cdot k = -1$ More often represented as a pair scalar-vector : q = [w,v] where v = (x,y,z)

13.1.14 The algebra of quaternions

Magnitude : $||q|| = \sqrt{w^2 + x^2 + y^2 + z^2}$

Normalization to unit-length quaternion : q = q/||q||

Sum and product:

given two quaternions

$$q_1 = w_1 + x_1 i + y_1 j + z_1 k \quad \text{and} \quad q_2 = w_2 + x_2 i + y_2 j + z_2 k$$

or equivalently $q_1 = [w_1, \mathbf{v}_1]$ and $q_2 = [w_2, \mathbf{v}_2]$

where $\mathbf{v}_1 = (x_1, y_1, z_1)$ and $\mathbf{v}_2 = (x_2, y_2, z_2)$

we define

$$q_1 + q_2 = [w_1 + w_2, \quad \mathbf{v}_1 + \mathbf{v}_2]$$

$$q_1 * q_2 = [w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, \quad \mathbf{v}_1 w_2 + w_1 \mathbf{v}_2 + \mathbf{v}_1 \times \mathbf{v}_2]$$

- Identities
  - sum
  - product

$$q_{1+} = [0, \quad (0,0,0)]$$

$$q_{1*} = [1, \quad (0,0,0)]$$

13.1.15 Quaternions and rotations

Each unit-length quaternion corresponds to a rotation in 3D:

q = [w,v] axis = v/—v— and angle = 2arccos(w)

Multiplication between two quaternions corresponds to composing the two rotations : Store current rotation in a quaternion q_curr; compute new rotation into another quaternion q_new; update current rotation $q_curr \leftarrow q_curr * q_new$

13.1.16 Conversions

- From quaternion to rotation matrix

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

- From quaternion to axis/angle

$$q = [w, \mathbf{v}] \quad axis = \mathbf{v}/|\mathbf{v}| \text{ and } angle = 2\arccos(w)$$

- From axis/angle to quaternions

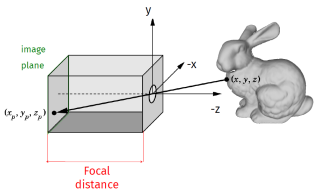
$$axis = (a_x, a_y, a_z) \text{ and } angle = \theta$$
$$q = [\mathbf{v}, w]$$
$$\mathbf{v} = (a_x \cdot \sin(\tfrac{\theta}{2}), a_y \cdot \sin(\tfrac{\theta}{2}), a_z \cdot \sin(\tfrac{\theta}{2}))$$
$$w = \cos(\tfrac{\theta}{2})$$

- From Euler angle to quaternions

$$q_x = [\cos(\tfrac{\alpha}{2}), \quad (\sin(\tfrac{\alpha}{2}),0,0)]$$
$$q_y = [\cos(\tfrac{\beta}{2}), \quad (0,\sin(\tfrac{\beta}{2}),0)]$$
$$q_z = [\cos(\tfrac{\gamma}{2}), \quad (0,0,\sin(\tfrac{\gamma}{2}))]$$
$$q = q_x * q_y * q_z$$

13.1.17 Projection Transformation

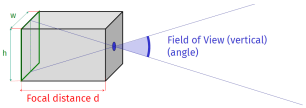
Pin-hole camera model.



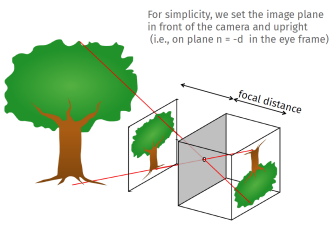
13.1.18 Intrinsic parameters of camera

Size of image plane (w,h) Focal distance (d) or angle Field of View (FoV) How to get one from the other??

d used for computations , FoV more intuitive to set

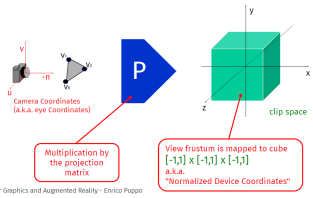


13.1.19 Pin Hole camera and Clip Space



Actually, we do not discard the third dimension:

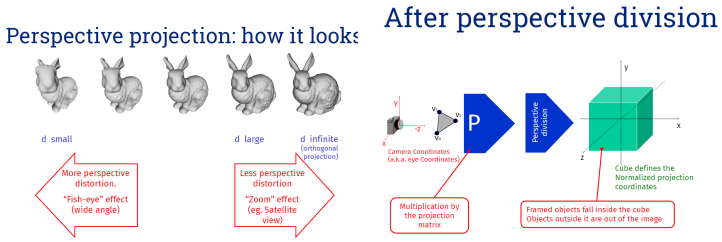
- We deform space to a more suitable frame



13.1.20 Projection

Projection affects all vertices that get to the projector : each vertex is converted to a different frame. Projection parameters define the projection matrix , which is not always affine : parallel projection is a combination of translation and scaling (affine) whereas perspective projection is a projective projection (not affine)

Project matrix maps points of the view volume to points of the normalized cube; points outside the view volume fall outside the cube.



13.1.21 Viewport transformation

The viewport is the portion of output window to which the frame buffer is mapped (rectangle w x h pixels) Vertices in the Normalized projection coordinate systems are further transformed to express them in Window coordinates. 3D coordinate system , with x,y in pixels and z in range [0,1]. Origin at lower left corner.

13.1.22 Eventually 2d + depth

All primitives in devices coordinates are projected onto plane z = 0. These 2D primitives are passed on to the rasterizer. Actually the third coordinate of each vertex is preserved as an attribute for the depth test

13.1.23 Aspect Ratio

The combined effect of projection and viewport transformations may deform the image. In order to keep the aspect ratio we need to have the same ratio w/h for the viewport as well as for the cross section of the view volume. Upon resize of the view window , we need to keep the aspect ratio consistent. Two ways : 1. Change the viewport , 2. Change the view volume