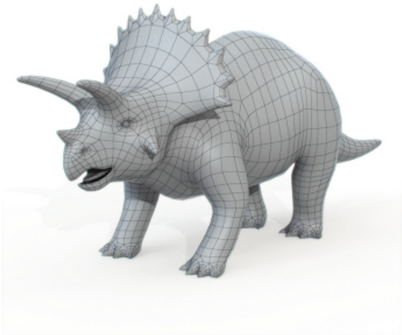
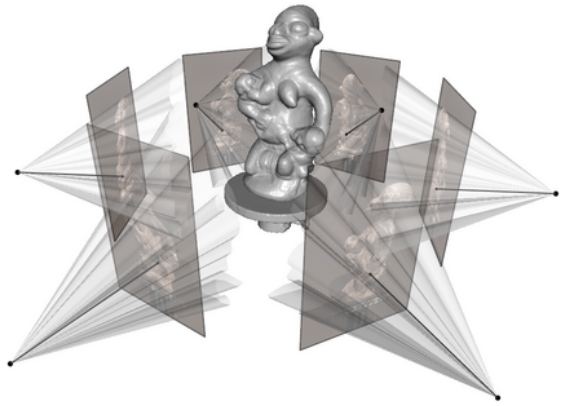
Computer Graphics notions and CG programming

Computer Graphics main SubFields

Modeling

How to create static shapes

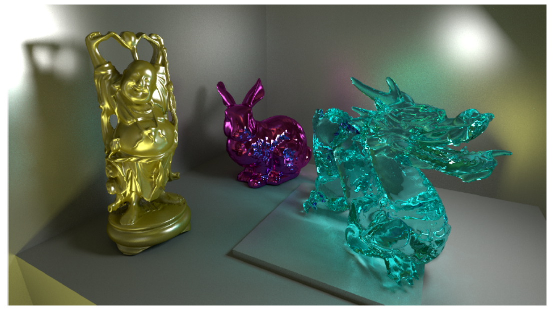
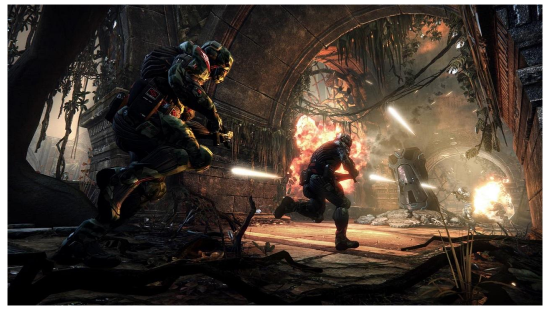
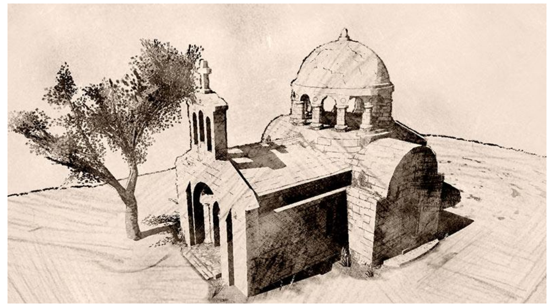
Animation

How to create and author time varying shapes



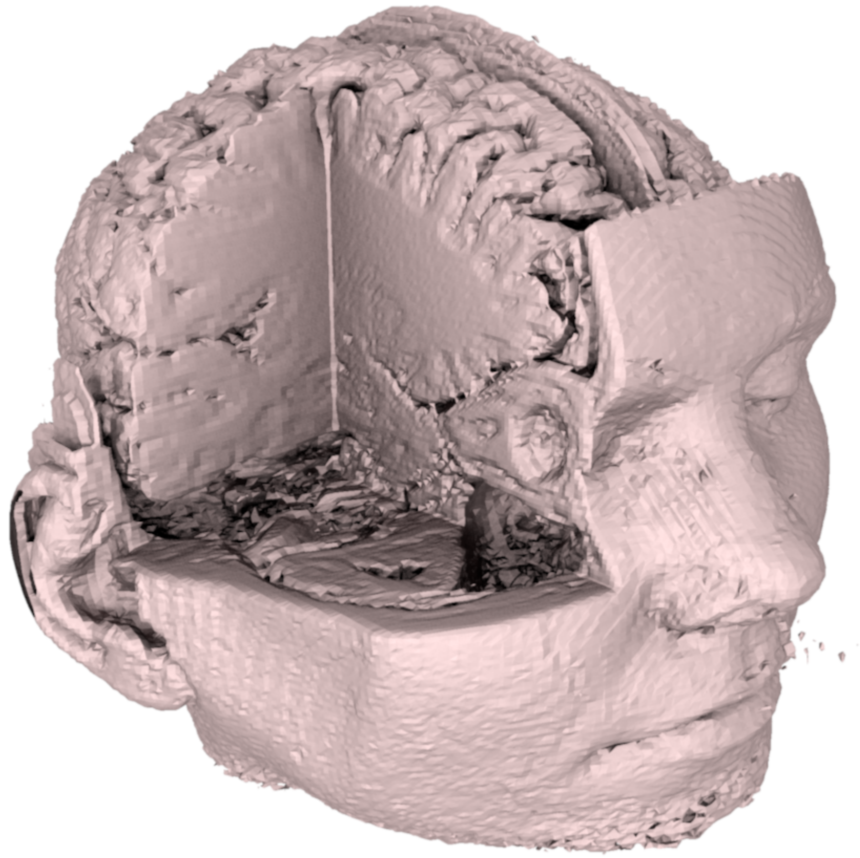
Rendering

How to generate 2D images from 3D data

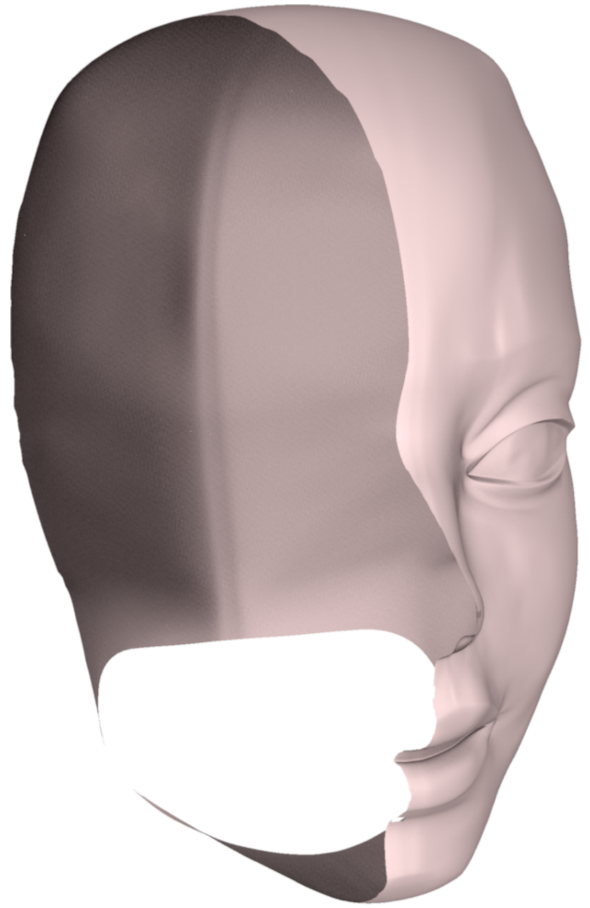
Representing 3D shapes for Graphics Applications

Volume representation



+ Accurate, handle density

Surface representation



+ Focus on visible part
+ Fast GPU rendering, low memory footprint

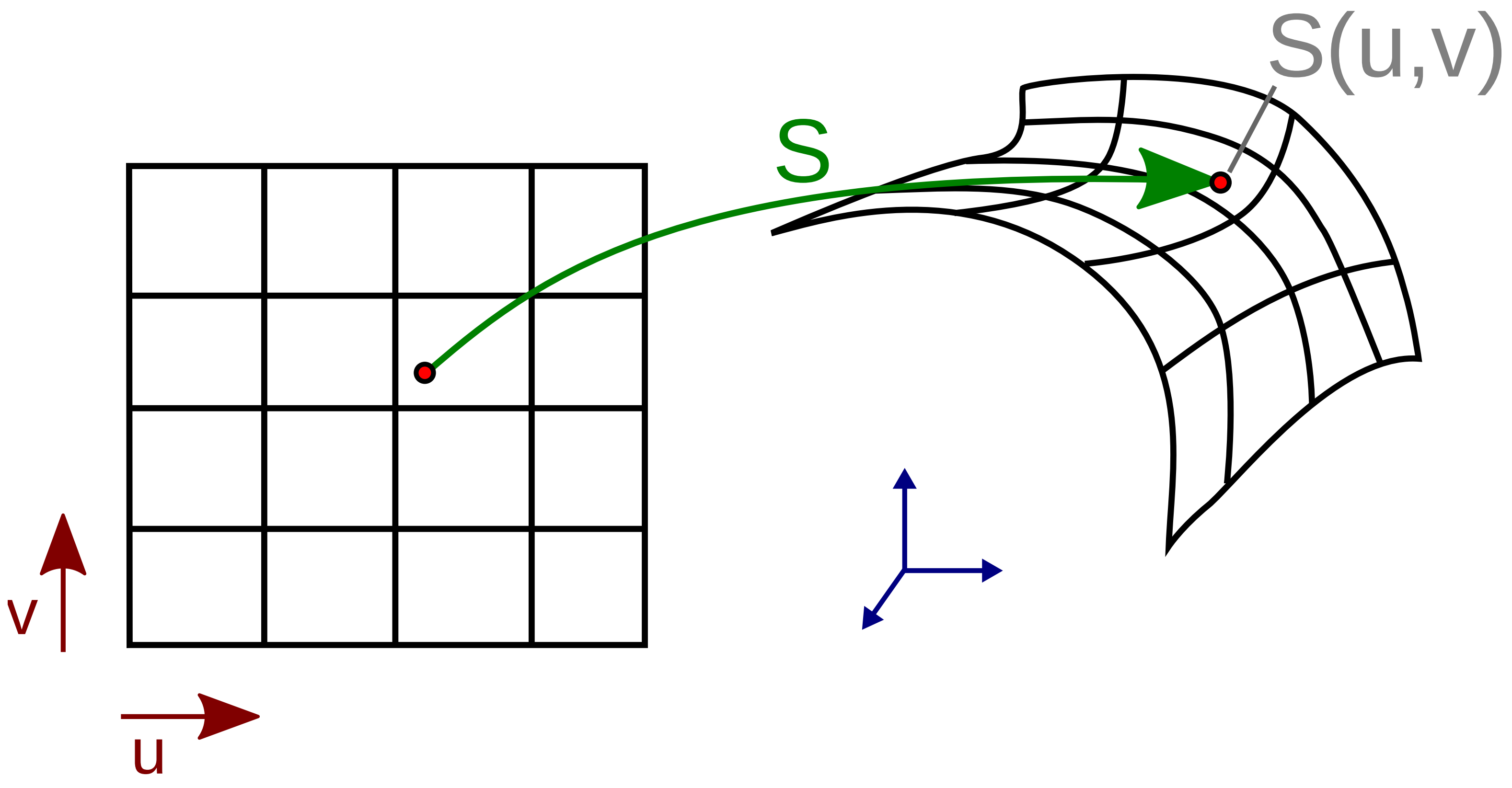
=> **Computer Graphics**: Mostly focus on representing **Surfaces**
=> **Scientific visualization**: Volume data

Two main representations for surfaces

Explicit representation

$$S(u,v) = (x(u,v),\ y(u,v),\ z(u,v))$$

Parametric map

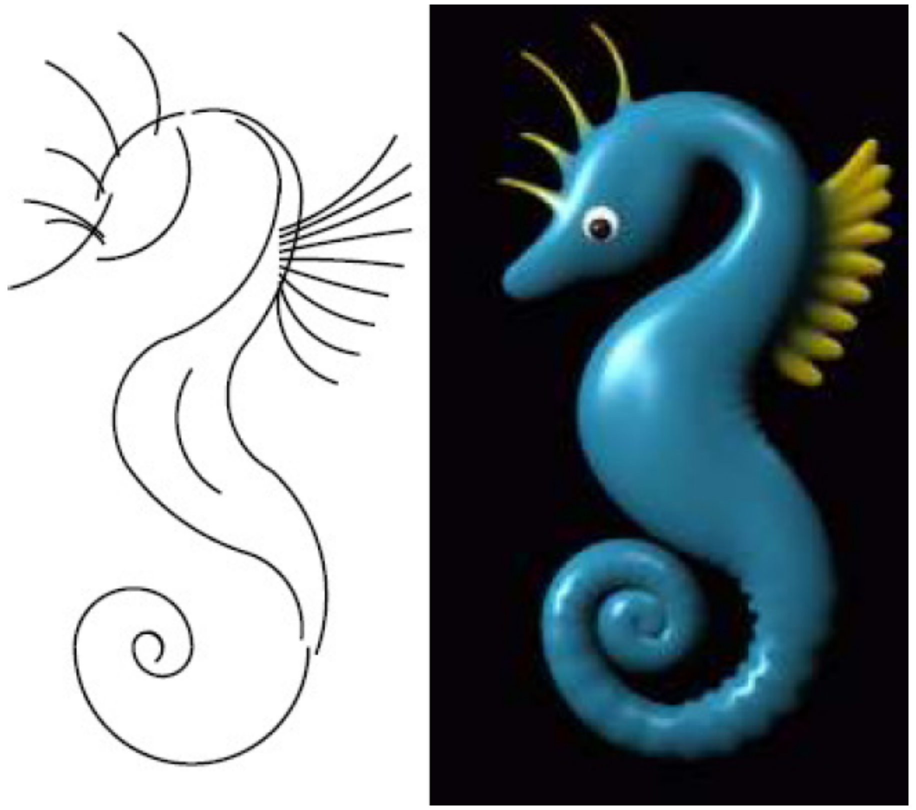


+ Neighborhood information

Implicit representation

$$S = \{(x,y,z) \in \mathbb{R}^3 \mid F(x,y,z) = 0\}$$

Isosurface of scalar field



+ Topological modification

Two main representations for surfaces

Example for a sphere

Explicit representation

$$S(u,v) = (x(u,v),\ y(u,v),\ z(u,v))$$

Parametric map

$$S(u,v) = \begin{cases} x(u,v) = R\sin(u)\cos(v) \\ y(u,v) = R\sin(u)\sin(v) \\ z(u,v) = R\cos(u) \end{cases}$$

Implicit representation

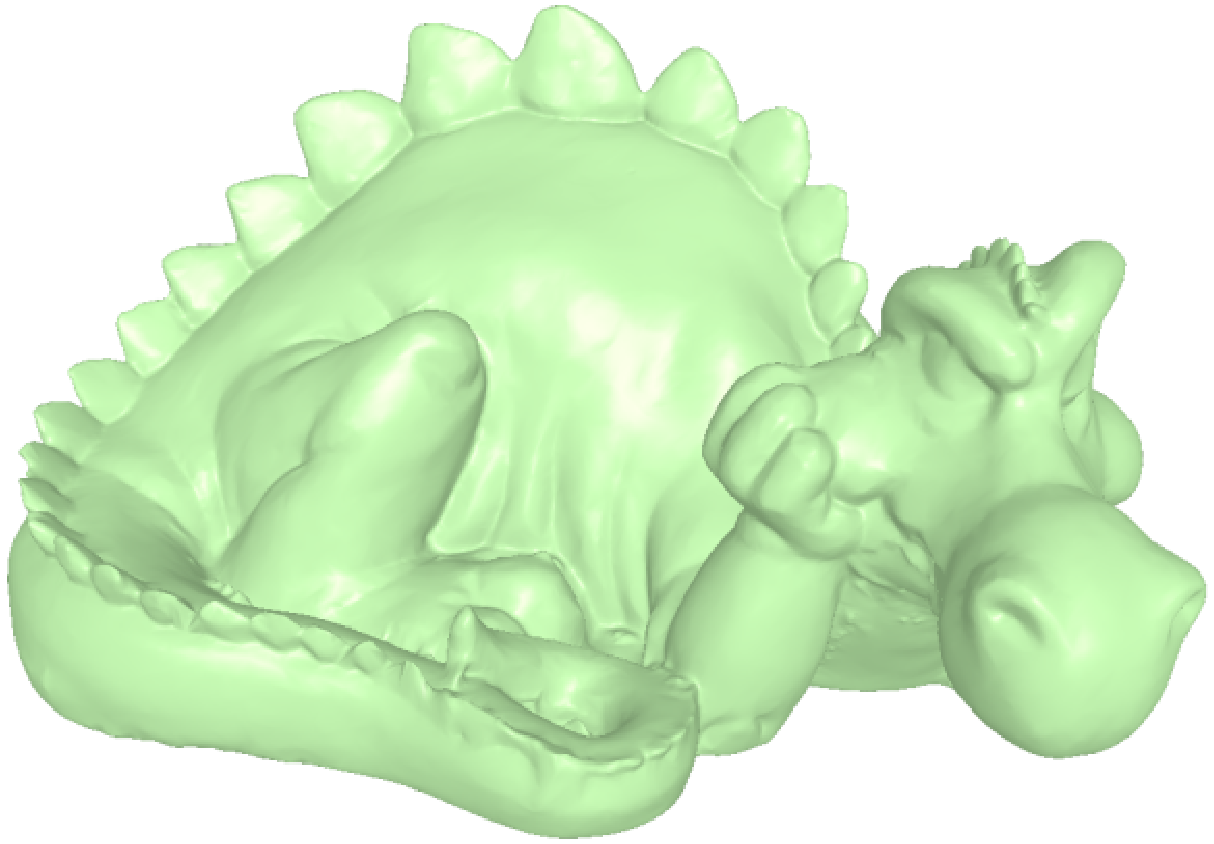
$$S = \{(x,y,z) \in \mathbb{R}^3 \mid F(x,y,z) = 0\}$$

Isosurface of scalar field

$$F(x,y,z) = x^2 + y^2 + z^2 - R^2$$

Difficulty of surface representation using function

Which function can represent this shape ?



$$S(u, v) = \,?$$

$$F(x, y, z) = \,?$$

Objective of surface representation

Main Idea => Use of **piecewise approximation**

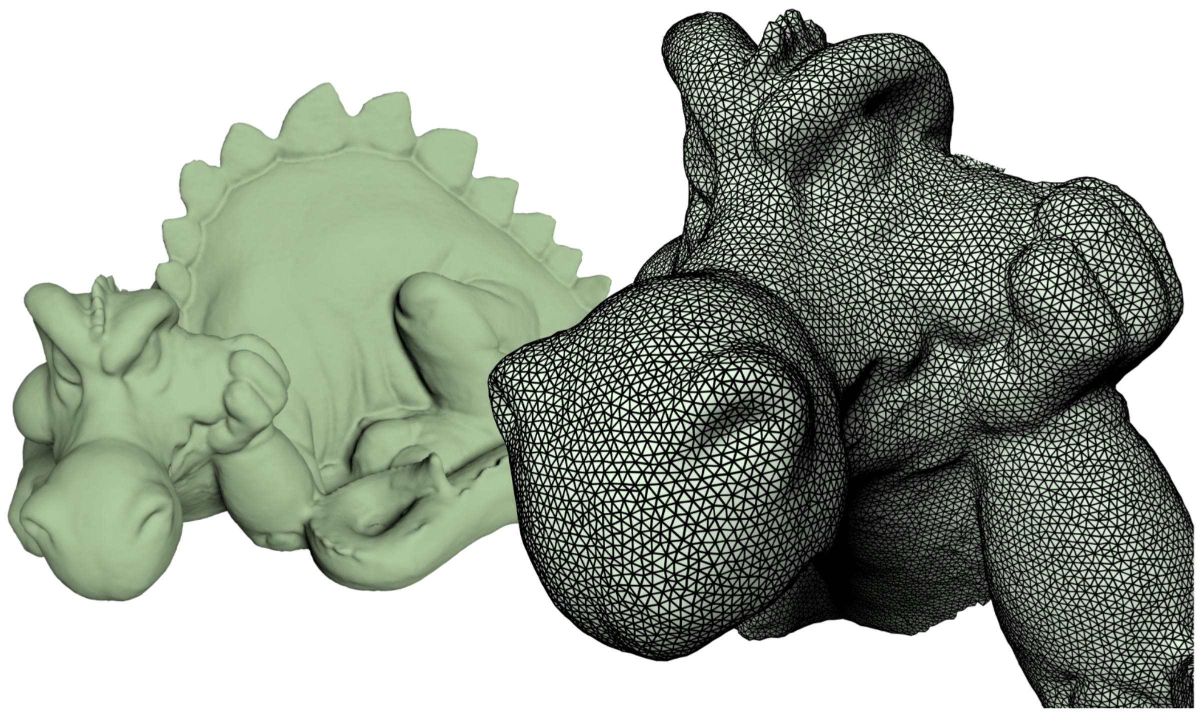
Ideal surface representation
- **Approximate** well any surface
- Require **few samples**
- Can be **rendered** efficiently (GPU)
- Can be manipulated for **modeling**

Example of models:
- *Mesh-based: Triangular meshes, Polygonal meshes, Subdivision surfaces*
- *Polynomial: Bezier, Spline, NURBS*
- *Implicit: Grid, Skeleton based, RBF, MLS*
- *Point sets*

=> For projective/rasterization render pipeline : always render **triangular meshes** at the end

+ Simplest representation
+ Fit to GPU Graphics render pipeline
- Requires large number of samples: complex modeling
- Tangential discontinuities at edges

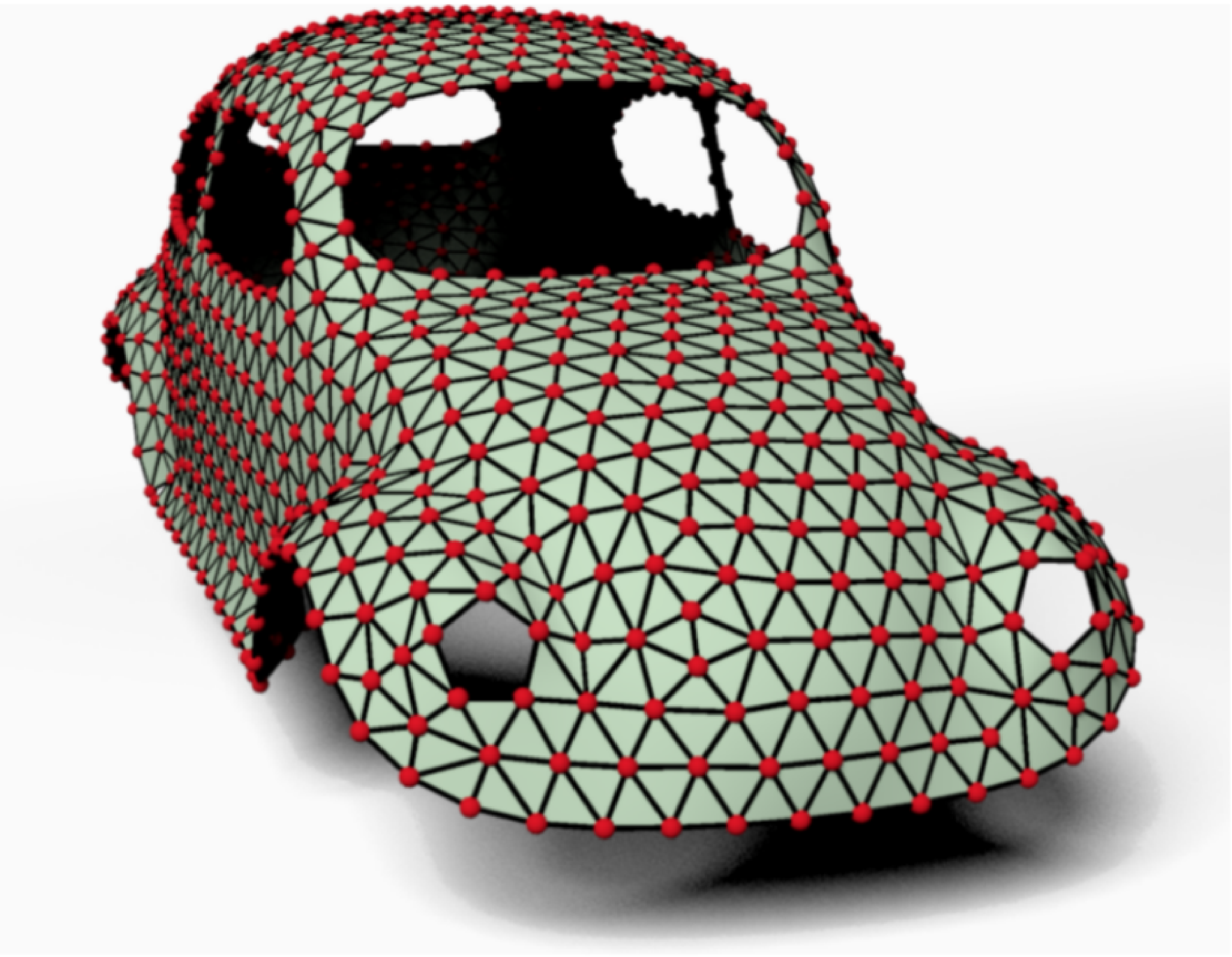
Meshes



Simplest possible representation of 3D surfaces: set of triangles

Described as a triplet: (Vertices, Edges, Faces)

$$S = (\mathcal{V}, \mathcal{E}, \mathcal{F})$$



- Vertex

$$\mathcal{V} = (v_1, \ldots, v_N)$$

- Edge

$$\mathcal{V} = (v_1, \ldots, v_{N_e}) \in (\mathcal{V}^2)^{N_e}$$

- Face

$$\mathcal{F} = (f_1, \ldots, f_N) \in (\mathcal{V}^3)^{N_f}$$

Mesh encoding

Exemple for a tetrahedron

- 1st Solution: *Soup of polygons*

```
triangles = [(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 0.0, 1.0),
             (0.0, 0.0, 0.0), (0.0, 0.0, 1.0), (0.0, 1.0, 0.0),
             (0.0, 0.0, 0.0), (0.0, 1.0, 0.0), (1.0, 0.0, 0.0),
             (1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)]
```

- 2nd solution: *Geometry, Connectivity*

```
geometry = [(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)]
connectivity = [(0,1,3), (0,3,2), (0,2,1), (1,2,3)]
```

=> Prefered solution

+ more space efficient

+ modifying 1 vertex = 1 operation

Mesh buffer encoding in C++

```
#include <vector>
#include <array>

struct vec3 {float x,y,z;};
using index3 = std::array<unsigned int, 3>;

int main()
{
    std::vector<vec3> geometry = { {0.0f, 0.0f, 0.0f}, {1.0f, 0.0f, 0.0f},
             {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f, 1.0f} };
    std::vector<index3> connectivity = { {0,1,3}, {0,3,2}, {0,2,1}, {1,2,3} };

    return 0;
}
```

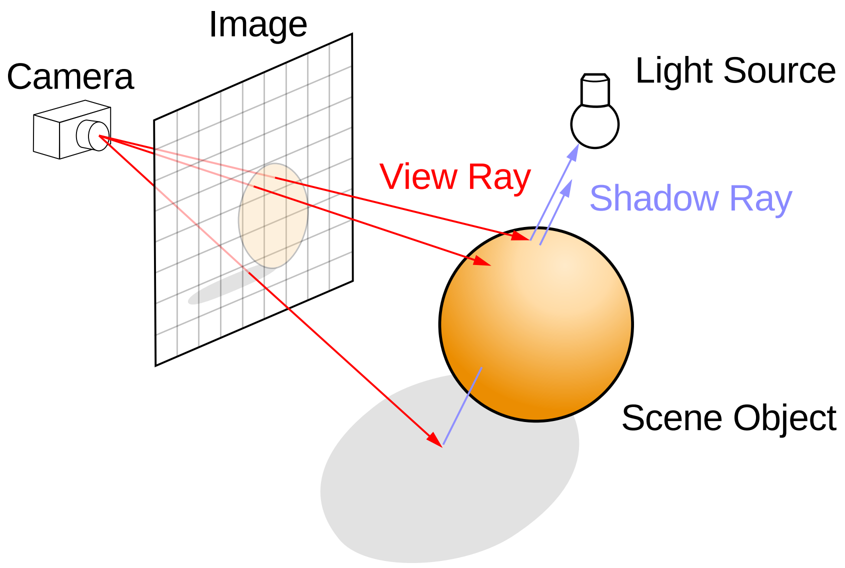
Example of 3D Mesh file

```
v -0.000000 0.620276 0.108446
v -0.000000 0.685780 0.104094
v 0.011128 0.685780 0.102245
v 0.014793 0.620276 0.106125
v 0.034724 0.684975 0.079817
v 0.040413 0.620278 0.086828
v 0.029160 0.619800 0.099405
v 0.024110 0.685194 0.093530
v 0.046714 0.554312 0.085764
v 0.033793 0.547222 0.100284
v 0.015067 0.542780 0.113608
v -0.000000 0.541146 0.117759
v 0.051177 0.430214 -0.047903
v 0.049948 0.435812 -0.035967
v 0.028863 0.449897 -0.050037
v 0.028839 0.444346 -0.059194
v 0.017691 0.251925 0.023686
v 0.034131 0.252216 0.014535
v 0.036689 0.275442 0.012672
v 0.015166 0.271140 0.025837
v 0.014869 0.285441 0.024957
```

Open question: How to display it efficiently on screen?

How to render surfaces

Ray tracing

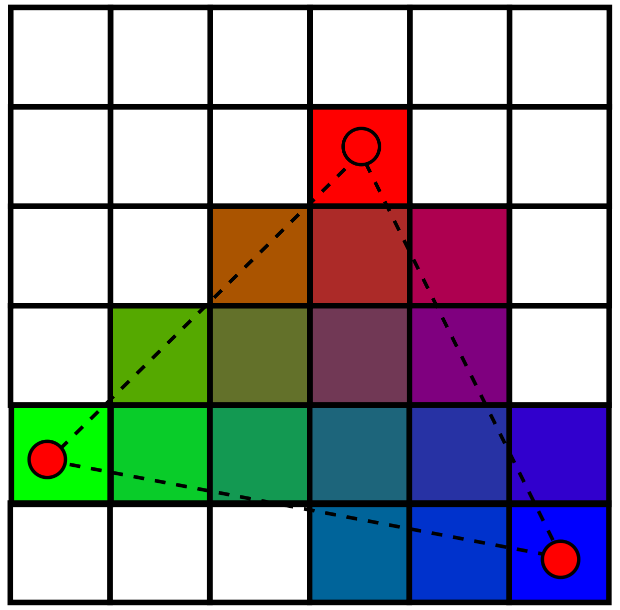


- *Throw rays from light-sources/camera*
- *Intersect rays with 3D shapes*
- *Pixel-wise computation*

\+ Photo-realistic rendering
  (Soft shadows, reflection, caustics)
\+ Handle general surfaces
\- High computational cost

=> Restricted to offline rendering (but developing more and more)

Projection/Rasterization



- *Assume shapes made of triangles*
  1. *Project each triangle onto camera screen space*
  2. *Rasterize projected triangle into pixels*
- *Triangle-wise computation*

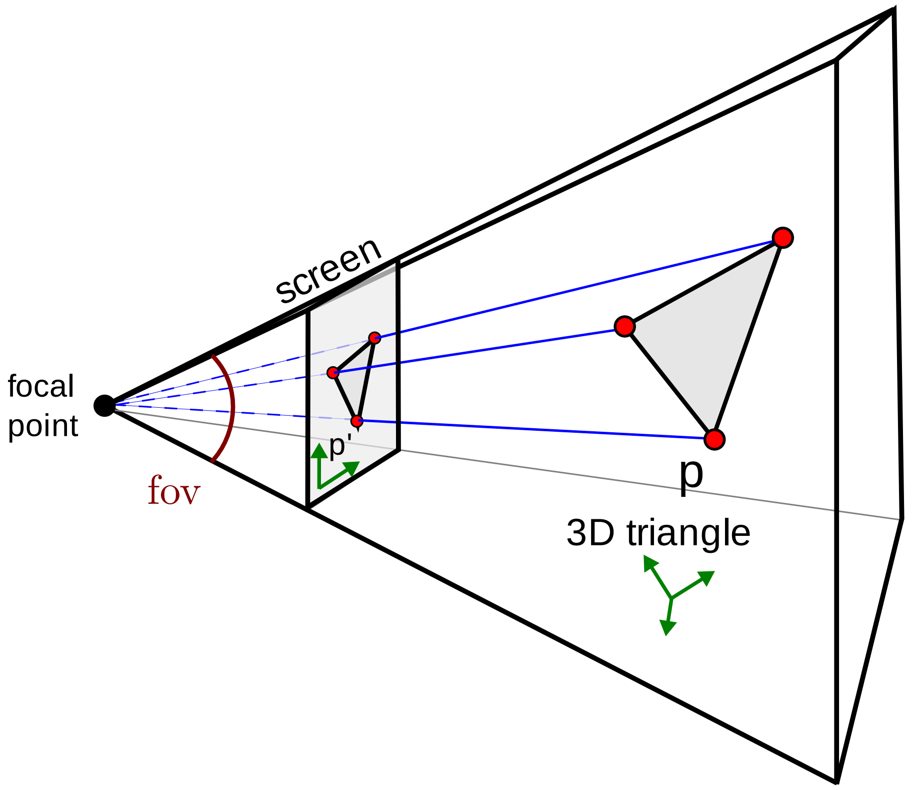
\+ Efficiently implemented on GPU
\- Limited to triangles
\- No native effects (shadows, transparency, etc)

=> The standard real time rendering with GPU

Projection/Rasterization

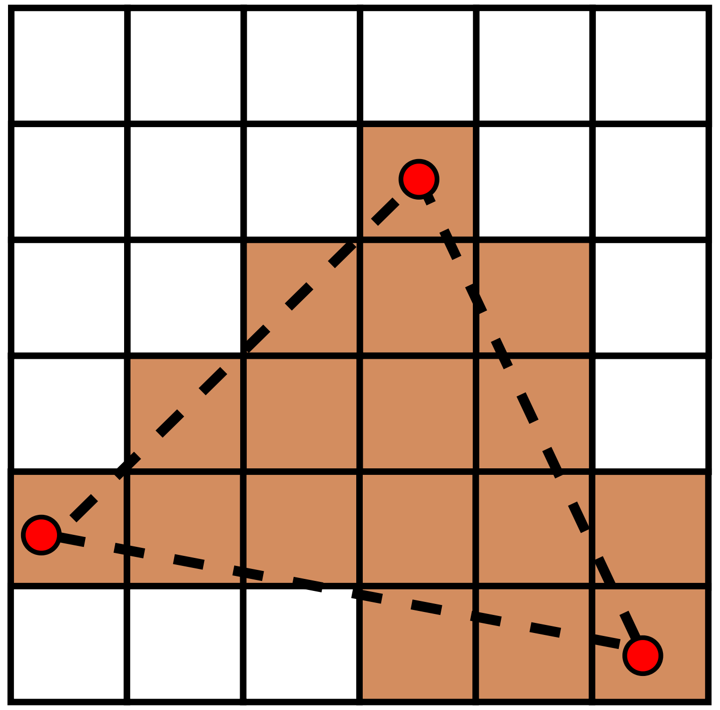
Object made of **triangles only**

1. **Project vertices of triangles**



- Projection computed as matrix operation (projective space $p' = \mathrm{M}\,p$)

2. **Rasterization**



- Discrete geometry
- Interpolate attributes (colors, etc) on each pixel

=> At interactive frame rate ($\geq$ 25 fps)

- Project all triangles of shapes
- Fill all pixels of each projected triangle

Quick fundamental notions for practical 3D programming

- Affine transform as 4D matrices
- Perspective and projective space
- Illumination and normals

Affine transforms and 4D vectors/matrices

Preliminary note

- We use a lot affine transformations to place shapes in 3D space

  Translation, Rotation, Scaling
- In CG vectors are often expressed in 4D, and matrices are 4×4.

=> Reason: Affine transforms can be expressed linearly (with matrices) in 4D

$$p = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \mathrm{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & t_x \\ m_{10} & m_{11} & m_{12} & t_y \\ m_{20} & m_{21} & m_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Affine transform in 2D

General principle in the 2D case

Example for a point $p = (x, y)$

Rotation $\mathrm{R} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$, Scaling $\mathrm{S} = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$, Translation $(x + t_x, y + t_y)$ *(not linear)*

Cannot express conveniently composition b/w several rotation, scaling, translation.

Trick - Add an extra coordinates to points $p = (x, y, 1)$ (homogeneous coordinates).

Then translation can be expressed linearly $p' = \mathrm{T}\, p$, with $p' = \underbrace{\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathrm{T}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$

Similarly with rotation $\mathrm{R} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$, and scaling $\mathrm{S} = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Affine transform matrix

With the extra dimension (in 2D):

Translation T, rotation R, scaling S can be composed as matrix products representation

ex. $\mathrm{M} = T_0\, R_0\, S_0\, T_1\, R_1\, S_1\, \ldots \qquad \mathrm{M} = \left(\begin{array}{cc|c} m_{00} & m_{01} & t_x \\ m_{10} & m_{11} & t_y \\ \hline 0 & 0 & 1 \end{array}\right).$

m_{ij} : linear part (rotation and scaling); $t_{x/y}$: translation part

Similar in **3D** but with **4-components vectors**, and $\mathbf{4} \times \mathbf{4}$ **matrices**.

$p = (x, y, z, 1)$ - *represents 3D position*

$$\mathrm{M} = \left(\begin{array}{ccc|c} m_{00} & m_{01} & m_{02} & t_x \\ m_{10} & m_{11} & m_{12} & t_y \\ m_{20} & m_{21} & m_{22} & t_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$ - *represents 3D affine transformation (rotation, scaling, translation)*

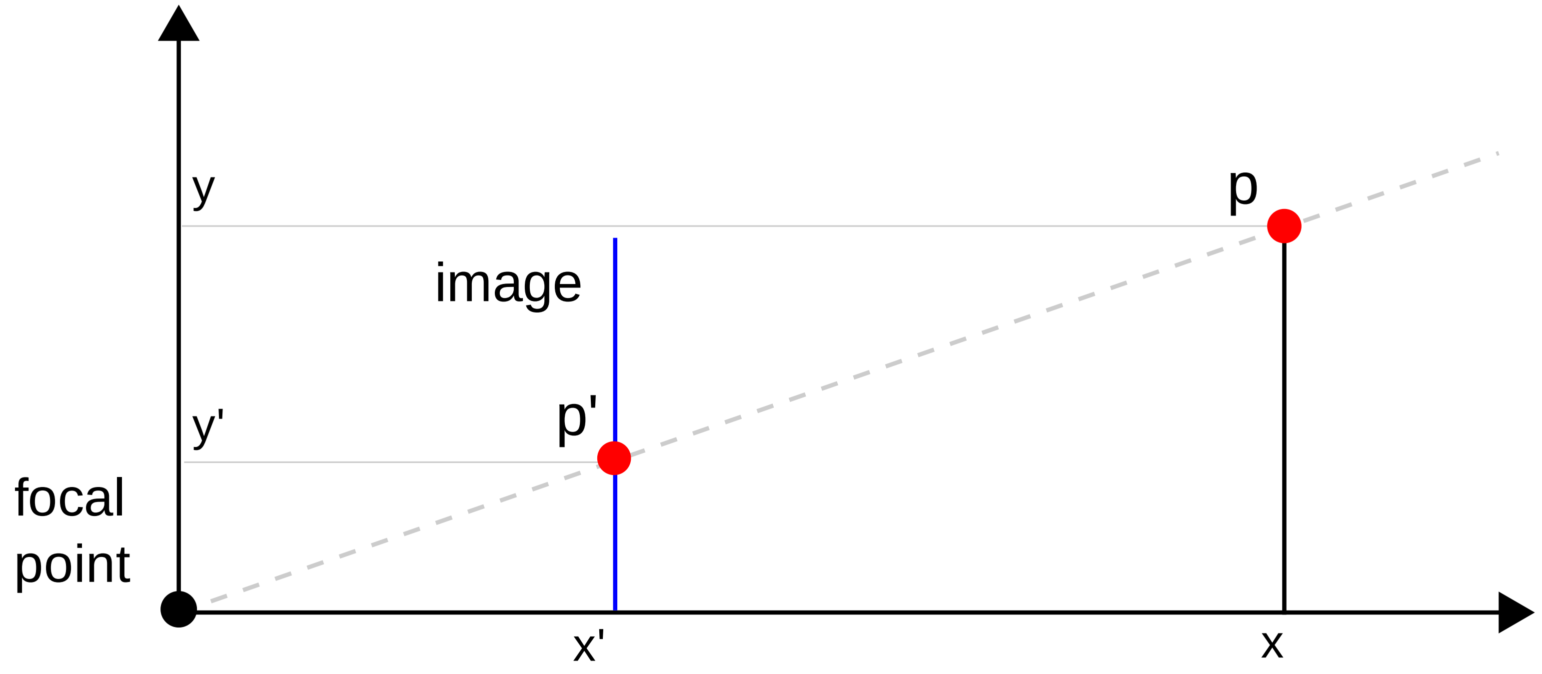
Note: vectors and points can be expressed

- 3D point $(x, y, z, 1)$ - translation applies.
- 3D vector $(x, y, z, 0)$ - translation doesn't apply.

Perspective and projective space

Modeling perspective projection requires division.

ex. in 2D (1D projection)



$$y' = x' \, \frac{y}{x} = f \, \frac{y}{x} \qquad (f\text{: focal})$$

Linear model using 3D vectors in projective space.

$$p' = \begin{pmatrix} f \\ f\,\frac{y}{x} \\ 1 \end{pmatrix} \underbrace{=}_{normalization} \begin{pmatrix} fx \\ fy \\ x \end{pmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

considering that the last coordinate must always be normalized to 1 (for points).

Projective space

- *Real* points lie on $z = 1$
- Vectors lie on $z = 0$

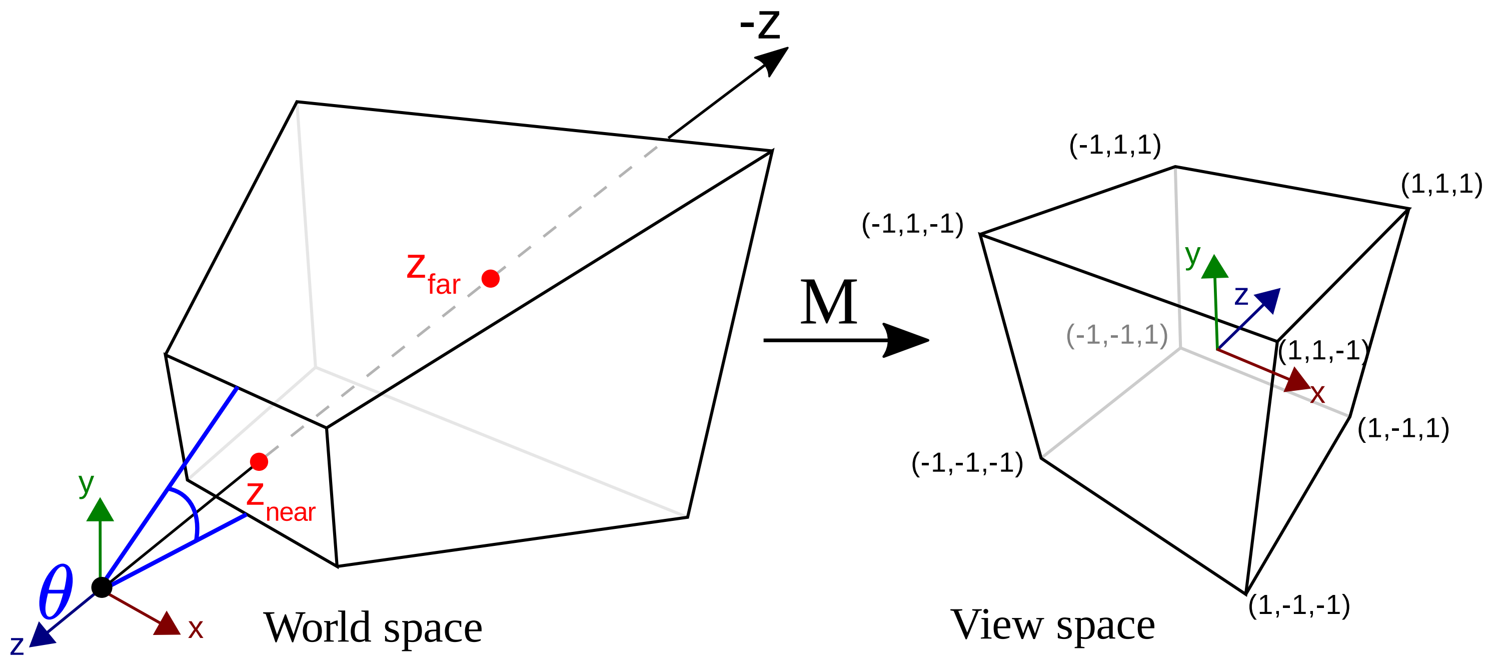
Real coordinates of points are obtained after normalization (division by z).

Perspective matrix

Perspective space : Allows perspective projection expressed as matrix.

Common constraints (in OpenGL)

- Wrap the viewing volume (truncated cone with rectangular basis called *frustum*) $(z_{near}, z_{far}, \theta)$ to a cube.
  - θ: view angle
  - $p = (x, y, z, 1) \in \text{frustum} \Rightarrow p' = (x', y', z', 1) \in [-1, 1]^3$.



$$M = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix} \qquad \begin{aligned} f &= 1/\tan(\theta/2) \\ L &= z_{near} - z_{far} \\ C &= (z_{far} + z_{near})/L \\ D &= 2\, z_{far}\, z_{near}/L \end{aligned}$$

In practice

=> You must define z_{near}, z_{far}

=> $z_{far} - z_{near}$ should be as small as possible for maximum depth precision.

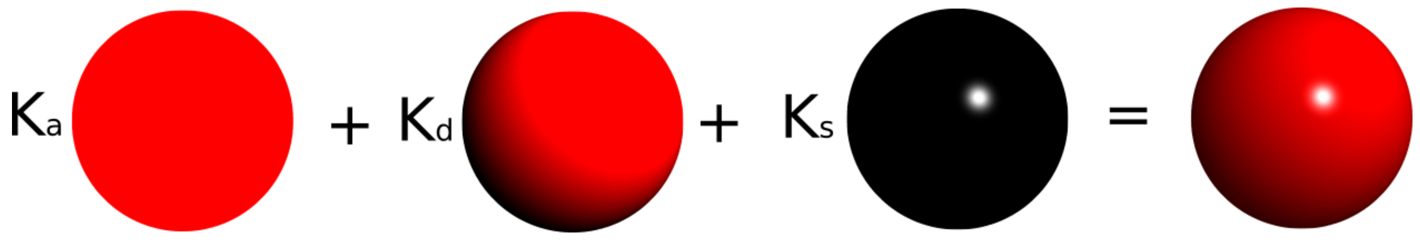
To which view space coordinates are mapped 3D world space points at z_{near}, z_{far} ?

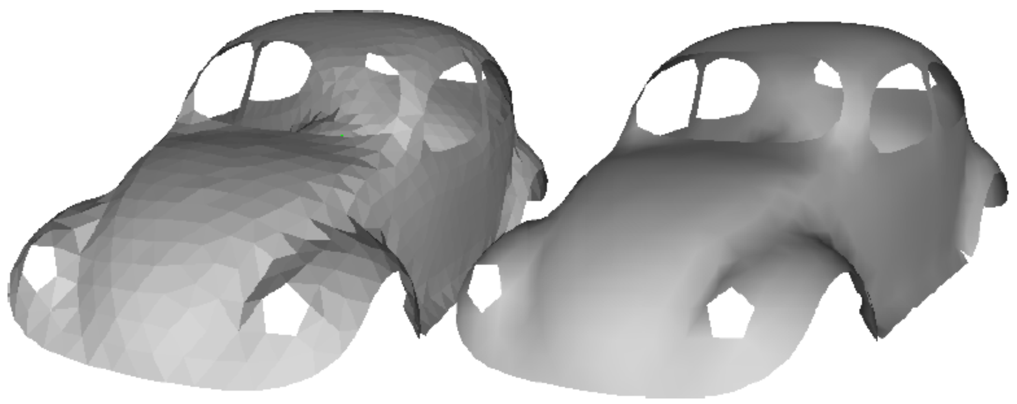
Per-vertex normal and illumination

For smooth looking meshes, we define a **normal per-vertex**.

- Vertices are seen as samples on a smooth underlying surface

- Normals are used for illumination
  ambiant, diffuse, specular components





- *Phong shading* interpolates normals on each *fragments* of triangles, and compute illumination.

Possible automatic computation of normals: averaged normals of surrounding triangle.

$$n_k = \frac{\sum_{j \in \mathcal{V}_k} \hat{n}_j}{\| \sum_{j \in \mathcal{V}_k} \hat{n}_j \|}, \ \mathcal{V}_k\text{: neighboring triangles.}$$