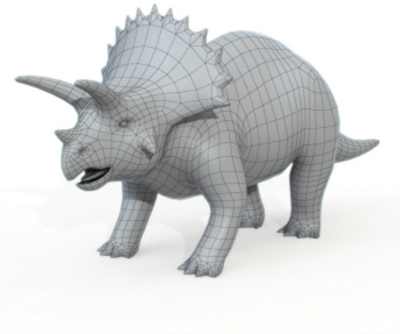
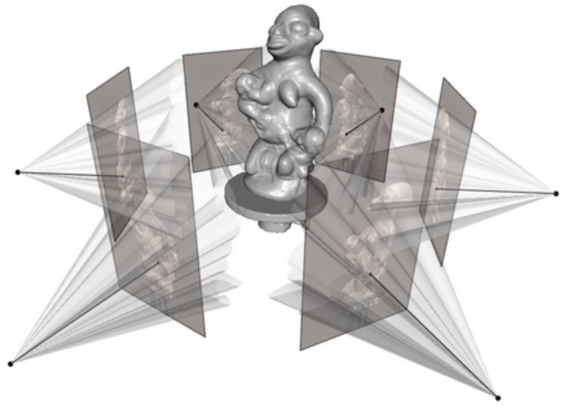
Computer Graphics notions and CG programming

INF630 - Refresher on Computer Science

Computer Graphics main SubFields

Modeling

How to create static shapes

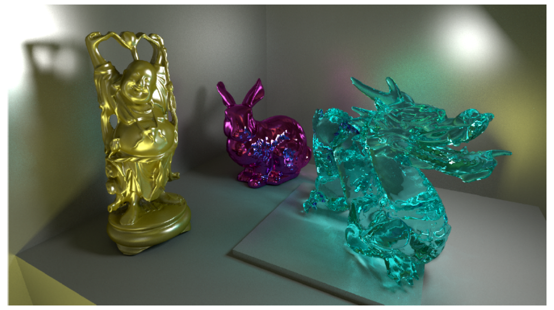
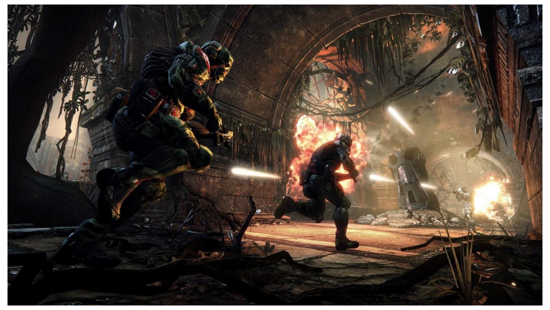
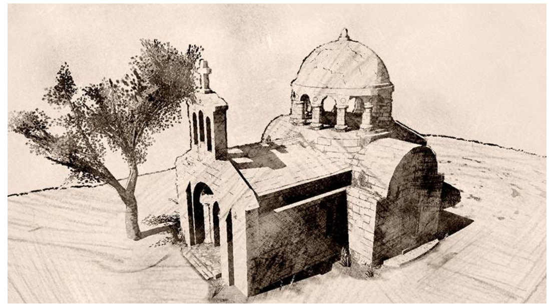
Animation

How to create and author time varying shapes



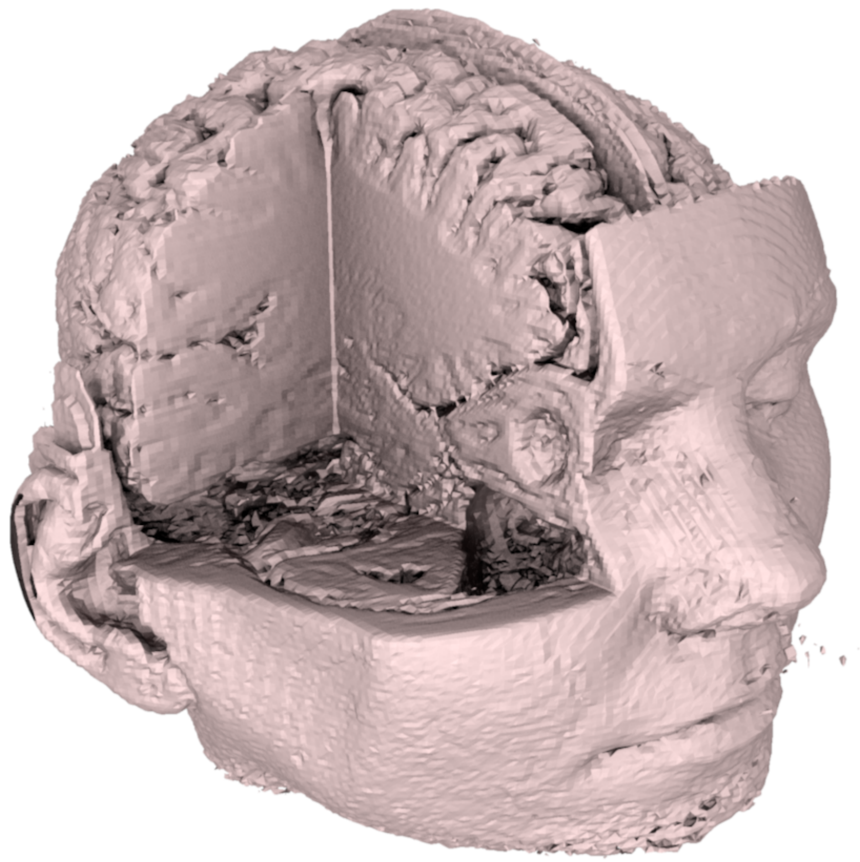
Rendering

How to generate 2D images from 3D data

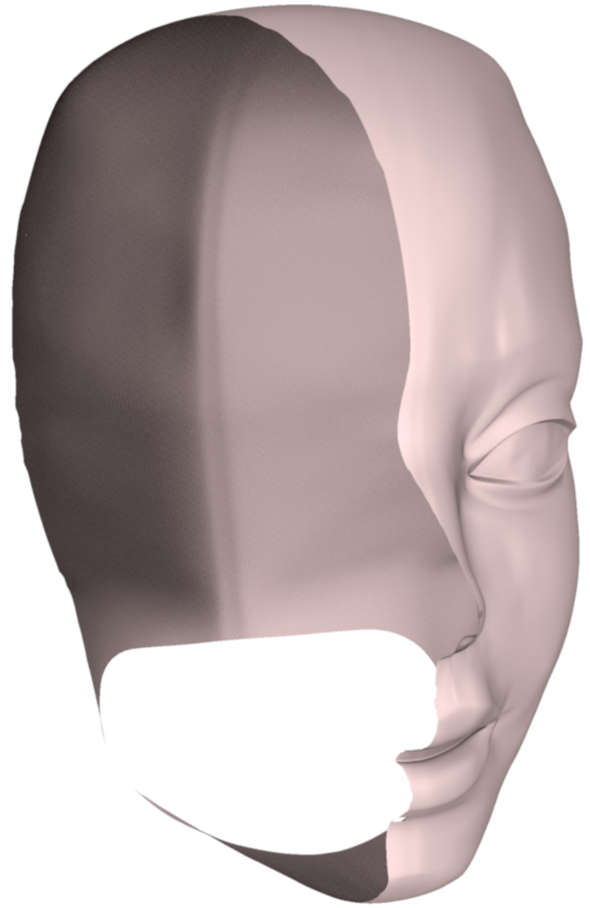
Representing 3D shapes for Graphics Applications

Volume representation



+ Accurate, handle density

Surface representation



+ Focus on visible part

+ Fast GPU rendering, low memory footprint

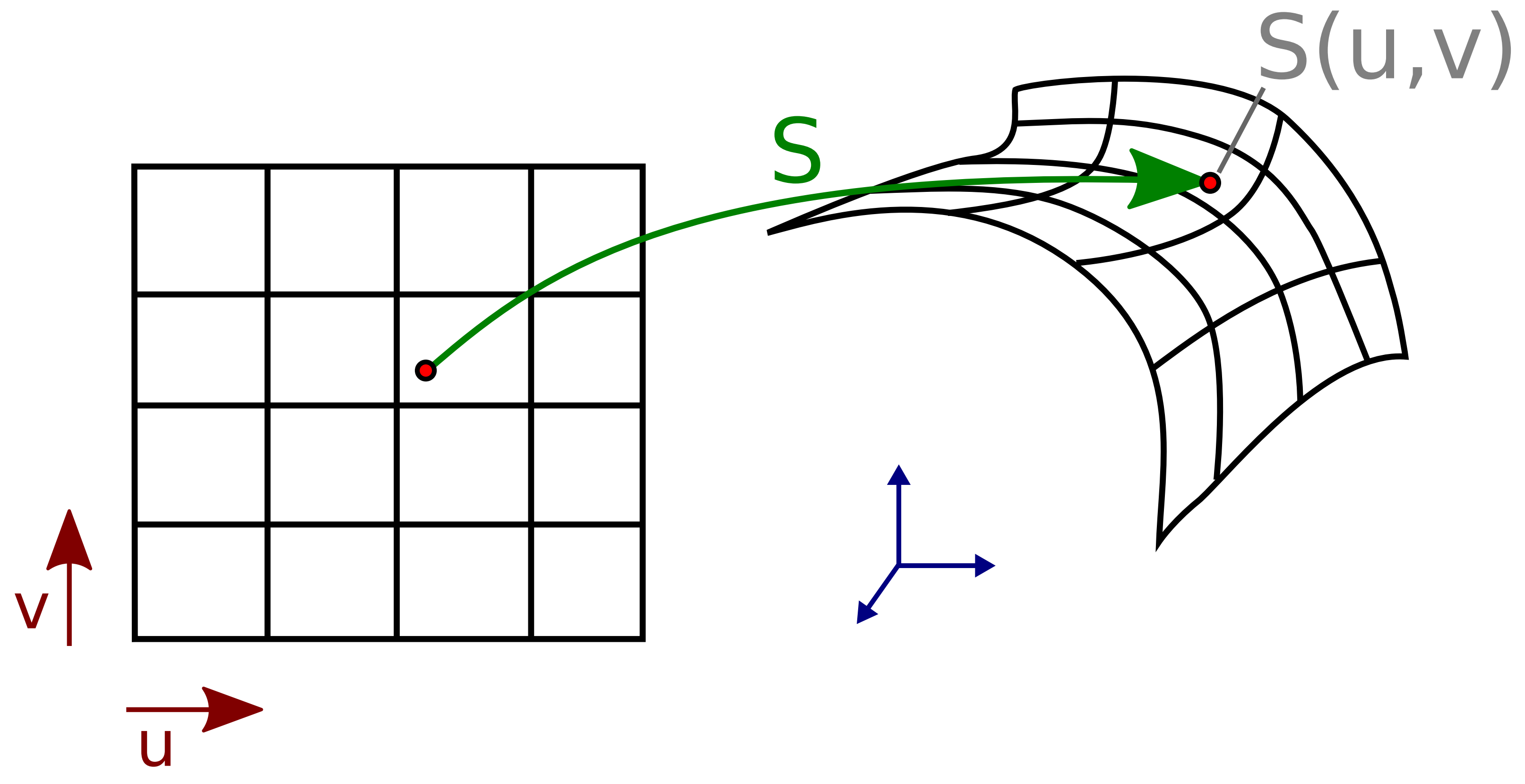
=> **Computer Graphics**: Mostly focus on representing **Surfaces**
=> **Scientific visualization**: Volume data

Two main representations for surfaces

Explicit representation

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

Parametric map

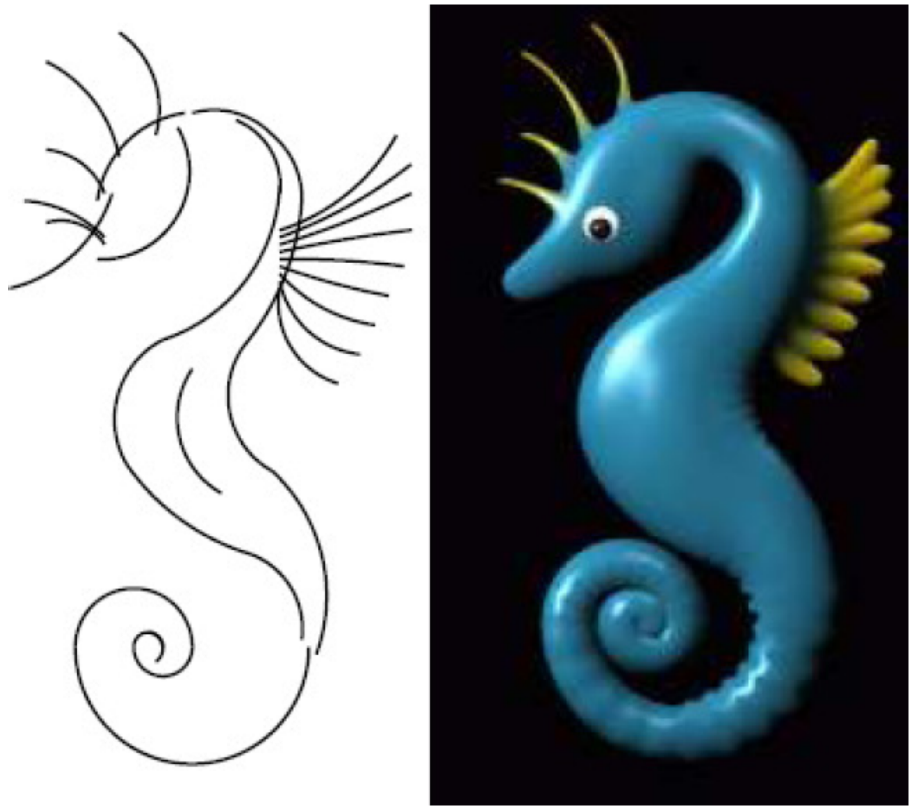


+ Neighborhood information

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

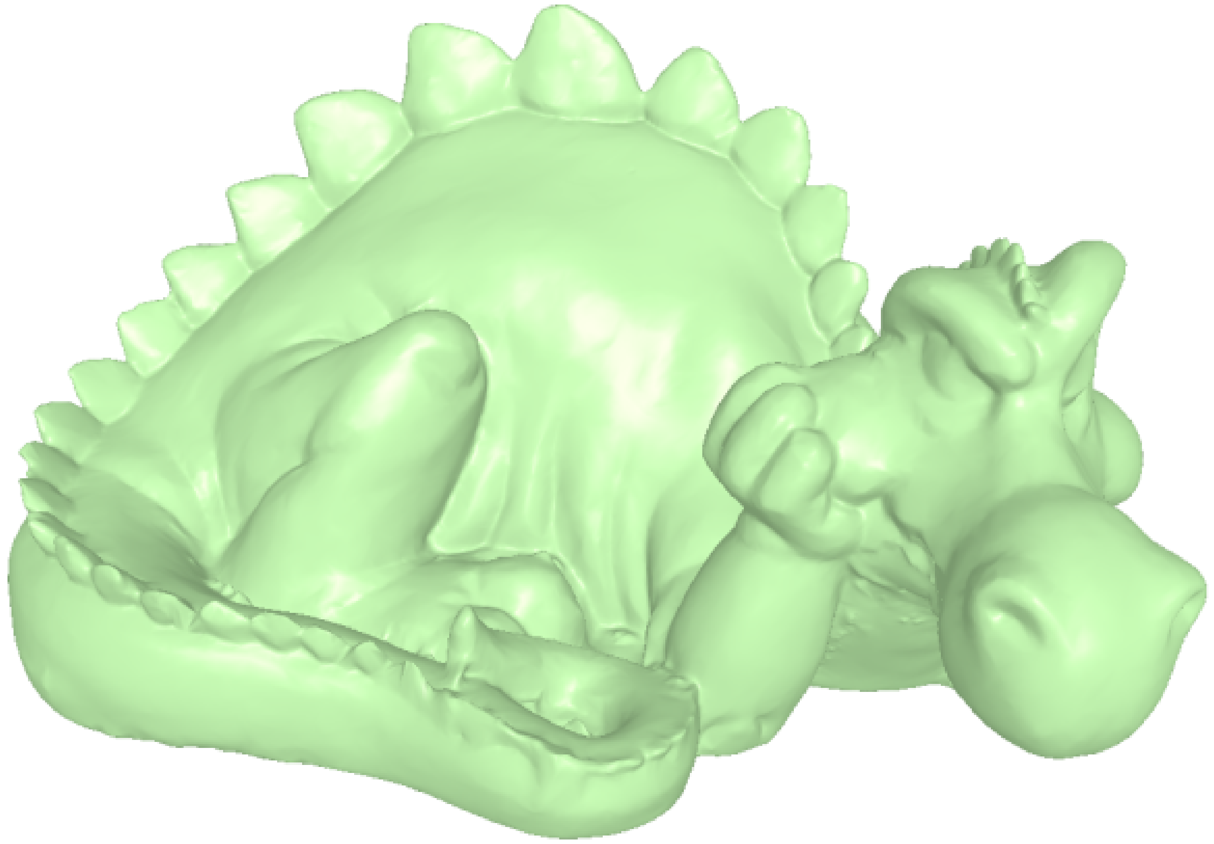
$$S = \{(x,y,z) \in \mathbb{R}^3 \mid F(x,y,z) = 0\}$$

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**

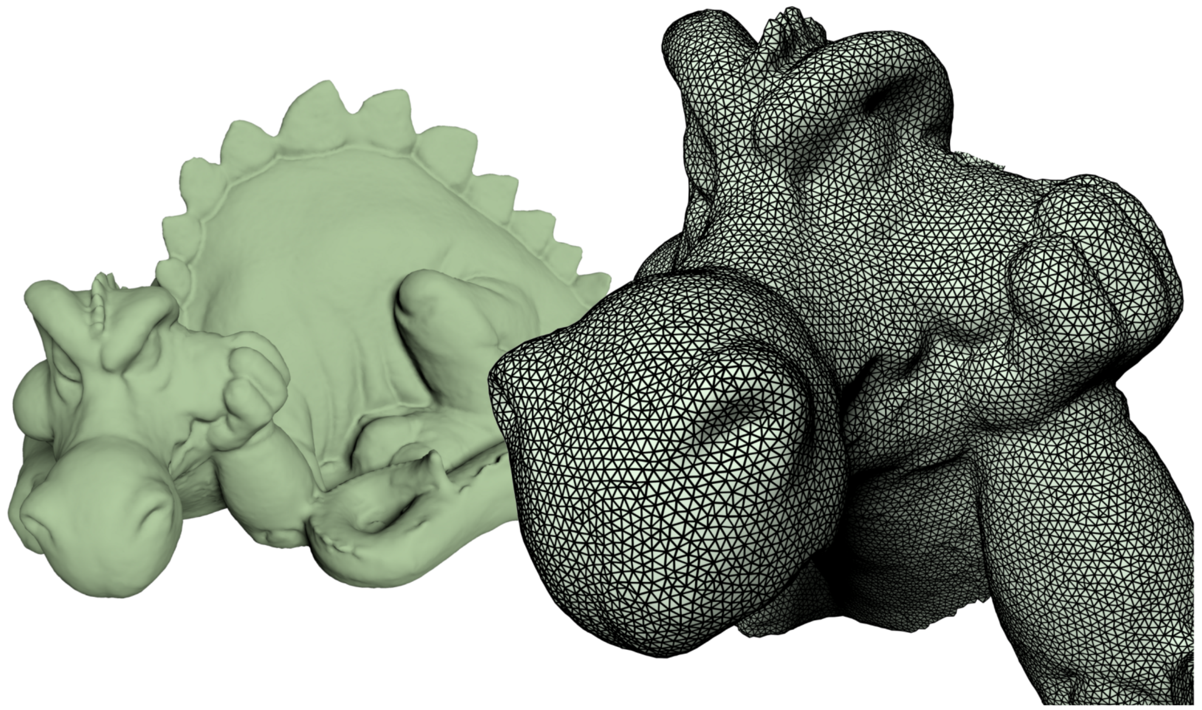
Ideal surface representation
- **Approximate** well any surface
- Require **few samples**
- Can be **rendered** efficiently (GPU)
- Can be manipulated for **modeling**

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

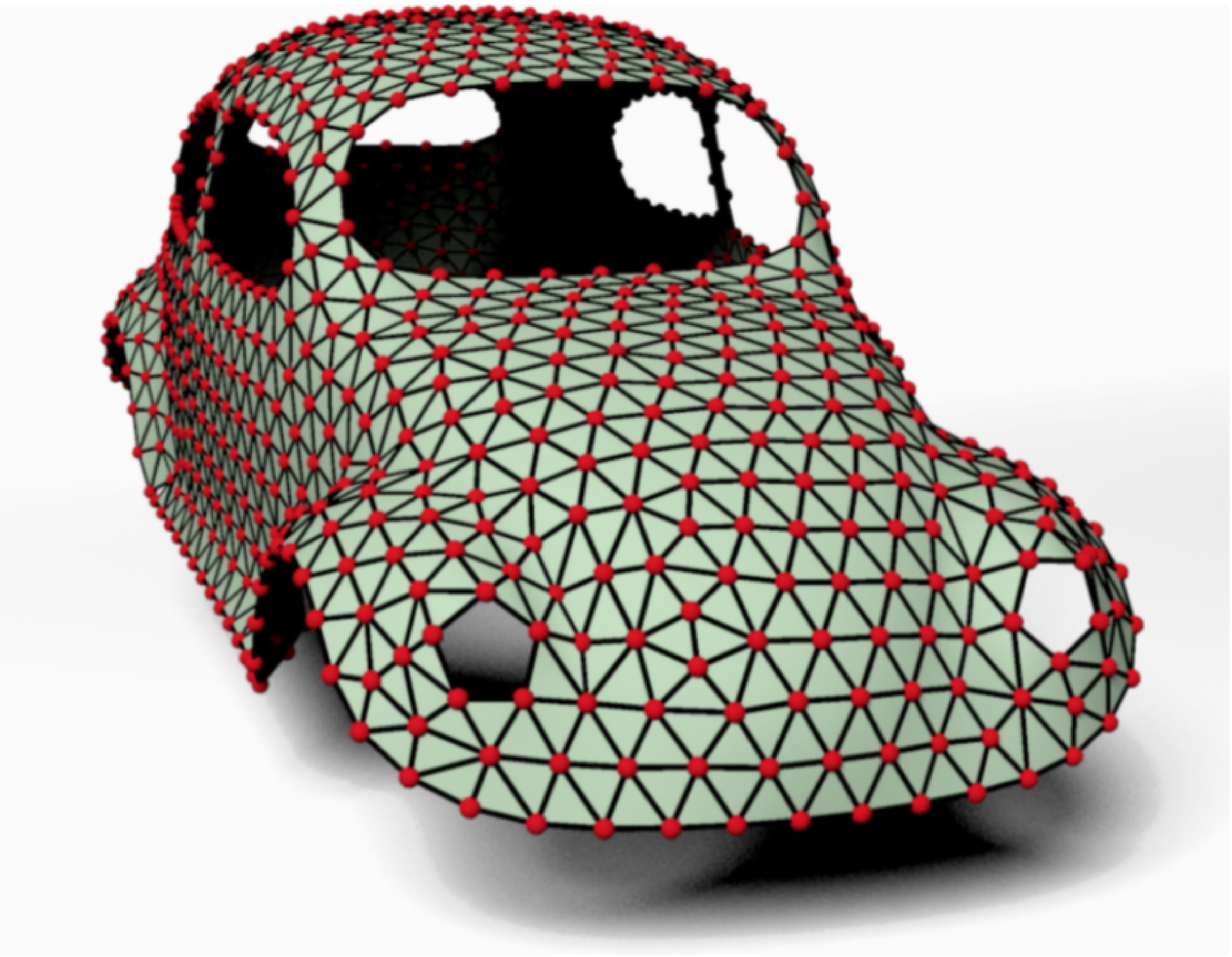
Meshes



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++

```cpp
#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;
}
```

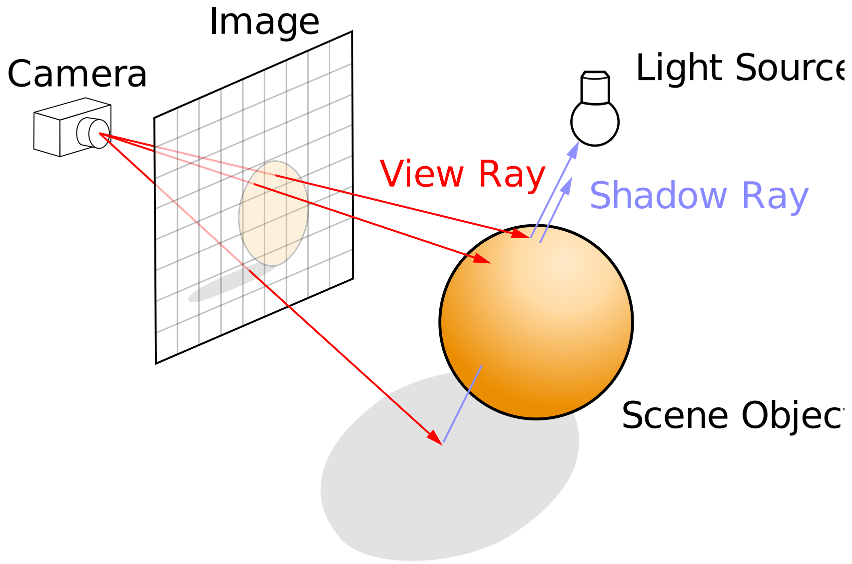
Example of 3D Mesh file

```
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
```

Open question: How to display it efficiently on screen?

How to render surfaces

Ray tracing

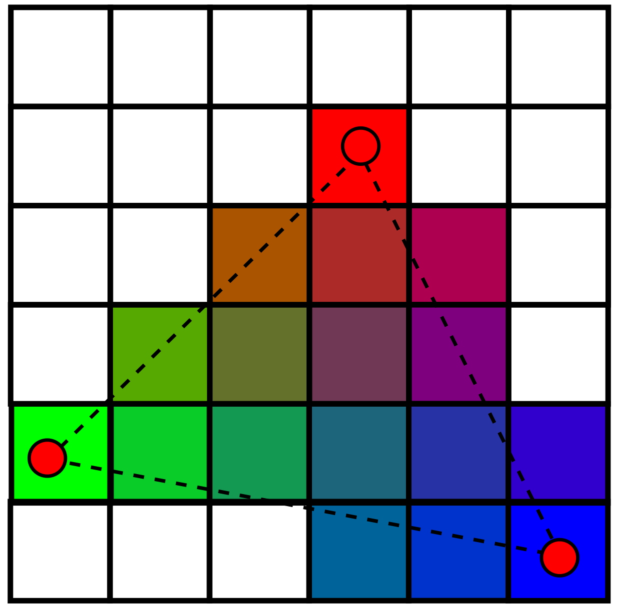


- *Throw rays from light-sources/camera*
- *Intersect rays with 3D shapes*
- *Pixel-wise computation*

\+ 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*

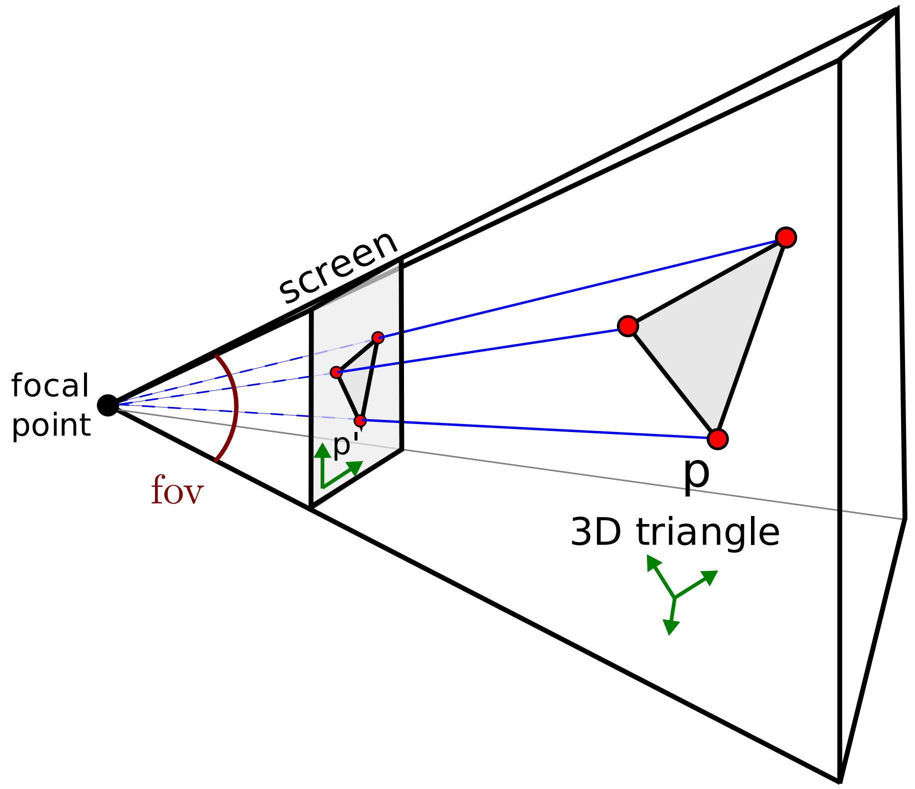
\+ Efficiently implemented on GPU
\- Limited to triangles
\- No native effects (shadows, transparency, etc)

=> The standard real time rendering with GPU

Projection/Rasterization

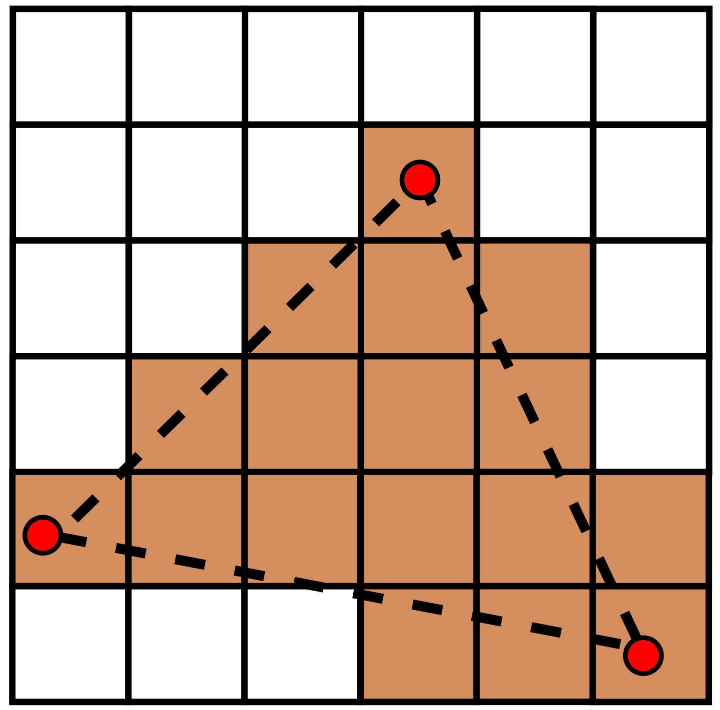
Object made of **triangles only**

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 $p' = \underbrace{\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathrm{T}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$

Similarily 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

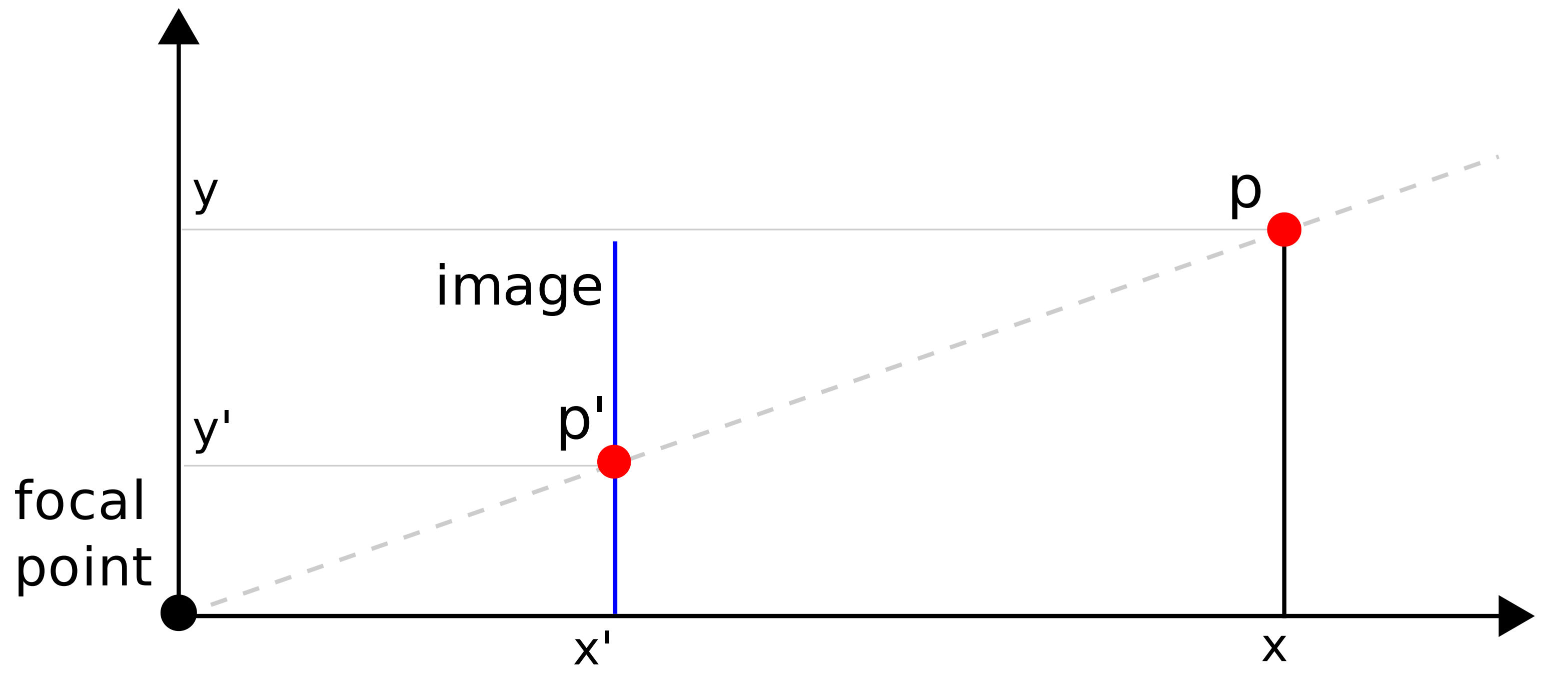
- 3D point $(x, y, z, 1)$ - translation applies.
- 3D vector $(x, y, z, 0)$ - translation doesn't apply.

Perspective and projective space

Modeling perspective projection requires division.

ex. in 2D (1D projection)

$$y' = x' \, \tfrac{y}{x} = f \, \tfrac{y}{x} \quad (f\text{: focal})$$



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 $z = 0$

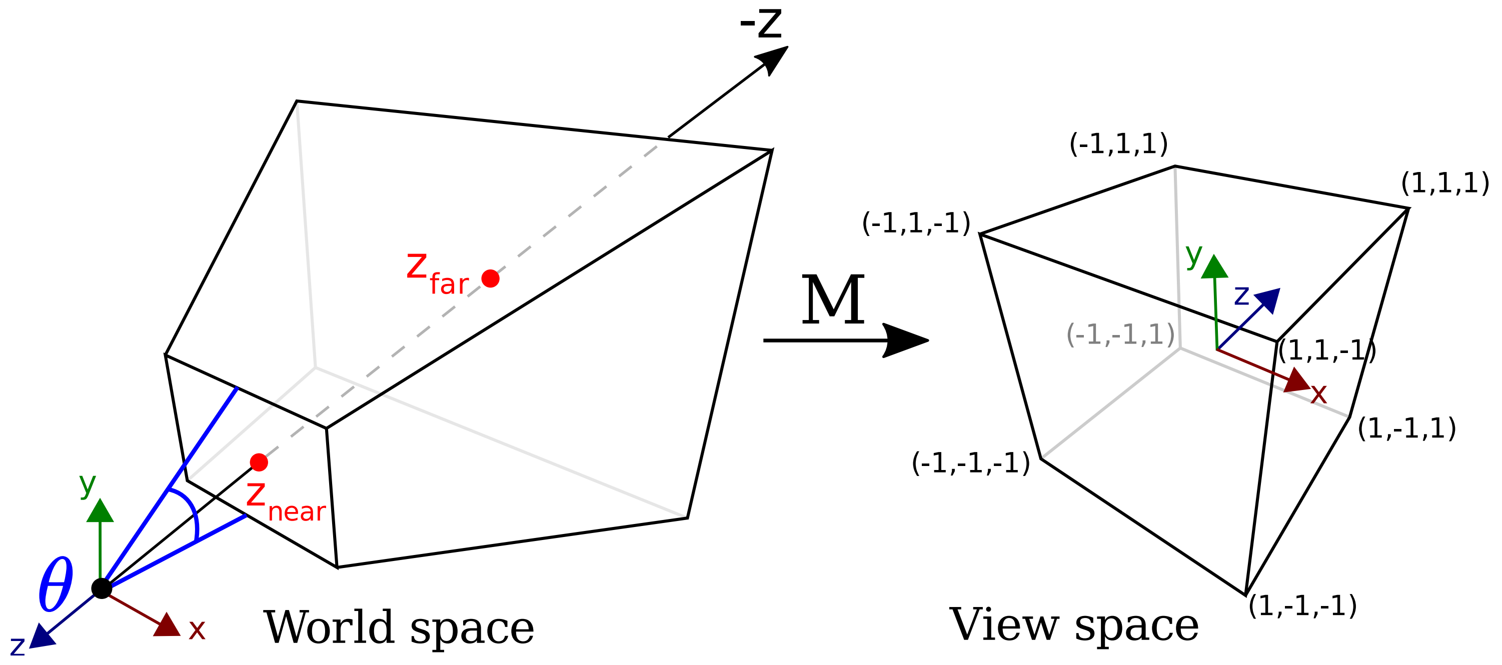
Real coordinates of points are obtained after normalization (division by z).

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
    - $p = (x, y, z, 1) \in \text{frustum} \Rightarrow p' = (x', y', z', 1) \in [-1, 1]^3$.



$$M = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

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

In practice

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

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

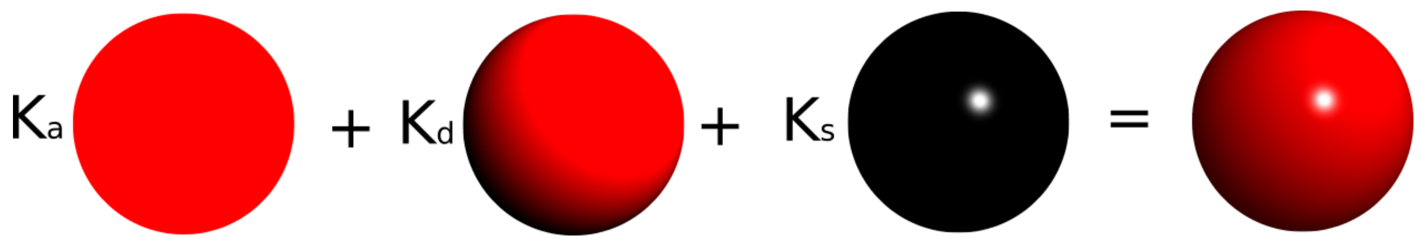
To which view space coordinates are mapped 3D world space points at z_{near}, z_{far} ?

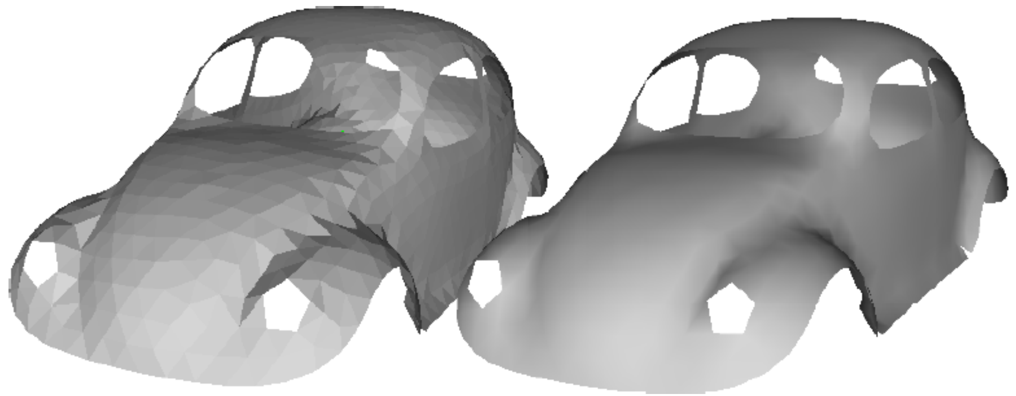
Per-vertex normal and illumination

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



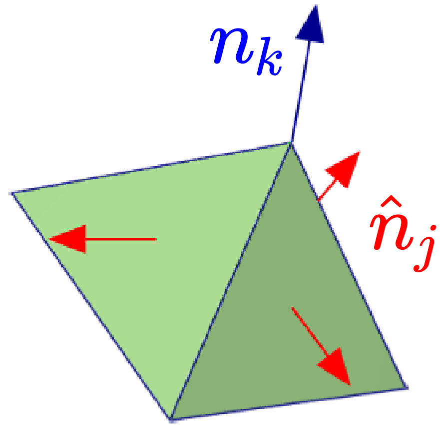


- *Phong shading* interpolates normals on each *fragments* of triangles, and compute illumination.

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.

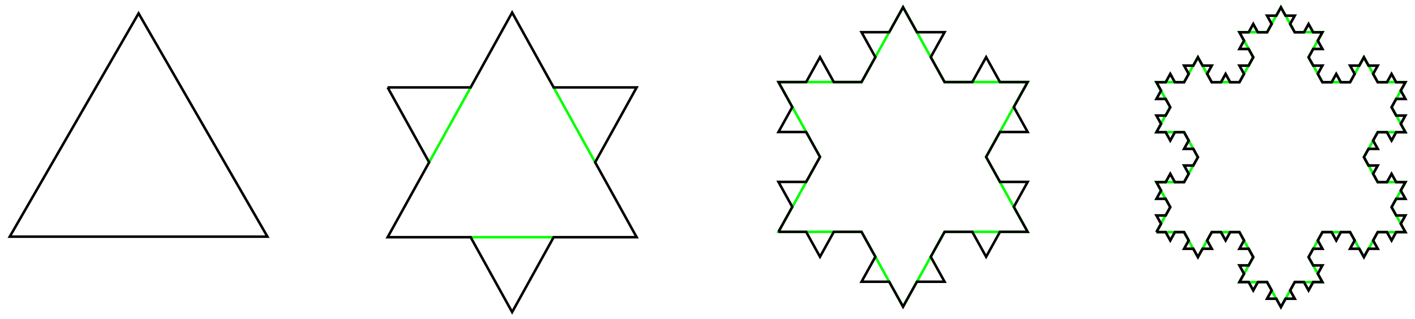


Fractals

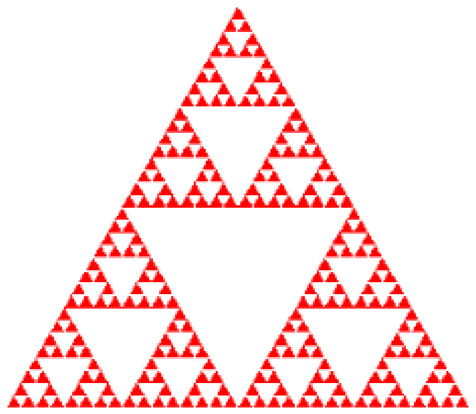
Idea: Recursively add self-similar details

Simple rule $\Rightarrow$ complex shapes

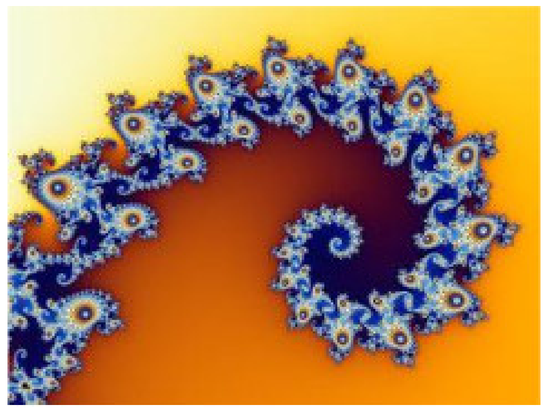
May look like complex natural details



Koch Snowflake



Sierpinski triangle, Shell:Oliva Porphyria



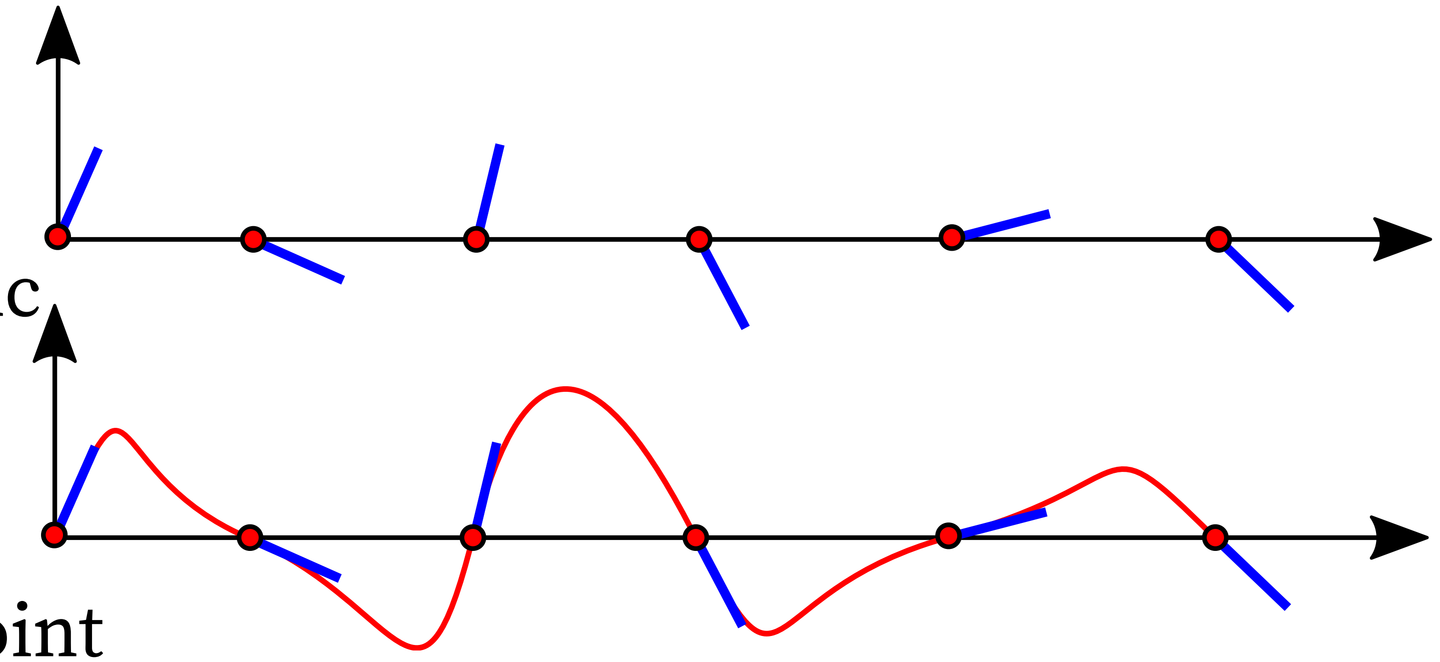
Perlin Noise

A widely used *noise* function.

Original article [An Image Synthesizer, Ken Perlin, SIGGRAPH 85]

Continuous Pseudo-random function

- Spatial variations are continuous, but non periodic
- Function can be evaluated at any point - deterministic



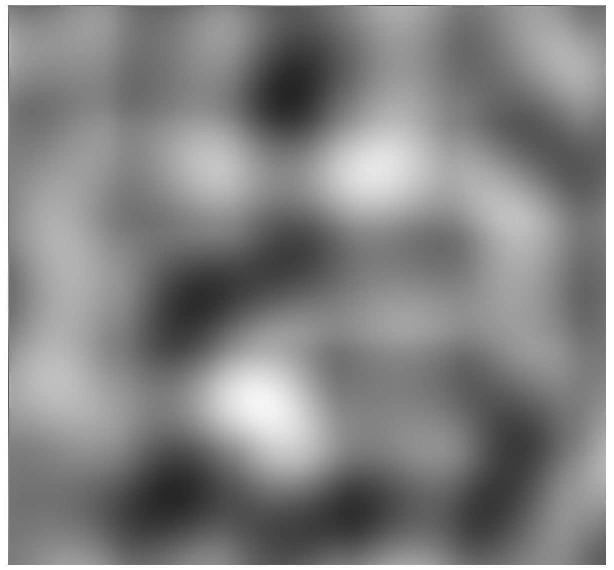
Creating a smooth function

- Compute pseudo-random gradient at each sample point

  Use some hash function

  ex. float hash(float n) { return fract(sin(n) * 1e4); }

- Interpolate smooth cubic curve between each points



(Algorithm for nD: Simplex noise)

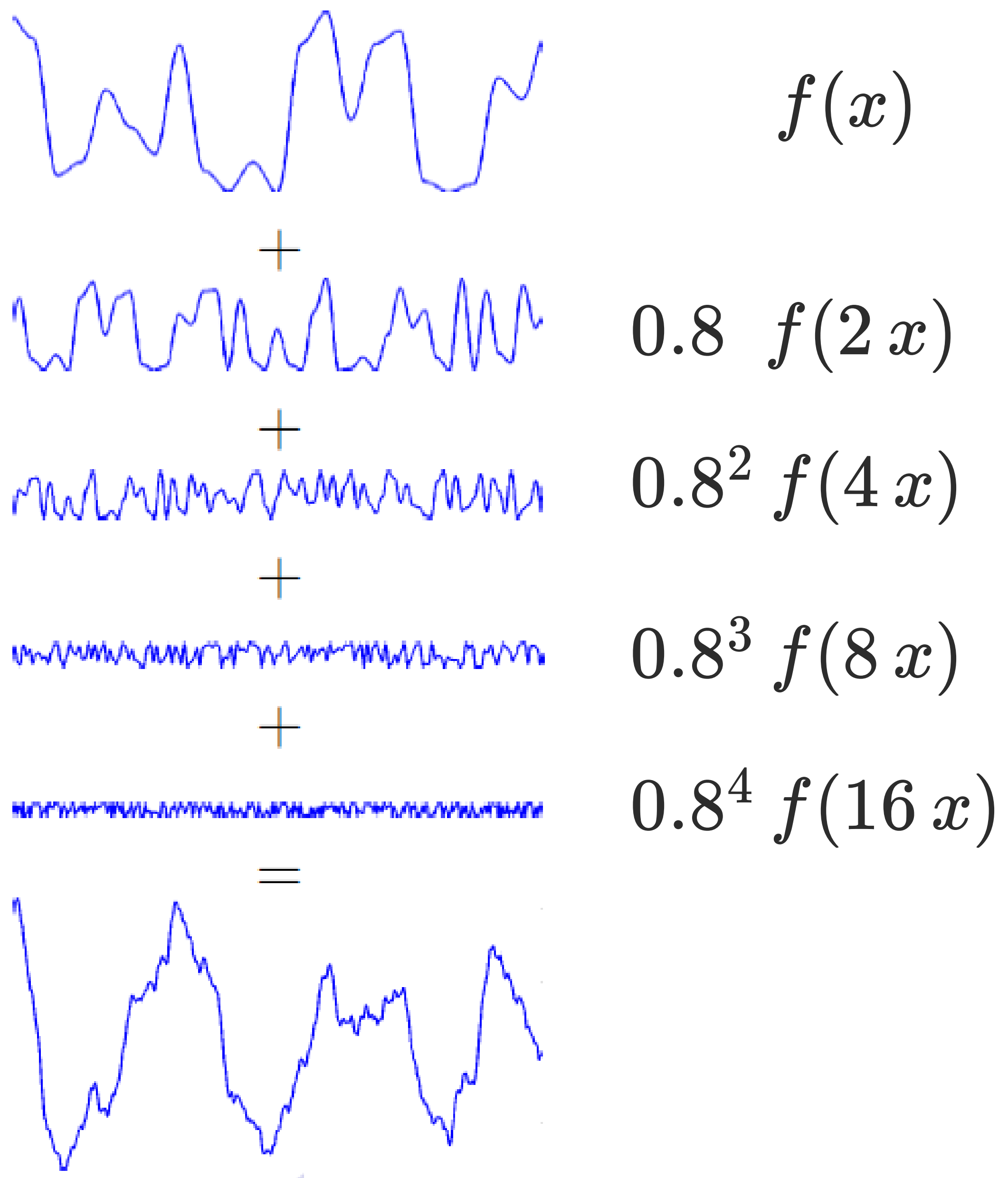
Fractal Perlin Noise

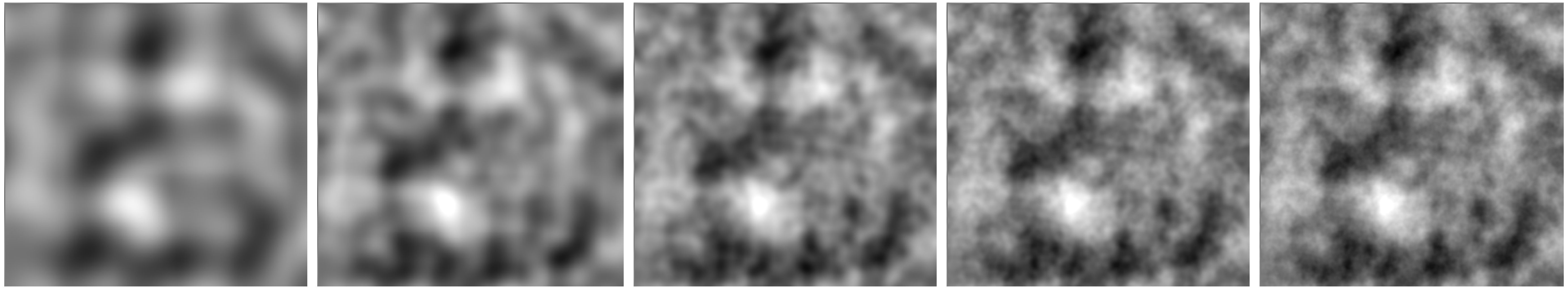
Sum over mutliple instance with increasing frequencies

f : Smooth Perlin Noise function

$$g(x) = \sum_{k=0}^{N} \alpha^k f(\omega^k\ x)$$

- N number of Octave
- α persistency ($1/\alpha$ attenuation)
- ω frequency gain



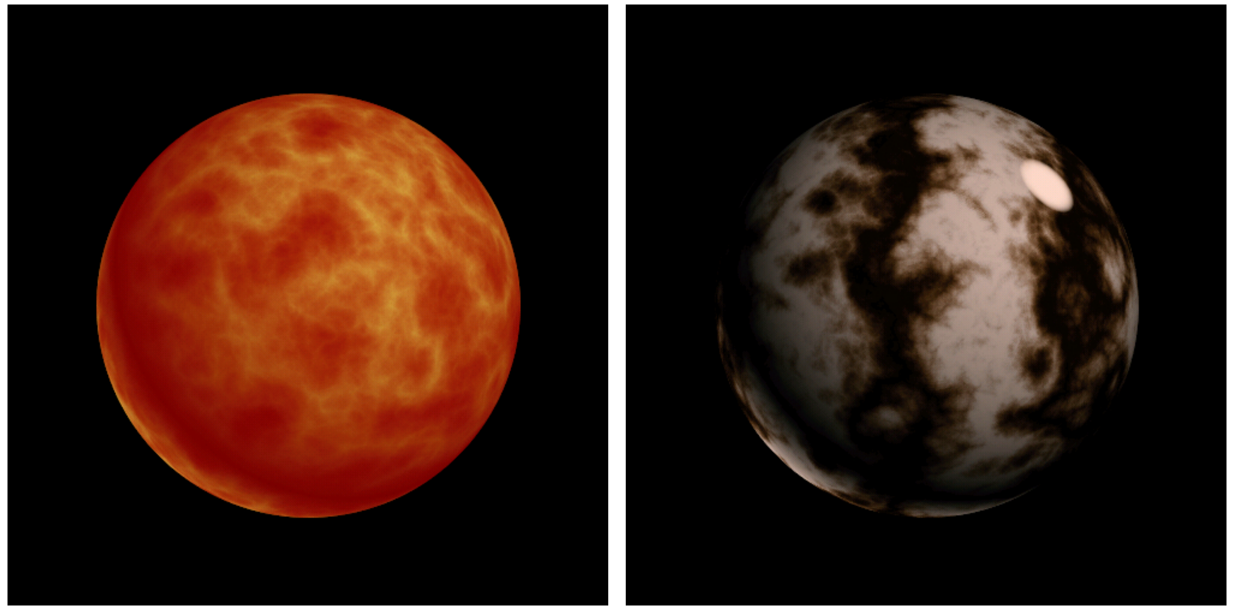


Perlin noise usage

Material texture

Ridge effect: $\sum_k \alpha^k |f(\omega^k x)|$

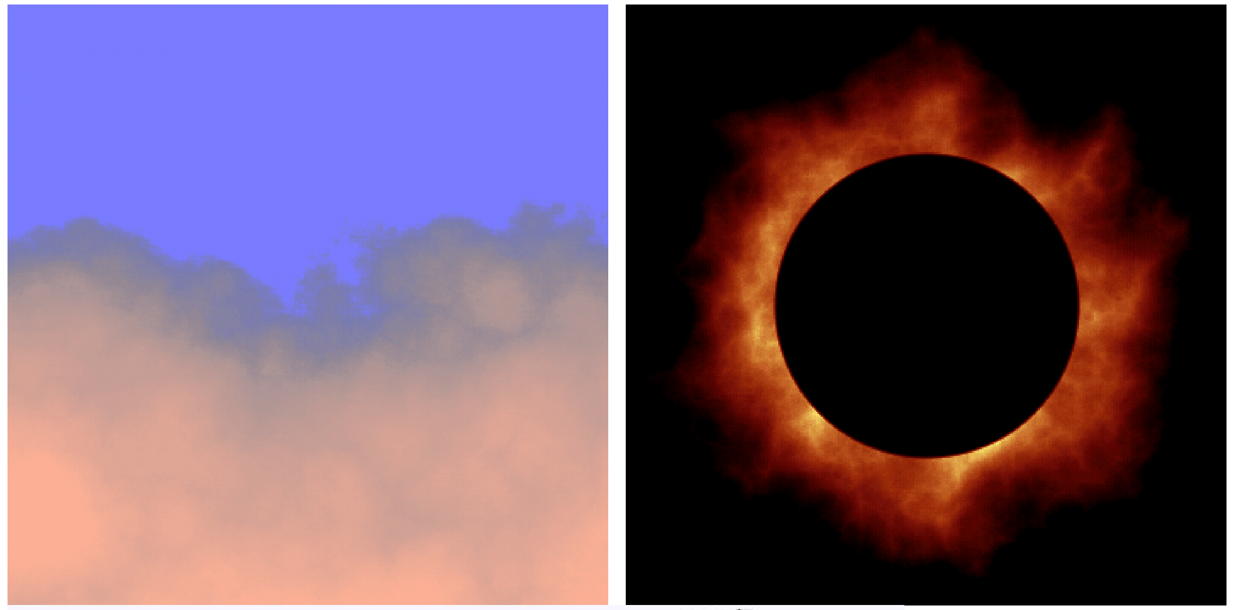
Marble effect: $\sin\left(x + \sum_k \alpha^k |f(\omega^k x)|\right)$



Animated textures

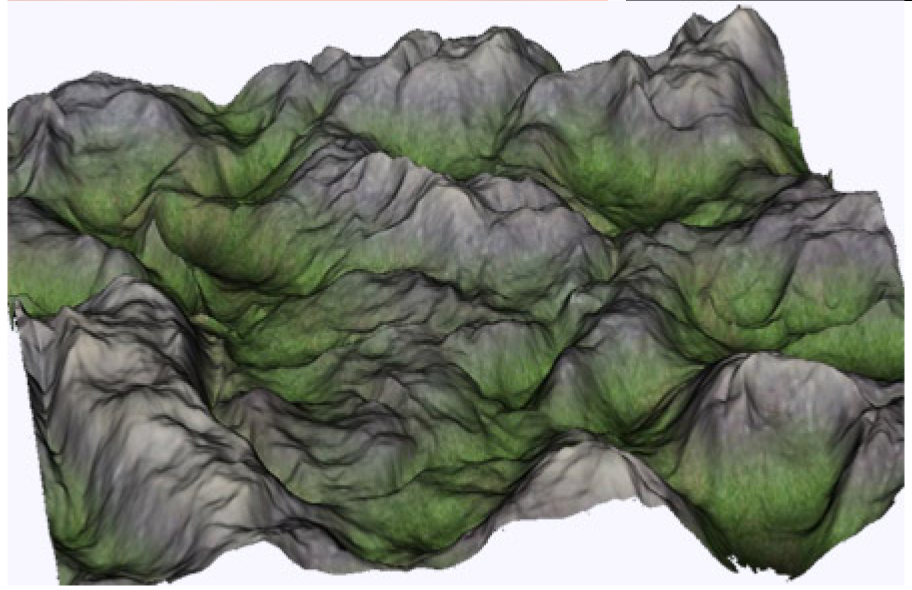
Translation: $f(x, y + t)$

Smooth evolution: $f(x, y, t)$



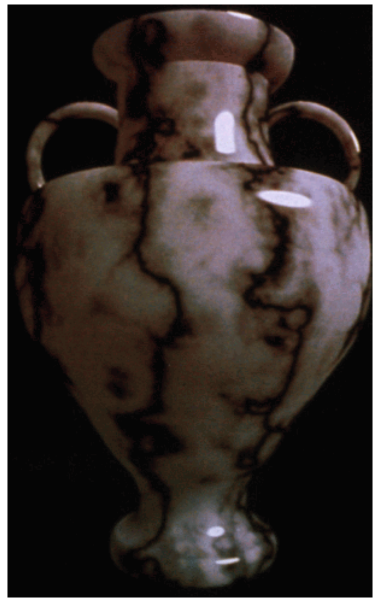
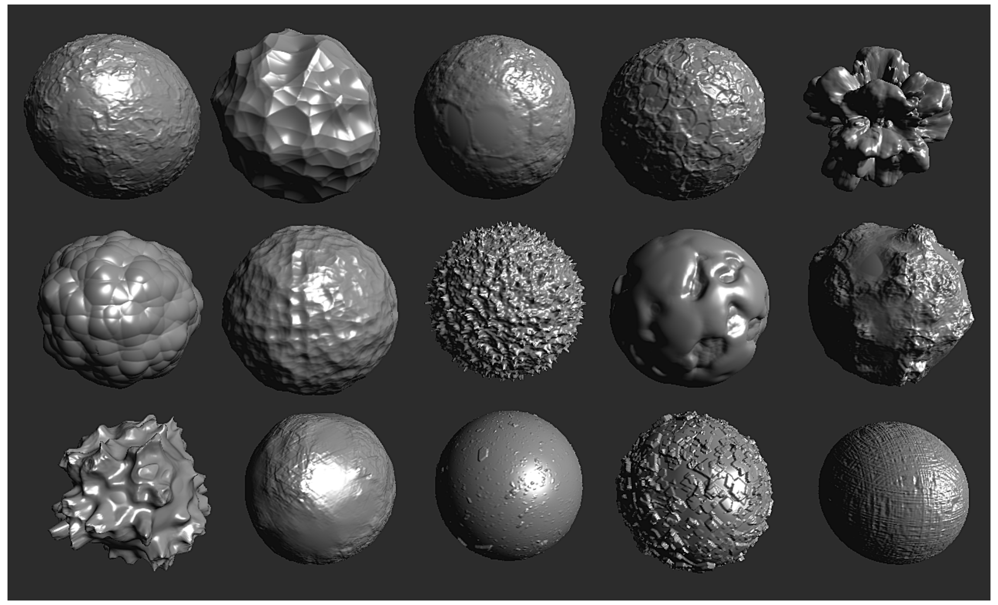
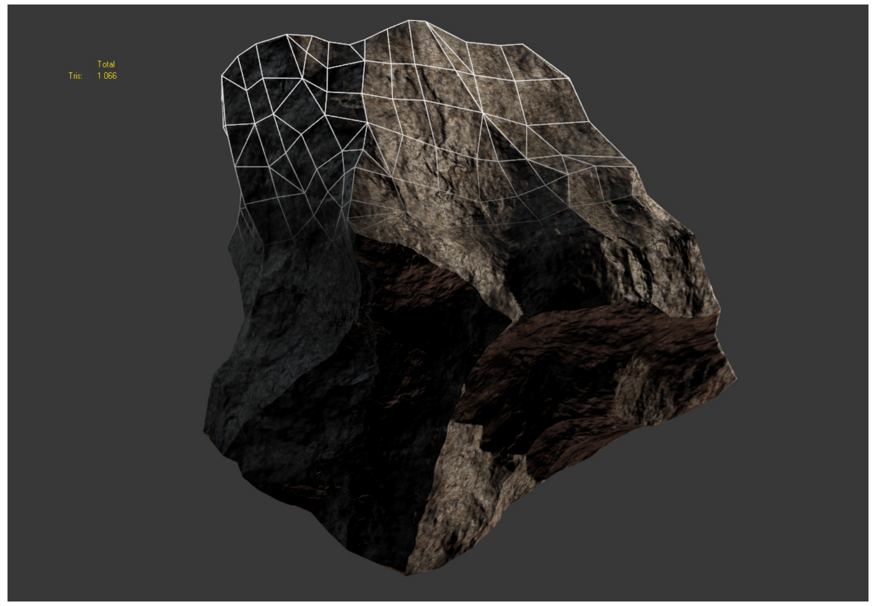
Mountain-looking terrain

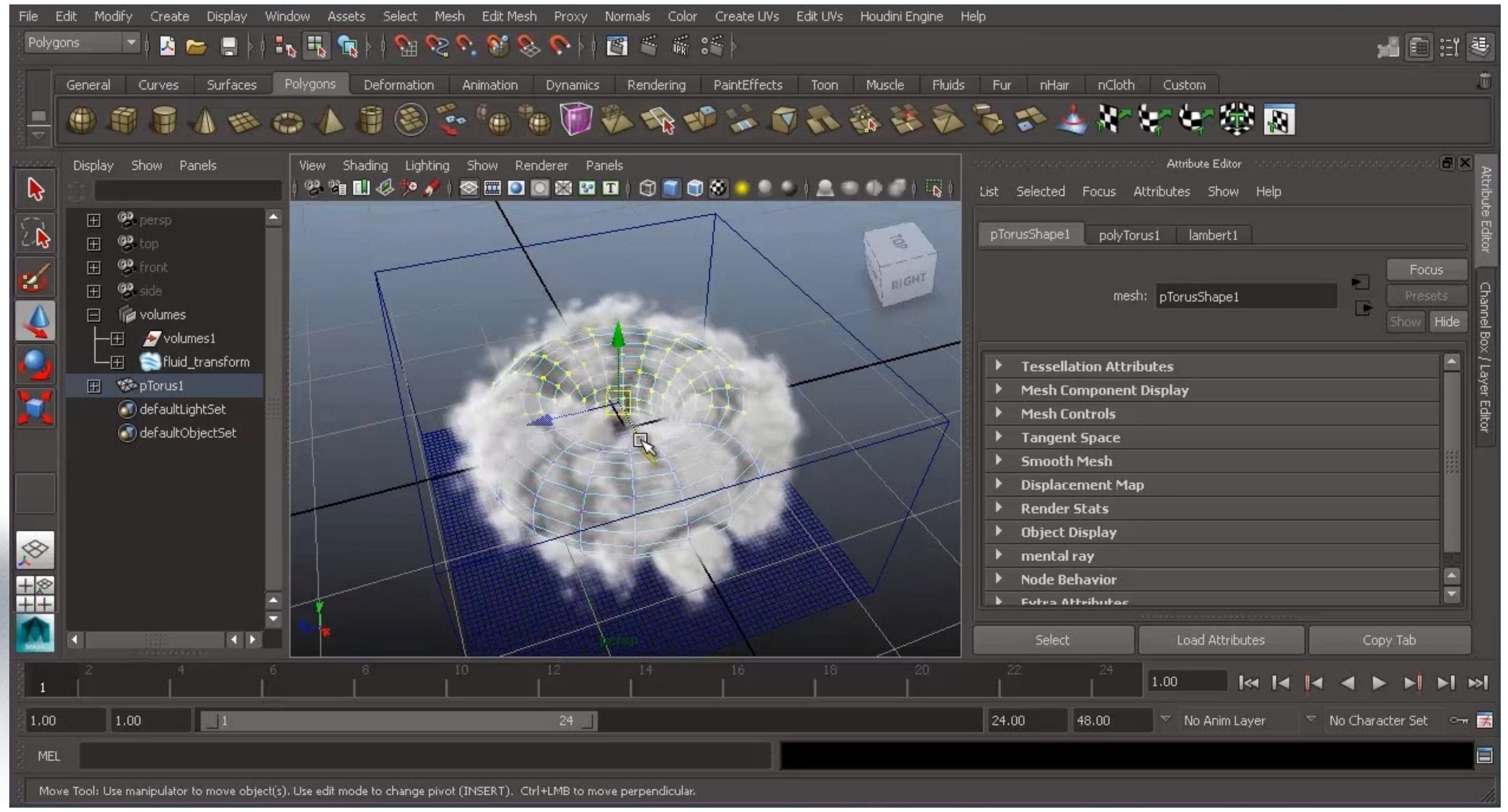
$z = f(x, y)$



Perlin noise applications

In almost every complex shapes ...

Look at Shader Toy + Noise example

Reminders on OpenGL

Objectives

Why C++ and OpenGL

- C++ is the dominant language for visual computing and (the core of almost every scientific softwares in general)

$\Rightarrow$ You will use (at least - if not develop) library that are coded in C++ underneath.

  - *Maya, 3DStudio, Houdini, Catia, Unreal Engine, etc.*

- OpenGL is the low-level base for real-time CG.

  You may use higher-level framework later (Unity, Unreal, etc.)

- However, **OpenGL** is *everywhere* underneath (/direct3D on Windows)

=> Give you (remind you) the fundamental low-level concept on which other tools are based

=> Explains origin of keywords, working behavior, you may find in high level libraries

  shaders, vertex, normals, phong, transform, etc

OpenGL

OpenGL : **Open** **G**raphics **L**ibrary

OpenGL = An API for drawing 2D and 3D graphics.

- Cross plateform
- Low level, high performance
- Design to communicate with GPU capability

OpenGL standard is defined by the **Khronos Group**

(*Consortium of several Graphics related companies NVIDIA, Intel, Samsung, Huawei, Sony, Valve, AMD, Google, etc.*).

What is OpenGL

OpenGL is **not** a software, nor a library of code

=> It is an API = **set of functions signatures and types** (*expressed in C language*).

- Implementation may depends on your Graphics Card/driver, OS.
- OpenGL functions may lead to GPU hardware implementation, or can be emulated by software

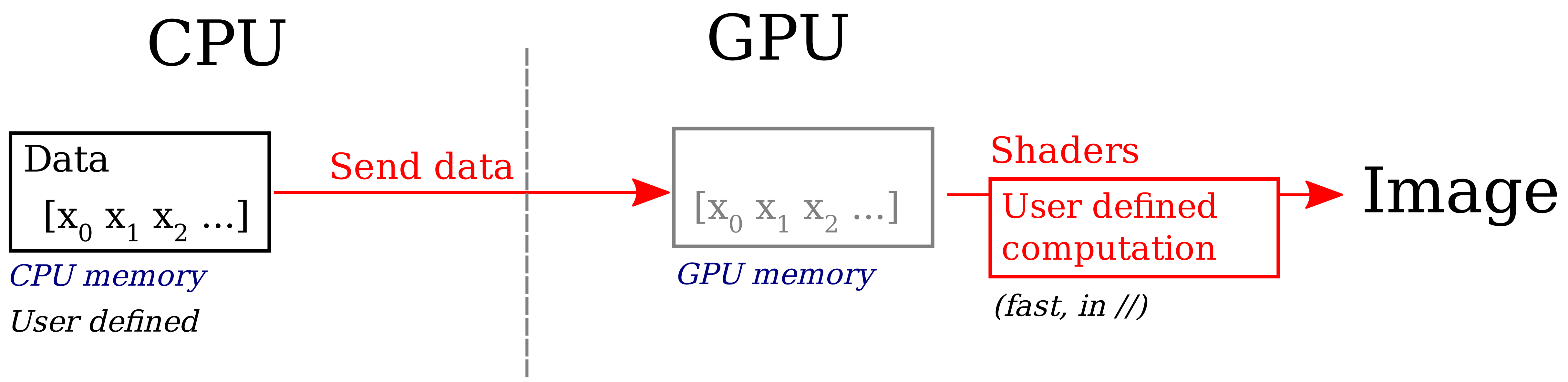
OpenGL focuss primarily on drawing Graphics

- It doesn't handle GUI (buttons, sliders, etc)
- It doesn't handle user events and other hardware (mouse, keyboard, sound, etc)
- It doesn't handle window in which your draw
- It doesn't provide high level 3D scene components (camera, lights, primitives, transformations, etc)

What OpenGL provides

Provides low level functions to **send raw data** (buffer of values) on your GPU memory (if available).
- You send your 3D model as buffer of coordinates and faces connectivity.
- Buffer = contiguous set of values in memory.

Allows to **manipulate your data** via *shaders*
- Shaders = Programs written in GLSL (openGL Shading Language), executed on the GPU (in parallel)
- Shaders take as input your data, and output a colored image (by default displayed on screen).
- You code all needed computation (coordinate/camera transformation, illumination) on shaders



Pro

Extremely fast: Direct communication CPU/GPU; only execute what you have coded.
Very generic: You compute what you want, not necessarily only graphics.

Cons

Very low level: You code everything (time consuming, error prone).
May be tricky to setup (driver compatibility, OpenGL version, manual window handling, etc).

When should you use OpenGL

Require full **speed** from your computation - only execute what you need

Require full **control** on your computation - specify directly what the GPU executes from your data

Ex.

- *Developing a Game Engine, or a dedicated rendering engine (browser, etc)*
- *Rendering huge dataset, unconventional transformation/rendering on your data*

When you can avoid OpenGL

Quickly setup and interact with standard 3D scene

ex. Common video game development.

Example of other higher tools:

- Havok, Unreal engine, Crytek engine, etc (existing game engine)
- Unity
- Three.js (web)

OpenGL History

[1992] OpenGL 1.0
- No shaders (fixed graphics pipeline)
- Higher level than today (vertex, normal, color, transformation matrix, lights)
- Immediate mode slower than today (every triangle is displayed one by one at every call)
- Depreciated functionality since 2008, not compatible with mobile/web.
- Still commonly encountered (easy to deploy on PC)

[2004] OpenGL 2.0
- GLSL vertex and fragment shaders (programmable graphics pipeline)
- Use of data buffers (named)

[2008] OpenGL 3.0
- Depreciate fixed pipeline functions
- Use of generic data buffers
- Addition of geometry shader

[2010] OpenGL 4.0
- Addition of compute and tesselation shaders

[2016] Vulkan 1.0
- Full new API design (multiple GPU handling, generic computation)
- Lower level than OpenGL (ex. manual synchronization, select GPU capability)

Example of a triangle

Low level: need to think about all steps

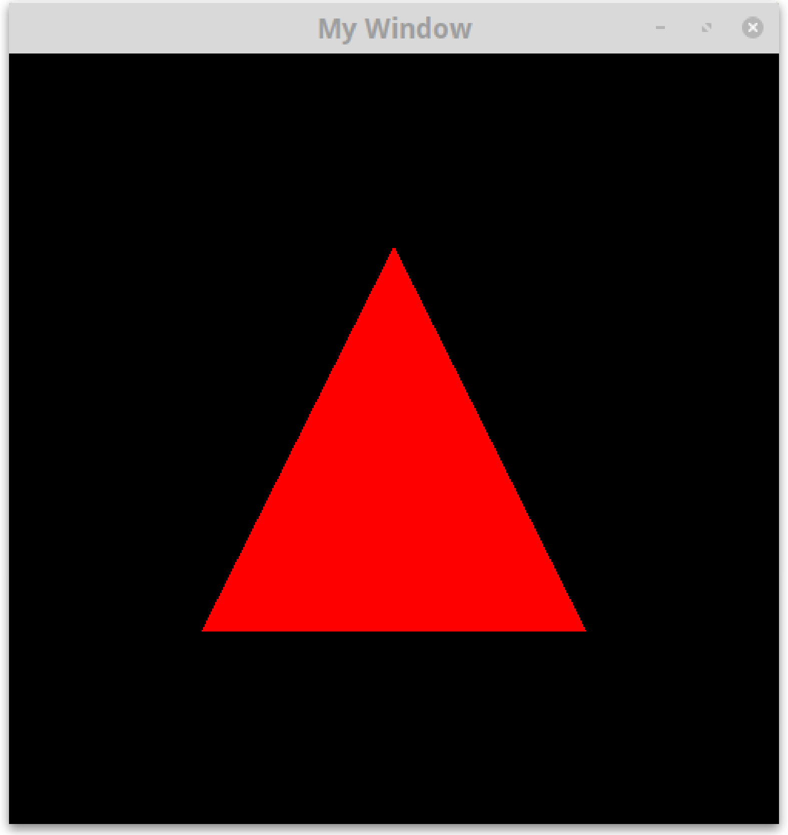
Example: Code to draw a single uniform colored triangle (~ 200 lines)

```
#include <glad/glad.hpp>
#include <GLFW/glfw3.h>
#include <iostream>
#include <vector>



int main()
{
    std::cout<<"*** Init GLFW ***"<<std::endl;
    const int glfw_init_value = glfwInit();
    if( glfw_init_value != GLFW_TRUE ) {
        std::cerr<<"Failed to Init GLFW"<<std::endl;
        abort();
    }


    std::cout<<"*** Create window ***"<<std::endl;
    const int window_width  = 500;
    const int window_height = 500;
```



FYI - Vulkan ~ 1500 lines.

Geometry processing libraries

Mesh structure, algorithms, possibly viewers.

Development libraries

- LibIGL : Simple to use, efficient, geometry processing library
  Origin: ETH Zurich, Interactive Geometry Lab (IGL)
- CGAL : Advanced generic geometry processing library, Heavily templated
  Origin: Inria (Sophia)
- GeoGram : KISS geometry processing lib. Surface and volume structure.
  Origin: Loria (Nancy)

Viewer (+lib)

- Graphite : Surface and Volume geometry processing. API (Geogram) for geometry processing.
  Origin: Loria (Nancy)
- Meshlab : Mesh viewer and processing
  Origin: Visual Computing Lab, Pisa.

Software

- Blender : Full 3D modeler, animation, renderer (Open source)
  Provide python API for coding
  Real swiss knife for 3D Graphics !

Usefull CG programming library

Usefull libs

- Eigen : Efficient C++ Matrix toolbox
- GLM : GLSL compatible C++ structures (vec3, mat3, etc.)
- Assimp : Mesh loader
- DevIL : Image loader

Minimalistic GUI (OpenGL compatible)

- ImGui
- NanoGui
- AnTweakBar (ancestor of ImGui and NanoGui)

Full framework

- Qt : Cross plateform development including classes, GUI, visual designer, etc.