INF585 - 3D Animation

Objective and organization of the class

- Give you fundamental notions of **3D computer animation**
- Train at **practical CG/animation programming**
  - TD in computer rooms - C++, OpenGL

Mostly practical-oriented class

Provisional TD schedule (may be adapted)

[1] (08/01) - Reminder C++, Tutorial OpenGL (not animation)
[2] (15/01) - Descriptive animation - Particle based animation
[3] (22/01) - Descriptive animation - Keyframe interpolation
[4] (29/01) - Physically based animation - Particle based, boids
[5] (05/02) - Physically based animation - Cloth animation
[6] (12/02) - Physically based animation - Fluid animation
[7] (19/02) - Character animation: Skeletal animation, skinning
[8] (05/03) - Project (by pair)
[9] (12/03) - Project (by pair)
[x] (19/03) - Exam

Grading based on

1. Your project: Code + small report
2. Written exam (19/03)

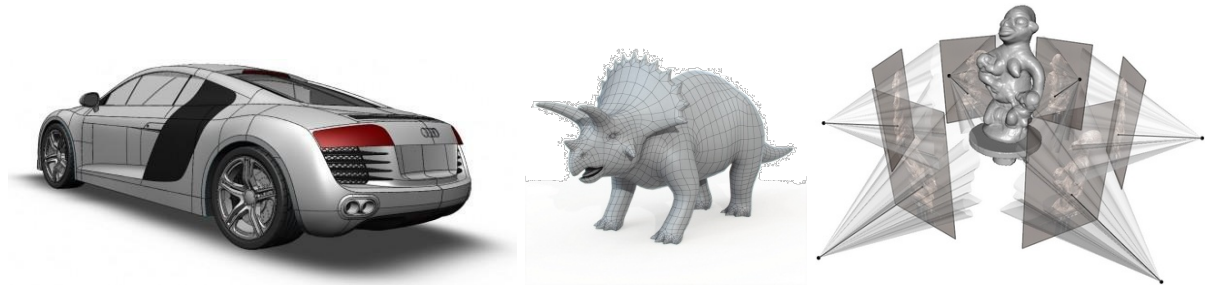
Brief reminder/introduction to Computer Graphics in general

- CG-Generalities
- Representing a surface
- Rendering a surface
- Fundamention notions
  - Projective space
  - Smooth shading

Computer Graphics main SubFields

Modeling

How to create static shapes

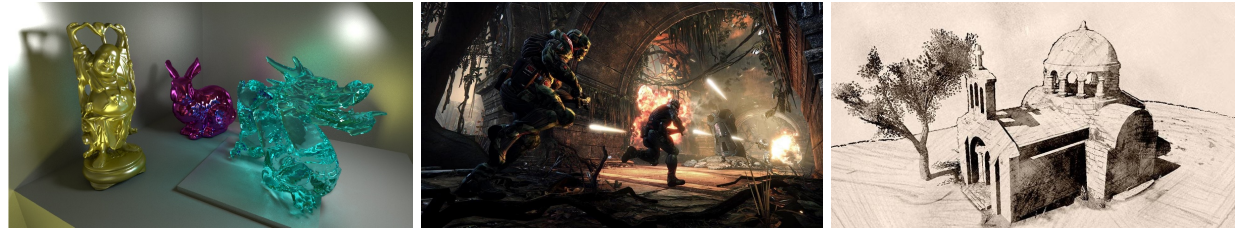


Animation

How to create and author time varying shapes

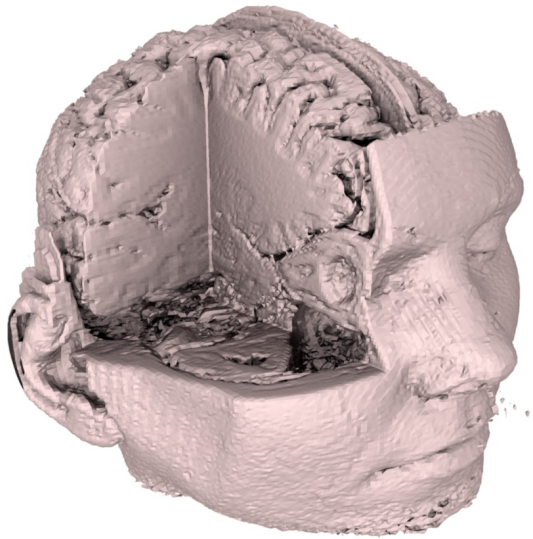
Rendering

How to generate 2D images from 3D data



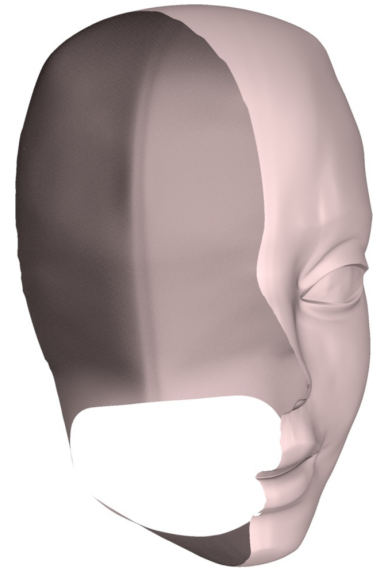
Representing 3D shapes for Graphics Applications

Volume representation



+ Accurate, handle density

Surface representation



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

=> **Computer Graphics**: Mostly focus on representing **Surfaces**

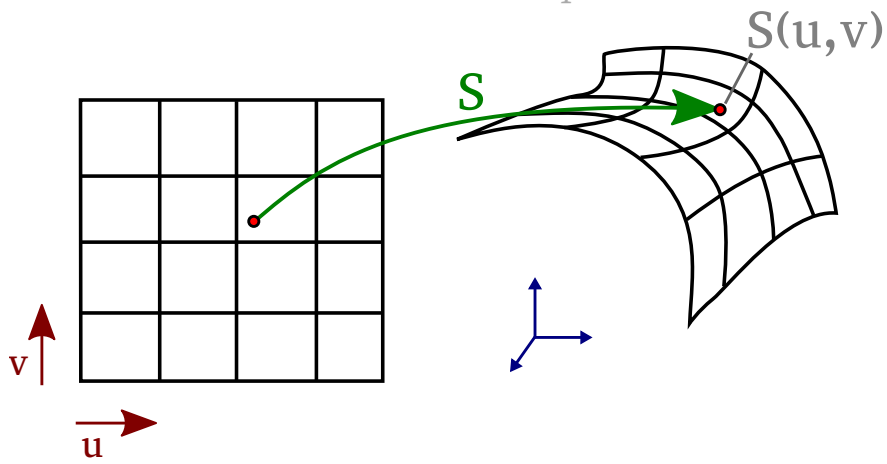
Digital representation of surfaces

Two main representations for surfaces

Explicit representation

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

Parametric map



+ Neighborhood information

Implicit representation

$$\{(x, y, z) \mid F(x, y, z) = 0\}$$

Isosurface from field function

+ Topological modification

Two main representations for surfaces

Example for a sphere

Explicit representation

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

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

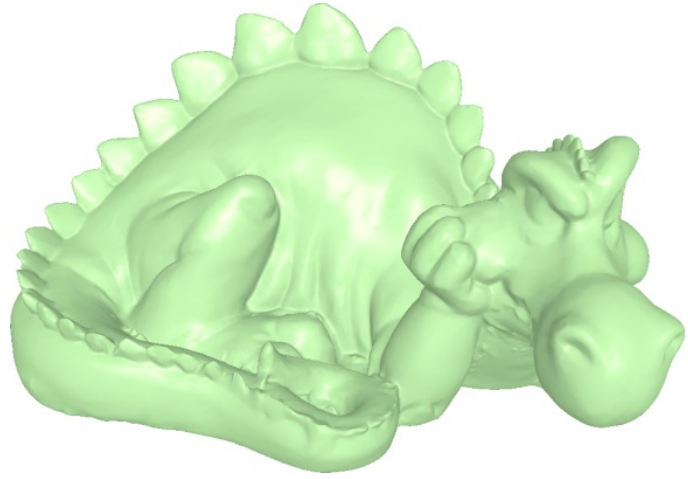
$$\{(x, y, z) \mid F(x, y, z) = 0\}$$

Isosurface from field function

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

Problematic of surface representation

What function represents this dragon ?



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

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

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*

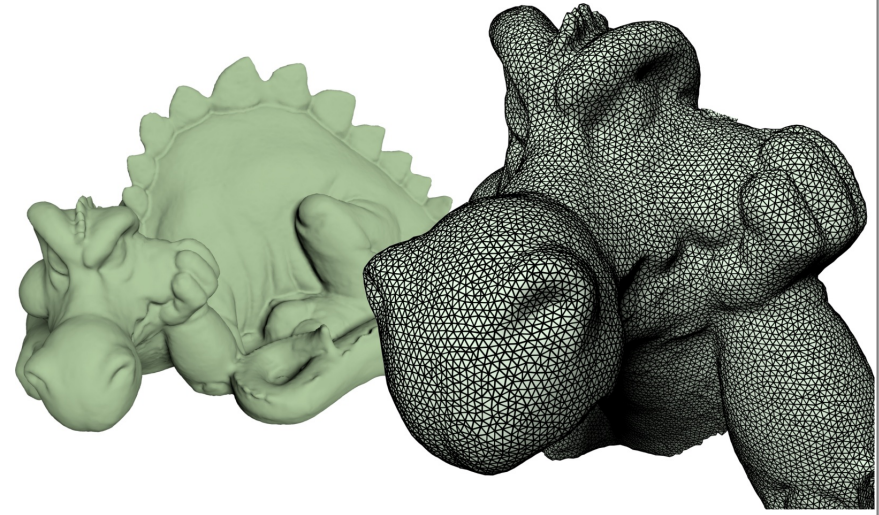
=> For projective/rasterization render pipeline : always render **triangular meshes** at the end

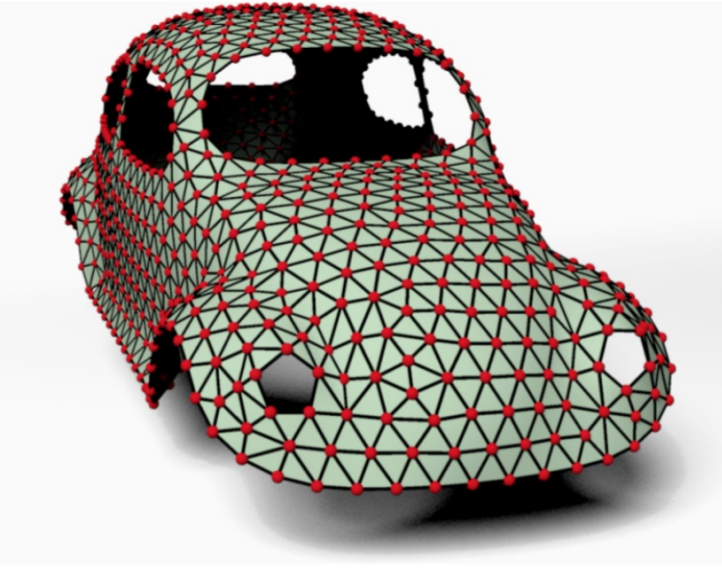
+ Simplest representation
+ Fit to GPU Graphics render pipeline
- Requires large number of samples: complex modeling
- Tangential discontinuities at edges

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

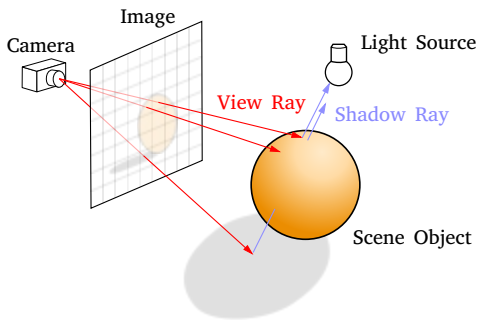
Mesh buffer encoding in C++

```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
    std::vector<index3> connectivity = { {0,1,3}, {0,3,2}, {0,2,1}, {1,2,3} };

    return 0;
}
```

How to render surface on a screen

How to render surfaces

Ray tracing

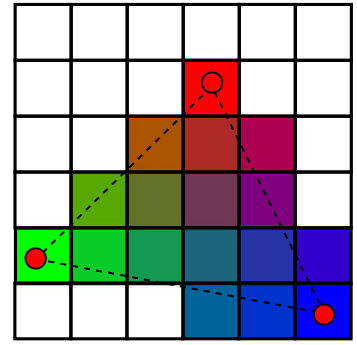


- *Throw rays from light-sources/camera*
- *Intersect rays with 3D shapes*
- *Pixel-wise computation*

+ 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*

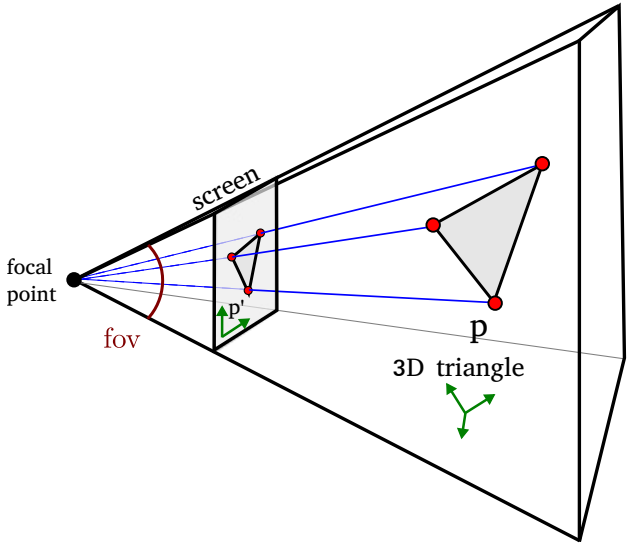
+ Efficiently implemented on GPU
- Limited to triangles
- No native effects (shadows, transparency, etc)

=> The standard real time rendering with GPU

Projection/Rasterization

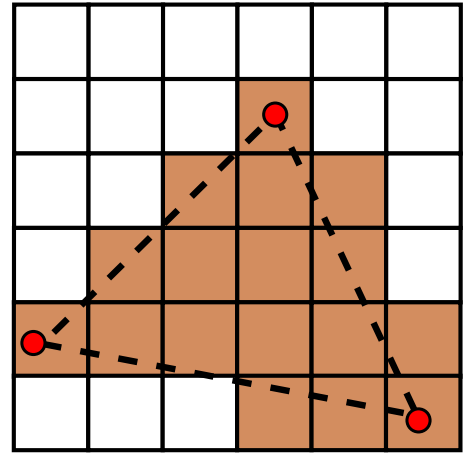
Object made of **triangles only**

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

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

Similar in **3D** but with **4-components vectors**, and 4×4 **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)*

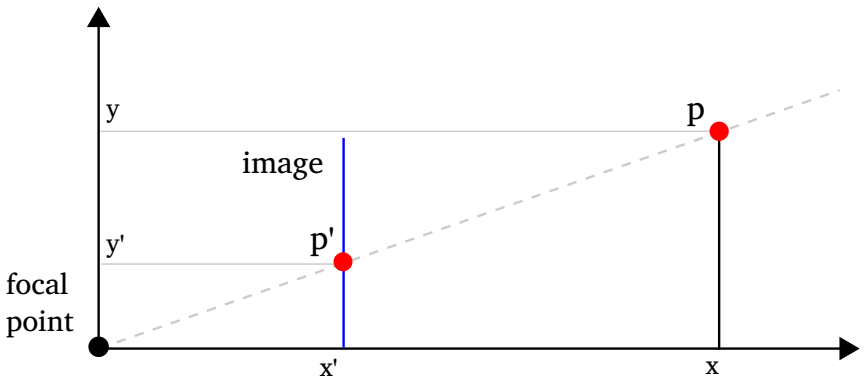
Note: vectors and points can be expressed

- 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)



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

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 $z = 0$

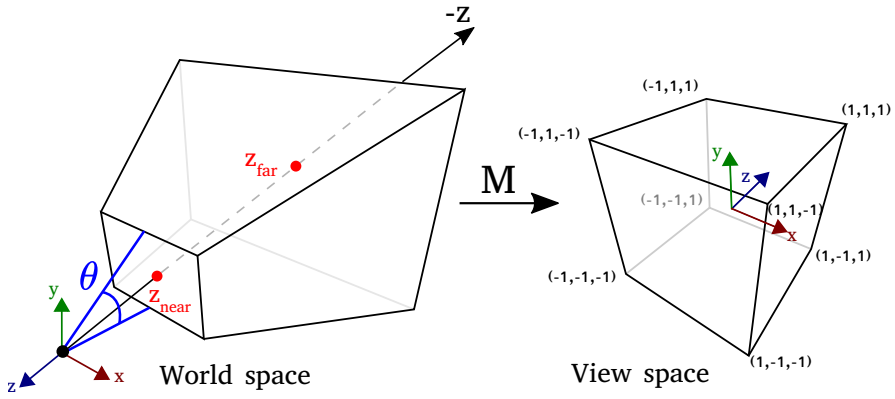
Real coordinates of points are obtained after normalization (division by z).

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



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

In practice

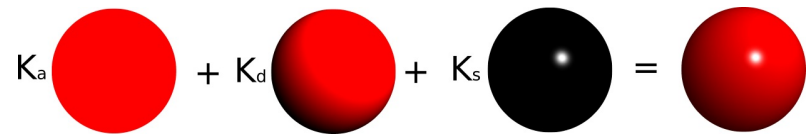
=> You must define z_{near}, z_{far}

Per-vertex normals and illumination

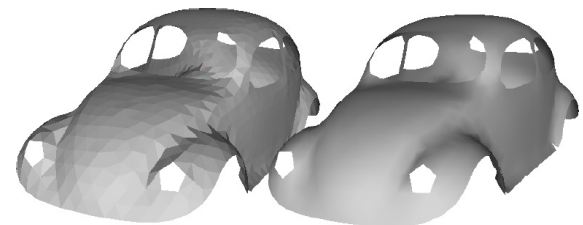
For smooth looking meshes, we define a **normal per-vertex**.

- Vertices are seen as samples on a smooth underlying surface

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

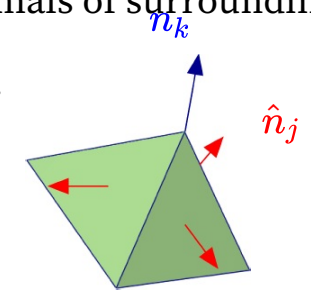


- *Phong shading* interpolates normals on each *fragments* of triangles, and compute illumination.

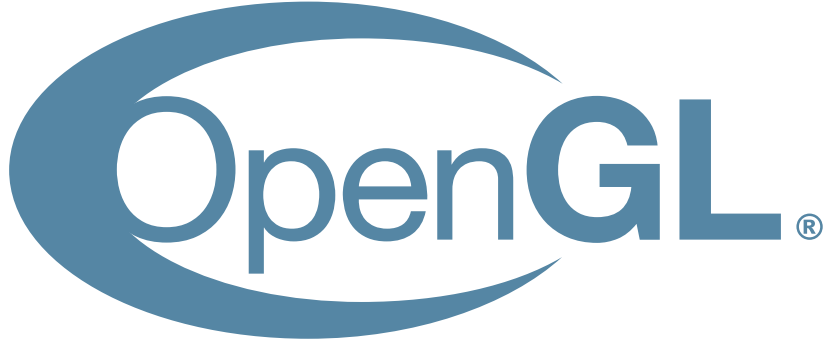


Possible automatic computation of normals: averaged normals of surrounding triangle.

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



Introduction to OpenGL



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)

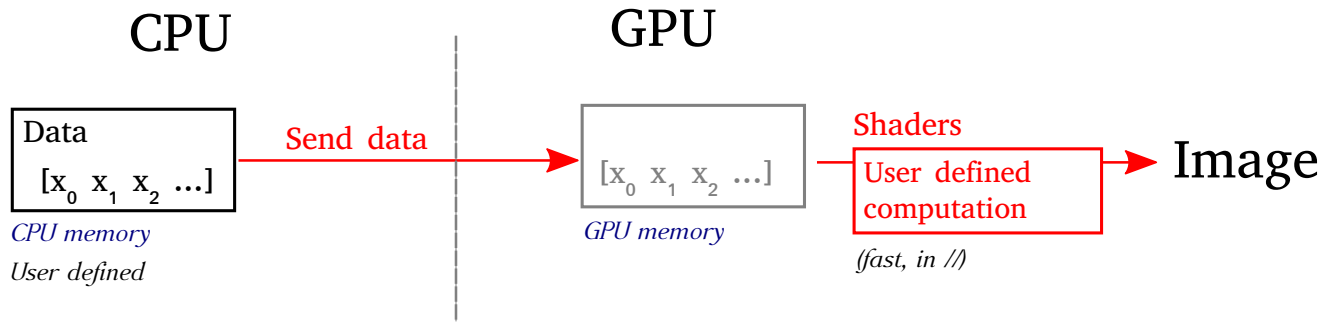
What OpenGL provides

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)

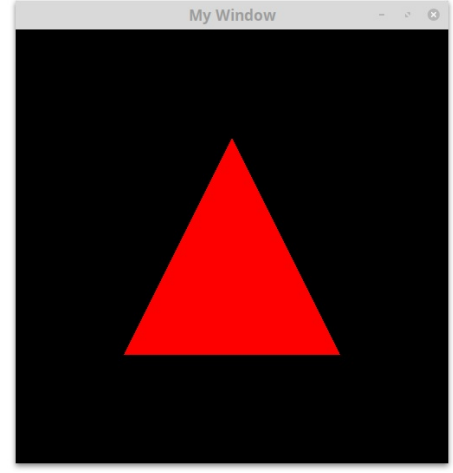
Setting up 3D scene with OpenGL

Do not rush to complex 3D code

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;
```



FYI - Vulkan ~ 1500 lines.

Steps to start Graphics Programming with OpenGL

1. Create a **window** (independant from OpenGL)
2. Start an **OpenGL context** (allows window to synchronize with OpenGL rendering)
3. **Call OpenGL functions** (within a loop for animation)

Creating a Window

First step for all Graphics program.

Not related to OpenGL, but should allow *OpenGL context*.

Creating a window from scratch is complex, depends on the OS.

Can use external libray to ease this task

- GLFW (lightweight, only window + events) < our case
- GLUT (lightweight, only window + events)
- SDL (made for video game, window + events + sounds)
- GTK (full GUI, more heavy)
- Qt (very complete, full GUI, even more heavy)

GLFW

- Lightweight library to create window compatible with OpenGL context.
- Handle events from mouse, keyboard, joystick
- Doesn't handle GUI: buttons, menus, etc.

Ex. Minimalistic window creation
(empty window without anything displayed)

```
int main()
{
    // Initialize GLFW - library handling the window
    glfwInit();
    // Create a new Window
    GLFWwindow* window = glfwCreateWindow(500, 500, "My Window", nullptr, nullptr);

    // Loop until receiving closing signal
    while( !glfwWindowShouldClose(window) ) {
        glfwSwapBuffers(window); // Double buffering (will be used to avoid flickering when animating a scen
        glfwPollEvents();        // Handle GLFW events (ex. mouse clicks, etc)
    }

    glfwTerminate(); // Close the GLFW Window
    return 0;
}
```

OpenGL functions loading

- By default, your system only loads OpenGL 1.x functions (gl.h).
- Access to more recent OpenGL function version requires specific **OpenGL loaders**
  (link low level pointers to function names)

Interest of loaders: can dynamically load the latest supported OpenGL version
(In our case, we only target OpenGL 3.3)

- Can be performed manually (tedious)
- Can use external OpenGL loaders
  - GLAD, GL3W, glLoadGen (recent, focuss on OpenGL > 3)
  - GLEW (older, dynamic loading of all OpenGL Version)

GLAD and GLFW - OpenGL functions loading

1. Create glad.h and glad.c for OpenGL 3.3 from generator
2. Compile glad.c
3. Init OpenGL 3.3 context and load functions in the code

```
#include "glad.h" // glad.h should be included before glfw or any OpenGL related include
#include <GLFW/glfw3.h>
#include <iostream>
int main()
{
    glfwInit();
    // Indicate to GLFW to setup context compatible with OpenGL 3.3 core profile
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,GLFW_OPENGL_CORE_PROFILE);
    GLFWwindow* window = glfwCreateWindow(500, 500, "My Window", nullptr, nullptr);
    // Enable the window to handle OpenGL Context
    glfwMakeContextCurrent(window);
    // Load OpenGL Functions
    gladLoadGL();
    while( !glfwWindowShouldClose(window) ) {
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    glfwTerminate();
    return 0;
```

First OpenGL call

Clear screen with a uniform color (not yet a triangle)

```
int main()
{
    // ... init
    while( !glfwWindowShouldClose(window) ) {

        // Set the (R,G,B,A) color to clear the screen
        glClearColor(1.0f, 1.0f, 0.5f, 1.0f);
        // Clear the screen (designated by the color buffer)
        glClear(GL_COLOR_BUFFER_BIT);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
glfwTerminate();
return 0;
}
```

OpenGL naming convention

- glName(...) *OpenGL function*

  ex. glClearColor(r, g, b, a), glClear(), etc.

- GLtype *OpenGL type*

  ex. GLint, GLuint, GLfloat, etc.

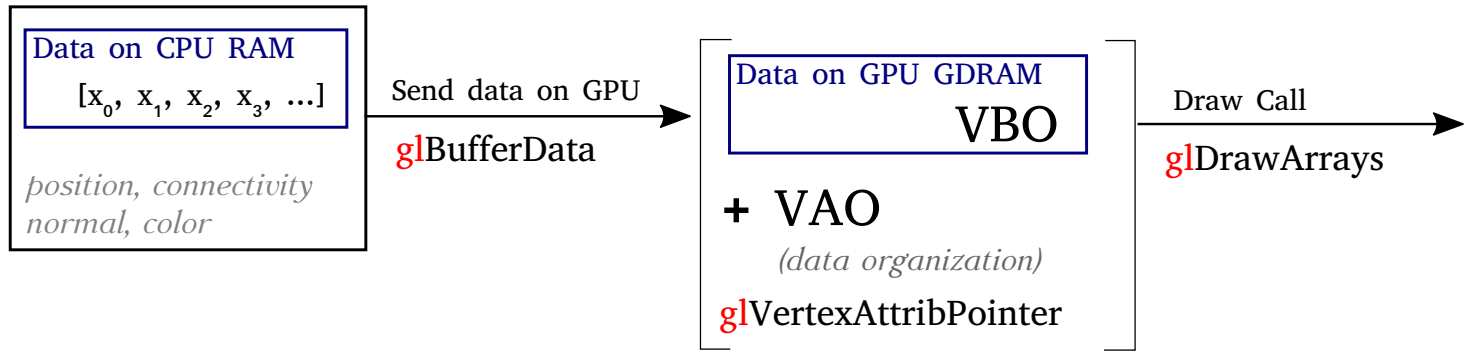
- GL_NAME *OpenGL constant enum*

  ex. GL_COLOR_BUFFER_BIT, GL_FLOAT, etc.

Drawing a triangle - OpenGL calls on CPU

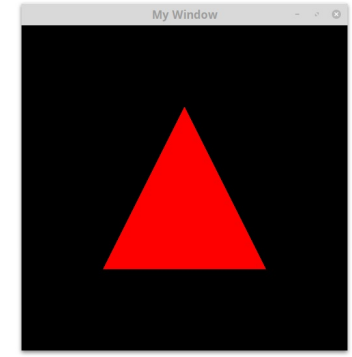
```
int main()
{
    // Init window, opengl, ...

    // CPU data
    const std::vector<vec3> position = { {-0.5f, -0.5f, 0.0f}, {0.5f, -0.5f, 0.0f}, {0.0f,  0.5f, 0.0f} };

    // Send data to VBO
    GLuint vbo = 0;
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, position.size()*sizeof(GLfloat), &position[0], GL_STATIC_DRAW );

    // Indicate data organization in VAO
    GLuint vao = 0;
    glGenVertexArrays(1,&vao);
    glBindVertexArray(vao);
    glEnableVertexAttribArray( 0 );
    glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, nullptr ); // only position data

    while( !glfwWindowShouldClose(window) ) {

        glDrawArrays(GL_TRIANGLES, 0, 3); // Draw 3 vertices

        glfwSwapBuffers(window);
        glfwPollEvents();
    }
}
```

Drawing a triangle - OpenGL calls on CPU




```cpp
int main()
{
    // Init window, opengl, ...

    // CPU data
    const std::vector<vec3> position = { {-0.5f, -0.5f, 0.0f}, {0.5f, -0.5f, 0.0f}, {0.0f,  0.5f, 0.0f} };

    // Send data to VBO
    GLuint vbo = 0;
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, position.size()*sizeof(GLfloat), &position[0], GL_STATIC_DRAW );

    // Indicate data organization in VAO
    GLuint vao = 0;
    glGenVertexArrays(1,&vao);
    glBindVertexArray(vao);
    glEnableVertexAttribArray( 0 );
    glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, nullptr );

    while( !glfwWindowShouldClose(window) ) {

        glDrawArrays(GL_TRIANGLES, 0, 3); // Draw 3 vertices

        glfwSwapBuffers(window);
        glfwPollEvents();
    }
}
```

Shaders



Vertex Shader

```
#version 330 core
layout (location = 0) in vec4 position;
void main()
{
    gl_Position = position;
}
```

Fragment shader

```
#version 330 core
out vec4 FragColor;
void main()
{
    FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```