# INFO 0939: Project 1 – Due on October 26, 2021

Updated 13/10/2021

(Intermediate deadline: October 12, 2021)

## Goals of the Project

- Refresh your C language knowledge.
- Experiment with the SLURM job scheduler.
- Write your first parallel program using OpenMP.
- Implement a simple 3D rendering pipeline on CPU.

## Statement

3D rendering is a process that converts a 3D model into a 2D image. The process can be carried on in real-time (e.g. in a video game or for interactive scientific visualization) or in batch mode (e.g. to produce photorealistic movie frames). Two techniques are commonly implemented: *rasterization*, which is the least computationally expensive, or *ray tracing*, which is much more expensive but can produce very sophisticated images, especially to represent liquids or complex optical phenomena.

While these algorithms are usually implemented on Graphics Processing Units (GPUs), in this project you will implement a simple rasterization algorithm on a (multicore) CPU, to convert 3D models consisting of a finite number of small triangles approximating the boundary of an object, into 2D images. Each triangle is defined by the coordinates of its three vertices, i.e. each triangle is fully characterized by a list of 9 single precision numbers (`float` in C). The algorithm is divided into the following three main steps:

**The camera view** The triangle coordinates are expressed in an orthonormal frame $\mathcal{W} = (O, \mathbf{e}_X, \mathbf{e}_Y, \mathbf{e}_Z)$ such that $(\mathbf{e}_X, \mathbf{e}_Y, \mathbf{e}_Z)$ is an orthonormal basis. This frame $\mathcal{W}$ is called the *world frame*. The first parameter that must be computed is the shading $s$ of each triangle. To evaluate it, the scalar product between the normal of the triangle $\vec{n}$ and the lightbeam $\vec{L}$ is computed. The normal of the triangle $(A, B, C)$ is obtained through the vector product

$$\vec{n} = (\overrightarrow{AB} \times \overrightarrow{AC}) / (\|\overrightarrow{AB} \times \overrightarrow{AC}\|) \tag{1}$$

and the lightbeam vector is assumed to be constant everywhere and equal to

$$\vec{L} : \left(-1/\sqrt{3},\ -1/\sqrt{3},\ -1/\sqrt{3}\right)$$

in the world frame. The shading parameter $s$ is defined as follows:

$$s = \begin{cases} -\vec{n} \cdot \vec{L} & \text{if } \vec{n} \cdot \vec{L} < 0 \\ 0 & \text{ortherwise} \end{cases}. \tag{2}$$

This shading will be used in the last algorithm step but must be computed before any change of coordinate is applied.

The 2D image is generated from a virtual camera located at $M : (p_{M,x}, p_{M,y}, p_{M,z})$ and looking to the origin $O$. The triangle coordinates must be changed to the so-called *view frame*, $\mathcal{V} = (M, \mathbf{e}_U, \mathbf{e}_V, \mathbf{e}_W)$, associated to the camera. These coordinate systems are depicted in Figure 1a. Let us introduce the following change of coordinates:

$$\begin{cases} \mathbf{e}_U = \dfrac{1}{D_g} \left[p_{M,z}\mathbf{e}_X - p_{M,x}\mathbf{e}_Z\right], \\ \mathbf{e}_V = \dfrac{1}{D_g D} \left[-p_{M,x}p_{M,y}\mathbf{e}_X + \left(p_{M,z}^2 + p_{M,x}^2\right)\mathbf{e}_Y - p_{M,y}p_{M,z}\mathbf{e}_Z\right], \\ \mathbf{e}_W = \dfrac{1}{D} \left[p_{M,x}\mathbf{e}_X + p_{M,y}\mathbf{e}_Y + p_{M,z}\mathbf{e}_Z\right], \end{cases} \tag{3}$$

where $D = \sqrt{p_{M,x}^2 + p_{M,y}^2 + p_{M,z}^2}$ is the camera distance and $D_g = \sqrt{p_{M,x}^2 + p_{M,z}^2}$ is the ground-plan camera distance, such that the coordinates of a point $P : (x, y, z)$ in the world frame becomes $P : (u, v, w)$ in the view frame according to

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} e_{U,x} & e_{U,y} & e_{U,z} & -\overrightarrow{OM} \cdot \mathbf{e}_U \\ e_{V,x} & e_{V,y} & e_{V,z} & -\overrightarrow{OM} \cdot \mathbf{e}_V \\ e_{W,x} & e_{W,y} & e_{W,z} & -\overrightarrow{OM} \cdot \mathbf{e}_W \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \tag{4}$$

Note that the matrix involved in the previous expression is called the *world to view matrix* (WtoV).

**The projection** Once our object is seen from the camera point of view, the next operation is the projection into a normalized cube $(x, y, z) \in [-1, 1]^3$ depending on camera parameters. These parameters are the vertical field of view angle $\theta_Y$, a position of a near and far clip plane ($n$ and $f$) such that points before the near clip plane and after the far clip plan are invisible, and an aspect ratio $r = {}^{W}/_{H}$ where $W$ and $H$ is the width and the height of the image (in pixels). Only points between the clip planes and the field of view pyramid will be displayed. This is the resulting truncated pyramid, called the *view frustum* (see Figure 1b), that is mapped into the normalized cube $\mathcal{N} = (M, \mathbf{e}_{X,S}, \mathbf{e}_{Y,S}, \mathbf{e}_d)$ (see Figure 1c) with the following projection:

$$\begin{pmatrix} x_S \\ y_S \\ d \end{pmatrix} = \frac{n - f}{2nfw} \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & \dfrac{n+f}{n-f} & -1 \end{pmatrix} \begin{pmatrix} u \\ v \\ w \\ 1 \end{pmatrix}, \tag{5}$$

where $S_y = {}^1\!/\!\tan(\theta_Y/2)$ and $S_x = {}^{S_y}\!/r$ are the y scaling and x scaling, respectively, $x_S$ and $y_S$ are the screen coordinates and $d$ is the depth coordinate. Note that this matrix is called the *view to projection matrix* (VtoP).

**The rasterization** The 2D image is created during this last step. An array of $w \times h$ pixels is used to store the image as follows:

$$
\begin{array}{|lllll|}
\hline
p[0][0] & p[0][1] & p[0][1] & \cdots & p[0][w-1] \\
p[1][0] & p[1][1] & p[1][1] & \cdots & p[1][w-1] \\
p[2][0] & p[2][1] & p[2][1] & \cdots & p[2][w-1] \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
p[h-1][0] & p[h-1][1] & p[h-1][1] & \cdots & p[h-1][w-1] \\
\hline
\end{array}
$$

where $p[0][0]$ corresponds to the upper left pixel and $p[h-1][w-1]$ is the lower right one. Each pixel is made by three 8-bit unsigned integers (`unsigned char` in C) representing the red, green and blue component ($RGB$) of the color and the depth coordinate (`float` in C) initialized to 1. At the beginning, each pixel has the color of the background.

A pixel $p[i][j]$ with $i \in \{0, 1, 2, \cdots, h-1\}$ and $j \in \{0, 1, 2, \cdots, w-1\}$ has screen coordinates $x_S = 2(j + 0.5)/w - 1$ and $y_S = -2(i + 0.5)/h + 1$. Knowing that a point $P$ is inside a triangle $(A, B, C)$ if the following inequalities are satisfied

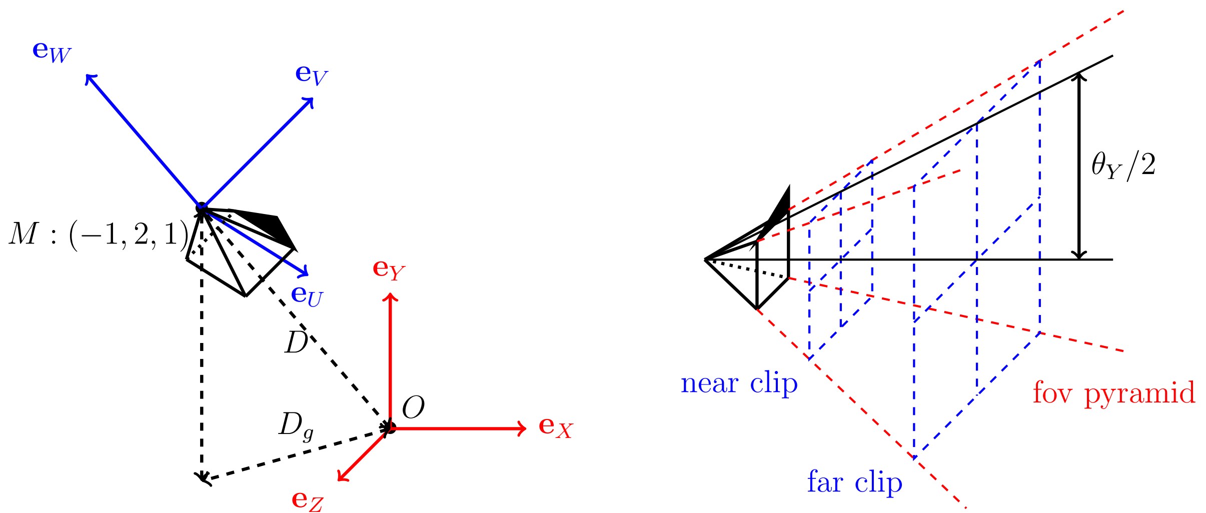
$$
\begin{cases}
\left(\overrightarrow{AB} \times \overrightarrow{AP}\right) \cdot \left(\overrightarrow{AP} \times \overrightarrow{AC}\right) \geq 0, \\
\left(\overrightarrow{BC} \times \overrightarrow{BP}\right) \cdot \left(\overrightarrow{BP} \times \overrightarrow{BA}\right) \geq 0,
\end{cases}
\tag{6}
$$

it is possible to determine for every triangle $t$ if $p[i][j]$ is inside or not (see Figure 1d). If it is not, then the pixel $p[i][j]$ is not influenced by the triangle $t$. If it is inside, then the color of the pixel $p[i][j]$ is set to the input object color $C_o$ multiplied by the shading $s$ if and only if the depth coordinate $d$ of the triangle $t$ evaluated at the pixel coordinates is lower than the depth of the triangle that previously updated this pixel (a pixel that has not been updated yet has a depth set to 1). In this way, only points that are closer to the camera will be displayed. Note that the depth at any point $P : (x_P, y_P)$ inside a triangle $(A, B, C)$ is given by

$$
d_P = w_A d_A + w_B d_B + w_C d_C, \tag{7}
$$

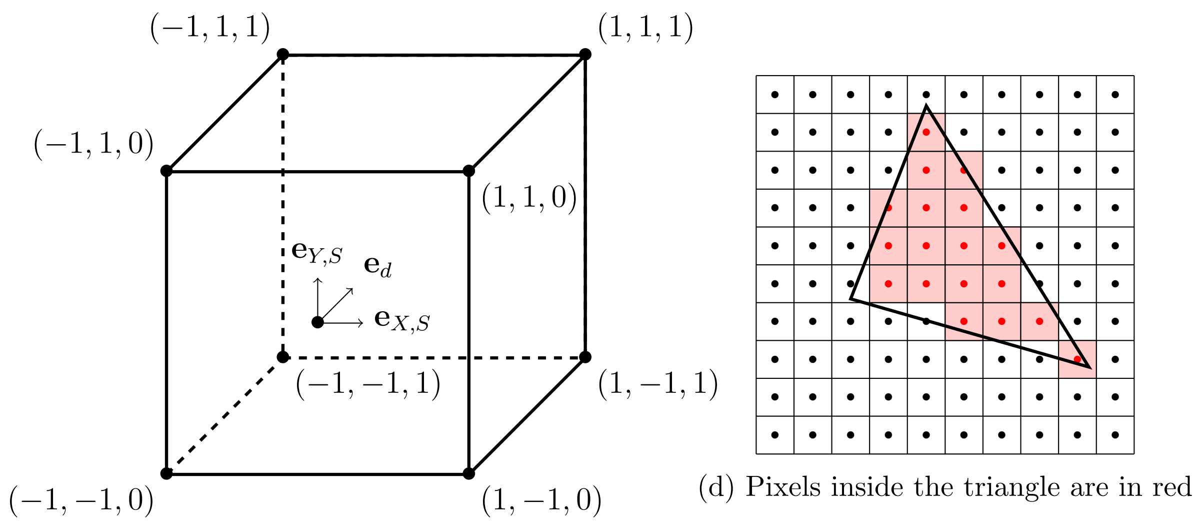
where $d_A$, $d_B$ and $d_C$ are the depths at vertices $A$, $B$ and $C$, respectively, and $w_A$, $w_B$ and $w_C$ are weights given by

$$
\begin{cases}
\color{red}{w_C} = \dfrac{(x_B - x_A)(y_P - y_A) - (y_B - y_A)(x_P - x_A)}{(x_B - x_A)(y_C - y_A) - (y_B - y_A)(x_C - x_A)}, \\
w_B = \dfrac{(x_A - x_C)(y_P - y_C) - (y_A - y_C)(x_P - x_C)}{(x_A - x_C)(y_B - y_C) - (y_A - y_C)(x_B - x_C)}, \\
\color{red}{w_A} = 1 - w_B - \color{red}{w_C}.
\end{cases}
\tag{8}
$$




(a) The world frame with a camera located at point $M$ and the associated view frame.

(b) The field of view (fov) pyramid and the near and far clips that mark out the view frustum.




(c) The normalized coordinate system

(d) Pixels inside the triangle are in red

Figure 1: Each step of the 3D rendering algorithm

### The algorithm

To implement a 3D rendering algorithm, please follow these different steps:

- take from the command line a 3D model (composed of $T$ triangles) and a scene parameter file (see sections hereafter for format specifications);
- store the triangle vertex positions in an array;

- allocate an array of $T$ `float` floating point numbers to store the $T$ shading parameters and apply to each triangle:
  - the formula (1) to compute the normal of the current triangle;
  - the change of coordinate (4)-(5) to the three vertices of the current triangle;
  - the formula (2) to compute the shading $s$ of the current triangle;
- allocate an array of $w \times h$ pixels to store the R, G, B components of the color and the depth coordinate (the color must be initialized to the background color $C_{bg}$ and the depth coordinate must be initialized to 1);
- loop over each triangle and:
  - check for each pixel if it is inside the current triangle (optimization hint: do we need to loop over all pixels for each triangle?);
  - if it is inside, compute the depth at the pixel position with (8);
  - if the depth is smaller than the pixel depth, set the pixel color to $sC_o$ and update the pixel depth;
- save the rendered image in `.ppm` format as described hereafter.

**The model format**

A model is stored in a `.dat` **binary** file that contains a list of triangles represented by the coordinates of their three vertices. At the beginning of the file, the number of triangles $T$ is stored as a four-byte unsigned integer (`unsigned int` in C). Then the following of the file is filled with the $9T$ coordinated of the vertices stored as four-byte floating point numbers (`float` in C). For instance, let us consider any triangle $t$ with $t = \{0, 1, \cdots, T-1\}$ and its vertices $A : (p^t_{A,x}, p^t_{A,y}, p^t_{A,z})$, $B : (p^t_{B,x}, p^t_{B,y}, p^t_{B,z})$ and $C : (p^t_{C,x}, p^t_{C,y}, p^t_{C,z})$. Then a model made by $T$ triangles is stored in a binary `.dat` file structured as follows:

| $T$ | $p^0_{A,x}$ | $p^0_{A,y}$ | $p^0_{A,z}$ | $p^0_{B,x}$ | $p^0_{B,y}$ | $p^0_{B,z}$ | $p^0_{C,x}$ | $p^0_{C,y}$ | $p^0_{C,z}$ | $\cdots$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $p^t_{A,x}$ | $p^t_{A,y}$ | $p^t_{A,z}$ | $p^t_{B,x}$ | $p^t_{B,y}$ | $p^t_{B,z}$ | $p^t_{C,x}$ | $p^t_{C,y}$ | $p^1_{C,z}$ | $\cdots$ |
| $\cdots$ | $p^{T-1}_{A,x}$ | $p^{T-1}_{A,y}$ | $p^{T-1}_{A,z}$ | $p^{T-1}_{B,x}$ | $p^{T-1}_{B,y}$ | $p^{T-1}_{B,z}$ | $p^{T-1}_{C,x}$ | $p^{T-1}_{C,y}$ | $p^{T-1}_{C,z}$ | |

### The scene parameter format

The scene parameters are stored int a `.txt` **ASCII** file following this format:

| | | |
|---|---|---|
| $C_{o,R}$ | $C_{o,G}$ | $C_{o,B}$ |
| $C_{bg,R}$ | $C_{bg,G}$ | $C_{bg,B}$ |
| $p_{M,x}$ | $p_{M,y}$ | $p_{M,z}$ |
| $h$ | $w$ | |
| $\theta_Y$ | | |
| $n$ | $f$ | |

where:

- $C_{o,R}$, $C_{o,G}$ and $C_{o,B}$ are the red, green and blue components of the object color $C_o$, stored as `unsigned char` integers;
- $C_{bg,R}$, $C_{bg,G}$ and $C_{bg,B}$ are the red, green and blue components of the background color $C_{bg}$, stored as `unsigned char` integers;
- $p_{M,x}$, $p_{M,y}$ and $p_{M,z}$ are the coordinates of the camera, stored as `float` floating point numbers;
- $h$ and $w$ are the height and width of the 2D image (in pixels), stored as `unsigned int` integers;
- $\theta_Y$ is the vertical field of view angle, stored as a `float` floating point number;
- $n$ and $f$ are the near and far clip point positions, stored as `float` floating point numbers.

### The PPM format

The image format used in this project is the **binary** PPM format encoded as follows (see `https://en.wikipedia.org/wiki/Netpbm#File_formats`):

1. The characters "P6" (a "magic number" for identifying the PPM file type)
2. Whitespace (blanks, TABs, CRs, LFs)
3. A width, formatted as ASCII characters in decimal
4. Whitespace
5. A height, again in ASCII decimal
6. Whitespace

7. The maximum color value (Maxval), again in ASCII decimal. Must be less than 65536 and more than zero. In this project this value is set to 255 (corresponding to a `unsigned char` type in C).

8. A single whitespace character (usually a newline).

9. A raster of Height rows, in order from top to bottom. Each row consists of Width pixels, in order from left to right. Each pixel is a triplet of red, green, and blue samples, in that order. Each sample is represented in **pure binary** by either 1 or 2 bytes. If the Maxval is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first. A row of an image is horizontal. A column is vertical. The pixels in the image are square and contiguous.

Lines starting with "#" are comments.

## Instructions

### Intermediate deadline

By group of **two students** (this is mandatory) you are asked to implement a C program that

1. takes from the command line a 3D model and a scene parameter file. A call to your program must thus look like `./graphics bun.dat param_bun.txt`;

2. implements the given 3D rasterization algorithm;

3. generates (and saves) a 2D image of the given model in PPM format;

4. is written in a single C file `main.c` (no header and no other source files).

For this deadline, the program should not be multithreaded. Only a working sequential version is needed.

### Final deadline

With the code written for the intermediate deadline, implement a multithreaded version using OpenMP. This program should run on the CECI clusters: to this end, you should write a SLURM script that reserves a given number of cores on the supercomputer, with appropriate memory and job duration.

Write a report of maximum 5 pages where you:

1. describe your implementation;

2. study the effect of the loop scheduling on your parallel implementation;
3. analyze the efficiency (strong scaling) of your parallel implementation;
4. present some 2D images obtained with your code.

Submit your C code on the Montefiore submission platform `https://submit.montefiore.ulg.ac.be/`. Note that no errors have to be reported during the automatic tests performed on this platform. **If your last submission generates error(s), a default grade of 0/20 will be attributed.**

When your code passes on the submission platform, send your report in PDF format together with your C code and SLURM submission script to `anthony.royer@uliege.be` and `matteo.cicuttin@uliege.be`. The files (report, C code, SLURM script) should be named

```
project1_Lastname1_Lastname2.pdf
project1_Lastname1_Lastname2.c
project1_Lastname1_Lastname2.sh
```

## Remarks

The number of arguments and their values are passed as arguments (`argc` and `argv`) to the `main` routine: `int main(int argc, char **argv)`. For `./graphics bun.dat param_bun.txt`; `argc` will be 3, `argv[0]` will be `"./graphics"`, `argv[1]` will be `"bun.dat"` and `argv[2]` will be `"param_bun.txt"`.

If your operating system cannot natively display PPM images, you could use the following commands in Matlab: `img=imread('file.ppm'); image(img);`.

3D models and parameters files can be downloaded on NIC5 at:
`/home/ulg/ace/aroyer/info0939/hw1`.

Make your code easily readable by other people and by "future you". This implies to:

- correctly indent your code;
- make things as simple as possible... while still being efficient;
- use pertinent, meaningful variable and function names, even if it makes them longer;
- make functions when necessary.

As usual, don't write the code all at once without testing. Write it part by part and test every small functionality separately.