Project on performance oriented programming

J. Daniel Garcia (coordinator)
Computer Architecture
Computer Science and Engineering Department
University Carlos III of Madrid

2025

1 Objective

The fundamental goal of this project is that students get familiar with **sequential programs optimizations**.

Specifically, the project will focus in the development of sequential software using the C++ programming language (including improvements up to C++23).

2 Overview

In this project, a 3D image rendering application will be developed. The software will take a file that describes a 3D scene and use a set of configuration parameters to generate an image with a two-dimensional representation.

The developed software will use two configuration files in text format:

- A rendering engine configuration file.
- A scene description file.

The output image format will be PPM (type P3), which represents an image as a sequence of pixels in RGB encoding.

Figure 1 presents an image generated with many spheres and cylinders.

2.1 Scene description

A scene is composed of a collection of objects in three-dimensional space. Additionally, each of these objects has an associated material that defines the properties that an object has when rays hit it.

2.1.1 Objects in three-dimensional space

The image rendering application generates a 2D image from the representation of a scene with 3D objects. The objects that must be supported are cylinders and spheres. Each of these objects will have an associated material with certain properties.

A **sphere** is given by the following properties:

- The position of its **center** $C \equiv (c_x, c_y, c_z)$.
- The length of its **radius** r.

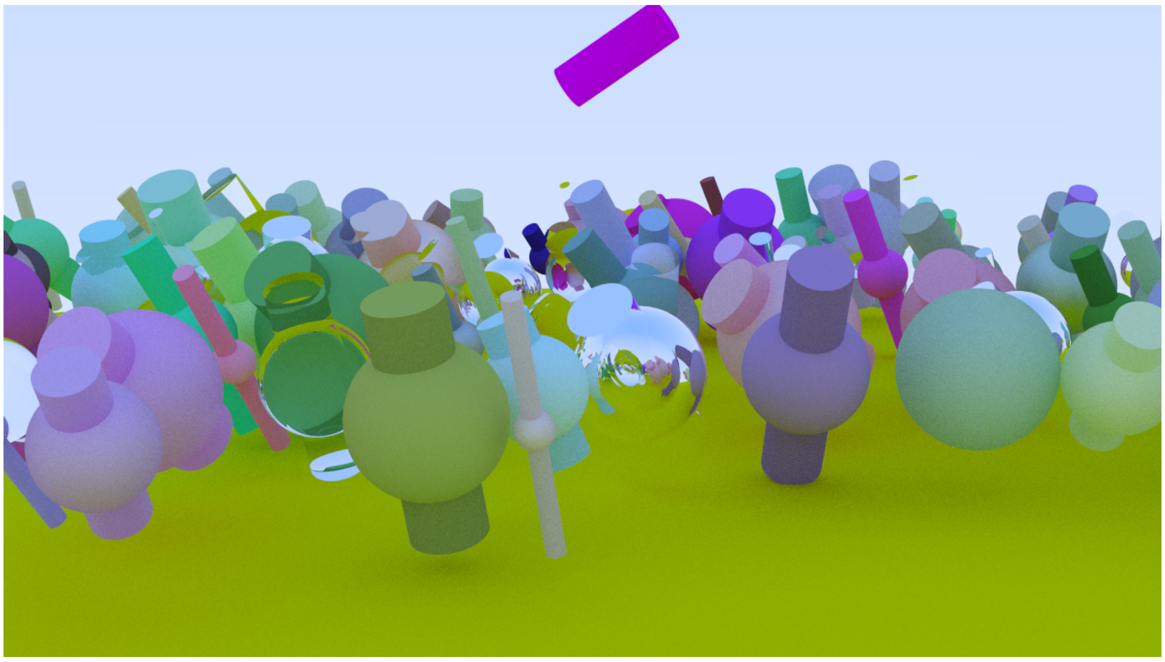

Figure 1: Image generated by the application with multiple spheres and cylinders.

A **cylinder** is given by the following properties:

- The position of its **center** $C \equiv (c_x, c_y, c_z)$. This position corresponds to the midpoint between the centers of its upper and lower bases.
- The **radius** r of the cylinder.
- The vector that defines the **axis** of the cylinder $\vec{a} = (a_x, a_y, a_z)$.

  Note that the height of the cylinder h can be obtained as the magnitude of that vector.

  $$h = \|\vec{a}\|$$

2.1.2 Object materials

Each object has an associated material that defines its properties when performing image synthesis. The application considers three types of materials: **matte**, **metal**, and **refractive**.

A **matte** material has an associated **reflectance** which is the ratio between the amount of light reflected by an object with respect to the received light. The reflectance is represented as a three-dimensional vector with the values for the RGB channels. Each of the values is in the interval $[0, 1]$.

A **metallic** material also has an associated **reflectance**. Additionally, it has an associated light **diffusion** factor which is a real value.

Finally, a **refractive** material has an associated **refraction index**, which indicates the ratio between the sines of the incidence angle and the refraction angle.

2.1.3 Scene description file

A scene description file contains a set of text lines. Each text line can have either a material specification or an object specification.

Material specifications A material specification line contains the following fields separated by whitespace:

- **Material type**: Can be one of the following labels: **matte:**, **metal:** or **refractive:**. This label indicates the type of material being defined.
- **Material name**: Is a single string without whitespace that identifies the defined material (e.g. **mat1**, **r512** or **new**).
- **Additional parameters**: These parameters depend on the selected material type:
  - **Matte**: Three numerical values that specify the material reflectance.
  - **Metal**: Four numerical values. The first three specify the object reflectance and the fourth the diffusion factor.
  - **Refractive**: A single numerical value that indicates the refraction index.

Object specifications An object specification line contains the following fields separated by whitespace:

- **Object type**: Can be one of the following labels: **sphere:** or **cylinder:**. This label indicates the type of object being defined.
- **Object parameters**: Are parameters that depend on the object type
  - **Sphere**: Three numerical values that indicate the coordinates of the sphere center and another numerical value for the radius.

    If the value for the radius is less than or equal to zero, it is considered that the value is invalid.
  - **Cylinder**: Three numerical values that indicate the coordinates of the cylinder center, another numerical value for the radius and three other numerical values for the vector that defines the cylinder axis.

    If the value for the radius is less than or equal to zero, it is considered that the value is invalid.
- **Material name**: Name of the material the object is made of. It must be a material previously defined in the file.

As an example, listing 1 presents the content of a configuration file for a scene.

Listing 1: Configuration file for a simple scene

```
matte: mat1 0 0.8 0.8
metal: metal1 0 0.8 0 2.0
refractive: ref99 1.3
sphere: 0 0 0 0.65 mat1
cylinder: 0 0 0 0.5 20 10 -5 metal1
```

This scene configuration file contains three materials whose names are **mat1**, **metal1** and **ref99**. It also includes two figures: a sphere and a cylinder.

The material **mat1** is a **matte** type material with a reflectance of $(r = 0, g = 0.8, b = 0.8)$. The material **metal1** is a **metal** type material with reflectance of $(r = 0, g = 0.8, b = 0)$ and diffusion factor of 2.0. Finally, the material **ref99** is a **refractive** type material with a refraction index of 1.3.

The sphere included in line 4 has its center at coordinates $C \equiv (c_x = 0, c_y = 0, c_z = 0)$ and a radius $r = 0.65$.

Similarly, the cylinder included in line 5 has its center at coordinates $C \equiv (c_x = 0, c_y = 0, c_z = 0)$, a radius of $r = 0.5$ and as axis the vector $\vec{a}(v_x = 20, v_y = 10, v_z = -5)$. Consequently, its height is $h = 22.91$.

2.1.4 Error handling

When loading a scene file, possible errors in it must be taken into account. If an error is found, the error will be reported and execution will terminate.

Unrecognized labels If an invalid entity name is found, an error message like the following will be printed:

```
Error: Unknown scene entity: triangle
```

Line structure The file structure is organized in lines. Thus, a line must contain all the information for a material or an object.

If a line is empty or composed exclusively of whitespace and tab characters, that line is ignored.

Lines with insufficient information If a line does not contain all the necessary information for the corresponding entity type, an error message like the following will be issued:

```
Error: Invalid matte material parameters
Line: "matte: mat1 0 0.8 "
```

Lines with excessive information If a line contains more information than necessary for the corresponding entity type, an error message like the following will be issued:

```
Error: Extra data after configuration value for key: [sphere:]
Extra: "3"
Line: "sphere: 0 0 0 0.65 mat1 3"
```

Lines with invalid information If a line contains information that is not of the expected type or has a value outside the admissible range for the corresponding entity type, an error message like the following will be issued:

```
Error: Invalid sphere parameters
Line: "sphere: 0 0 0 a mat1 3"
```

Repeated material names If two lines appear defining a material with the same name, an error message like the following will be issued:

```
Error: Material with name [mat1] already exists
Line: "matte: mat1 0 0.8 0.8"
```

References to undefined materials If an object references a material that has not been previously defined, an error message like the following will be issued:

```
Error: Material not found: [metal12]
Line: "cylinder: 0 0 0 0.5 20 10 -5 metal12"
```

2.2 Image generation parameters

The application has a series of configurable parameters, each with a default value.

Each parameter is described below, including its default value, if specified in the corresponding configuration file.

- **Aspect ratio**: Is the ratio between the width and height of an image. It is specified by two positive integer values. The first is the width and the second is the height. Both values must be positive integers.

  If not specified, its default value is the pair $(16, 9)$.

  If either of the two values is less than or equal to zero, it is considered to have an invalid value.

  Its label in a configuration file is **aspect_ratio:**.

- **Image width**: Is the width in pixels of the image to be generated.

  If not specified, its default value is 1920.

  If the value is less than or equal to zero, it is considered to have an invalid value.

  Its label in a configuration file is **image_width:**.

  Note that the **image height** can be computed by dividing the width by the aspect ratio.

- **Gamma parameter**: Is the value of the parameter for gamma correction to be applied to the resulting image.

  If not specified, its default value is 2.2

  Its label in a configuration file is **gamma:**.

- **Viewpoint parameters**: Are specific parameters of the viewpoint from which the scene is observed.

  - **Position** of the viewpoint: Are the coordinates of the viewpoint.
    If not specified, its default value is $(0, 0, -10)$.
    Its label in a configuration file is **camera_position:**.
  - **Target** of the vision: Are the coordinates of the target point of the vision. That is, the point being looked at.
    If not specified, its default value is $(0, 0, 0)$.
    Its label in a configuration file is **camera_target:**.
  - **North direction** of the viewpoint: Is the direction vector that points toward the north of the viewpoint.
    If not specified, its default value is $(0, 1, 0)$.
    Its label in a configuration file is **camera_north:**.
  - **Field of view** angle: Is the angle that determines the field of view, expressed in degrees.
    If not specified, its default value is 90.
    If the value is less than or equal to zero or greater than or equal to 180, it is considered to have an invalid value.
    Its label in a configuration file is **field_of_view:**.

- **Samples per pixel**: Is the number of rays projected in the vicinity of each pixel.

  If not specified, its default value is 20.

  If the value is less than or equal to zero, it is considered to have an invalid value.

  Its label in a configuration file is **samples_per_pixel:**.

  The number of samples per pixel impacts image quality, since with a higher number of samples per pixel, an image with smoother edges is obtained.

- **Maximum depth**: Is the number of bounces of each ray on the scene.

  If not specified, its default value is 5.

  If the value is less than or equal to zero, it is considered to have an invalid value.

  Its label in a configuration file is **max_depth:**.

  A higher depth level results in a more realistic image.

- **Material seed**: Is a seed for the random number generator used for materials. Note that all materials share a single random number generator, which they use in reflection computations.

  If not specified, its default value is 13.

  If the value is less than or equal to zero, it is considered to have an invalid value.

  Its label in a configuration file is **material_rng_seed:**.

- **Ray seed**: Is a seed for the random number generator used for ray generation.

  If not specified, its default value is 19.

  If the value is less than or equal to zero, it is considered to have an invalid value.

  Its label in a configuration file is **ray_rng_seed:**.

- **Dark background color**: Is the darkest value for the background. The background color is generated as a gradient through the vertical coordinate between the dark background color and the light background color.

  If not specified, its default value is $(0.25, 0.5, 1)$.

  If the value of any of the three components is less than 0 or greater than 1, it is considered to have an invalid value.

  Its label in a configuration file is **background_dark_color:**.

- **Light background color**: Is the lightest value for the background. The background color is generated as a gradient through the vertical coordinate between the dark background color and the light background color.

  If not specified, its default value is $(1, 1, 1)$.

  If the value of any of the three components is less than 0 or greater than 1, it is considered to have an invalid value.

  Its label in a configuration file is **background_light_color:**.

Listing 2 shows an example configuration file.

Listing 2: Example configuration file config.txt

```
image_width: 1200
gamma: 2.2

camera_position: 13 2 3
camera_target: 0 0 0
camera_north: 0 1 0
field_of_view: 20

samples_per_pixel: 10
max_depth: 5

material_rng_seed: 45
ray_rng_seed: 133

background_dark_color: .25 .5 1
background_light_color: 1 1 1
```

2.2.1 Error handling

If an error is detected when reading a configuration file, an error message should be printed and the program execution should be terminated.

Unrecognized labels All labels consist of a single word ending with the colon character. If an unrecognized label is found, an error message indicating the situation will be written.

For example, if a line beginning with the label **image_xwidth:** is found, the following error message will be presented:

```
Error: Unknown configuration key: [image_xwidth:]
```

Similarly, if a line beginning with a valid label is found, but that does not end with the colon character (like **width**), the following error message will be presented:

```
Error: Unknown configuration key: [width:]
```

Line structure Each configuration parameter must appear exactly on one line that will always be formed by the parameter name, followed by its value. There can be whitespace and tab characters before the label name, between the different parameters, and at the end of the line. However, all information for a parameter must appear on one line.

If there is an error processing a line in the file with the information for a parameter, an error message like the following will be presented:

```
Error: Invalid value for key: [aspect_ratio:]
Line: "aspect_ratio: nulo"
```

Repeated parameters If a configuration file contains several lines with the same parameter label, the information provided in the last occurrence will prevail.

In the following configuration file example, the value **2.5** will be taken for the **gamma** parameter:

```
gamma: 1.8
image_width: 900
  gamma: 2.5
```

Blank lines A configuration file can have blank lines, which will be ignored.

Line with unexpected information If a line in the configuration file contains unnecessary information, an error message will be issued.

For example, if a configuration file contains a line like the following:

```
gamma: 2.1 2.2 99
```

The following error message will be issued:

```
Error: Extra data after configuration value for key: [gamma:]
Extra: "2.2 99"
```

Line with insufficient information If a line does not contain all the necessary information for a parameter, an error message will be issued indicating that the value is not valid.

For example, if a configuration file contains a line like the following:

```
camera_position: 500 500
```

The following error message will be issued:

```
Error: Invalid value for key: [camera_position:]
Line: "camera_position: 500 500"
```

2.3 Image representation

The program output is a file that represents a 2D image. However, a distinction must be made between the in-memory representation and the file representation. In the in-memory representation, two different alternatives will be implemented (structure of arrays versus arrays of structures). However, the output file will be in PPM format, variant P3.

2.3.1 In-memory image representation

An image in memory can be represented in two ways as a **structure of arrays** or as an **array of structures**. Regardless of the chosen in-memory representation, an image has a size, which is defined by a width (the number of columns) and a height (the number of rows). By convention, the pixel at row 0 and column 0 corresponds to the upper left corner of the image.

Each pixel in the image is represented by a color given by three values for the red (R), green (G), and blue (B) components. Each of these values is an integer in the range 0 to 255.

In the **structure of arrays** type in-memory representation, the image is represented by three independent arrays or vectors (one for each of the three R, G, and B channels). Each of these three vectors contains the channel values for all pixels in the image.

In the **array of structures** type in-memory representation, the image is represented by a single array or vector. This array contains at each position the R, G, and B values for the corresponding pixel.

2.3.2 PPM format image representation

When an image is generated in a file, the PPM format will always be used, in its P3 variant. In this variant, the file is a text file composed of a header and a body.

The file header is composed of three lines:

- The first line contains the string **P3** followed by a newline.

- The second line contains two numerical values separated by a space. These values correspond to the width (number of columns) and the height (number of rows) of the image The line ends with a newline.
- The third line contains the value **255** followed by a newline.

The rest of the file is a succession of lines, each line corresponds to a pixel in the image and is composed of three numerical values separated by spaces. The pixels appear in the file by traversing the image by rows. First all pixels of the first row appear, then those of the second row, and so on.

In listing 3 a PPM file is presented for an image of 2 rows by 3 columns. The color corresponding to row 1, column 0, is the white color $(255, 255, 255)$. In the same way, the color corresponding to row 2, column 1, is the blue color $(0, 0, 255)$.

Listing 3: PPM file example

```
P3
2 3
255
0 0 0
128 128 128
255 255 255
255 0 0
0 255 0
0 0 255
```

2.3.3 Example image

As an example of the expected result, if you process the scene presented in listing 1 with the configuration presented in listing 2, you get an image like the one in figure 2.

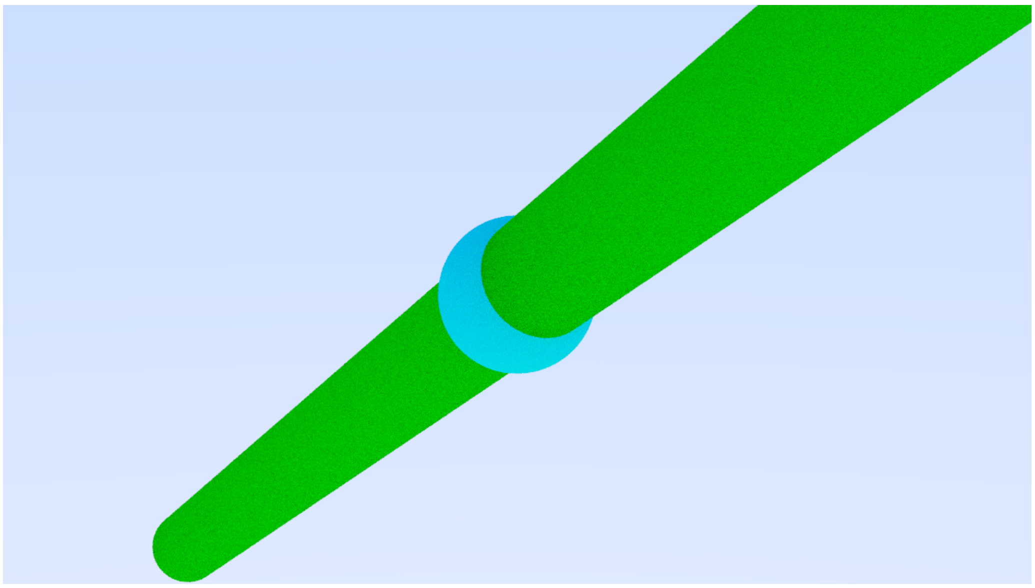

Figure 2: Image generated with a sphere and a cylinder.

3 Generating an image

Image generation has three stages:

- Generation of the **viewpoint**. This is the location where the camera is placed, pointing to the scene and defining the projection window.
- **Ray tracing** for each of the pixels in the image on the projection window.
- Determination of the **color contribution** of each traced ray.

3.1 Image viewpoint

To generate an image, besides the scene description, it is necessary to define a viewpoint. The viewpoint is given by three parameters: the viewpoint geometry, the size of the resulting image, and a random number seed.

Viewpoint geometry The **viewpoint geometry** is given by the following parameters:

- The **position** of the viewpoint $P \equiv (p_x, p_y, p_z)$. If not specified, its default value is position $(0, 0, -10)$.
- The **target** point of the vision $D \equiv (c_x, c_y, c_z)$. That is, the point being looked at from the viewpoint position. If not specified, its default value is position $(0, 0, 0)$.
- The direction indicating the **north** of the vision, given by the vector $\vec{n} = (n_x, n_y, n_z)$. Note that with a **position** and a **target**, one could look in different directions. Therefore, this vector defines the north direction of the image that is generated. If not specified, its default value is $(0, 1, 0)$.
- The **field of view** that indicates the lens aperture with which the image is observed. This angle is expressed in degrees. If not specified, its default value is $90°$.

Size of the image to be generated The **image size** indicates the width and height measured in pixels of the resulting image. These values are important for determining the projection window, which is a rectangle in three-dimensional space onto which objects are projected.

Random number generation The viewpoint also receives an integer parameter that is the **seed of a random number generator** that is used during the image synthesis algorithm. In this application, a 64-bit *Mersenne-Twister* generator will be used for this process. In C++, the type **std::mt19937_64** can be used (see `https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine.html`).

Projection window computation From these parameters, the value of the projection window can be derived. The projection window is a rectangle in three-dimensional space that has the same size as the finally generated 2D image.

This window, in addition to its size, is defined by an **origin** point (coordinates of the upper left corner) and the **horizontal** and **vertical** direction vectors. These vectors determine the orientation plane of the window.

To determine the projection window, the following steps are followed:

1. Determination of the **focal vector**. This is the vector ($\vec{v}_f$) that goes from the vision **target** point to the viewpoint **position**.

$$\vec{v}_f = P - D$$

2. Determination of the **focal distance**. This is computed as the magnitude of the **focal vector**.

$$d_f = \|\vec{v}_f\|$$

3. Determination of the **projection window height**. This height (h_p) is computed using the following expression:

$$h_p = 2 \cdot tan(\frac{\alpha}{2}) \cdot d_f$$

   Where α is the **field of view** angle (converted to radians) and d_f is the **focal distance**.

4. Determination of the **projection window width** (w_p). Since the window must have the same height-to-width ratio as the final image, the width w_p is determined by multiplying the width by the **aspect ratio** (ratio between the width and height of the image in pixels).

$$w_p = h_p \frac{w}{h}$$

5. Determination of the **director vectors** of the projection window ($\vec{u}$ and $\vec{v}$).

$$\hat{v}_f = \frac{\vec{v}_f}{\|\vec{v}_f\|} \qquad \vec{u} = \frac{\vec{n} \times \hat{v}_f}{\|\vec{n} \times \hat{v}_f\|} \qquad \vec{v} = \hat{v}_f \times \vec{u}$$

   where $\vec{n}$ is the vector that marks the **north** direction.

6. Determination of the **horizontal** ($\vec{p_h}$) and **vertical** ($\vec{p_v}$) vectors. These vectors are obtained by multiplying the unit vectors $\vec{u}$ and $\vec{v}$ by the width w_p and height h_p of the projection window respectively. For the vertical vector, it must also be taken into account that in an image the coordinate of the upper left corner has the value 0 and its values increase downward, which is corrected by changing the sign of the corresponding vector.

$$\vec{p_h} = w_p \cdot \vec{u} \qquad \vec{p_v} = h_p \cdot (-\vec{v})$$

7. Determination of the **projection window origin**. The origin O starts from the viewpoint **position** and is displaced by subtracting the **focal vector** (thus reaching) the **target point** position. Since that point is at the center of the **projection window**, a negative offset of half of the vectors $\vec{p}_h$ and $\vec{p}_v$ is performed to reach the upper left corner of the projection window. From here, to reach the center of the upper left pixel, half of the offset vectors $\vec{\Delta x}$ and $\vec{\Delta y}$ are added. The latter are obtained by dividing the vectors $\vec{p}_h$ and $\vec{p}_v$ by the image dimensions in pixels.

$$\vec{\Delta x} = \frac{\vec{p}_h}{w} \qquad \vec{\Delta y} = \frac{\vec{p}_v}{h} \qquad O = P - \vec{v_f} - \frac{1}{2} \cdot (\vec{p}_h + \vec{p}_v) + \frac{1}{2} \cdot (\vec{\Delta x} + \vec{\Delta y})$$

3.2 Ray tracing process

To generate a 2D image, rays are traced from the viewpoint and pass through each pixel of the projection window until they collide with an object.

Note that each pixel in the projection window does not correspond to a point but to a rectangle that has a width $\|\Delta x\|$ and a height $\|\Delta y\|$. To improve image quality, instead of tracing a single ray, a set of rays passing through random positions within the pixel rectangle is determined. The number of rays cast per pixel is called the **number of samples per pixel**.

Given a position in the final image determined by its coordinate (f, c), a position Q is generated in the projection window. To do this, two random values δ_x and δ_y are first generated, both in the interval $[-\frac{1}{2}, \frac{1}{2}]$. Next, position Q is computed as a random position in the following intervals:

$$Q = P + \vec{\Delta x} \cdot (c + \delta_x) + \vec{\Delta y} \cdot (f + \delta_y)$$

For each of the positions Q_i generated in the rectangle defined by the pixel, a ray is traced whose origin is the **position** of the viewpoint P and that passes through position Q_i. In this way, a ray can be seen as a semi-line that has an origin P and a direction vector $\vec{d_r}$.

For each generated ray, the color obtained by projecting the ray onto the scene is computed. This color is obtained as an RGB value where each component is found in the real number space $[0, 1]$. Subsequently, the contributions of all rays that correspond to the same pixel are averaged, with this average value being the value assigned to that pixel.

Before generating the final value on the scale from 0 to 255, two final operations must be performed:

1. Application of **gamma correction**. Each intensity value in the range $[0, 1]$ is raised to the exponent $\frac{1}{\gamma}$. This correction improves image quality.
2. **Intensity** scaling. Each value in the range $[0, 1]$ obtained after gamma correction is scaled to the discrete interval $[0, 255]$. In this way, the original value 0 corresponds to a discrete value of 0 and the original value 1 corresponds to a discrete value of 255. If necessary, the obtained value is truncated.

3.3 Color contribution of a ray

When a ray is traced from the viewpoint to the scene, a **depth** is incorporated. This value is the number of bounces that the ray produces on other objects. That is, if the depth is 1, only the collision of the ray with the first object is taken into account. However, if this depth were greater than 1, subsequent bounces of the ray with other objects would be taken into account. These bounces will depend on the material of the object with which the ray collides.

In this way, there are three elements that intervene in ray projection: the **ray** itself, the **scene** with the objects, and the level of **depth** with which the computation is performed. With all this, the color contribution of each ray collision is computed.

- When the depth is less than or equal to 0, there is no contribution to color.
- If the ray depth is positive, the **intersection** of the ray with the scene is computed.
  - If **several objects** have intersection with the ray, the **closest intersection** to the **viewpoint** is selected.
  - Once the intersection is determined, the **contribution** to the color corresponding to this **intersection** is computed.
- If no intersection of the ray with any object in the scene occurs, the **background color contribution** is generated.

3.4 Ray intersection with the scene

A ray can have an intersection with several objects in the scene. In this case, the intersection of the ray with each object in the scene will be computed, maintaining in each iteration the information of the closest intersection and the distance from the ray origin to that intersection. In any case, intersections at a distance less than 10^{-3} will not be taken into account.

The computation of the intersection of a ray with an object depends on the type of specific object. In this project, intersections with **spheres** and **cylinders** are considered.

3.4.1 Ray intersection with a sphere

All points of a ray can be represented by the expression $P(\lambda)$, as the position of the ray origin O_r added to the direction vector $\vec{d_r}$ multiplied by the value of parameter λ:

$$P(\lambda) = O_r + \vec{d_r}\lambda$$

On the other hand, if a point $P(\lambda)$ is on the surface of a sphere with center C and radius r, it must satisfy:

$$\|C - P(\lambda)\|^2 = r^2$$

Combining these two expressions and remembering the relationship between magnitude and dot product ($\|v\|^2 = \vec{v} \cdot \vec{v}$), we have the following expression:

$$(C - (O_r + \vec{d_r}\lambda)) \cdot (C - (O_r + \vec{d_r}\lambda)) = r^2$$

If we define the vector $\vec{r_c}$ as the vector from the ray origin O_r to the sphere center (that is, $\vec{r_c} = C - O_r$), we have:

$$(\vec{r_c} - \vec{d_r}\lambda) \cdot (\vec{r_c} - \vec{d_r}\lambda) = r^2$$

This vector equation can also be represented as:

$$\vec{d_r} \cdot \vec{d_r}\lambda^2 - 2\vec{d_r} \cdot \vec{r_c}\lambda + \vec{r_c} \cdot \vec{r_c} - r^2 = 0$$

Which is a quadratic equation in which:

$$a = \vec{d_r} \cdot \vec{d_r} \qquad b = 2\vec{d_r} \cdot \vec{r_c} \qquad c = \vec{r_c} \cdot \vec{r_c} - r^2$$

And to compute the values of λ, the solution of the quadratic equation is applied:

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

Three cases must be considered depending on the discriminant value:

- If the discriminant is negative, there is no intersection between the ray and the sphere.
- If the discriminant is 0, there is a single intersection point.
- If the discriminant is positive, there are two intersection points.

Note that only the intersection point closest to the viewpoint should be considered and only if no other closer intersection with another object has been found.

If an intersection point is located, the following are also computed:

- The position in space of the **intersection point**, which is obtained by evaluating the vector expression $I = O_r + \vec{d_r}\lambda$.
- The **normal vector** to the sphere at the intersection point, which is obtained by evaluating the vector expression $\vec{d_n} = (I - C)/r$.
- The **ray length** to the intersection point, which is given by the value λ.
- The **orientation** of the normal vector which can be **outward** or **inward** of the sphere. If the dot product of the ray direction $\vec{d_r}$ and the normal vector $\vec{d_n}$ is negative, the orientation is **outward**. Otherwise, it is **inward**.

  Note that if the **orientation** is **inward**, the sign of the **normal vector** must be changed.

3.4.2 Ray intersection with a cylinder

The computation of intersection with a cylinder has two stages:

1. Calculation of the closest intersection with the curved surface of the cylinder.
2. Calculation of intersections with each of the two bases of the cylinder if either of them is closer than the intersection already found.

To perform the computations, the following values are used:

- C: Point that is the center of the cylinder.
- r: Radius of the cylinder.
- $\hat{a}$: Unit vector of the cylinder axis direction.
- h: Height of the cylinder.

Intersection with the curved surface of the cylinder The intersection of the ray with the curved surface of the cylinder can result in zero intersections (no intersection), one intersection (the ray is tangent to the cylinder), or two intersections (the ray cuts the curved surface of the cylinder).

First, the intersection of the ray with an equivalent cylinder but of infinite height is computed.

To do this, we start with the vector $\vec{r_c}$ that goes from the ray origin O_r to the cylinder center C (that is, $\vec{r_c} = O_r - C$).

Note that for any vector $\vec{v}$, its component perpendicular to axis $\hat{a}$ can be computed as:

$$\vec{v}_{\perp\hat{a}} = \vec{v} - (\vec{v} \cdot \hat{a})\hat{a}$$

On the other hand, any point P on the ray of origin O_r and direction $\vec{d_r}$ can be expressed as a function of parameter λ as:

$$P(\lambda) = O_r + \vec{d_r}\lambda$$

For a point $P(\lambda)$ to be on the cylinder surface, it must satisfy:

$$\|(P(\lambda) - C)_{\perp\hat{a}}\|^2 = r^2$$

In the case of a point on the ray $P(\lambda)$ (which is $O_r + \vec{d_r}\lambda$)

$$\|(O_r + \vec{d_r}\lambda - C)_{\perp\hat{a}}\|^2 = r^2$$

Taking into account that $\vec{r_c} = O_r - C$

$$\|(\vec{r_c} + \vec{d_r}\lambda)_{\perp\hat{a}}\|^2 = r^2$$

Applying the definition of component perpendicular to axis $\hat{a}$, we have:

$$\|(\vec{r_c} + \vec{d_r}\lambda) - ((\vec{r_c} + \vec{d_r}\lambda) \cdot \hat{a})\hat{a}\|^2 = r^2$$

Which can be expressed as:

$$\|\vec{r_c} - (\vec{r_c} \cdot \hat{a})\hat{a} + \vec{d_r}\lambda - (\vec{d_r}\lambda \cdot \hat{a})\hat{a}\|^2 = r^2$$

Taking into account the definition of component perpendicular to axis $\hat{a}$, it can be expressed as:

$$\|\vec{r}_{c_{\perp\hat{a}}} + \vec{d}_{r\perp\hat{a}}\lambda\|^2 = r^2$$

Applying the relationship between dot product and norm we have:

$$\|\vec{d}_{r_{\perp\hat{a}}}\|^2\lambda^2 + 2(\vec{r}_{c_{\perp\hat{a}}} \cdot \vec{d}_{r\perp\hat{a}})\lambda + \|\vec{r}_{c_{\perp\hat{a}}}\|^2 - r^2 = 0$$

And we obtain a quadratic equation in which the coefficients a, b and c are:

$$a = \|\vec{d}_{r_{\perp\hat{a}}}\|^2 \qquad b = 2(\vec{r}_{c_{\perp\hat{a}}} \cdot \vec{d}_{r\perp\hat{a}}) \qquad c = \|\vec{r}_{c_{\perp\hat{a}}}\|^2 - r^2$$

And to compute the values of λ, the solution of the quadratic equation is applied:

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

Three cases must be considered depending on the discriminant value:

- If the discriminant is negative, there is no intersection between the ray and the cylinder surface.
- If the discriminant is 0, there is a single intersection point.
- If the discriminant is positive, there are two intersection points.

Note that only the intersection point closest to the viewpoint should be considered and only if no other closer intersection with another object has been found.

If an intersection point is located, the following are also computed:

- The position in space of the **intersection point**, which is obtained by evaluating the vector expression $I = O_r + \vec{d_r}\lambda$.

  It must be verified that the intersection point is within the limits of the cylinder. To do this, the vector from the intersection point I to the center C is computed and its dot product with the unit axis $\hat{a}$ is computed ($(I - C) \cdot \hat{a}$). If the obtained distance is greater than $\frac{h}{2}$, the intersection is discarded.

- The **normal vector** to the cylinder at the intersection point, which is obtained by evaluating the vector expression

$$\vec{d_n} = (I - C)_{\perp\hat{a}} = (I - C) - ((I - C) \cdot \hat{a})\hat{a}$$

- The **ray length** to the intersection point, which is given by the value λ.
- The **orientation** of the normal vector which can be **outward** or **inward** of the cylinder. If the dot product of the ray direction $\vec{d_r}$ and the normal vector $\vec{d_n}$ is negative, the orientation is **outward**. Otherwise, it is **inward**.

  Note that if the **orientation** is **inward**, the sign of the **normal vector** must be changed.

Intersection with the cylinder bases The intersection with each of the cylinder bases must also be considered. To do this, we start with the planes defined by each of the bases. Each plane is defined by a point and a normal vector to that plane:

- The **upper base** is defined by:
  - The **point** $P = C + \frac{h}{2}\hat{a}$.
  - The **normal vector** $\vec{p_n} = \hat{a}$.
- The **lower base** is defined by:
  - The **point** $P = C - \frac{h}{2}\hat{a}$.
  - The **normal vector** $\vec{p_n} = -\hat{a}$.

For each plane, the length of the intersection of the ray with the corresponding plane must be determined:

1. The vector from the ray origin O_r to the plane point P is determined as $\vec{r_p} = P - \vec{o_r}$.
2. The distance d_p is computed as:

   $$\lambda = \frac{\vec{r_p} \cdot \vec{p_n}}{\vec{r_d} \cdot \vec{p_n}}$$

   Note that if the absolute value of the denominator is very small (less than 10^{-8}), it is considered that there is no intersection because a very high distance would be obtained.

If the obtained distance is closer than other previously obtained distances, the intersection is computed.

- The position in space of the **intersection point**, which is obtained by evaluating the vector expression $I = O_r + \vec{d_r}\lambda$.

  If the distance between the intersection point I and the plane point P is greater than the radius r, it is considered that there is no intersection.
- The **normal vector** to the base is $\vec{p_n}$.
- The **ray length**, which is given by the value λ.
- The **orientation** of the normal vector which can be **outward** or **inward** of the cylinder. If the dot product of the ray direction $\vec{d_r}$ and the normal vector $\vec{d_n}$ is negative, the orientation is **outward**. Otherwise, it is **inward**.

  Note that if the **orientation** is **inward**, the sign of the **normal vector** must be changed.

3.5 Color contribution at an intersection

When the closest intersection of a ray with the scene has been determined, the color corresponds to the intersection. This is done in several steps:

- **Generation** of a **new ray** r_n, which has as origin O_r the intersection position I and as direction vector $\vec{d_r}$ the vector determined by reflection depending on the material type.
- **Calculation** of the **reflected color** by the new ray in the scene decreasing the depth level by one unit.
- **Attenuation** of the reflected color obtained in the previous step with the reflectance of the material of the current intersection.

$$c_x = c_x \cdot r_x \qquad c_y = c_y \cdot r_y \qquad c_z = c_z \cdot r_z$$

3.5.1 Reflection in matte materials

In the case of a matte material, the reflected ray is generated in a random direction. To do this, the reflection direction $\vec{d_r}$ is computed by adding to the normal vector $\vec{d_n}$ of the intersection a random value between -1 and 1 in each of its three components.

It is possible that the resulting direction is too small. A vector v is considered too small if the following conditions are met (all must be satisfied):

$$|v_x| < 10^{-8} \qquad |v_y| < 10^{-8} \qquad |v_z| < 10^{-8}$$

In that case, the reflection direction $\vec{d_r}$ is the normal vector $\vec{d_n}$.

As a result of the reflection we have:

- The resulting **reflectance** is that of the matte material.
- The **direction** of the new ray is $\vec{d_r}$.

3.5.2 Reflection in metallic materials

In the case of a metallic material, the reflection direction $\vec{d_r}$ is computed from the original ray direction $\vec{d_o}$ and the normal vector $\vec{d_n}$ of the intersection.

First, the initial reflection direction $\vec{d_r^1}$ is computed:

$$\vec{d_r^1} = \vec{d_o} - 2(\vec{d_o} \cdot \vec{d_n})\vec{d_n}$$

Next, the final reflection vector is computed by adding a diffusion vector from the material's diffusion factor Φ. To do this, a random vector $\vec{\phi}$ is generated in which each component takes a value between $-\Phi$ and Φ. The reflection vector $\vec{d_r}$ is computed by adding to the initial reflection direction (normalized) the diffusion vector:

$$\vec{d_r} = \frac{\vec{d_r^1}}{\|\vec{d_r^1}\|} + \vec{\phi}$$

As a result of the reflection we have:

- The resulting **reflectance** is that of the metallic material.
- The **direction** of the new ray is $\vec{d_r}$.

3.5.3 Reflection in refractive materials

In the case of a refractive material, the refraction angle θ must be determined from the unit direction vector of the original ray $\hat{d}_o$ (which is computed as $\frac{\vec{d_o}}{\|\vec{d_o}\|}$) and the normal vector of the intersection $\vec{d_n}$:

$$\cos\theta = \min(-\hat{d}_o \cdot \vec{d_n}, 1) \qquad \sin\theta = \sqrt{1 - \cos^2\theta}$$

When determining the direction of the new ray, the corrected refraction index is used:

$$\rho' \begin{cases} \rho & \text{if direction is outward} \\ \frac{1}{\rho} & \text{if direction is inward} \end{cases}$$

Once the refraction index is corrected, two cases must be considered:

- The value $\rho' \cdot \sin\theta$ is greater than 1:
  - The reflection direction is obtained as

    $$\vec{d_r} = \hat{d}_o - 2(\hat{d}_o \cdot \vec{d_n})\vec{d_n}$$
- The value $\rho' \cdot \sin\theta$ is less than or equal to 1:
  - The reflection direction is obtained from the sum of two vectors $\vec{u}$ and $\vec{v}$:

    $$\vec{u} = \rho'(\hat{d}_o + (\cos\theta)\vec{d_n}) \qquad \vec{v} = -(\sqrt{|1 - \|\vec{u}\|^2|})\vec{d_n}$$

As a result of the reflection we have:

- The resulting **reflectance** is always $(1, 1, 1)$. That is, 100% in all its components.
- The **direction** of the new ray is $\vec{d_r}$.

3.6 Background color

The background color is computed as a combination of two colors: the **dark background color** $\vec{c_d}$ and the **light background color** $\vec{c_l}$. The percentage of each of these two colors in the mixture depends on the Y coordinate.

To do this, the unit vector of the direction of the ray being traced is determined:

$$\hat{d}_r = \frac{\vec{d_r}}{\|\vec{d_r}\|}$$

The Y component of this unit vector $\hat{d}_{r_y}$ will be a value in the interval $[-1, 1]$. The average between this value and the maximum value which would be 1 can be used as mixing factor m.

$$m = \frac{\hat{d}_{r_y} + 1}{2}$$

The resulting color for the background is composed by mixing the colors $\vec{c_d}$ and $\vec{c_l}$ using factor m:

$$\vec{c} = (1 - m) \cdot \vec{c_l} + m \cdot \vec{c_d}$$

4 Tasks

4.1 Application to be developed

Two applications with different implementation strategies shall be developed. The **render-soa** application will use the SOA (*structure of arrays*) implementation strategy and the **render-aos** application will use the AOS (*arrays of structures*) implementation strategy. Both applications will have a command-line interface.

IMPORTANT

For simplicity, in the rest of this document, the name **render** is used to refer to both **render-soa** and **render-aos** interchangeably.

4.1.1 Application Parameters

The application will take the following parameters:

- Path to the configuration file.
- Path to the scene file.
- Path to the output file.

4.1.2 Analysis of Application Arguments

If the number of arguments received by the application is not exactly 3, an error message will be generated and the program will terminate with an error code.

```
$ render
Error: Invalid number of arguments: 0
$ render cfg.txt
Error: Invalid number of arguments: 1
$ render cfg.txt scn.txt
Error: Invalid number of arguments: 2
$ render cfg.txt scn.txt out.ppm out2
Error: Invalid number of arguments: 4
```

4.2 Software Development

4.2.1 Program to be developed

This task consists of developing a sequential version of the described application using C++23. Please note that this version does not allow the use of multiple execution threads (*multithreading*) or any kind of parallelism.

Two independent executable programs will be developed: **render-soa** and **render-aos**, which will respectively implement the **SOA** and **AOS** strategies.

- **AOS – Arrays of Structures**: The pixels of an image will be represented as a single sequence of values. Each value in the sequence will be composed of three fields that must be in the range of **0** to **255**.
- **SOA – Structure of Arrays**: The pixels of an image will be represented as three independent sequences. Each of the sequences will contain elements that must be in the range of **0** to **255**.

Additionally, other points in the program where the use of structures of array might improve performances should be studied.

Recommendations

- Evaluate the different sequence containers offered by the standard library. You can find a list of them at https://en.cppreference.com/w/cpp/container.
- Identify the most appropriate integer types for each context offered by the language (https://en.cppreference.com/w/cpp/language/types#Integral_types) and by the standard library (https://en.cppreference.com/w/cpp/types/integer).

4.2.2 Project structure

In the GitHub repository https://github.com/comparchuc3m/render-2025-template you will find a template for your project. You can generate your initial project from this project.

This repository contains the following characteristics:

- The **.devcontainer** directory contains the configuration files for a Docker development container that will allow you to generate a container with all the necessary development tools.
- The **cmake** directory contains a file with additional utilities that will be useful for performing coverage analysis of your unit tests.
- The **common**, **aos** and **soa** directories are designed for you to include the source code you must develop.
- The **utcommon**, **utaos** and **utsoa** directories are designed for you to include the source code of your unit tests.
- In the root directory you will find, among others, the following files:
  - Configuration files **.clang-format** and **.clang-tidy**. Please do not modify these files unless you receive explicit instructions to do so.
  - File **CMakeLists.txt** with the project build instructions.
  - File **CMakePresets.json** with the predefined configuration of compilation options.

Project configuration: main CMake file The main **CMakeLists.txt** file establishes the following options:

- Minimum CMake version: 4.0.
- Languages to use in the project: exclusively C++.
- No external libraries are allowed, with the exception of the following libraries:
  - **GSL** (C++ Core Guideline Support Library). This library offers utilities that extend the standard library and that may be useful to you.
  - **GoogleTest**. You can use this library to prepare your unit tests.

During the compilation process, these libraries are downloaded and configured.

- An **ENABLE_CLANG_TIDY** option is established that can be enabled and disabled. When this option is active, the rules established by the **clang-tidy** tool are applied to all source code in the project.

- The **common**, **soa** and **aos** directories are added, as well as the corresponding unit test directories **utcommon**, **utaos** and **utsoa**.

IMPORTANT

All source files must compile without problems and will not emit any compiler warnings. They must also compile without any problems when the **ENABLE_CLANG_TIDY** option is enabled.

Project Configuration: *presets* files The **CMakePresets.json** file, which is also supplied for download, contains several sets of pre-established options:

- Configuration options:
  - **default**: This is the default configuration used for compiling without running **clang-tidy**.
  - **clang-tidy**: Enables **ENABLE_CLANG_TIDY** on top of the default configuration.

- Build options:
  - **gcc-debug**: Debug mode compilation with default options.
  - **gcc-release**: Optimized mode compilation with default options.
  - **clang-tidy-debug**: Debug mode compilation with default options. It also performs static code analysis.
  - **clang-tidy-release**: Optimized mode compilation with default options. It also performs static code analysis.

The options that all configuration and compilation profiles established in this file share are the following:

- Use of the internal **Ninja-Multiconfig** build generator.

- Compilation options with C++ modules are disabled.

- Use of the **g++-14** compiler.

- Compile in strict C++23 mode.

- Warning and error options for a stricter compilation are enabled.

Project Compilation Compilation can be carried out in **Debug** mode (targets **gcc-debug** and **clang-tidy-debug**) or **Release** mode (targets **gcc-release** and **clang-tidy-release**).

Important

Remember that all evaluations will be carried out with compiler optimizations enabled, using the **gcc-release** compilation profile. If you deem it appropriate, you can add additional optimization options in the **CMakePresets.json** file, and stating it in the project report.

Exception

The use of the *C++ Core Guideline Support Library* is permitted. The latest version can be obtained at: `https://github.com/microsoft/GSL`.

Code Quality Rules The source code must be well-structured and organized, as well as appropriately documented. It is recommended (although not required) to follow the **C++ Core Guidelines** (`http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`).

Nevertheless, all rules specified in the document **Coding rules for the C++ language** which is published separately must be followed.

To facilitate the work of the teams, a configuration file for the **clang-tidy** tool will also be supplied.

4.2.3 Software components

The development performed is structured into three software components, which are placed in the existing folders **common**, **aos** and **soa**.

Common folder This folder contains all source files with data types and functions that are common to both versions of the program. In turn, it contains two subdirectories **include** (with all header files) and **src** (with all **.cpp** files).

The **CMakeLists.txt** file defines a library called **common**. In the code that is supplied, the **target_sources** directive depends only on the file **src/vector.cpp**. You must add to this list all source files that you want to include in the **common** library.

The **common** target uses **include** as the include directory. Additionally, a dependency on the **GSL** library is established.

AOS folder This folder contains all source files with data types and functions that are specific to the **AOS** version. It contains two subdirectories **include** (with all header files) and **src** (with all **.cpp** files).

Additionally, in the **src** directory there must be a **main.cpp** file that will contain exclusively the **main()** function.

The **CMakeLists.txt** file defines an executable called **render-aos**. In the code that is supplied, the **target_sources** directive depends only on the file **src/main.cpp**. You must add to this list all source files that you want to include in the **render-aos** executable.

The **render-aos** target uses **include** as the include directory. Additionally, dependencies on the **GSL** and **common** libraries are established.

SOA folder This folder contains all source files with data types and functions that are specific to the **SOA** version. It contains two subdirectories **include** (with all header files) and **src** (with all **.cpp** files).

Additionally, in the **src** directory there must be a **main.cpp** file that will contain exclusively the **main()** function.

The **CMakeLists.txt** file defines an executable called **render-soa**. In the code that is supplied, the **target_sources** directive depends only on the file **src/main.cpp**. You must add to this list all source files that you want to include in the **render-soa** executable.

The **render-soa** target uses **include** as the include directory. Additionally, dependencies on the **GSL** and **common** libraries are established.

4.2.4 Unit Tests

A set of unit tests will be defined and also submitted. The use of GoogleTest (`https://github.com/google/googletest`) is recommended. If another framework for unit testing is desired, authorization from the course coordinator must be obtained.

In any case, evidence of sufficient unit tests must be provided.

utcommon tests The **utcommon** directory contains all unit tests that correspond to the components of the **common** library.

In **CMakeLists.txt** two variables are defined:

- **COMMON_SRC_FILES**: Contains the list of source files being tested.
- **CURRENT_DIR_SRC_FILES**: Contains the list of source files that contain the tests that should be executed.

As an example, a test file **test_vector.cpp** is presented with tests for the **vector** type.

utaos and utsoa tests They contain the tests for the corresponding **aos** and **soa** components, following a similar structure.

4.2.5 Use of AI Tools

The use of AI-based tools during the project is permitted. However, the following must be taken into account:

- If you use an AI-based tool, you must declare its uses in the design section of the project report. No penalty will be incurred for the declaration as long as a statement of use is included.
- No support will be given for the use of such tools. In particular, if you have questions about your code, you must be capable of explaining said code.
- Keep in mind that some AI tools can generate insecure code or code with low performance.
- Any of your professors may require an explanation of your own code.

4.3 Performance and Energy Evaluation

This task consists of carrying out a comparative evaluation of performance and energy consumption. To carry out the performance evaluation, the total execution time will be measured using the **perf** tool. In addition, energy will also be measured and power will be derived.

All performance evaluations will be carried out on a node of the **avignon** cluster.

Plot the total execution times, energy usage, and power for different images. Perform a scalability analysis for those application options for which it makes sense.

The project report will include conclusions derived from the results. Please do not limit yourself to a mere description of the data. Try to find convincing explanations for these results.

5 Report structure to be submitted

The report should be limited to describing the work performed. Conciseness in descriptions will be especially valued. It must be submitted in PDF format. It must contain, at least, the following sections:

- **Title page**: will contain the following data:
  - Project name.
  - Name of the reduced group in which the students are enrolled.
  - Assigned team number.
  - Name and NIA of all authors.
- **Design**. It should explain the global design of the application, complementing it with design diagrams using some commonly used notation (e.g. structure diagrams, behavior diagrams, ...).

  It must also include the main design decisions.

  > **IMPORTANT**
  >
  > Do not repeat in this section what can be specified in code comments, such as what are the parameters of a function, or its algorithm.

  Maximum length: 3 pages.
- **Optimization**. It must contain a discussion of the applied optimizations and their impact.

  In particular, optimizations performed on the original source code should be indicated, as well as optimizations enabled with additional compilation flags that are deemed appropriate.

  Maximum length: 1 page.
- **Tests performed**: Description of the test plan performed to ensure correct execution. It must include unit tests, as well as functional system tests.

  In the report, only a general overview of the testing approach should be given. The list of specific tests and how to execute them should be found in execution scripts (**utest.sh**/**utest.py** and **ftest.sh**/**ftest.py**).

  Maximum length: 2 pages.

- **Performance and energy evaluation**: It should include the performance and energy evaluations carried out.

  Maximum length: 5 pages.

- **Work organization**: It should describe the work organization among team members, making explicit the tasks carried out by each person.

  - It must contain a division of the project into tasks.
  - Tasks must be small enough so that a single person can be assigned to a task.
  - All team members must make relevant contributions in the software development process.
  - No more than one person can be assigned to a task. In such case, the task must be subdivided into subtasks.
  - The time dedicated by each person to each task must be indicated.

  Maximum length: 2 pages.

  A table in ODS format can be submitted with at least the following columns: **task name**, **person name**, **number of hours dedicated**.

- **Conclusions**. Those derived from the performance evaluation results will be especially valued, as well as those that relate the work performed with the content of the subject.

  Maximum length: 2 pages.

6 Submission procedure

The project report submission will be delivered through a submitter enabled for this purpose in Aula Global. In this submitter a single file that must be called **report.pdf** will be placed.

The entire project developed in the assigned GitHub project will also be delivered in a ZIP file through Aula Global in a submitter enabled for this purpose. In this submitter a single compressed file in ZIP format that must be called **render.zip** will be placed.

7 Grading

Final grades for this project is obtained in the following way:

- Performance: 20%.
- Energy use: 20%.
- Unit tests: 7%.
- Functional tests: 3%.
- Design quality: 5%.
- Code quality: 5%.
- Performance and energy evaluation in the report: 15%.
- Contributions from each team member: 20%.

- Conclusions: 5%.

RELEVANT WARNINGS

- If the submitted code does not compile, the final grade for the project will be **0**.
- If code contains unauthorized modifications to provided CMake files, the final grade for the project will be **0**.
- If a quality coding rule is ignored without justification, the final grade for the project will be **0**.
- If the execution time is deemed extremely high, the final grade for the project will be **0**.
- In case of copying all implied groups will get a grade of **0**. Moreover, the head of the school will be notified for the corresponding disciplinary actions.

The following are specific evaluation criteria for each of the evaluation dimensions. Please note that these criteria are **merely indicative** and are subject to the interpretation of your instructors.

Performance and energy Different evaluation scenarios and configurations with different levels of difficulty will be defined. For each evaluation, reference values will be established to obtain 5 points out of 10 in the evaluation. A threshold will also be established to obtain 0 points out of 10 in the evaluation.

The final grade for performance and energy will be obtained as the arithmetic mean of the grades from all tests.

Important: These reference values will be published between October 13 and 17.

Unit tests The requirement is that there be a sufficient number of unit tests for all software developed using the GoogleTest framework. No specific coverage criteria have been defined, so coverage will not be analyzed. It will be considered that there are sufficient tests if there are tests for each function or member function of each class.

The following levels are defined:

- **Excellent**
  - There are three unit test executables generated: **utest-common**, **utest-soa** and **utest-aos**.
  - There is a script (**utest.sh**/**utest.py**) that launches all unit tests.
  - GoogleTest is used as the framework.
  - There are test cases for each function and for each member function that consider both basic cases and different error cases.
  - Unit tests do not depend on the existence of specific files in any directory.
- **Acceptable**
  - GoogleTest is used as the framework.
  - There is a set of tests that corresponds to each .cpp file of the source code, but there are no tests for all functions or member functions.

- **Insufficient**
  - GoogleTest is used as the framework.
  - There are some tests but they are insufficient.
  - Some segmentation fault or other catastrophic error occurs when executing the tests.
- **Deficient**
  - No unit tests using Google Test are provided.
  - There are no explanations of unit tests in the report.

Functional tests The requirement here is that there be evidence that functional tests of the complete application have been performed. An attempt should be made to automate the tests.

- **Excellent**
  - Complete functional tests exist and they consider both success cases and error cases.
  - Functional tests have been automated with some script and are reproducible.
- **Acceptable**
  - Functional tests exist but they are incomplete and only consider non-error cases.
  - Functional tests have been automated with some script and are reproducible.
- **Insufficient**
  - Functional tests are described in the report but no automated way to execute them is provided.
  - An attempt is made to perform functional tests from a program written in C++ instead of checking the executable functionality from a script.
- **Deficient**
  - There is no evidence of the existence of functional tests.
  - Functional tests are not mentioned in the report.

Design quality Design comprises logical design (data structures, classes, functions,...) and physical design (source code structure).

- **Excellent**
  - Diagrams are provided and they give a clear vision of the structure of the software.
  - The design of classes and functions is clearly justified.
  - The difference between the structures used for AOS and SOA is clear and these are adequate.
  - Auxiliary data structures are the most adequate.
  - The main decisions for each functionality and the performance-oriented design decisions, including optimizations, are justified.
  - Specific optimization flags used are mentioned.

- **Acceptable**
  - The design of classes and functions is justified, although there are unclear or unjustified decisions.
  - There is a difference between the structures used for SOA and AOS and these are adequate.
  - Auxiliary data structures are not very adequate.
- **Insufficient**
  - The design is not sufficiently justified and is difficult to understand.
  - The structures used for AOS and SOA are not adequate.
- **Deficient**
  - The application design is not described.

Code quality This section evaluates whether the code is of good quality and there are no bad practices. Most of these checks are already performed with **clang-tidy**.

- **Excellent**
  - The code is clear, concise and well formatted. The names of variables, functions and data types are consistent.
  - There are comments documenting each function and data type.
- **Acceptable**
  - The code is clear, concise and well formatted. The names of variables, functions and data types are consistent.
  - There are no comments documenting each function and data type.
- **Insufficient**
  - The code lacks clarity and readability.
  - There are no comments documenting each function and data type.
  - There are some violations of **clang-tidy**, but they are few.
  - Rules of **clang-tidy** have been disabled using **NOLINT** in an unjustified way, but they are few.
- **Deficient**
  - There are violations of **clang-tidy** or rules of **clang-tidy** have been disabled in an unjustified way (i.e., unauthorized **NOLINT**).

Performance and energy evaluation in the report This section will evaluate the discussion in the report about performance and energy evaluation.

- **Excellent**
  - A performance and energy evaluation has been carried out for the different program options.
  - A scalability analysis has been performed and graphs are provided.
  - Conclusions are included based on experimental data.
- **Acceptable**
  - A performance and energy evaluation has been performed for the different program options.
  - A scalability analysis has been performed for at least one significant parameter.
  - No graphs are provided, but data tables are provided.
  - Conclusions are included based on experimental data.
- **Insufficient**
  - A performance or energy evaluation has been performed only for some program options or only screenshots are provided without any graph.
  - No scalability analysis has been included.
  - There are no conclusions based on experimental data.
- **Deficient**
  - No performance and energy evaluation has been performed.

Individual member contributions This is the only section that is evaluated independently for each project team member.

Initial observation: All members of a team will receive a grade of **DEFICIENT** if any of the following circumstances occur:

- No division of work into individual tasks is presented.
- No list of completed tasks is presented.
- **Excellent**
  - Contributions are well described and have adequate granularity.
  - For each contribution, the person's dedication in hours is quantified. The contributions and their quantification are credible.
  - The person's contributions are adequate in quantity and quality.
- **Acceptable**
  - Contributions are well described and have adequate granularity.
  - For each contribution, the person's dedication in hours is quantified. The contributions and their quantification are credible.
  - The person's contributions are somewhat lower than expected in quantity and quality.

- **Insufficient**
  - Only the person's total dedication to the project in hours is quantified, but not the effort dedicated to each specific task.
  - The total number of hours dedicated by the person to the project cannot be determined.
- **Deficient**
  - The effort dedicated by the person to the project is not quantified in hours.

Conclusions This section evaluates the conclusions of the work.

- **Excellent**
  - Conclusions include what is inferred from the experimental data of the project and the statements are coherent and adequate.
  - Conclusions are presented in terms of performance and energy.
  - Conclusions about what was learned are included.
  - The work is connected with topics seen in the course.
  - The conclusions presented do not contain inaccuracies and are relevant.
- **Acceptable**
  - Conclusions are a mere repetition of experimental results without going further.
  - Conclusions about what was learned are included or the work is connected with topics seen in the course.
  - The conclusions presented do not contain inaccuracies and are relevant.
- **Insufficient**
  - Conclusions are a mere repetition of experimental results without going further.
  - No conclusions about what was learned are included nor is the work connected with topics seen in the course.
  - The conclusions presented contain slight inaccuracies, but in general are relevant.
- **Deficient**
  - There are no conclusions or they are irrelevant.