On the structure of this practise, and on how to submit the solutions

This practise consists on exercises identified by number. You need to upload in canvas **one single `.zip` file** with the filename

`{your-full-name}-graded-practise-3.zip`.

If the following instructions are not properly followed, the automatic grader will give you 0 points

This `zip` file should contain **only** directories named `exercise01`, `exercise02`, ..., `exercise15`. **Notice that if you compress *the directory containing these directories*, your `.zip` will not follow the correct structure**.

For example, imagine that you have your directories `exercise01`, `exercise02`, ..., `exercise15` inside a directory named `pc3`. If you right-click on top of `pc3` and choose *compress*, **the zip file that you will generate will be incorrect and you will receive 0 points**. The reason of this is: your `zip` file will contain, inside it, the directory `pc3` *which will contain, inside it*, the directories `exercise01`, `exercise02`, ..., `exercise15`. **And this should not be in this way. The directories `exercise01`, `exercise02`, ..., `exercise15` should be at the root level inside the `zip` file, without having the *container* directory named `pc3` (or any other name)**.

The way to achieve this effect is simple: just select with your mouse *all the directories* `exercise01`, `exercise02`, ..., `exercise10` (NOT the directory containing them, but all the directories), press the right button of your mouse and select *compress*. This procedure will generate a `zip` file that will contain the exercises in the root level.

Note: not following the correct directory structure will result in a grade of 0 points.

Convention for the names of the files containing the solutions for the exercises.

Inside the directory `exercise01` must be a file named `solution.py` that must implement the function requested in the description of the exercise 1, inside the directory `exercise02` must be a file named `solution.py` that must implement the function requested in the description of the exercise 2, and so on. **The filename must follow this convention in a very rigorous manner. Solutions that does not follow this convention will receive 0 points in the exercise**.

How will the evaluation work

Each problem will be automatically judged through the usage of automatic unit tests written in python. Perhaps (not guaranteed), you will be provided with *some* of these unit tests, that you can use as a *basic* confirmation that your code works. You should not rely *only* on the few tests that will be provided. You are encouraged to confirm the correctness of your algorith using any means that you consider. The tests that will be provided are just very basic cases that intends to help you checking very simple cases.

Runtime constraints

All the calls to all the functions in the present practise have a runtime limit of 30 seconds. Although it is desirable that the functions work in *just a few seconds, if not less than 1 second*, situations in which the functions take up to 30 seconds will be accepted. However, cases taking more than 30 seconds will be graded as incorrect.

Problems (Total: 126 points. Optional exercises: 4, 10, 15)

Basic concepts. Textures. Rotations. Translations.

1. (2 points) OFF and PLY meshes of a cube with square faces.

Description

Implement a function named `cube_with_square_faces` that creates either an `OFF` or a `PLY` file representing a cube whose faces are squares. `cube_with_square_faces` receives the full path of the output file and inferes from its extension wether the output has to be an `OFF` or a `PLY` file.

For the `OFF` file format, you can refer to:

`http://www.geomview.org/docs/html/OFF.html`

For the `PLY` file format, you can refer to:

`https://gamma.cs.unc.edu/POWERPLANT/papers/ply.pdf`

The mesh representing the cube should be composed by the following points and faces:

```
Points: 0: -1 -1 -1    1: -1 1 -1    2: -1 1 1;    3: -1 -1 1
        4:  1 -1 -1    5:  1 1 -1    6:  1 1 1;    7:  1 -1 1
```

Faces: `0 3 2 1  3 7 6 2  7 4 5 6  4 0 1 5  1 2 6 5  3 0 4 7`

Prototype

```
cube_with_square_faces( full_path_output_file )
```

Examples

```
cube_with_square_faces(
  full_path_output_file = '/home/someone/cube-squares.off'
  )
  --> produces an OFF file

cube_with_square_faces(
  full_path_output_file = '/home/someone/cube-squares.ply'
  )
  --> produces a PLY file
```

2. (2 points) OFF and PLY meshes of a cube with triangular faces.

Description

Implement a function named `cube_with_triangular_faces` that creates either an `OFF` or a `PLY` file representing a cube whose faces are triangles. `cube_with_triangular_faces` receives the full path of the output file and inferes from its extension wether the output has to be an `OFF` or a `PLY` file.

For descriptions of the `OFF` and `PLY` file formats, please use the references given in the exercise 1.

The mesh representing the cube should be composed by the following points and faces:

Points: `0: -1 -1 -1  1: -1 1 -1  2: -1 1 1;  3: -1 -1 1`
`4:  1 -1 -1  5:  1 1 -1  6:  1 1 1;  7:  1 -1 1`

Faces: `0 3 1  3 2 1  3 7 2  7 6 2  7 4 6  4 5 6`
`4 0 5  0 1 5  1 2 5  2 6 5  3 0 7  0 4 7`

Prototype

```
cube_with_triangular_faces( full_path_output_file )
```

Examples

```
cube_with_triangular_faces(
  full_path_output_file = '/home/someone/cube-triangles.off'
  )
  --> produces an OFF file

cube_with_square_faces(
  full_path_output_file = '/home/someone/cube-triangles.ply'
  )
  --> produces a PLY file
```

3. (4 points) OFF and PLY meshes of sphere with quadrilateral faces.

Description

Implement a function named `sphere_with_quadrilateral_faces` that creates either an `OFF` or a `PLY` file with a mesh representing a sphere of radius `radius`, centered at `center`, using quadrilaterals.

`sphere_with_quadrilateral_faces` receives the full path of the output file, `radius` and `center`. The number of faces in the output mesh is 360×180. The faces are described as follows:

The position of one point P in the sphere S is determined by two angles $\phi \in [0°, 360°), \theta \in [0°, 180°]$. Let $C = (C_x, C_y, C_z)$ be the center of S. ϕ is the angle formed by $\overline{PC}$ with the plane $x = C_x$. θ is the angle formed by $\overline{PC}$ with the plane $z = C_z$. Consider the points whose corresponding angles ϕ and θ are an integer number of degrees. Those are the points that we want to consider as vertices of the quadrilaterals.

For example, the following point will be connected and form a quadrilateral:

$(5°, 7°)$ — $(5°, 8°)$ — $(6°, 8°)$ — $(6°, 7°)$

In general, the points

$(N°, M°)$ — $(N°, M + 1°)$ — $(N + 1°, M + 1°)$ — $(N + 1°, M°)$

will form a quadrilateral. Notice that $N + 1$ and $M + 1$ are to be taken modulo 360.

Prototype

```
sphere_with_quadrilateral_faces(
  full_path_output_file,
  radius,
  center)
```

Examples

```
sphere_with_quadrilateral_faces(
  path_input_off = '/home/someone/sphere-rectangles.off',
  radius = 5,
  center = (2,3,5))
  --> Produces an OFF mesh of a sphere of radius 5 centered at (2,3,5)

sphere_with_quadrilateral_faces(
  path_input_off = '/home/someone/sphere-rectangles.ply',
  radius = 5,
  center = (2,3,5))
  --> Produces a PLY mesh of a sphere of radius 5 centered at (2,3,5)
```

4. (4 points) OFF and PLY meshes of sphere with triangular faces.

Description

Implement a function named `sphere_with_triangular_faces` that creates either an `OFF` or a `PLY` file with a mesh representing a sphere of radius `radius`, centered at `center`, using triangles.

`sphere_with_triangular_faces` receives the full path of the output file, `radius` and `center`. The number of faces in the output mesh is $2 \times 360 \times 180$. The faces are described as follows:

The vertices are defined exactly in the same way as in the previous exercise.

For each face A — B — C — D of the previous exercise, we create the faces A — B — C and A — C — D

Prototype

```
sphere_with_triangular_faces(
  full_path_output_file,
  radius,
  center)
```

Examples

```
sphere_with_triangular_faces(
  path_input_off = '/home/someone/sphere-rectangles.off',
  radius = 5,
  center = (2,3,5))
  --> Produces an OFF mesh of a sphere of radius 5 centered at (2,3,5)
```

```
sphere_with_triangular_faces(
  path_input_off = '/home/someone/sphere-rectangles.ply',
  radius = 5,
  center = (2,3,5))
  --> Produces a PLY mesh of a sphere of radius 5 centered at (2,3,5)
```

5. (7 points) PLY mesh of sphere with texture.

Description

Implement a function named `sphere_with_texture` that receives a mesh representing a sphere, the coordinates of the center of the represented sphere and creates a `PLY` file with the same input mesh with texture coordinates added to the vertices.

Consider one of the `PLY` files representing a sphere that were generated in the exercise 4. `sphere_with_texture` will associate to the vertex $v = (v_x, v_y, v_z)$ the texture coordinates that are given by the *polar coordinates* (neglecting the radius) of the point v in relation to the center of the sphere.

In other words: consider the map between a rectangle (i.e.: the texture) and a sphere that is defined by the *polar coordinates*. Use this map to assign texture coordinates to the vertices of the sphere.

`sphere_with_texture` should work easily when changing the texture file. The example textures that are provided are `texture1.png`, `texture2.png`, and `texture3.png`.

Prototype

```
sphere_with_texture(
    full_path_input_ply,
    full_path_texture,
    center,
    full_path_output_ply)
```

Examples

```
sphere_with_texture(
    full_path_input_ply='/home/someone/sphere-rectangles.ply',
    full_path_texture='texture1.png',
    center=(2,3,5),
    full_path_output_ply='sphere-with-texture-1.ply')
```

6. (7 points) Rotation of a mesh around a line.

Description

Implement the function `rotate_mesh_around_line` that receives a mesh, a line L (specified by a point p and a vector $d = (d_x, d_y, d_z)$, such that the points in the line are the points $p + kd$, with $k \in \mathbb{R}$) and an angle α, and produces an output mesh where the vertices are rotated α degrees around L.

`rotate_mesh_around_line` works both for input meshes in OFF format and in PLY format. The format of the output should be the same as the format of the input.

Prototype

```
rotate_mesh_around_line(
  full_path_input_mesh,
  axis_of_rotation, # Has the structure ((p_x, p_y, p_z), (d_x, d_y, d_z))
                    # where (p_x, p_y, p_z) is a point and
                    # (d_x, d_y, d_z) is a direction
  alpha, ## number of degrees that we want to rotate the vertices
  full_path_output_mesh)
```

Examples

```
rotate_mesh_around_lin(
    full_path_input_mesh='/home/someone/sphere-rectangles.off',
    axis_of_rotation = ((0,0,0),(0,0,1)),
    angle = 35,
    full_path_output_mesh='sphere-rectangles-rotated.off'
)

rotate_mesh_around_lin(
    full_path_input_mesh='/home/someone/sphere-rectangles.ply',
    axis_of_rotation = ((0,0,0),(0,0,1)),
    angle = 35,
    full_path_output_mesh='sphere-rectangles-rotated.ply'
)
```

7. (2 points) Translation of a mesh by a given vector.

Description

Implement the function `translate_mesh` that receives a mesh, a vector $d = (d_x, d_y, d_z)$, and produces an output mesh where the vertices are translated by d.

`translate_mesh` works both for input meshes in OFF format and in PLY format. The format of the output should be the same as the format of the input.

Prototype

```
translate_mesh(
  full_path_input_mesh,
  d, ## d has the format (d_x, d_y, d_z)
  full_path_output_mesh)
```

Examples

```
translate_mesh(
    full_path_input_mesh='/home/someone/sphere-rectangles.off',
    d = (25, 2, 3)
    full_path_output_mesh='sphere-rectangles-translated.off'
)

translate_mesh(
    full_path_input_mesh='/home/someone/sphere-rectangles.off',
    d = (25, 2, 3)
    full_path_output_mesh='sphere-rectangles-translated.off'
)
```

Algorithms of Catmull-Clark and Loop.

8. (15 points) Catmull-Clark.

Implement the function:

```
catmull_clark(
  full_path_input_mesh,
  number_of_iterations,
  full_path_output_mesh)
```

that performs `number_of_iterations` iterations of the Catmull-Clark algorithm on the input mesh received as a parameter.

Examples

```
catmull_clark(
```

```
    full_path_input_mesh = '/home/someone/cube-with-square-faces.off',
    number_of_iterations=3,
    full_path_output_mesh='catmull-clark-from-cube-3-iterations.off')

catmull_clark(
    full_path_input_mesh = '/home/someone/cube-with-square-faces.off',
    number_of_iterations=3,
    full_path_output_mesh='catmull-clark-from-cube-3-iterations.off')
```

9. (15 points) Loop.

Implement the function:

```
loop(
  full_path_input_mesh,
  number_of_iterations,
  full_path_output_mesh)
```

that performs `number_of_iterations` iterations of the Loop algorithm on the input mesh received as a parameter.

Examples

```
loop(
    full_path_input_mesh = '/home/someone/cube-with-triangular-faces.off',
    number_of_iterations=3,
    full_path_output_mesh='loop-from-cube-3-iterations.off')

loop(
    full_path_input_mesh = '/home/someone/cube-with-triangular-faces.ply',
    number_of_iterations=3,
    full_path_output_mesh='loop-from-cube-3-iterations.ply')
```

Drawing implicit surfaces: Marching Squares and Marching Cubes.

10. (7 points) Marching squares.

Implement the function:

```
marching_squares(
    json_object_describing_curve,
```

```
output_filename,
x_min, y_min, x_max, y_max,
precision)
```

that is able to draw the curve described in `json_object_describing_curve` as a tree of boolean operations on implicit curves, using the marching squares algorithm and the adaptative subdivision that we saw in class.

Parameters:

- `json_object_describing_curve` is a json object that describes boolean operations between curves described by implicit functions. This object is exactly as we saw in class. Example:

```
example_json = {
   op = "union",
   function = "",
   childs = [
      {
        op = "",
        function = "(x-2)^2 + (y-3)^2 - 4^2",
        childs=[]
      },
      {
        op = "",
        function = "(x+1)^2 + (y-3)^2 - 4^2",
        childs=[]
      },
   ]
}
```

  OBSERVATION: The posisible values for `op` are: `"union"`, `"intersection"`, and `"diff"`. `"union"` performs the union of all the child objects. `"intersection"` performs the intersection of all the child objects. `"diff"` removes from the first child all the other childs.

- `output_filename` is the filename of the file that should be generated. **It has to be a file in EPS format.**

- $x_{\min}$, $y_{\min}$, $x_{\max}$, $y_{\max}$ are the coordinates of the rectangle that we want to draw in the output image

- `precision` is the stop condition for the subdivision. When both $x_{\max} - x_{\min} <$ `precision` and $y_{\max} - y_{\min} <$ `precision`, we stop subdividing.

Examples

```
marching_squares(
  example_json,
  '/home/someone/example-marching-squares.eps',
  -5, -5, 6, 6,
  0.1)

marching_squares(
  # one circle of radius 1 centered at (2, 2)
  {op:"", function:"(x-2)^2+(y-2)^2-1", childs:[]},

  '/home/someone/example-marching-squares.eps',
  -5, -5, 6, 6,
  0.1)

marching_squares(
  {op:"union", function:"",childs:[
    # circles of radius 1 centered at (2, 2) and (4, 2)
    {op:"", function:"(x-2)^2+(y-2)^2-1", childs:[]},
    {op:"", function:"(x-4)^2+(y-2)^2-1", childs:[]}
  ]},
  '/home/someone/example-marching-squares.eps',
  -5, -5, 6, 6,
  0.1)
```

11. (15 points) Marching cubes.

Implement the function:

```
marching_cubes(
    json_object_describing_surface,
    output_filename,
    x_min, y_min, x_max, y_max, z_min, z_max,
    precision)
```

that is able to draw the surface described in `json_object_describing_surface` as a tree of boolean operations on implicit surfaces, using the marching cubes algorithm and the adaptative subdivision that we saw in class.

Parameters:

- `json_object_describing_surface` is a json object that describes boolean operations between surfaces. It exactly equal to the object described in the previous problem; but this time with functions in 3 variables (this is: x, y, z instead of in 2.
- `output_filename` is the filename of the file that should be generated. It can be wither `OFF` or `PLY`.

- $x_{\min}$, $y_{\min}$, $x_{\max}$, $y_{\max}$ are the coordinates of the rectangle that we want to draw in the output image
- `precision` is the stop condition for the subdivision. When both $x_{\max} - x_{\min} <$ `precision` and $y_{\max} - y_{\min} <$ `precision`, we stop subdividing.

Examples

```
marching_cubes(
  example_json,
  '/home/someone/example-marching-squares.eps',
  -5, -5, -5, 6, 6, 6
  0.1)

marching_cubes(
  # sphere of radius 1 centered at (2, 2, 2)
  {op:"", function:"(x-2)^2+(y-2)^2+(z-2)^2-1", childs:[]},
  '/home/someone/example-marching-cubes.eps',
  -5, -5, -5, 6, 6, 6,
  0.1)

marching_cubes(
  {op:"union", function:"",childs:[
    {op:"", function:"(x-2)^2+(y-2)^2+(z-2)^2-1", childs:[]},
    {op:"", function:"(x-4)^2+(y-2)^2+(z-2)^2-1", childs:[]}
  ]},
  '/home/someone/example-marching-cubes.eps',
  -5, -5, 6, 6,
  0.1)
```

Drawing 3D meshes in 2D.

12. (7 points) Painter's algorithm with simple cosine illumination.

Implement the function:

```
painter_algorithm_simple_cosine_illuminatio(
    full_path_input_mesh,
    full_path_output_image,
    min_x_coordinate_in_projection_plane,
    min_y_coordinate_in_projection_plane,
    max_x_coordinate_in_projection_plane,
    max_y_coordinate_in_projection_plane,
    width_in_pixels,
    height_in_pixels)
```

that takes an input polygonal mesh (with all its faces assumed to be white), and *takes a photo* of it. The cammera is assumed to be in $(0, 0, 0)$, the vision ray is assumed to be $(0, 0, 1)$, and the projection plane is $z = 1$.

When projecting a triangle T, the color of the projected triangle T' is

$$white \times \cos(\alpha)$$

where α is the angle formed by the normal to T and the opposite vector to the vision vector (the vision vector is, in our case, $(0, 0, 1)$).

For the implementation of the Painter's algorithm of this exercise, the ordering for painting the triangles is: order the triangles by *the maximum distance to the origin among the three vertices*.

Examples

```
simple_cosine_illumination(
    full_path_input_mesh='/home/someone/sphere-rectangles.off',
    full_path_output_image='photo-of-sphere.png'
    min_x_coordinate_in_projection_plane = -1.0,
    min_y_coordinate_in_projection_plane = -1.0,
    max_x_coordinate_in_projection_plane = 1.0,
    max_y_coordinate_in_projection_plane = 1.0,
    width_in_pixels = 640,
    height_in_pixels = 480
)
```

13. (15 points) Painter's algorithm with textures.

Consider a mesh with texture. For example: one of the meshes generated in the exercise 5. Implement:

```
painter_algorithm_textures(
    full_path_input_mesh,
    full_path_input_texture,
    full_path_output_image,
    min_x_coordinate_in_projection_plane,
    min_y_coordinate_in_projection_plane,
    max_x_coordinate_in_projection_plane,
    max_y_coordinate_in_projection_plane,
    width_in_pixels,
    height_in_pixels)
```

that takes an input polygonal mesh containing texture coordinates for its vertices, and the texture, and it generates a *2D photo* of the mesh. The cammera is assumed to be in $(0, 0, 0)$, the vision ray is assumed to be $(0, 0, 1)$, and the projection plane is $z = 1$.

For mapping the texture coordinates, use baricentric coordinates.

For the implementation of the Painter's algorithm of this exercise, the ordering for painting the triangles is: order the triangles by *the maximum distance to the origin among the three vertices*.

Examples

```
simple_cosine_illumination(
    full_path_input_mesh='/home/someone/sphere-rectangles.off',
    full_path_input_texture='texture1.png',
    full_path_output_image='photo-of-sphere.png'
    min_x_coordinate_in_projection_plane = -1.0,
    min_y_coordinate_in_projection_plane = -1.0,
    max_x_coordinate_in_projection_plane = 1.0,
    max_y_coordinate_in_projection_plane = 1.0,
    width_in_pixels = 640,
    height_in_pixels = 480
)
```

Putting all together: simple animations.

14. (7 points) Spheres rotating.

Using the concepts that were practised in exercises 13, 5 and 6, implement

```
frames_of_a_textured_sphere_rotating(
    full_path_input_mesh,
    full_path_input_texture,
    rotation_line
    filename_without_suffix_output_frames)
```

that takes an input polygonal mesh containing texture coordinates for its vertices, and the texture, and a rotation line. It generates 360 files named `{filename_without_suffix_output_frames}-1.png`, `{filename_without_suffix_output_frames}-2.png`, ...

If the texture is a planisphere and the rotation line is choosen strategically, we can produce the effect of the earth rotating.

Examples

```
frames_of_a_textured_sphere_rotating(
    full_path_input_mesh='/home/someone/sphere-rectangles.ply',
    full_path_input_texture='planishere.png',
    rotation_line=(0,0,1),
    filename_without_suffix_output_frames='frame-')
```

15. (15 points) A more complex animation.

This problem is free for you to be creative and implement any animation that you want. You can, for example, implement several spheres with different textures rotating and at the same time moving (moving a distance that is computed from the angular speed of the rotation and the radius of the sphere), producing in this way the impression of *rolling*, and they hit other spheres and bounce accordingly.

Produce, first, a sequence of frames. After that, you can put all your frames together and generate an MP4.