2 Particle

Points: 13 (+ max 5 bonus)
The goal of this exercise is the implementation of ray-casted spheres and cylinders. With this you will be able to visualize molecules as a ball and stick model. The example molecule for this exercise is $C_{34}H_{40}F_2N_4O_4$ which you can find in Protein Data Bank (PDB).

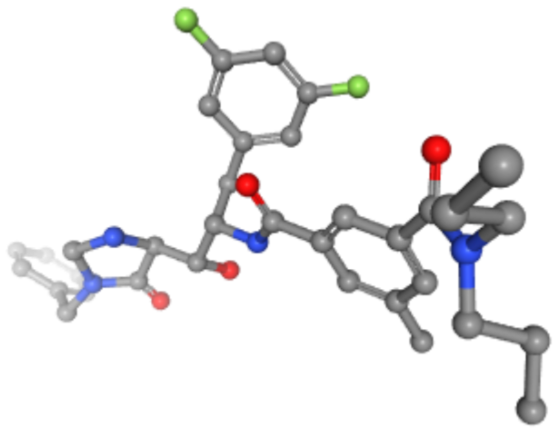

Figure 1: Visualization of molecule $C_{34}H_{40}F_2N_4O_4$ from PDB

The plugin of this exercise does not render anything at the very beginning except for a plane that can be utilized to test the depth implementation afterwards. The `particle` plugin reads the atom center positions, the atom color and connections between the atoms from a sdf-file, stores them and creates vertex buffer objects.

2.1 Sphere Raycasting (8 points)

The spheres are used to render the atoms. Therefore, the `sphere_raycast`-shader gets the positions of the atom center and a color depending on the chemical element as input. The radius of the sphere can be adjusted by using the radius-slider in the GUI and is the same for all spheres (and therefore given as a uniform-variable in the shader). Your task is to ray cast the spheres with this information. You can use the presented formulas of `2-particle-intro.pdf`.

a) Render the **silhouette** quad. At first compute all variables that parametrize the silhouette in the vertex shader (`sphere_raycast.glvs`). Use the already present `sphere_parameter_space` struct for this (2 points). Then compute the positions of the corners of the silhouette quad which are used to create a rectangle specified by a triangle strip in the geometry shader (`sphere_raycast.glgs`) (2 points). (**4 point**)

b) **Discard fragments** not belonging to the sphere in the fragment shader by returning *false* in the method `compute_ray_sphere_intersection()`. Use the simplified intersection test using texture coordinates. (**1 point**)

c) Compute the ray intersection with the sphere in `compute_ray_sphere_intersection()`. Store the position and normal at the intersection in eye coordinates.(**2 point**)

d) Update the **depth** buffer by setting the correct value to gl_FragDepth in the fragment shader. Some invariant parts could be computed in the vertex- or geometry shader as well. You can test your depth computation by moving the provided plane through the scene and look at the sphere-plane intersections. (**1 point**)

2.2 Cylinder Ray casting (5 points)

We will use cylinders to render the connections between the atoms. Fragments that cover the cylinder have to be specified in order to efficiently ray cast a cylinder. In this exercise we will use an object orientated bounding box for this. The bounding box is tessellated by calling gl_draw with lines determining the start and end point of each cylinder. The position and color information are passed through the vertex shader to the geometry shader which has to tessellate the bounding box. Finally, the ray intersection with the cylinder has to be computed in the fragment shader.

a) Render the object orientated **bounding box** of the cylinder in the geometry shader. First, you have to compute the vectors u,v and t of the unit cylinder coordinate system. Then compute the transformation matrix from unit cylinder to world coordinates. (**2 point**)

b) Compute the ray intersection with cylinder in the fragment shader `cylinder_raycast.glfs`. First check for fragments covering the planar side of the cylinder (1 point). If not compute the ray intersection with the cylinder mantle by implementing the `compute_cylinder_intersection function` (1 point). The corresponding formulas can be found in the `2-particle-intro.pdf`. (**2 point**)

c) Update the **depth** buffer by setting the correct value to gl_FragDepth in the fragment shader. The formula is the same as for the spheres. You can test your depth computation by moving the provided plane through the scene and look at the sphere-plane intersections. (**1 point**)

2.3 Bonus tasks (max. 5 Points)

a) Implement the double connections between atoms (see Figure 1) that are neglected this far. Expand the already used function `read_sdf_file()` in `particle.cxx` in order to add the information of the connection count from the sdf-file. (**3 point**)

b) Implement the early depth test proposed in the lecture and compare its performance with the previous depth test on a large molecule visualization that benefits from an early depth test. You can find other molecules in the PDB. (**3 point**)

Submission

Put all files in the exercise folder that you modified or added in a Zip archive (named according to the pattern `[YourName]_ex2.zip`) and upload that archive to Opal. Make sure that your solution runs on the type of system used for evaluation (Windows, Visual Studio $\geq$ 2017). If you do not have access to such a system and are in doubt whether your solution will work in this configuration, contact your tutor or supervisor.