Coursework I:
Ray-tracing

COMP0027 Team
Tobias Ritschel, Michael Fischer, Siddhant Prakash, Chen Liu

October 6, 2023

We have shown you the framework for solving the coursework at `https://uclcg.github.io/uclcg/`.

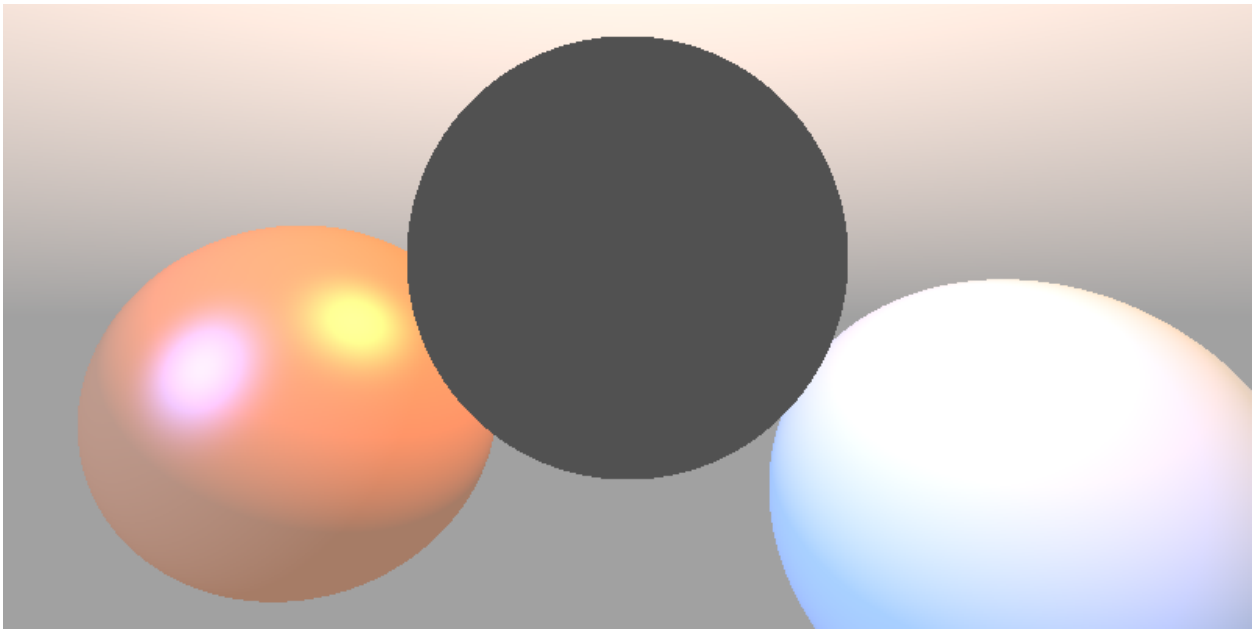
You should start by extending the respective example there. The programming language is WebGL and the OpenGL ES Shading Language (GLSL). Here is a quick reference for GLSL functions that you will commonly use in your coursework, e.g., `reflect` and `sin`.

This should run in any browser, but we formerly experienced problems with Safari and thus recommend using a different browser such as Chrome. Do not write any answers in any other programming language, in paper, or pseudo code. Do not write code outside the `#define` blocks.

Remember to save your solution often enough to a `.uclcg` file. In the end, hand in that file via Moodle.

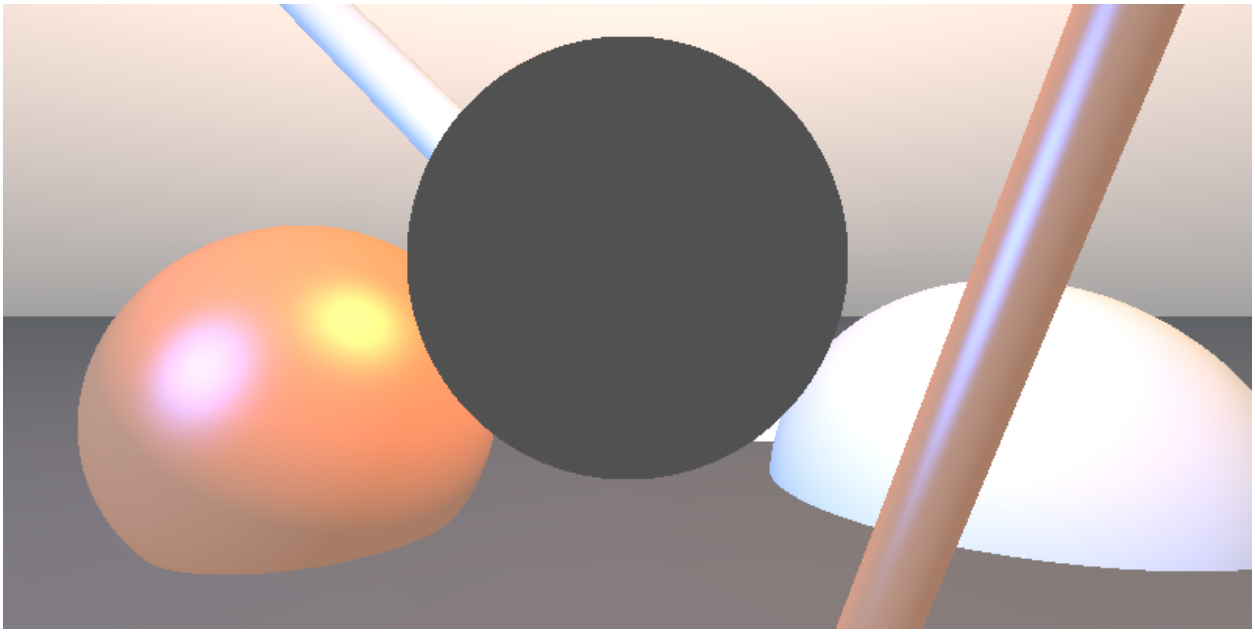
The total points for this exercise is **100**.

Please refer to Moodle for the due dates.



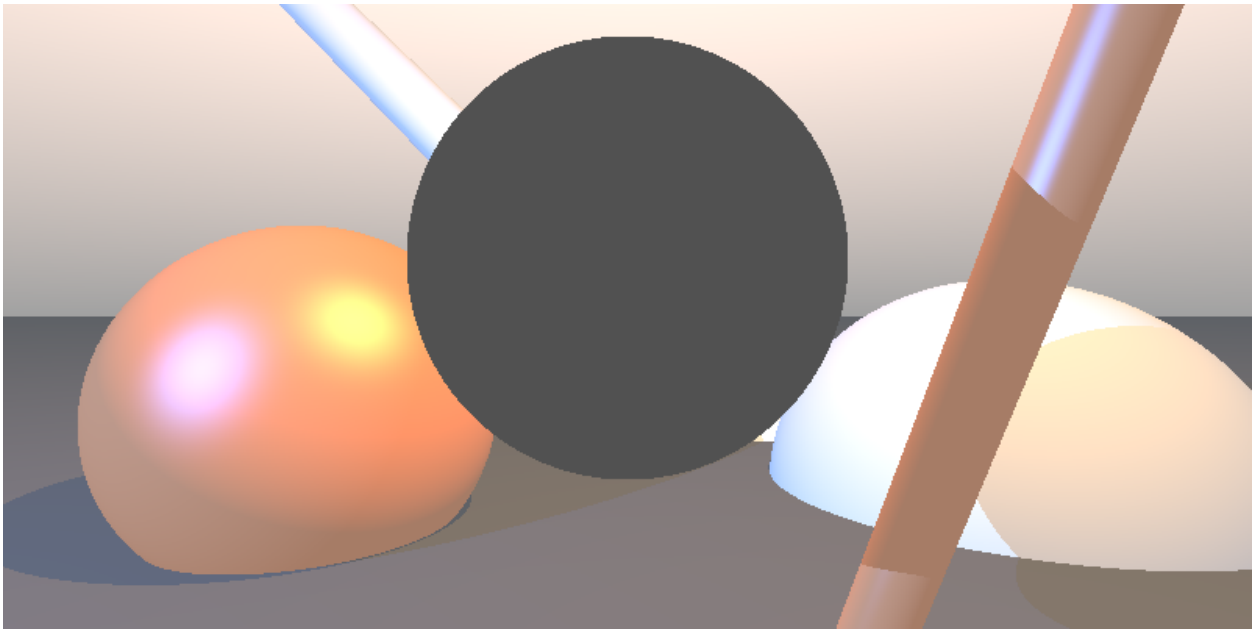
Above, the scene from this coursework, rendered with the simple initial ray-tracer you are asked to extend.

1 New primitives: Plane and cylinder (20 points)



We have added definitions of planes (a normal $\mathbf{n}$ and a distance r to the origin) and cylinders (an orientation $\mathbf{o}$ and a radius r) to the scene. You are asked to code the ray-plane (**5 points**) and ray-cylinder (**10 points**) intersections, and explain how they work (**5 points**). Pay attention on how `Sphere` is intersected. Stick to the same function signature for `Plane` and `Cylinder` that contains `HitInfo`, including normals and material.

2 Casting shadows (10 points)

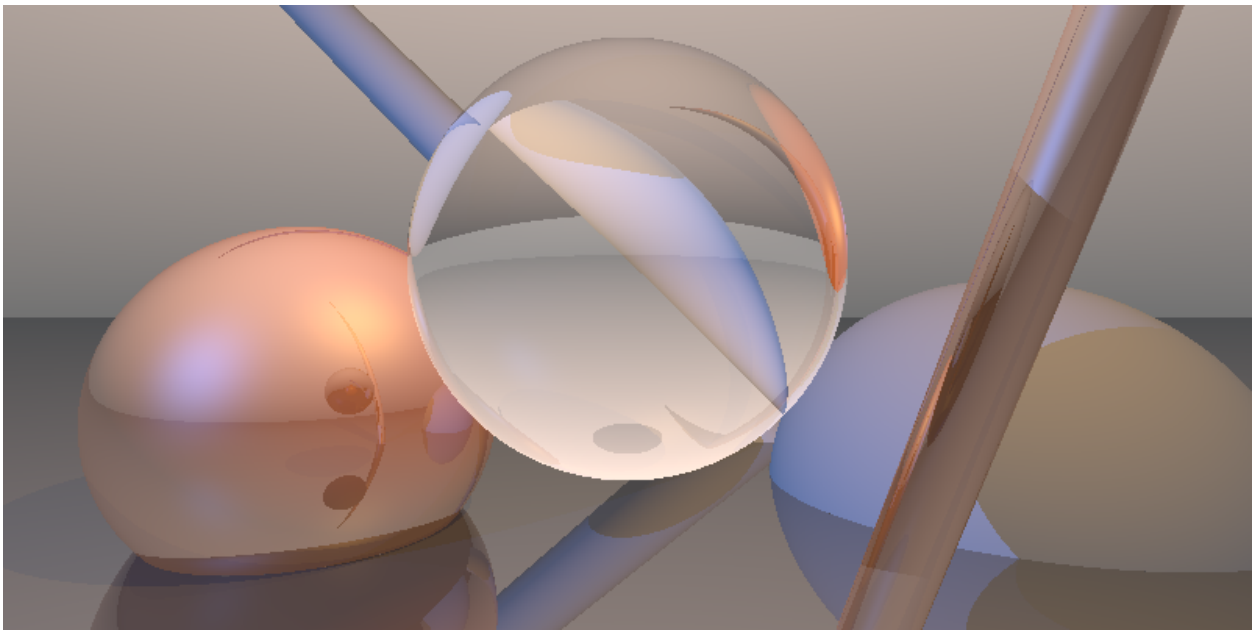


We have discussed how shadows, reflections and refractions work and how recursions can be unrolled into

loops under some conditions. The framework contains the skeleton of such a traversal, but without the code for shadows, reflection and refraction.

First, add shadow tests to the shading (**10 points**).

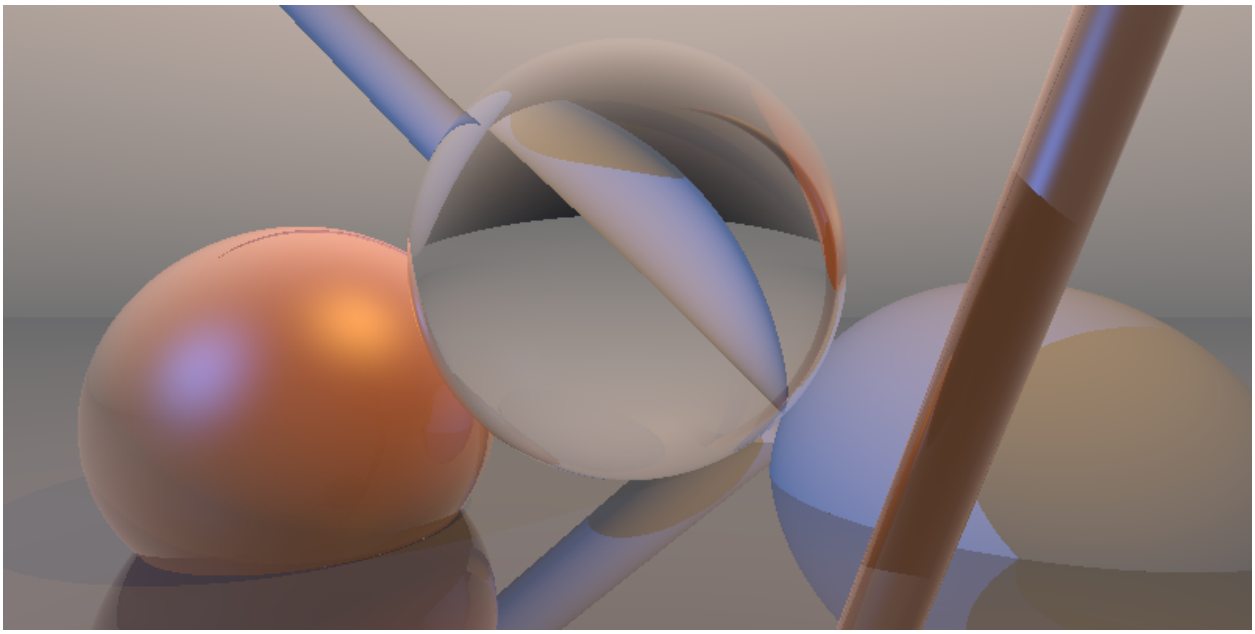
3 Adding reflections and refractions (24 points)



The code already contains the loop to perform the ray traversal iteratively, what remains to be added is code for computing the reflection direction (**12 points**) and refraction direction. Refraction direction is subdivided into two tasks: 1) the main code flow for bouncing rays and including the refractive constant (*10 points*) and 2) using the `enteringPrimitive` flag properly for keeping track of IORs (*2 points*) for a total of (**12 points**).

Note how objects far away enough behind the glass sphere appear mirrored. Although not shown here, objects that are behind and near the glass sphere would just appear distorted.

4 Fresnel (10 points)



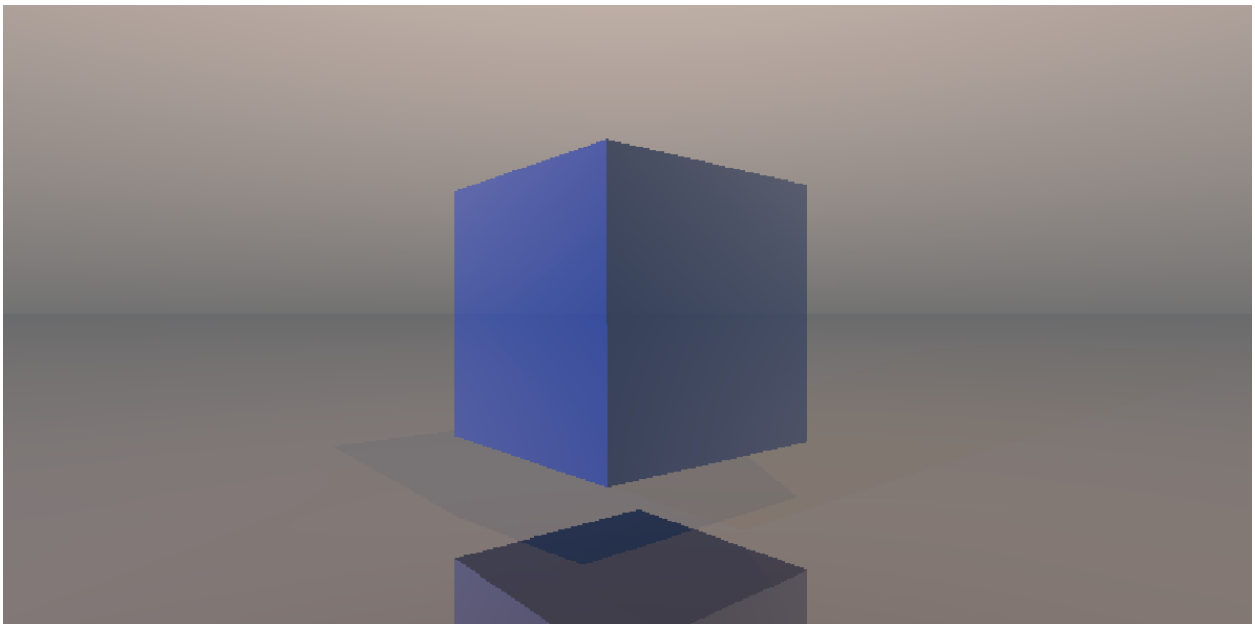
The image with reflection and refraction looks okay, but the glass likely does not look like glass as it shows reflection and refraction at equal strength all over the sphere. Such an image is shown above. Here, and in reality, reflection and refraction strength vary so that w_{reflect} and w_{refract} sums to one or less, so here we simply assume $w_{\text{reflect}} = 1 - w_{\text{refract}}$. This weighting depends on the view direction and the normal at the hit point. The reflection is typically strong on grazing angles close to the edge of the sphere, while the refraction is strongest in the centre. Do some research to find out how those weights could be computed and tell us what you used (**5 points**) and implement it (**5 points**) in the function `fresnel`. A simple implementation such as using a single dot product or the approximation by Schlick could be a good starting point. The image above was produced using the dot product solution, the Schlick one might look different.

5 Shine on, you crazy diamond! (36 points)

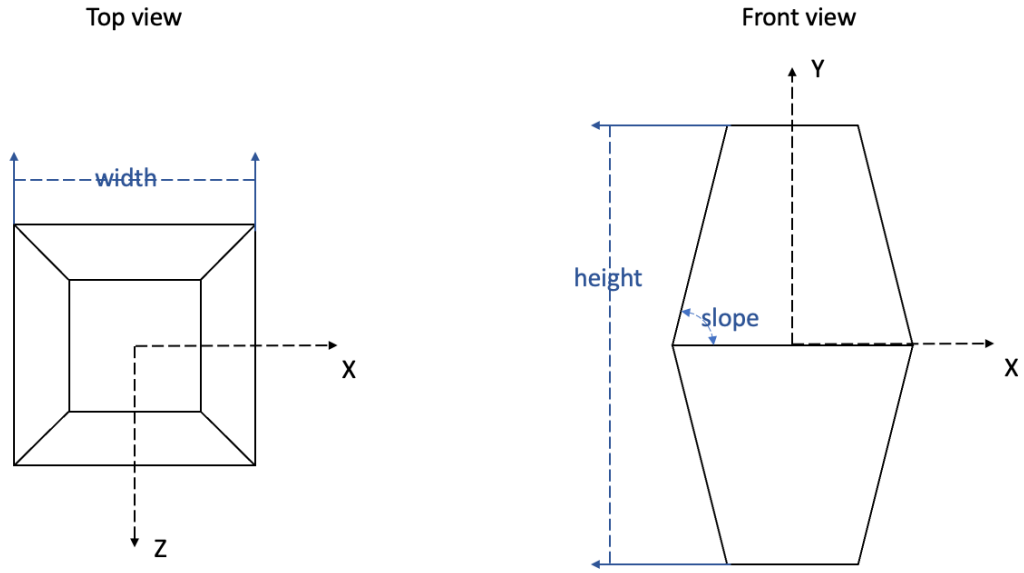
In this task, you are asked to develop a convex polytope, which hopefully will be a diamond.

A convex polytope is comprised of multiple planes. Planes should be familiar to you as defined in Sec. 1 by a normal and a distance. The normal of each plane is defined to point towards the "outside" of the polytope so each plane splits the 3D space into an outer and an inner part. The union of all inner parts will be the polytope.

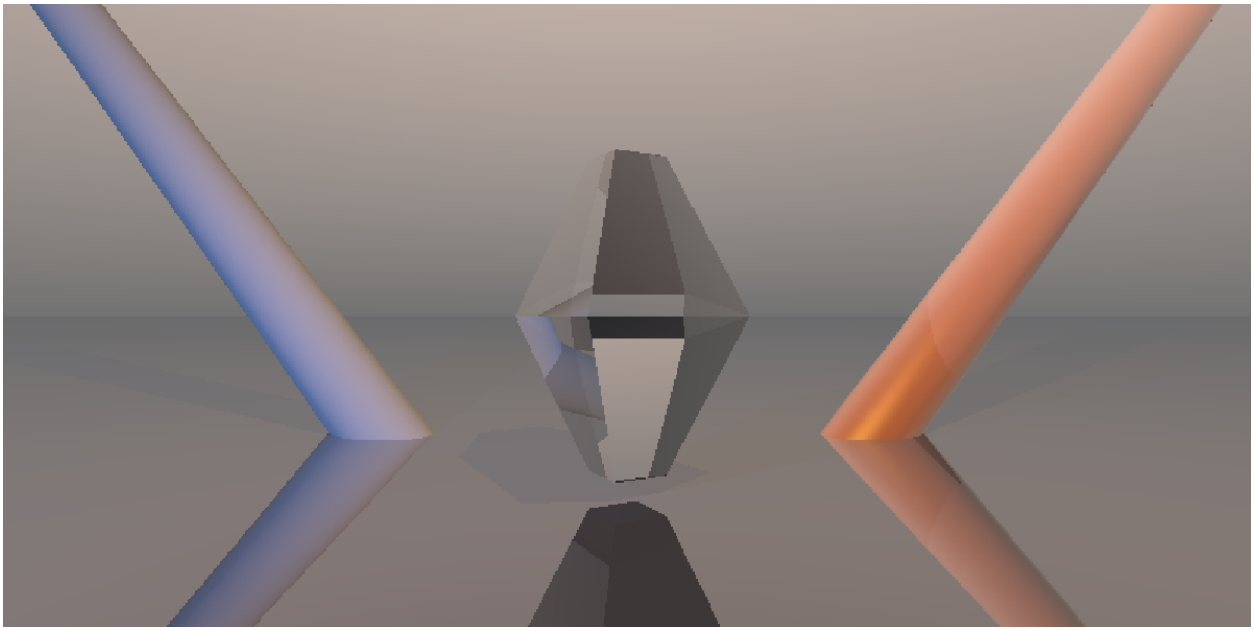
The first component asks you to implement an intersection routine for the polytope (**18 points**) in the function `intersectPolytope`. By doing so correctly, you will see a spinning cube as below (you can press the play button on the top to watch the animation, and if it runs too slowly please adjust the resolution). Please do not use direct quad or triangle intersection code from the Internet but develop something specific to this object.



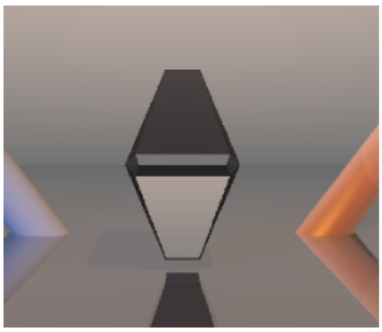
Next, let's carve it into a diamond. We will not directly give the definition code of the diamond, and only show you the desired image, with a couple of key numbers. Our diamond shape has three degrees of freedom: They are `height`, `width`, and `slope`, all illustrated by the following figures. Note that we have a right-hand coordinate system whose X-Y plane is parallel to our 2D screen.



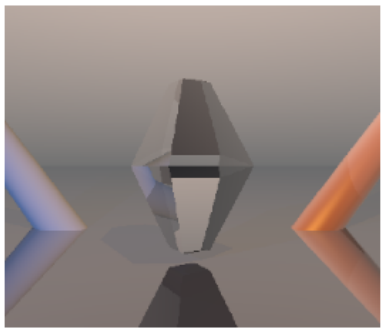
You are asked to implement a function called `loadDiamond` (**12 points**), which parses the given diamond specifications and returns the corresponding diamond polytope centered at the origin (0, 0, 0). One example with the specification `(height=8.0, width=4.37, slope=70°)` is shown below and also what you need to display. Please refer to `loadCube` function for an example that loads six planes and forms a cube polytope.



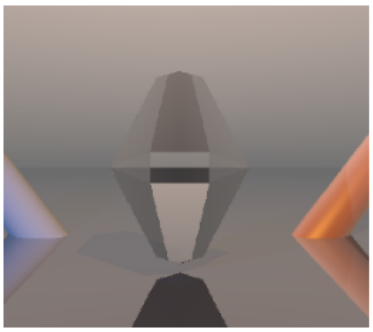
Finally, to ensure correct reflections and refractions as you did for spheres, your `intersectPolytope` function now needs to check if the ray is entering or leaving the polytope and return the correspondingly oriented normal (**6 points**). For marking, we will test the rotating angles of 0°, 30°, and 45°, which are supposed to have the appearance as below. You can test your implementation by modifying the rotating angle `theta` in `main` function.



0°



30°



45°