Virtual Reality
Winter Term 2018/2019

Assignment Sheet 2

To start with this assignment, download the source code from Moodle to your home directory and extract it. Execute `./start.sh` on a terminal to run the application. For your submission, please pack your exercise folder into a `*.zip` file and upload it to Moodle. You are required to do so by **8 November 2018, 11:55 PM**. Group work in pairs of two is permitted. You will present and explain the submitted code in the lab class on **9 November 2018**. Individual time slots for presentation will be arranged with the teaching assistants.

This assignment sheet contains tasks worth **12 points** and will be weighted by **15%** for your total lab class grade.

The solar system is a good example case for the hiearchical organization of a set of objects. The absolute position and orientation of a solar object is determined by its orbital movements around a reference solar object (see Figure 1) and by the local rotation about its own axis (see Figure 2).

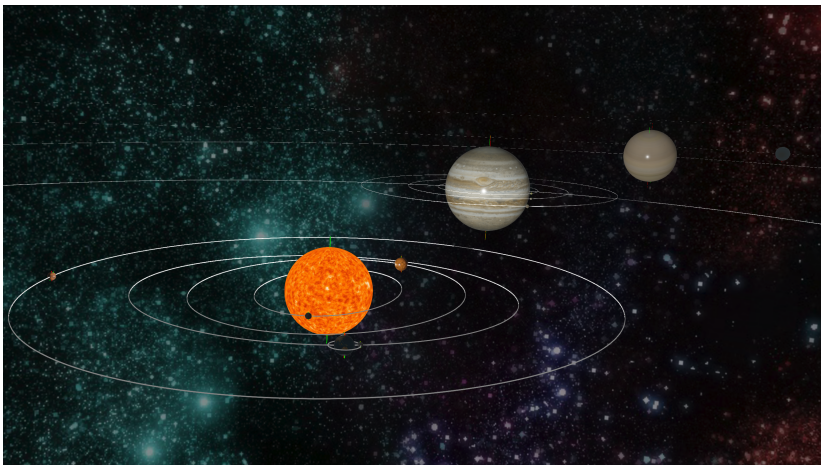

Figure 1: A model of the solar system using multiple instances of the `SolarObject` class (not true to scale)

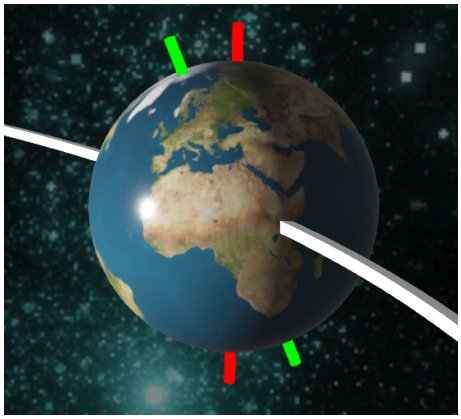

Figure 2: Inclined (green) versus upright (red) rotation axis of the Earth

The given code of this assignment contains the class `SolarObject`, which is used to represent all objects in the solar system. The sun, the planets and their moons are all instances of `SolarObject` and have to be attached to each other in the correct manner. For this purpose, the `SolarObject` class already offers two internal scenegraph nodes. The first node, `object_geometry`, is used to load an external trimesh sphere stored in an obj-file. The varying appearances of the planets and moons are realized through parametrization of the sphere geometry and by applying different textures to the materials. The available textures can be found in the folder `data/textures`. The second node, `orbit_radius_node`, contains the relative spatial transformation regarding the higher-level solar object. The node `object_geometry` is a child node of `orbit_radius_node`. The function `update_orbit` is called every frame and accumulates the respective movement inputs to the local matrix of `orbit_radius_node`. For simplicity, the orbit is implemented as a circular motion.

A simple navigation interface is already provided in the application. The virtual camera can be controlled by using either a Spacemouse or the keyboard. In keyboard mode, the WASD keys are used for translations of the camera on two axes (x and z). PageUp and PageDown enable translations on the third axis (y). The arrow keys rotate the camera on two axes. In Spacemouse mode, all six degrees of freedom are controlled by movements of the Spacemouse. The navigation mode can be switched at the top of `main.py`.

Exercise 2.1 (0 points)

Analyze the code of the provided application in order to understand what functions are already given and how the different classes work together. Start from the file `main.py` and proceed with the imported classes and functions.

Exercise 2.2 (2 points)

In the file `lib/SolarSystem.py`, create two instances of `SolarObject` representing the Earth and its moon. The necessary parameters and texture paths can be found in `lib/SolarParameters.py`. Especially pay attention to set the correct `PARENT_NODE` when creating the solar objects.

After this task, you should see the Earth rotating around the sun and the moon rotating around the Earth. The orbit inclination and the proprietary rotation parameters are still ignored. You can adjust the simulation speed by `+/-` on the numpad. Analyze the scenegraph using the function `print_graph(scenegraph.Root.value)` on the interactive console.

Exercise 2.3 (2 points)

The class `OrbitVisualization` provides code for rendering circular motion paths of solar objects. Extend the class `SolarObject` with an instance of `OrbitVisualization` such that the movement path of each solar object becomes visible. To do so, import the source file containing the `OrbitVisualization` class.

After this task, you should see orbit visualizations for both the Earth and its moon. Analyze the scenegraph using the function `print_graph(scenegraph.Root.value)` on the interactive console.

Exercise 2.4 (2 points)

Extend the behavior of solar objects such that the parameter `ORBIT_INCLINATION` is considered properly. For this purpose, add another scenegraph node `orbit_inclination_node`, fill it with the correct transformation matrix and fit it into the structure of nodes already present in the class `SolarObject`. Since both the Earth and its moon have an orbit inclination of $0°$, set these values to $45°$ for testing.

After this task, the orbits of the Earth and its moon (including their visualizations) should be properly inclined, and the solar objects should still

move along the visualized paths. Analyze the scenegraph using the function `print_graph(scenegraph.Root.value)` on the interactive console.

Exercise 2.5 (3 points)

Extend the behavior of solar objects such that the remaining parameters `ROTATION_INCLINATION` and `ROTATION_DURATION` are considered properly. For this purpose, add another scenegraph node `rotation_inclination_node`, fill it with the correct transformation matrix and fit it into the scenegraph structure of nodes already present in the class `SolarObject`. Finally, implement the frame callback function `update_rotation(self)` such that the solar object's rotation is updated every frame with the corresponding rotation velocity.

After this task, all parameters given in the constructor of `SolarObject` should be considered properly. The Earth should move along the inclined orbit around the sun and rotate itself around the specified axis. Analyze the scenegraph using the function `print_graph(scenegraph.Root.value)` on the interactive console.

Exercise 2.6 (2 points)

Attach the two given axis nodes (`axis_red_geometry` and `axis_green_geometry`) in the class `SolarObject` correctly such that they represent the upright and the inclined orientation of the solar object, respectively.

After this task, all solar objects should contain the visualization of the upright and the inclined rotation axis (see Figure 2).

Exercise 2.7 (1 point)

Initialize the remaining planets Mercury, Venus, Mars, Jupiter (including its three moons), Saturn, Uranus, Neptune and verify again that your functionalities implemented in the previous exercises work correctly.

After this task, your solar system implementation should look like the one illustrated in Figure 1.

Glossary

The goal of this glossary is to list some useful classes and functions of the `avango.gua` module used for this assignment sheet. This list is not a full documentation or exhaustive in any way. If you are uncertain about the variables and functions an object offers, consider tabbing in the interactive console.

`avango.gua.Mat4()`	class providing a 4x4 matrix interface, usually instantiated by `make_*_mat` functions
`avango.gua.Vec3(x,y,z)`	class providing a 3-component vector interface; the elements are accessed by `vec.x`, `vec.y` and `vec.z`; the length can be computed by `vec.length()`
`avango.gua.make_trans_mat(x,y,z)`	creates a translation matrix along `x`, `y` and `z` (floats)
`avango.gua.make_rot_mat(d, vector)`	creates a rotation matrix of `d` (float) degrees around the axis specified by `vector` (Vec3)
`avango.gua.make_scale_mat(sx,sy,sz)`	creates a scale matrix by `sx`, `sy` and `sz` (floats)
`avango.gua.nodes.SceneGraph()`	class representing a scenegraph; the constructor creates an empty scenegraph only containing the `Root` node
`avango.gua.nodes.TransformNode(Name = "node")`	class representing a transformation node in the scenegraph; important fields are `Transform` and `Children`
`avango.gua.nodes.TriMeshLoader()`	class providing helper functions to create `TriMeshNode` instances from .obj files on the hard disk

`loader.create_geometry_from_file(name, path, loader_flags)`	creates a single `TriMeshNode` called `name` or a subtree of nodes representing the hierarchies specified in the .obj file located at `path`; `loader_flags` specify additional options with `avango.gua.LoaderFlags.DEFAULTS` being the most simple one
`avango.gua.nodes.Material()`	class representing an object's material used for shading; each `TriMeshNode` has an associated material stored in the field `Material`
`material.set_uniform("UniformName", value)`	sets a uniform value associated with this material; userful uniforms are `("Emissivity", float)`, `("Roughness", float)`, `("Metalness", float)`, `("ColorMap", texture_path)`

Additionally, the code of this assignment offers two additional helper functions in `main.py`:

`print_graph(node)`	prints all scenegraph nodes attached below `node`
`print_fields(node, bool)`	prints all fields a specific scenegraph `node` offers; if `bool` is `True`, the field values are also printed