Virtual Reality
Winter Term 2019/2020

Assignment 1 - Transformations and Scenegraphs

Throughout the lab classes, we are going to use *Python3* to write virtual reality applications in our framework *avango-guacamole*. In this assignment, you will implement basic algorithms to deal with transformation matrices, vectors, and simple scenegraphs, which are essential for writing higher-level applications. A glossary of important classes and functions of our framework can be found in the corresponding Moodle activity.

Group work in pairs of two is encouraged. You are required to submit this assignment by **31 October 2019, 11:55 pm** on Moodle. Furthermore, you will be asked to present and discuss your results in the lab class on **01 November 2019**. Please register for an individual time slot with the teaching assistants on Moodle (one per group). This assignment contains tasks worth a total of **22 points** and will be weighted by **1/6** for your total lab class grade.

Getting Started

Download the source code package from the assignment page on Moodle and extract it to your local hard drive. You can start the application by typing `./start.sh` on a terminal in the extracted directory. This will set all environment variables correctly and execute the file `main.py` using *Python3*. The provided code specifies the objects to be rendered in the file `Scene.py`, the viewing setup used for rendering in the file `DesktopViewingSetup.py`, and the rendering settings themselves in the file `Renderer.py`.

This assignment focuses on the specification of the objects to be rendered (file `Scene.py`) using scenegraphs. The other classes provide you with a simple desktop viewing setup including keyboard and mouse controls and basic desktop rendering settings. To complete the exercises, modify the provided source code files with respect to the given instructions, compress the directory to a .zip file, and upload it back to Moodle. Please do only insert code between the corresponding `# YOUR CODE - BEGIN` and `# YOUR CODE - END` comments in `Scene.py`. Additional code outside of the marked areas will not be considered for grading.

Transformations in 3D Space

Exercise 1.1 (4 points)

When launching the assignment code for the first time, you will see a virtual environment consisting of eight solid and eight wireframe monkeys (see Figure 1). The axes of the world coordinate system are shown by colored arrows, and a grid on the xz-plane visualizes the extent of one length unit. You can adjust the camera by pressing the W, A, S, and D keys in combination with moving the mouse.

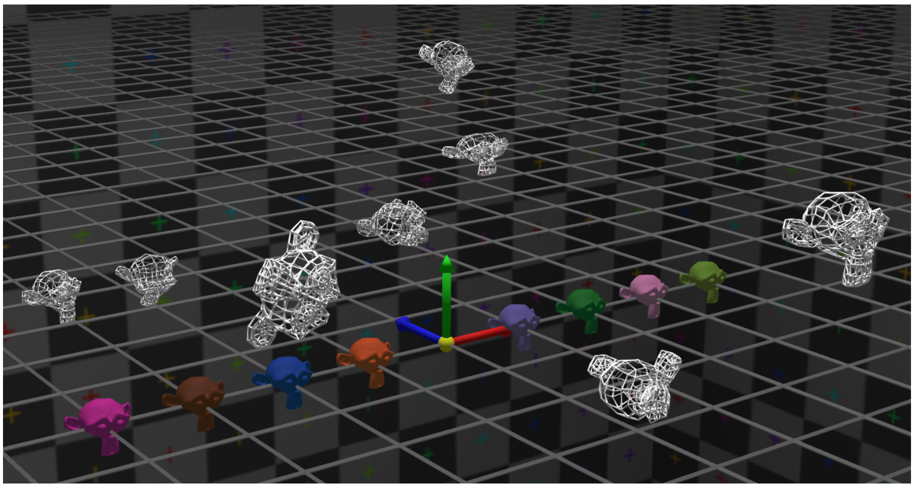

Figure 1: Screenshot of the virtual environment when launching the assignment

Your task is to set the appropriate transformation matrices for the solid monkeys such that they match the transformations of the wireframe monkeys. Write the corresponding matrices or matrix multiplications to the list `transformation_matrices` in the constructor of the class `Scene`. You should use the functions `avango.gua.make_trans_mat`, `avango.gua.make_rot_mat`, `avango.gua.make_scale_mat`, and multiplications thereof. A subtask is considered complete once a solid monkey fully matches its wireframe. Only translations between -4.0 and 4.0 in steps of 0.5, rotation angles divisible by 10, and scaling factors between 1.0 and 2.0 in steps of 0.5 are required to complete this task.

Exercise 1.2 (3 points)

In order to understand how translation, rotation, and scaling matrices are created internally, this exercise asks you to fill the member functions `make_trans_mat(self, tx, ty, tz)`, `make_rot_mat(degrees, ax_x, ax_y, ax_z)`, and `make_scale_mat(sx, sy, sz)` of the class `Scene` with the correct code. The outputs of these functions should be identical to the ones provided by the corresponding functions in the module `avango.gua`. In particular, each function should return an instance of `avango.gua.Mat4()` representing the specified transformation. You should use the function `set_element(row, column, value)` in order to fill the matrices correctly. In the case of `make_rot_mat`, it is sufficient for this exercise to focus on rotation around the cardinal axes, i.e. `(ax_x, ax_y, ax_z)` can only be equal to `(1,0,0)`, `(0,1,0)`, or `(0,0,1)`. As introduced in the lecture, pay attention to column-major representations of transformation matrices.

Once you have completed this exercise, illustrate that your functions are working correctly by reproducing your results from Exercise 1.1 using your own implementations. For this purpose, fill the list `own_transformation_matrices` in the constructor of `Scene` with the monkey transformation matrices of Exercise 1.1 but replace every call to the module `avango.gua` with a call to your previously written functions (e.g. `avango.gua.make_trans_mat(1,2,3)` becomes `self.make_trans_mat(1,2,3)`). In order to visualize the list `own_transformation_matrices` instead of the list `transformation_matrices` when running the application, adjust the call `self.load_solid_solution_monkeys(transformation_matrices)` to `self.load_solid_solution_monkeys(own_transformation_matrices)`.

Exercise 1.3 (2 points)

In order to understand how matrix multiplication works internally, this exercise asks to to fill the member function `mult_mat(self, lhs, rhs)` with the correct code such that a new instance of `avango.gua.Mat4()` representing the multiplication `lhs * rhs` is returned.

Once you have completed this exercise, illustrate that your functions are working correctly by reproducing your results from Exercise 1.1 using your own implementation. For this purpose, fill the list `own_multiplications` in the constructor of `Scene` with the monkey transformation matrices of Exercise 1.1 but replace every multiplication sign with a call to the function `self.mult_mat()` (e.g. `avango.gua.make_trans_mat(1,2,3) * avango.

gua.make_scale_mat(1,1,1) becomes self.mult_mat(avango.gua.make_trans_mat(1,2,3), avango.gua.make_scale_mat(1,1,1))). In order to visualize the list own_multiplications instead of the list transformation_matrices when running the application, adjust the call self.load_solid_solution_monkeys(transformation_matrices) to self.load_solid_solution_monkeys(own_multiplications).

Exercise 1.4 (2 points)

Representing rotations using angles around the cardinal axes (Euler angles) can suffer from ambiguities when multiple rotation matrices are multiplied to a single transformation matrix. This means that different combinations of Euler angles can lead to the same resulting transformation. As an example, you are tasked to find a pair of different angles alpha and beta such that make_rot_mat(90, 1, 0, 0) * make_rot_mat(alpha, 0, 0, 1) is equal to make_rot_mat(beta, 0, 1, 0) * make_rot_mat(90, 1, 0, 0).

To complete this task, uncomment the call to self.build_equal_rotation_task() in the constructor of the class Scene. As visualized in Figure 2, this will create two coordinate system visualizations four units above the origin of the world coordinate system. Change the variables alpha and beta in the function build_equal_rotations such that the two coordinate systems are identical.

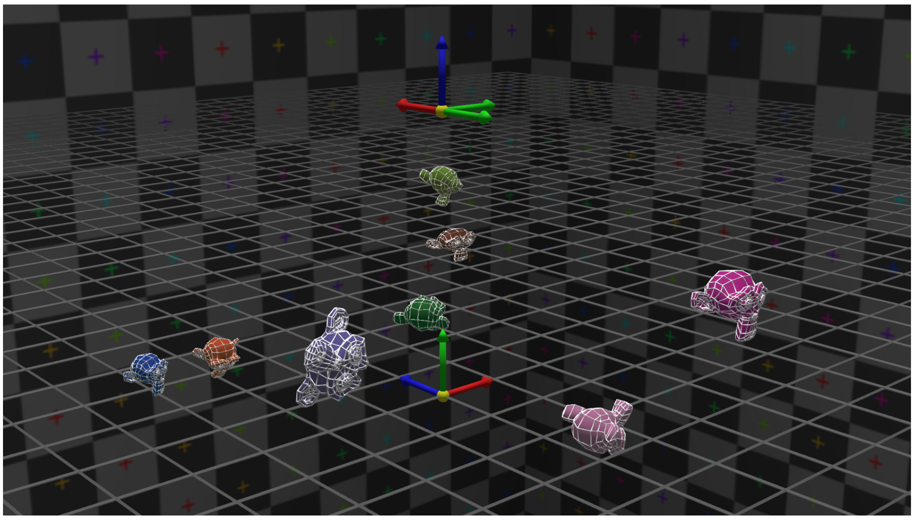

Figure 2: Screenshot of the virtual environment before completing Exercise 1.4.

Scenegraphs

Exercise 1.5 (3 points)

To place an object into the virtual environment, it needs to be attached as a geometry node to the scenegraph. The path from the root node to the geometry node defines the sequence of transformations that are applied to the respective geometry before rendering. For Exercise 1.1, all monkey geometries were directly attached to the root node such that their desired transformation had to be specified in the world coordinate system (see Figure 3).

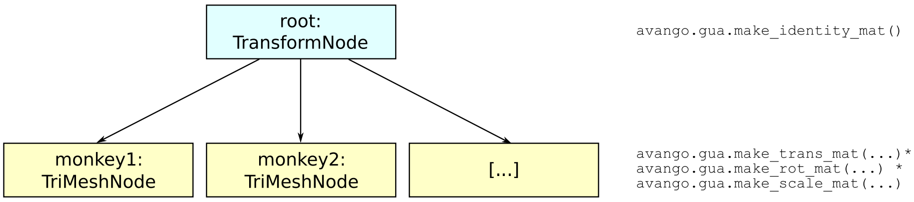

Figure 3: Scenegraph structure of the monkeys placed in Exercise 1.1.

Instances of `TriMeshNode` represent a geometry loaded from a `.obj` file while `TransformNode` instances represent a transformation without any geometry attached to it. The root node of the scenegraph is always a `TransformNode` with an identity matrix as transformation. All attributes of scenegraph nodes are encapsulated into *Fields*. A field stores its data in the `value` attribute and allows to establish dependencies to other fields. In particular, the following fields can be set:

Name The name of the node (e.g. monkey1, monkey2)

Children A list of other node instances that are the children of this transformation node

Transform The transformation matrix that represents the local coordinate system of this node (illustrated on the right of Figure 3)

Uncomment the call to `self.build_rotating_monkeys()` in the constructor of the class `Scene`. This will create two additional yellow monkeys in the virtual environment stored in the member variables `big_monkey` and `another_big_monkey`. By default, these monkeys will also be attached to the root node and filled with a transformation matrix each.

Similar to the monkeys in Exercise 1.1, the node `big_monkey` will be filled with a transformation matrix of the form $translation * rotation * scale$. As visualized in Figure 4, separate these three components into distinct scenegraph nodes by creating two additional internal instances of `avango.gua.nodes.TransformNode()`.

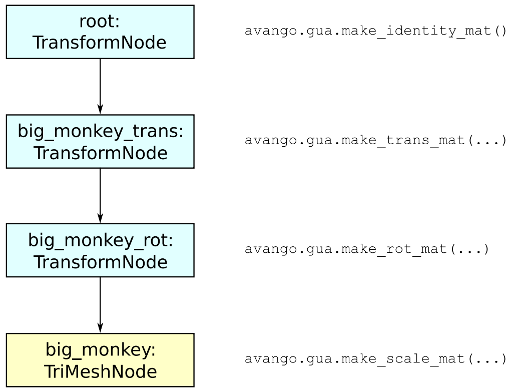

Figure 4: Scenegraph structure to be implemented for Exercise 1.5.

After the completion of this exercise, the resulting transformation of `big_monkey` in the virtual environment is unchanged since traversing the scenegraph from top to bottom multiplies the transformation matrices on the path in the same way as before.

Exercise 1.6 (1 point)

One advantage of using fields instead of raw attributes is that dependencies to other fields can be established. When defining a field connection between a source and a target field, for example, the value of the target field is automatically updated every time the value of the source field changes. A field connection can be established by writing code of the form `target_field.connect_from(source_field)`.

The class `RotationAnimator` in the file `Scene.py` provides an exemplary implementation of a field container that produces an animated rotation matrix. For this purpose, the matrix stored in the field `sf_rot_mat` is rotated by one degree around the y-axis every frame. Connect the field `sf_rot_mat` to the `Transform` field of your previously created node `big_monkey_rot` to see the animated rotation in action. The result should resemble the rotation of the left monkey illustrated in Figure 5.

Exercise 1.7 (3 points)

Use the knowledge gained in the previous two exercises to build a scenegraph structure for an animated rotation of the node `another_big_monkey` around the origin with a radius of 8 units and a height of 1.5 units. The result should resemble the rotation of the right monkey illustrated in Figure 5.

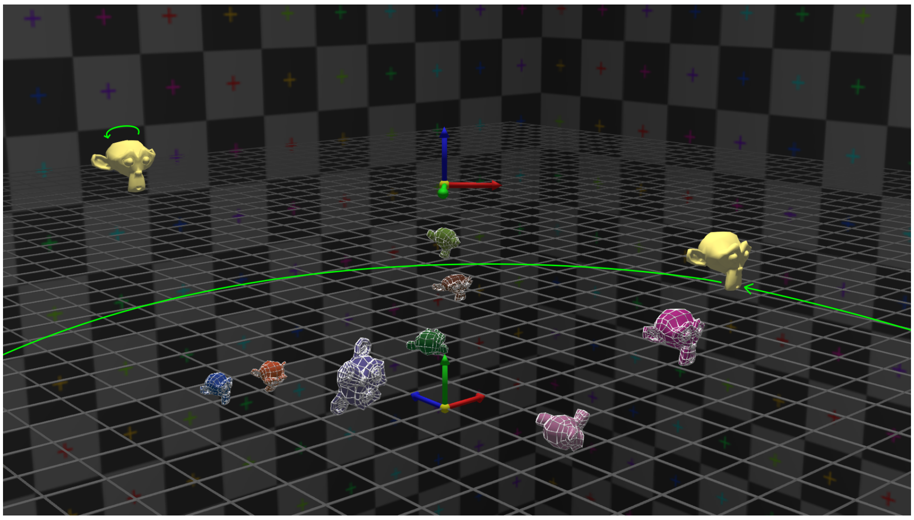

Figure 5: Screenshot illustrating the rotation animations to be implemented in Exercises 1.6 and 1.7.

Exercise 1.8 (3 points)

The previous exercises illustrate that the local and world transformation matrices of a node can be different. The local transformation matrix expresses the node's transformation in the coordinate system of the parent node while the world transformation matrix is given by the path towards the node in the scenegraph. While the world transformation matrix of a node is computed and stored in the read-only field `WorldTransform` for convenience, your task in this exercise is to implement an own computation of the world transformation matrix.

Uncomment the final two lines of the constructor of the class `Scene`. This will create an instance of the field container `WorldTransformComputer` defined at the bottom of the file `Scene.py`. The only input field `sf_node` stores the node for which the world transformation should be computed and is set to reference `another_big_monkey`. The function `evaluate`, which is called every frame, compares the value of the field `WorldTransform` with the output of the function `compute_world_transform(self, node)` that you are about to implement.

Implement the function `compute_world_transform(self, node)` such that it returns the correct world transformation matrix of the given node. For your implementation, you can use the read-only field `Parent` defined on each scenegraph node, which is automatically filled and updated correctly every frame.

Exercise 1.9 (1 point)

Use the functions `get_translate()`, `get_rotate()`, and `get_scale()` to decompose the computed world transformation matrix in the function `evaluate` of `WorldTransformComputer` and print the results to the console. Be prepared to explain what the outputs mean geometrically.