Virtual Reality
Winter Term 2018/2019

Assignment Sheet 3

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 **22 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 **23 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 **10%** for your total lab class grade.

In this assignment, you will implement a crane consisting of multiple hierarchical levels (hinges and arms). Movements applied to one level of the crane implicitly move all corresponding sub-levels. As an example, rotating the base hinge turns the whole crane, whereas input applied to the last hinge only affects the last arm segment. You will use different input evaluation policies and simple dataflow programming using dependency graphs. The video `demo_video.mp4` in the repository illustrates the intended final result. Basic information on datflow programming with Avango/Guacamole can be found in the lecture on **Graphics Basics and Scenegraphs**. Further details on event handling will be given in the lab class on **16 November 2018**.

Figure 1: A multi-level crane, consisting of three segments. Keyboard input is used to rotate the individual hinges (red discs), and all sub-structures move accordingly.

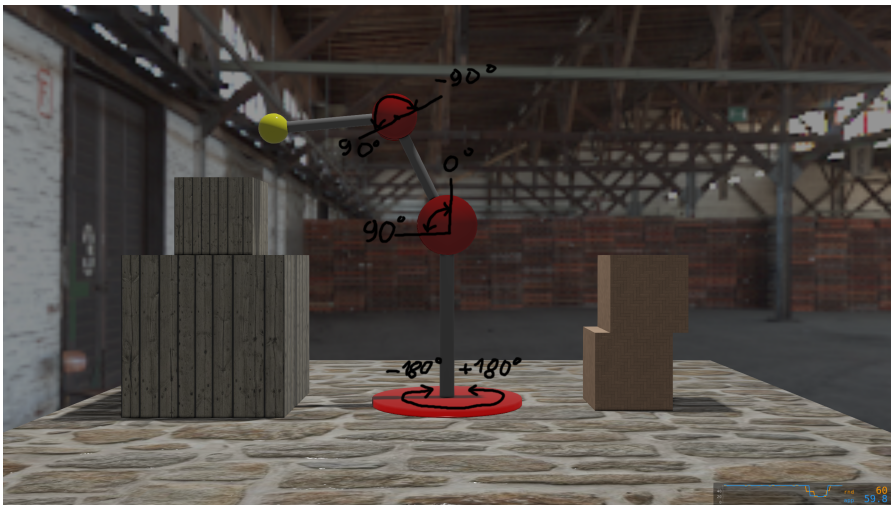

Figure 2: Rotation axes and rotation constraints of each hinge.

Figure 3: The hook at the final level of the crane (yellow sphere) checks for intersections with the boxes. If the hook is inside a geometry, the respective box is colored red.

Exercise 3.1 (3 points)

Set up a multi-level crane as shown in Figure 1. The crane should consist of three levels, which includes three hinges (red discs) and three arm-segments (grey cylinders). The classes `Hinge` and `Arm` are already given and contain internal scenegraph nodes for the representation of this structure. In the class `Crane`, the first segments of the crane are already initialized. Initialize and parametrize the remaining `Hinge` and `Arm` segments there. Try to proceed stepwise with one segment after the other to validate your results.

Exercise 3.2 (3 points)

The `KeyboardInput` class provides three rotation input values as field variables (`sf_rot_input0`, `sf_rot_input1`, `sf_rot_input2`). The left/right, up/down and page-up/page-down keys are used to define these rotations. Once these buttons are pressed, the respective hinge should rotate continuously with a velocity of 45°/sec until the key is released. Calculate the rotation input per frame accordingly and write it to the respective fields.

The class `Hinge` contains an instance of the class `Accumulator` as a member. Forward the rotation input fields to the internal `Accumulator` class of each hinge using a field connection. Then, accumulate the connected input in the `evaluate` callback of the `Accumulator` class to the given transformation matrix. Finally, connect the accumulated transformation matrix to the matrix of scenegraph node representing the hinge.

Exercise 3.3 (3 points)

Include rotation constraints for the hinges to clamp the rotation input accumulation to the minimum and maximum values given in Figure 2. For this purpose, create an instance of the given class `Constraint` in the `Hinge` class and modify the field connections accordingly to integrate this new stage into the data processing chain of the hinge. The constraint values can be stated as constructor parameters into the instances of `Hinge` and then forwarded into the `Constraint` class. The `evaluate` callback in `Constraint` provides code for the calculation of Euler angles that can be used to clamp the accumulated rotation values to the minimum and maximum values. The modifications of the transformation matrix in the `Constraint` is relevant for the subsequent execution of the code in the `Accumulator` class. Thus, a cyclic dependency occurs and needs to be handled carefully.

Exercise 3.4 (2 points)

At the final stage of the crane, a hook is attached (yellow sphere in Figure 3). The `Hook` class is already given. It provides a callback function that checks if the position of the hook is inside of any of the target boxes defined in the `Scene` class. If so, the intersected boxes are recolored. Initialize the necessary node(s) and connect the final hook transformation matrix to the callback field variable in order to provide the necessary data for the intersection check.

Exercise 3.5 (1 point)

Implement a frame rate-independent input mapping for all hinges. For this purpose, the rotation input values in the class `KeyboardInput` have to be adjusted with respect to the frame rate of the application. Pressing the CTRL key toggles the application frame rate between 20Hz and 60Hz for testing. The crane should move continuously with the same velocity independent of different frame rates.

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)`
`@field_has_changed(sf_value) def sf_value_changed(self):`	callback function that is immediately triggered if the corresponding field value is set
`self.sf_value1.connect_from(sf_value2)`	creates a field connection between fields `sf_value1` and `sf_value2`
`self.sf_value1.connect_weak_from(sf_value2)`	creates a weak field connection between fields `sf_value1` and `sf_value2`