Virtual Reality
Winter Term 2019/2020

Assignment 2 - Desktop Viewing Setups

Virtual cameras and projection screens define the visible parts of the scene to be included in the rendering result. This assignment focuses on the single-user desktop case, in which the viewing setup consists of only a single camera and a single screen node. You will study the relation between these nodes, the role of their paths in the scenegraph, and the implementation of basic camera controls using the keyboard as an input device.

Group work in pairs of two is encouraged. You are required to submit this assignment by **14 November 2019, 11:55 pm** on Moodle. Furthermore, you will be asked to present and discuss your results in the lab class on **15 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 **20 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 virtual environment of this assignment contains the model of an island and a bird flying above it (see Figure 1). As in the previous assignment, all objects to be rendered are specified in the file `Scene.py`. The viewing setup (defined in the file `DesktopViewingSetup.py`) consists of a camera with a fixed static transformation and a screen plane positioned at a distance of 0.6m in front of it. The scenegraph of the application is visualized in Figure 2.

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. Additional code outside of the marked areas will not be considered for grading.

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

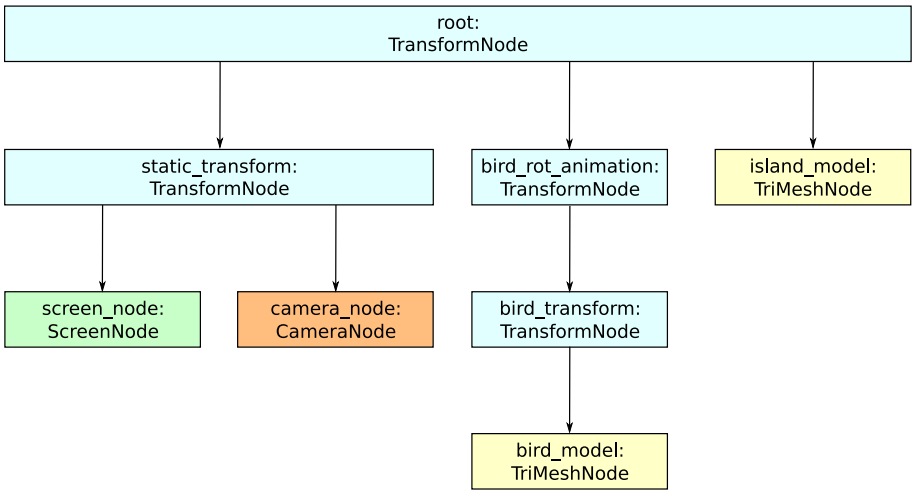

Figure 2: Initial scenegraph structure of the assignment. The left subtree of the root node is specified in the file `DesktopViewingSetup.py` while the right two subtrees are created in the file `Scene.py`.

Exercise 2.0 (no grading)

Start the application and measure the size of the window on your desktop monitor using a ruler or measuring tape. Fill the constant `SCREEN_SIZE` at the top of `DesktopViewingSetup.py` with the width and height of the window in meters. Repeat this process when switching to another monitor type.

Exercise 2.1 (2 points)

The viewing frustum is defined by the transformations of the virtual camera and the screen plane to project the virtual environment on. In the desktop case, the camera is placed at a fixed distance in front of the center of the virtual projection screen. The distance between the camera and the screen is directly related to the camera's angular field-of-view (see Figure 3).

Implement the function `compute_fov_in_deg()` in the class `DesktopViewingSetup` such that it reads the distance of the camera to the screen and converts it to the corresponding horizontal field-of-view in degrees. Initially, the distance between camera and screen is 0.6m. You can change the distance in the screen node to experiment with different values. The console prints defined in the end of the constructor of the class `DesktopViewingSetup` can help you to check the outputs of your function.

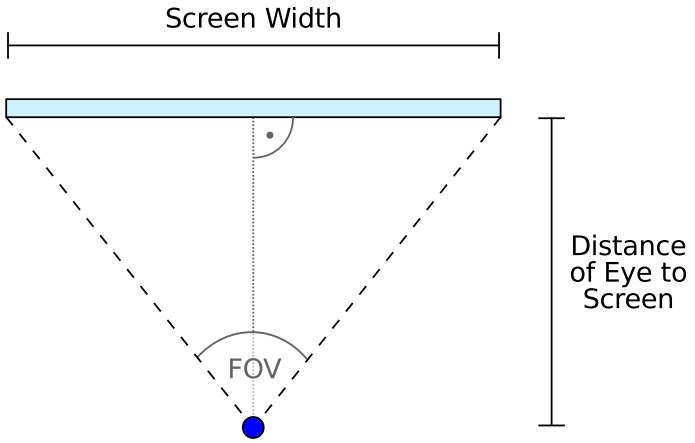

Figure 3: Schematic illustration of the relation between the virtual camera, the virtual screen plane, and the resulting field-of-view (FOV)

Exercise 2.2 (2 points)

Implement the function `set_fov_in_deg()` in the class `DesktopViewingSetup` such that it computes the necessary camera-screen distance for a given field-of-view in degrees and sets it on the screen node. In the end of the constructor of the class `DesktopViewingSetup`, the field-of-view is set to 45 degrees, but you can change this value to experiment with different field-of-view settings.

Exercise 2.3 (1 point)

Above the virtual island, a bird flies in circles using the same `RotationAnimator` class as in the previous assignment (defined in the file `Scene.py`). Attach the screen and camera nodes to the node `bird_transform` instead of `static_transform` such that they move along with the bird. In the class `DesktopViewingSetup`, you can use the bracket operator `[]` on the scenegraph instance to retrieve any node by its path: `node = self.scenegraph["/node_name_1/node_name_2/..."]`.

As visualized in Figure 4, the result of this exercise will not yet be appealing. Because camera and screen are now children of `bird_transform`, they are now inside of the flying bird model. You are going to fix this in the next exercise.

Exercise 2.4 (2 points)

To have a clear view onto the surroundings, an additional mechanism should allow to toggle the visibility of the bird model. Every scenegraph node has a field `Tags` that contains a list of strings that can be used to assign labels to nodes. Moreover, the camera node has a field `BlackList` that defines a list of tags, which is used to exclude objects from rendering. For this assignment, we also prepared the function `sf_visiblity_toggle_changed()` that is called every time the left `Alt` key on the keyboard is pressed.

Using these prerequisites, implement a mechanism that can toggle the visibility of the bird model. On the first press of `Alt`, the bird model should become invisible. On the second press of `Alt`, the bird model should become visible again. After the successful completion of this exercise, you can enjoy an unoccluded view on the environment while flying in circles (see Figure 5).

Figure 4: When attaching camera and screen directly to the node `bird_transform`, the result will not be appealing.

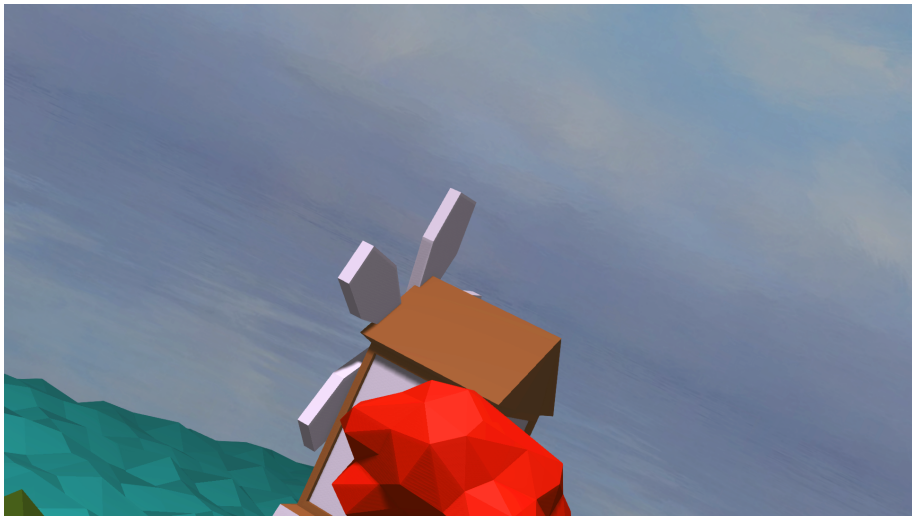

Figure 5: When the bird's geometry is hidden, you can enjoy flying in circles above the island.

Exercise 2.5 (2 points)

Modify your viewing setup such that the camera is always placed five units behind the bird instead of inside the bird. The result should resemble the one illustrated in Figure 6.

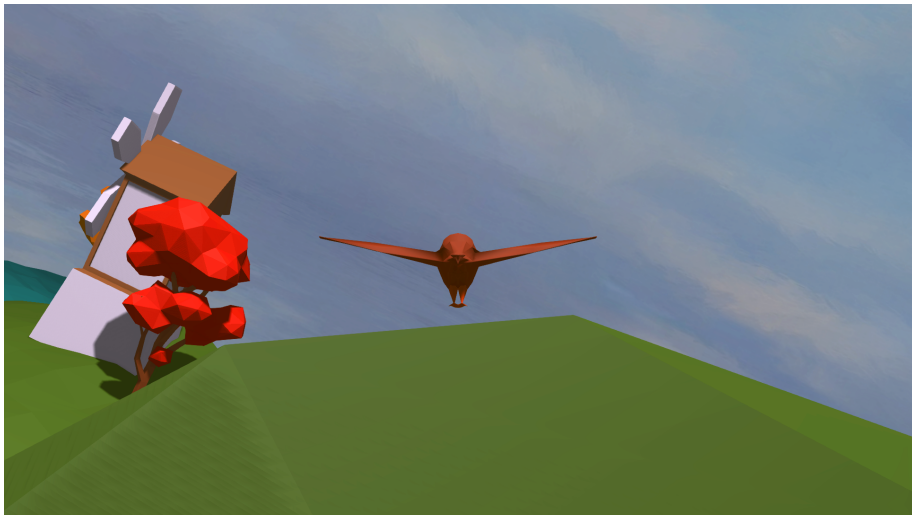

Figure 6: When placing the camera five units behind the bird, visibility toggling mechanisms are not needed anymore.

Exercise 2.6 (2 points)

The fields `sf_left_arrow_key` and `sf_right_arrow_key` in the class `Desktop ViewingSetup` are prepared to contain `True` when the left and right arrow keys are pressed, respectively. Moreover, the function `evaluate` is called every frame.

Modify your viewing setup such that you can use the left and right arrow keys to rotate the camera around itself while still following the bird. When rotating left, for example, the result should resemble the one illustrated in Figure 7 (note the left wing of the bird on the right side of the picture).

Exercise 2.7 (4 points)

Investigate how button inputs are created and processed in the application and integrate the usage of the up (`EV_KEY::KEY_UP`) and down (`EV_KEY::KEY_DOWN`)

Figure 7: When rotating the camera around itself while still following the bird, you have more freedom in the observation of the virtual island.

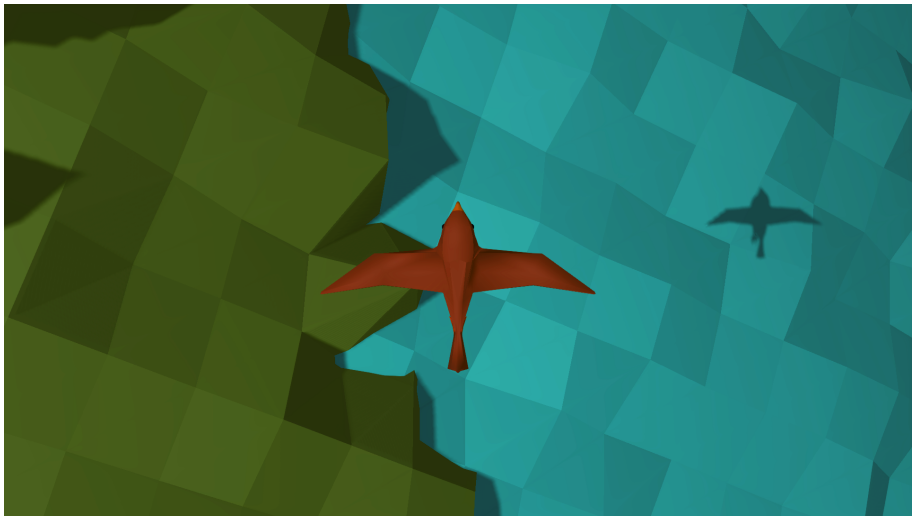

Figure 8: When orbiting the camera around the bird, you can adjust your perspective on the scene even more freely.

arrow keys. Start in the file `daemon.py`, where all input devices and their events are registered, and continue in the file `DesktopViewingSetup` to see how the values are retrieved from the daemon.

Modify your viewing setup such that you can use the up and down arrow keys to orbit the camera around the bird while still following the bird. When rotating upwards, for example, the result should resemble the one illustrated in Figure 8. Make sure that your solution approach is compatible with the previous exercise, i.e. both orbiting around the bird and rotating the camera around itself should be supported. For this purpose, think of an appropriate node structure in the scenegraph and use different nodes for the different types of rotation.

Exercise 2.8 (3 points)

Up to this point, the class `RotationAnimator` in `Scene.py` (and maybe also your arrow key mappings) are dependent on the frame-rate of the application. The rotation animator, for example, is implemented in a way that 0.1 degrees are added to the animated rotation matrix every frame. If the frame rate drops, however, the animation will be slower, which is highly undesired. In order to experience this effect, press the right `Ctrl` key on your keyboard to switch between a slow and a fast framerate.

Modify the frame-wise update in the function `evaluate` of the class `Rotation Animator` such that it is frame-rate independent. For this purpose, use Python's module `time` to measure the elapsed time between frames and adjust the matrix multiplications such that the animated matrix is updated with the rotation speed defined in the field `sf_rotation_speed` (unit: degrees per second). Adapt the same strategy to make your arrow key mappings frame-rate independent. After the successful completion of this exercise, every movement should have the same speed in both the normal and the slow frame-rate mode.

Exercise 2.9 (2 points)

Implement the function `compute_model_view_transform(self, node)` in the class `DesktopViewingSetup` such that it computes and returns the model-view matrix of a node with respect to the screen node. The constructor of the class `DesktopViewingSetup` calls this function for the bird model node in the first frame, which you can use to check your implementation. Be prepared to explain the meaning and the components of the model-view matrix.