Virtual Reality
Winter Term 2019/2020

Assignment 4 - Application Example

In this assignment, you will use the concepts learned throughout the previous three assignments in the realization of a small game involving simple navigation and object selection capabilities. You will demonstrate that you are familiar with the hierarchical organization of scene content, transformations in 3D space, camera controls, input devices, and transfer functions. Moreover, you can earn bonus points for implementing additional functionalities in a head-mounted display environment.

Group work in pairs of two is encouraged. You are required to submit this assignment by **19 December 2019, 11:55 pm** on Moodle. Furthermore, you will be asked to present and discuss your results in the lab class on **20 December 2019**. If these dates are too close to the Christmas break for you, the submission and presentation of the assignment can also be completed a week earlier. In this case, the submission deadline is 12 December 2019, 11:55 pm for a presentation on 13 December 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** (**24 points or 120% including bonus tasks**) 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*. 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. This time, you may add code at any position within the application folder.

Scenario

Your task is to implement a small game, in which the user navigates a figure to collect a set of items distributed across the virtual environment. The figure can be controlled using the inputs of a space navigator, and it should automatically follow the irregularly shaped ground of the scene. Moreover, the game should compute collisions with obstacles and prevent the figure from moving through them. An exemplary screenshot of the completed game is shown in Figure 1, but you are free to vary the aesthetics as long as the exercises are fulfilled correctly. You can test our reference implementation in the lab class.

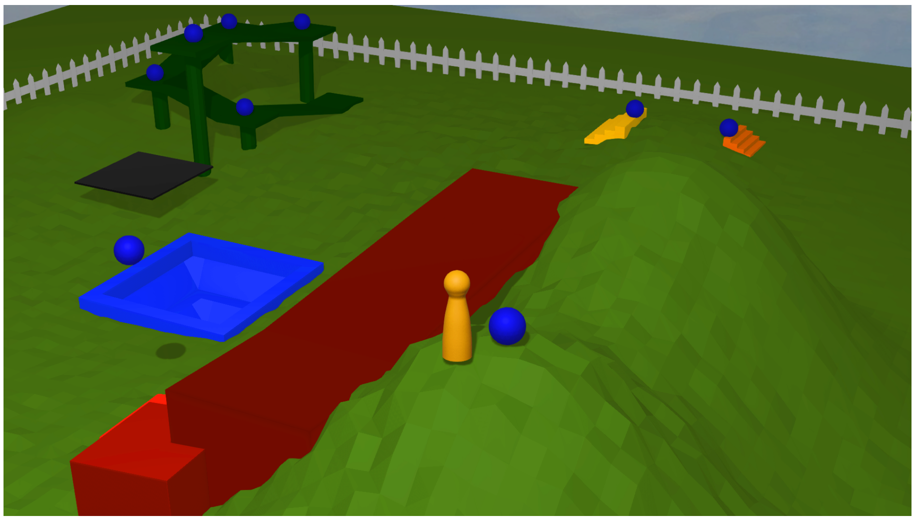

Figure 1: The yellow figure moves around the scene to collect blue spheres.

Regular Tasks (20 points)

Exercise 4.1 (3 points)

When launching the application for the first time, you can observe the virtual environment from the first-person view of the player figure. Implement the function `evaluate` of the class `NavigationControls` such that the figure can be navigated in the xz-plane of the virtual environment using a rate-control transfer function. The inputs of the space navigator are already connected to the respective fields of the class. Moreover, the user should be able to rotate the figure around the y-axis with a rate-control transfer function. Pay attention that rotating the camera also changes the forward direction in which inputs should be applied. This means that moving the space navigator forward should always result in a forward movement of the figure, too. Write your result to the field `sf_output_matrix`, which is already connected to the figure in the class `DesktopViewingSetup`. After this exercise, the figure will not yet interact with the scene geometry.

Exercise 4.2 (3 points)

Implement a third-person camera that automatically follows the figure at a fixed distance in the class `DesktopViewingSetup` (see Figure 1). The camera should always stay behind the figure. To control the viewing angle of the camera onto the figure, map the space navigator's x-rotation to an x-rotation of the camera around the figure using a position-control transfer function. Figures 2 and 3 provide examples of how this mapping should look like in the application. When no x-rotation is provided by the user, the camera should snap back to its default orientation.

Exercise 4.3 (5 points)

Import the class `Picker`, which provides a simple implementation of a ray-scene intersection mechanism. The function `compute_pick_result(pos, direction, length, blacklist)` casts a ray of the given length (`length`) from the given position (`pos`) in a given direction (`direction`). It returns an instance of `avango.gua.nodes.PickResult()` if an intersection was found or `None` if not. The `blacklist` is a list of strings that exclude objects with the respective tags from the picking operation.

The origin of the figure model is at the center of its head, which should always be 2.0 units above the ground. Implement a simple ground following mechanism

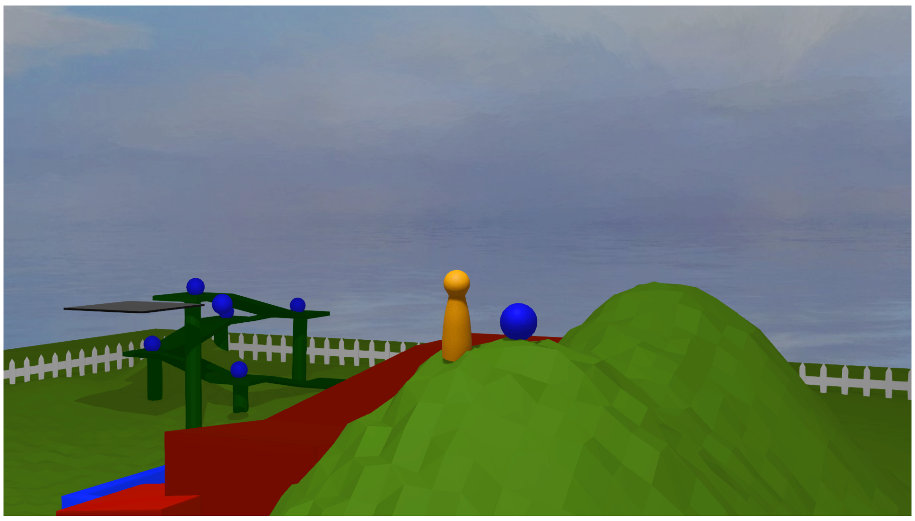

Figure 2: Rotating the space navigator downwards should result in a flatter viewing angle onto the figure.

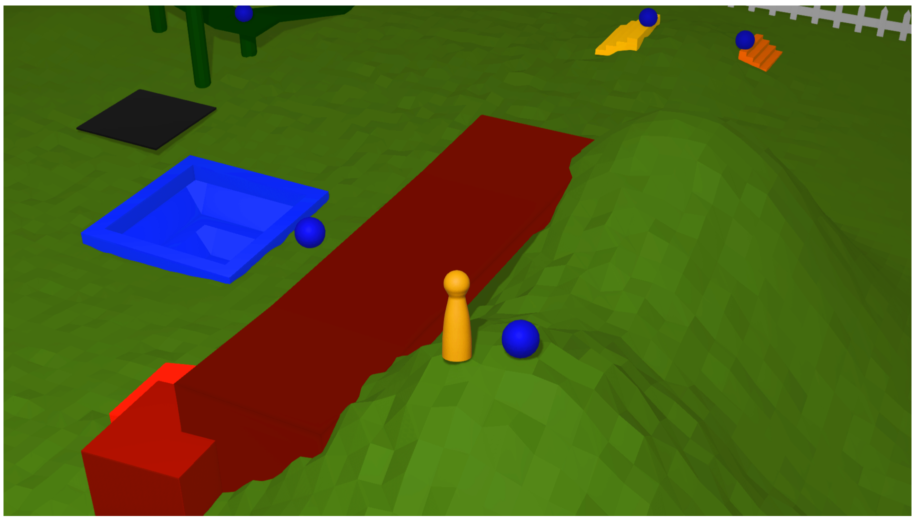

Figure 3: Rotating the space navigator upwards should result in a steeper viewing angle onto the figure.

that satisfies this criterion and therefore makes the figure automatically adjust height when walking around the irregularly shaped ground. For this purpose, ...:

- ... cast a vertical downward ray from the figure's head to determine its height above ground.
- ... if the distance to the intersection is smaller than the intended figure's height, move the figure upwards.
- ... if the distance to the intersection is larger than the intended figure's height, move the figure downwards.
- ... avoid too abrupt changes to the figure's height by adding only small values every frame; for moving upwards, this can be a constant, but when falling downwards, this value should increase with the duration of the fall.

The figure should now be able to climb hills and slopes and to fall down when jumping off ledges.

Exercise 4.4 (4 points)

While your implementation of ground following ensures that the figure is always at the correct height, it doesn't avoid collisions with obstacles on the same height level. Implement a simple collision mechanism that prevents the figure from moving through objects. For this purpose, ...:

- ... cast an intersection ray from the figure's current position to the figure's intended position in the next frame.
- ... determine if there is a collision between both positions.
- ... block or redirect the inputs to avoid moving through potential obstacles.

Exercise 4.5 (3 points)

Distribute some objects to be collected in the virtual environment. You can use spheres as in Figure 1 or add your own objects to the `data/objects` directory. Implement a mechanism that makes these objects disappear when they are hit by the figure. To increase the difficulty for the player, make at least one object move around using a pre-defined animation.

Exercise 4.6 (2 points)

Implement an effect that is triggered when all objects are collected. You could, for example, show the task completion time on the terminal or realize some flashing scene lights.

Bonus Tasks (4 bonus points)

Use one of the designated Windows machines in our lab to launch the application in a head-mounted display. We will offer an introduction on how to boot the machines to Windows, how to log in, and how to prepare the required hard- and software in the lab class on **6 December 2019**. Enter the IP address of your machine in the file `config.py` and launch the application by executing the script `start.bat`. To kill all application terminal windows at once, you can execute the script `kill.bat`.

When starting the application for the first time, you can observe the scene in first-person mode. Your head movements are tracked, so you can freely look and walk around in the available tracking area. An additional tracked controller serves as your input device (see Figure 4).

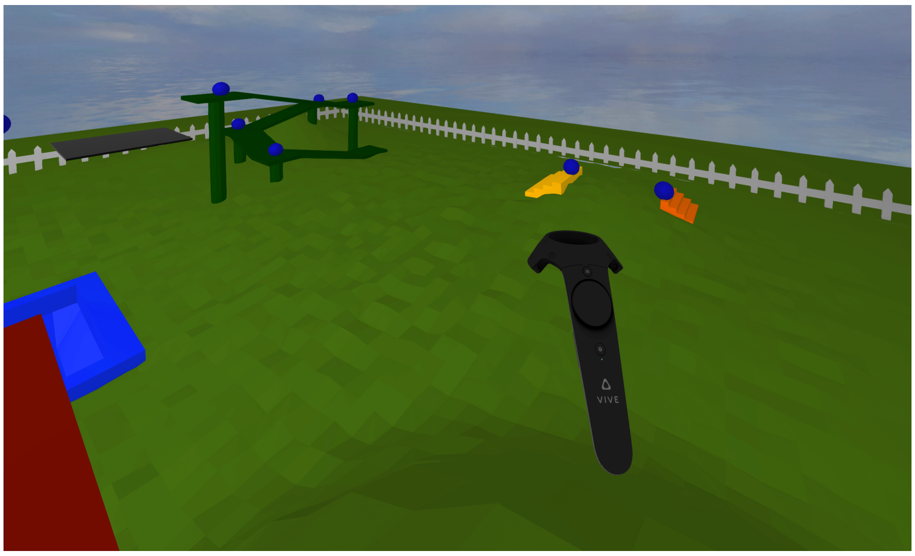

Figure 4: When launching the application in a head-mounted display, you will explore the virtual environment in an immersive first-person mode.

Exercise 4.7 (1 bonus point)

Implement the function `evaluate` of the class `ViveNavigationControls` such that you can navigate through the scene using pointer-directed steering. This means that pressing the rocker button on the controller should result in a 3D displacement of the user in the direction indicated by the pointer. The scenegraph node referenced by `navigation_node` contains the transformation of your tracked workspace in the scene, and the nodes referenced by `head_node` and `controller1_node` represent the tracked head transformation of the user and the tracked pointer transformation within the workspace, respectively.

Exercise 4.8 (2 bonus points)

Apply your previously written functions for ground following and collision detection to the user. Pay attention that while you want to compute all intersection rays starting from the user's head position, you can only apply changes to the transformation matrix of the navigation node. The head node is filled with tracking data every frame, so changes in its matrix would be overwritten immediately.

Exercise 4.9 (1 bonus point)

Adjust your implementation such that you can switch to gaze-directed steering instead of pointer-directed steering. In this mode, the forward direction of movement is specified by the viewing orientation of the user.