Virtual Reality
Winter Term 2022/2023

Assignment 1 - Build Your World

This assignment will introduce you to the basics of the Unity 3D game engine. You will learn how to use the Unity Editor and write scripted behavior to manage GameObjects at application runtime. Furthermore, you will use our `VRSYS-Core` Unity plugin for social VR development. It allows you to access your scene both from a head-mounted display (HMD) as well as from a desktop device without any coding overhead. This will allow you to meet each other in a shared virtual environment. To extend the features we provide, you will compute transformation updates and synchronize them between all clients in your distributed application.

As the title of this document states, the wider context of this sheet is to create a representative virtual scene to be used in upcoming assignments. Additionally, you will have had some initial experience with VR device integration, which should ease the process of joining our VR lecture in VR.

While artistic efforts are not graded, giving your environment a personalized flavor is always encouraged.

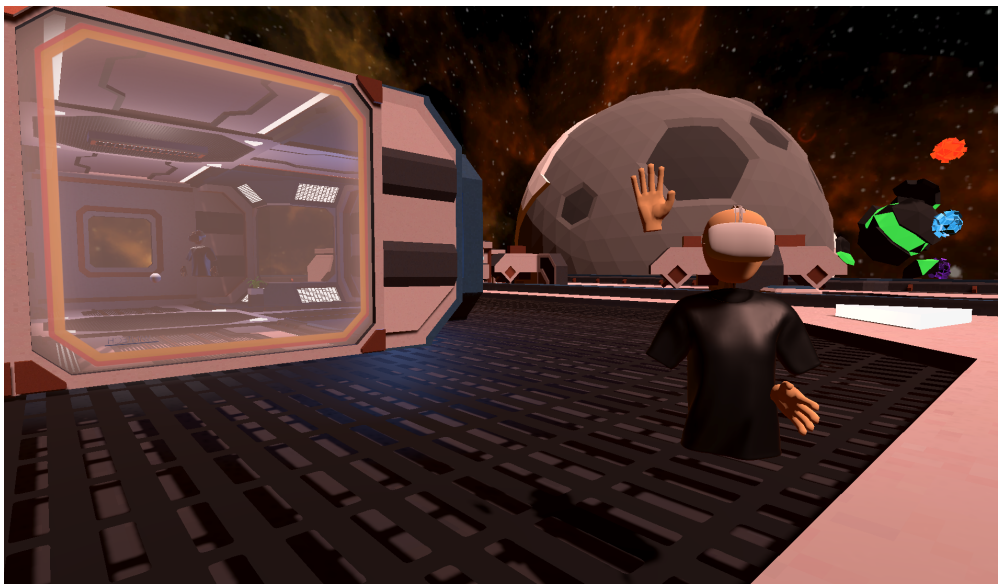

Figure 1: A snapshot from the completed assignment Unity project.

Submission and General Remarks

You are required to submit this assignment by **9th Nov 2023 23:55** on Moodle. Furthermore, you will be asked to present and discuss your results in the lab class on **10th of Nov 2023** to one of the teaching assistants. Your submission should consist of one `.zip`-archive named `<your-group-name>.zip`. The archive should contain a clean Unity project with your solutions for the tasks explained in this assignment sheet. Make sure to clean out all temporary files before zipping. Relevant folders are `Assets, Packages, ProjectSettings, UserSettings` (see Figure 2).

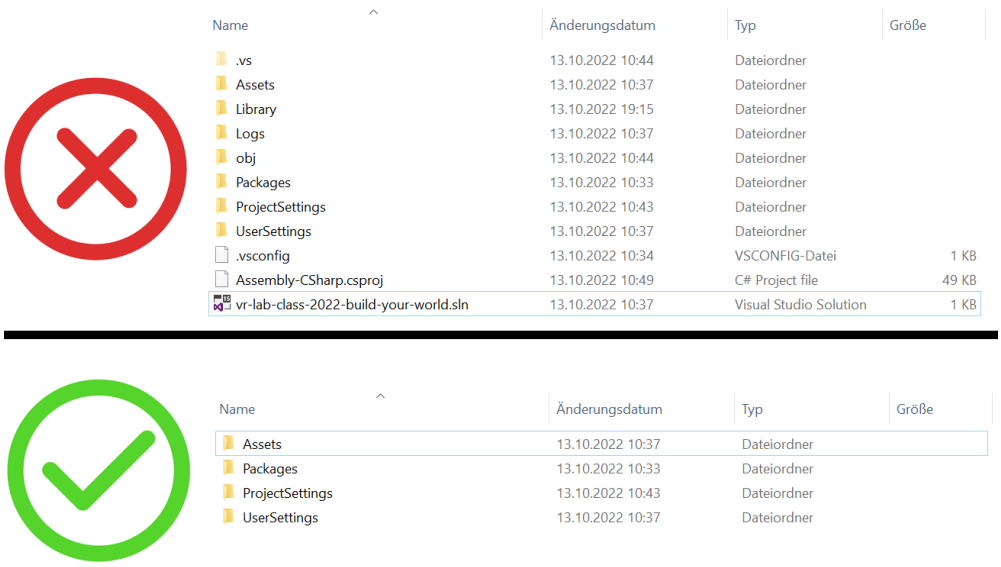

Figure 2: Top: A Unity project after generating temporary files (not ready to be zipped as submission). Bottom: A clean Unity project before temporary files are generated (ready to be zipped as submission).

Over the course of all assignments in this class you will be guided through a series of tasks that will teach you how to build social VR experiences. Due to the vast scope of challenges involved when designing and implementing multi-user VR applications, we will only focus on a subset of techniques to be implemented and environments to interact with. We want to encourage you to expand your journey into VR and explore beyond the tasks that we provide. At all times, feel free to reach out for our feedback and guidance, so that we may help to expand your skills!

Please note: If not mentioned otherwise, Unity Asset Packages supplied to you as part of the assignments are strictly for course use only and they may not be shared

or reused in projects unrelated to the VR Lecture. Using Asset Store packages for any of the coding related tasks is not allowed and will lead to a 0% assessment for the affected tasks.

Assessment

Each task in this assignment shows a percentage in brackets. The sum of task percentages add up to 100 (exluding bonus tasks) within each assignment. Consequently, you may treat task percentages both as a maximum score per task as well as an assignment progress indicator. The contribution of each assignment towards your overall lab class grade depends on its weight, which are as follows:

- **Assignment 1: 20%**
- Assignment 2: 20%
- Assignment 3: 20%
- Assignment 4: 40%

Unity

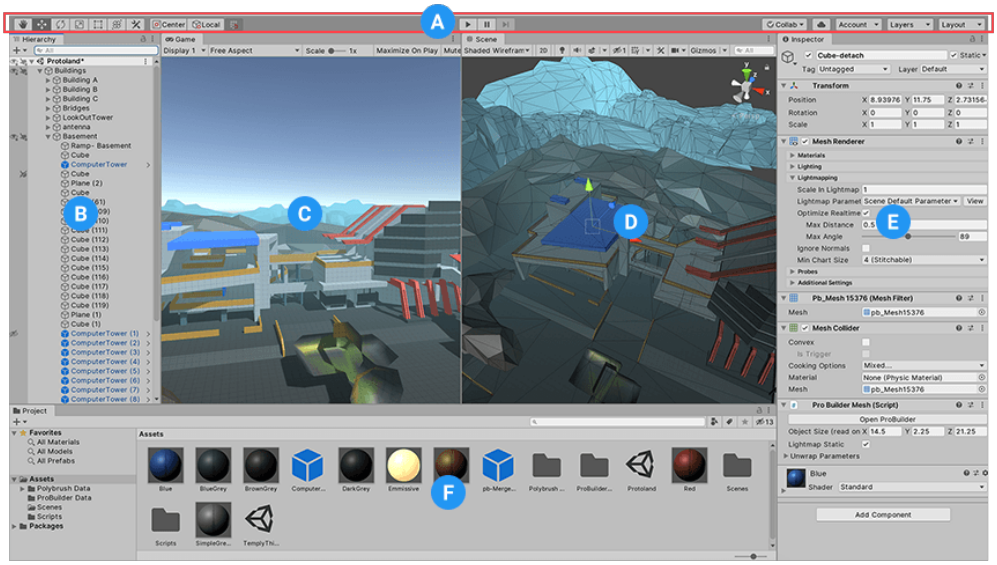

Figure 3: Unity Interface: Toolbar (A), Hierarchy (B), Game View (C), Scene View (D), Inspector (E) and Project View (F). More on https://docs.unity3d.com/Manual/UsingTheEditor.html

Throughout the lab classes, we are going to use *C#* to write applications in *Unity*. The Unity Editor's interface has multiple windows (see Figure 3). The *Toolbar* (A) gives you access to essential features of Unity and allows you to start, stop and pause your application, the *Hierarchy Window* (B) shows the scenegraph of your application in text form, the *Game View Window* (C) renders the actual view of your application's main camera, the *Scene View Window* (D) provides a graphical view of your scene and allows visual editing and navigation, the *Inspector Window* (E) displays components and properties of the currently selected GameObject and the *Project Window* (F) shows the file structure and files of your unity project.

To place an object into the virtual environment, it needs to be attached as a *GameObject* to the scene in Unity. The path from the root of the scene to the *GameObject* defines the sequence of transformations that are applied to the respective object before rendering. In this assignment, you will implement basic algorithms to deal with spatial transformation and simple hierarchical structures (scenegraphs), which are essential for writing higher-level applications.

VRSYS-Core

Figure 4: Snapshot from a social VR application (see Goethe-Live-3D https://uni-weimar.de/goethe-live-3d), which was built using VRSYS-Core.

The development of social VR applications relies heavily on custom code that runs across the WWW. `VRSYS-Core` wraps the core networking setup procedure to connect and manage your application using Unity *Netcode for GameObjects* (see https://docs-multiplayer.unity3d.com/netcode/current/about/). We realize that the conceptual and practical principles of network programming could easily fill the scope of a standalone university course, so we can merely teach you the basics that will kick-start your journey into social VR. Over the coming months we expect to release our plugin on an OpenSource license, so you can feel free to use it even outside of the course.

During the lab class, we will incrementally roll out relevant parts of `VRSYS-Core` and provide additional guidance to familiarize you with the necessary knowledge about distributed programming. But you will likely also need to cross-read into related (Unity and Netcode) documentation and look into tutorials self-sufficiently. Our foremost recommendation is to start working on the assignments early and raise your questions to us, so we can support your individual progress effectively.

Here is one comprehensive tutorial that addresses the most important concepts of multiplayer programming using *Netcode for GameObjects*: https://www.youtube.com/watch?v=swIM2z6Foxk *(please treat this as complementary material, we don't expect that you follow any or all of it)*

Getting Started

Download and extract `vrsys-lab-assignments-2023.zip`. It contains the Unity project, which you will use to complete the tasks in this assignment. The project was built and tested using Unity Editor 2021.3.26f.

You will also receive a *Meta Quest HMD* (Quest 2, Quest Pro or Quest 3 - depending on state of inventory) to complete the assignments and attend (some of) our VR lectures in VR. For the assignments you will need to have the *Oculus desktop app* installed. The borrowed HMDs come with our group account setup as a default user. We would like to encourage you to explore content in-the-wild, so you may feel free to do a factory reset, register a personal account (requires *Meta Quest mobile app*) and install any apps that you are curious about.

We encourage you to use git as a version control system for your lab assignments (see https://git-scm.com/downloads). For convenience, we supply a `.gitignore`-file within the archive mentioned above. It is setup to ignore all temporary files generated by Unity, when versioning your progress.

If you work from the VR lab, the aforementioned github project, Unity version and Oculus desktop app will already be available on your account. If you work from home, please install *Unity HUB* (see https://unity3d.com/get-unity/download) and add Editor version `2021.3.26f` (see https://unity3d.com/get-unity/download/archive).

You can now start your assignment by opening the unzipped `vrsys-lab-assignments-2023` folder from the Projects-tab within Unity HUB. Also, make sure your HMD is connected to the computer via USB-C. Test the connection and enter *Quest Link* mode (see Figure 5 or follow https://www.meta.com/help/quest/articles/headsets-and-accessories/oculus-link/connect-link-with-quest-2/).

We are happy to provide guidance with any of the steps involved. Make sure to watch the video introduction linked in the caption of Figure 6.

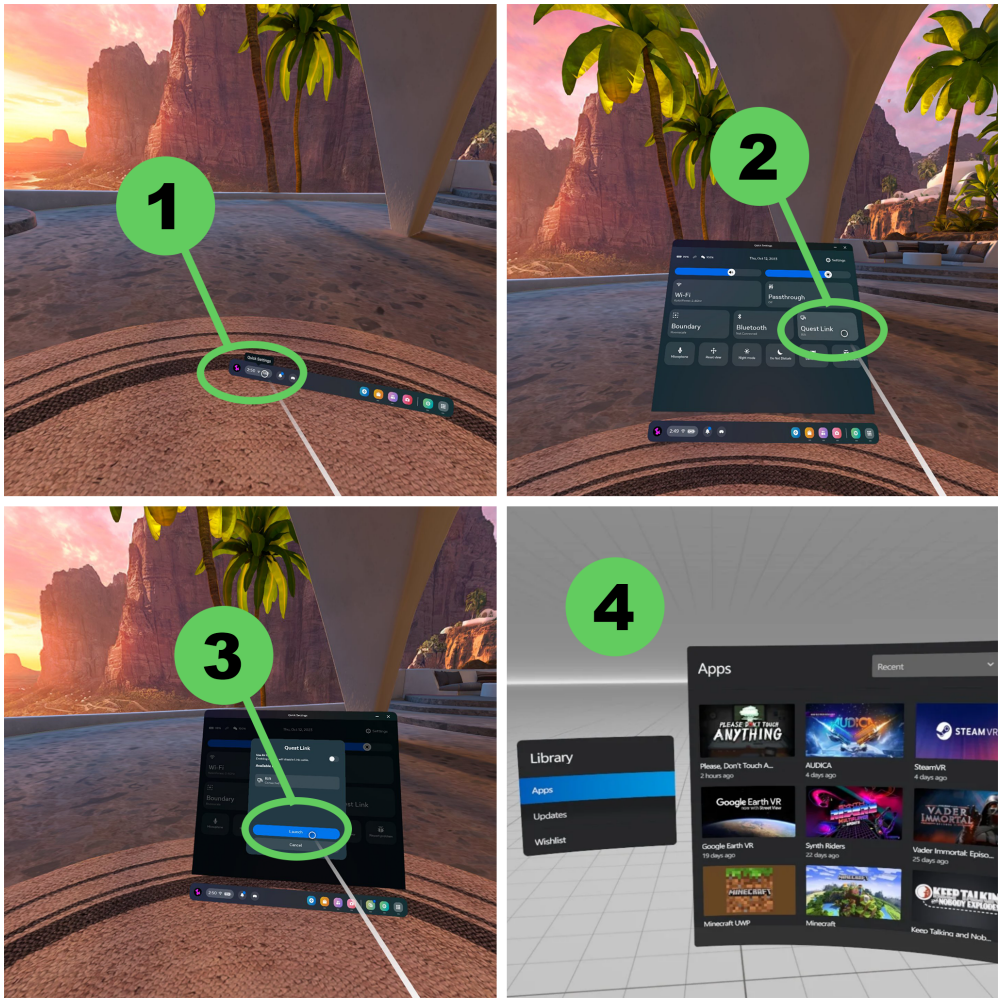

Figure 5: After setting up your HMD and installing the Oculus desktop app, connect your HMD via USB-C and enter Quest Link. To do that (1) select quick settings (2) select Quest Link (3) select Launch. If all works, your virtual environment should be showing a white void space with a grid texture (see (4)).

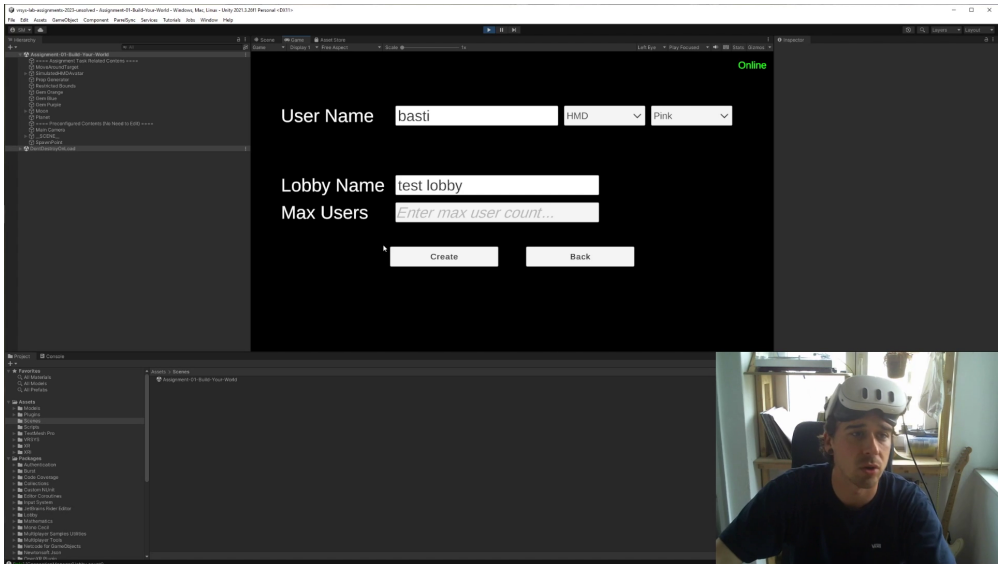

Figure 6: Snapshot of the intro tutorial that supplies an overview of how to start the Unity project for this assignment. The video is located at https://cloud.uni-weimar.de/s/HPz7zfXd8MMfZJX

The Relevance of 3D-Transformations in Scenegraphs

A GameObject is considered a child of another GameObject if it is nested underneath it. In this relationship, the nesting root is considered the parent GameObject. We refer to these hierarchical structures as scenegraphs. You can review the scenegraph for your Unity scene in the Hierarchy window. For better generalizablity, you should get used to referring to GameObjects as nodes. In the context of Unity, these terms are interchangeable. A parent node can have multiple child nodes, but any node can only ever have one parent node. Scenegraphs help us to model transformation dependencies and they can be used to organize the layout of our virtual scene.

Every node describes a transformation (expressed by a transformation matrix), which is composed of parameters for 3D translation, rotation and scaling. Child nodes inherit the transformation of their parent. Which means that a child nodes transformation is expressed w.r.t. to the parent nodes transformation as its origin (local space). To compute the transformation of a node in world space, one needs to traverse the scenegraph from the root node towards the respective child and multiply all matrices along the way. Unity conveniently wraps this concept, so that manual matrix multiplication is unnecessary in most cases. But as you will practically explore throughout this course, 3D computations in your application logic rely heavily on conceptual knowledge about local and world space transformations.

Object Creation and Transformation Basics

Tasks in this section are centered around instantiation and transformation of GameObjects from script. A demo video of the expected output produced by a finished implementation can be found here: https://cloud.uni-weimar.de/s/eKaNL4gagMD5pQm

Exercise 1.1 (30%)

The GameObject `Prop Generator` contains a script `EnvironmentGenerator.cs` that provides method stubs for the automatic instantiation of further GameObjects to populate the scene. The array `environmentPrefabs` contains a set of representative geometries that should be instantiated at random positions across the ground plane (feel free to import and modify this list using your own assets).

To complete this task, follow the steps below.

1.) Implement the method `GenerateEnvironment()` to perform the following operations:

- Use the `Random` module to select a random prefab from the array `environmentPrefabs`.
- Instantiate this prefab and set the `Prop Generator.gameObject` as its parent
- Move it to a random position on the ground plane within [`generatorBoundsMin`, `generatorBoundsMax`] and apply a random rotation around the instances' up-Vector.
- Store the instance inside the list of `instances`.
- Repeat this process until a total number of `numObjects` prefab instances are placed in the scene.
- Call the function when the app starts

When you start the application, your environment should now be populated with a unique random arrangement of `environmentPrefabs` at each execution.

2.) Complete the function `ResolveCollisions()` to ensure that the instantiated objects are not placed within any of the `restrictedBounds`(see https://docs.unity3d.com/ScriptReference/Bounds.Intersects.html). Move them if necessary. Upon execution of your game, you should now see all objects colliding with the house jump out 2 seconds after their instantiation.

3.) Extend the `Update()` function, so that it checks if the variable `reset` was set to true. If so, all `instances` should be destroyed (see https://docs.unity3d.com/ScriptReference/Object.Destroy.html), removed from the list of `instances` and the vegetation should be regenerated. Don't forget to flip `reset` again at the end of performing these operations, to avoid a looped reset execution. While your app is running, you should now be able to click the `reset`-checkbox on the `Environment Generator` component in the Inspector, which will trigger a new unique random environment.

Transformation in World Space

Tasks in this section of the assignment require you to compute transformation updates expressed w.r.t. world space. You will create a continuous motion path for a simulated HMD avatar. A demo video for the output of the completed tasks below can be found here: https://cloud.uni-weimar.de/s/bZPD4EKErHfKqaA

Exercise 1.2 (15%)

The `MoveAroundTarget.Transform` is set as the `target` of the `ServerMoveAroundTarget.cs` script, which is attached to the `SimulatedHMDAvatar` GameObject).
Extend the computation inside the `ServerMoveAroundTarget.cs` script, so that:

- `CalculatePositionUpdate()` calculates `newPosition` as a radial motion update around the `target` at given `degreesPerSecond`. It takes into account only the xz-Plane when calculating the position update. The y-Position should remain unaffected by `Update()`.
- `Time.deltaTime` is applied to the calculation of `newPosition` to ensure that varying frame rates of your application will not affect the `degreesPerSecond` movement speed produced by `ServerMoveAroundTarget.cs`.
- `CalculateRotationUpdate()` should adjust the forward vector of `SimulatedAvatarHMD.transform`, so that it matches the current movement direction.

Transform Networking

You will notice that the `Update` function of `ServerMoveAroundTarget.cs` returns if the condition `!IsServer` is met. The aforementioned property is inherited from the base class (`NetworkBehaviour`) of this script. As you might expect, this condition ensures that only the server is in charge of moving the `SimulatedHMDAvatar` GameObject. Clients in the application receive updated transformation data from the server.

Continuous updates of transform data are enabled by the `Network Transform` component, which needs to be added to each GameObject that should have consistently shared transform properties. You can adjust the sync settings by changing the respective component properties. Unchecking options that do not change during runtime, decreases the amount of information that needs to be sent over the network. So it is good practice to configure accordingly.

Any `NetworkBehaviour` (including the `Network Transform` component) requires to be associated with a `Network Object`. `Network Object` acts as a container for information that is sent over the network. One `Network Object` can carry information for multiple `NetworkBehaviours`. It also tracks who owns a resource and is allowed to change it (more on that later).

The link between a `NetworkBehaviour` and its corresponding `Network Object` is automatically established. You need only add it to the root GameObject that it should describe. All `NetworkBehaviours` added as components to the same GameObject, or any GameObjects further down the hierarchy, will now be associated with it.

Exercise 1.3 (10%)

Unfold the hierarchy of `SimulatedHMDAvatar` and investigate it carefully. You can find both a `Network Object` and `Network Transform` attached to it. Other moving parts of the avatar are the GameObjects `Head`, `HandLeft` and `HandRight`. These rely on the `Network Object` above and only add `Network Transform` components. Additionally, they are composed of scripts that drive their motion. Understand the behavior implemented in the `ServerArmSwingSimulation.cs` and `ServerGazeSimulation.cs` scripts.

During the assignment presentation, be prepared to quickly guide us through the Unity Editor (and the source code) to explain your answers to the following questions:

- Which transform properties are changed by `ServerArmSwingSimulation.cs` and `ServerGazeSimulation.cs`
- How does the behavior of `ServerArmSwingSimulation.cs` and `ServerGazeSimulation.cs` affect the sync setting on the `Network Transform` of `Head`, `HandLeft` and `HandRight`

Transformation in Local Space

Tasks in this section of the assignment require you to compute transformation updates in a scenegraph. You will create orbital motion for a planet system. A demo video for the output of the completed tasks below can be found here: https://cloud.uni-weimar.de/s/6LMaJ9KJpcRLnoq

Exercise 1.4 (10%)

Extend the `Update` function inside `ServerTimedRotate.cs` as follows:

- the script continuously adds a local rotation to the GameObject it is attached to (`degreesPerSecondX`, `degreesPerSecondY` and `degreesPerSecondZ` describe the amount of added rotation per second around each local axis)
- `Time.deltaTime` is applied to the calculation of the rotation update to ensure that varying frame rates of your application will not affect the rotation speed

Exercise 1.5 (35%)

Compose a scenegraph for the planet system, which satisfies the following requirements:

- the large grey `Planet` rotates around itself in the origin of the planet system
- the green `Moon` orbits around the `Planet`, while also rotating itself
- the `Gem Orange/Blue/Purple` orbit around the `Moon` independently
- orbital motion speed and direction is unaffected by any planet rotations
- all of the above transformation updates are computed solely by using your completed implementation of `ServerTimedRotate.cs` (see previous task in this section)
- moving the origin of the planet system inside of the Unity Editor at runtime moves all of the above nodes
- add `Network Object` and `Network Transform` components to appropriately distribute the motion within your scenegraph composition