Guideline

Document name
Content production guidelines: Deep Space Development Kit for Unity3D

Project Acronym:	IMMERSIFY
Grant Agreement number:	762079
Project Title:	Audiovisual Technologies for Next Generation Immersive Media

Revision:	1.4.1 (always corresponds to Dev Kit version)
Authors:	Clemens Scharfen Roland Haring Ali Nikrang
Reviewer:	Erik Sundén, NVAB
Delivery date:	28.12.19
Dissemination level (Public / Confidential)	Public

Abstract
The Ars Electronica Deep Space 8K Unity Development Kit shall make it possible for developers all over the world to easily create Unity applications for the Deep Space 8K, a large-scale multiuser VR environment. This document shall give an overview of the Deep Space 8K possibilities and explain the SDK in detail.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 762079.

Table of Contents

1 Introduction

The Ars Electronica Deep Space 8K Unity Development Kit shall make it possible for developers all over the world to easily create Unity[1] applications for the Deep Space 8K[2], a large-scale multiuser VR environment.

This document shall give an overview of the Deep Space 8K possibilities and explain the SDK in detail.

Please find the source, that is documented in this document on Github[3].

The current Unity Version is 2019.2.x. We try to always use the latest Unity release for new projects, but we figured out that not all versions work, e.g. when using the off-axis projection or stereo rendering (see chapter Known Unity Bugs). The latest tested version that works fine in the Deep Space 8K is 2018.3.5. Later versions might work too, though. However, do not worry too much about upcoming problems. If you are ready to test the application in the Deep Space 8K and recognize problems, we will help you with our experience.

1.1 Key Data

Projection: 2x 8K (8x 4K projectors in total), active stereo (120 Hz), L-shaped projection areas

Note: Net resolution depends on actual system setup and calibration due to edge blending and geometric calibration. Projected wall and floor resolution without edge blending is 8192x4320 each. The currently calibrated resolution for the wall is 6467x3830 pixels and for the floor 6467x3595 pixels.

Size: The wall's size is 14.96m (width) x 8.8m (depth). The floor's size is 14.96m (width) x 9.003m (height).

Note: When using the wall-floor-combined view (off-axis projection), that is provided in this SDK, the sweet spot is set at the center of the floor, at 1.7m height. This means that an "average tall person" standing in the middle of the room will have the feeling that there is no edge between the wall and the floor projection. This effect also applies to all people that are standing close to the sweet spot. The sweet spot position can be changed in the SDK at any time.

Audio: 5.1 Surround-Sound

Operating System: Windows 7 SP1 64 Bit

Cluster (First Signal Way): One machine for wall and floor each, i7 and 4 x P6000 GPUs

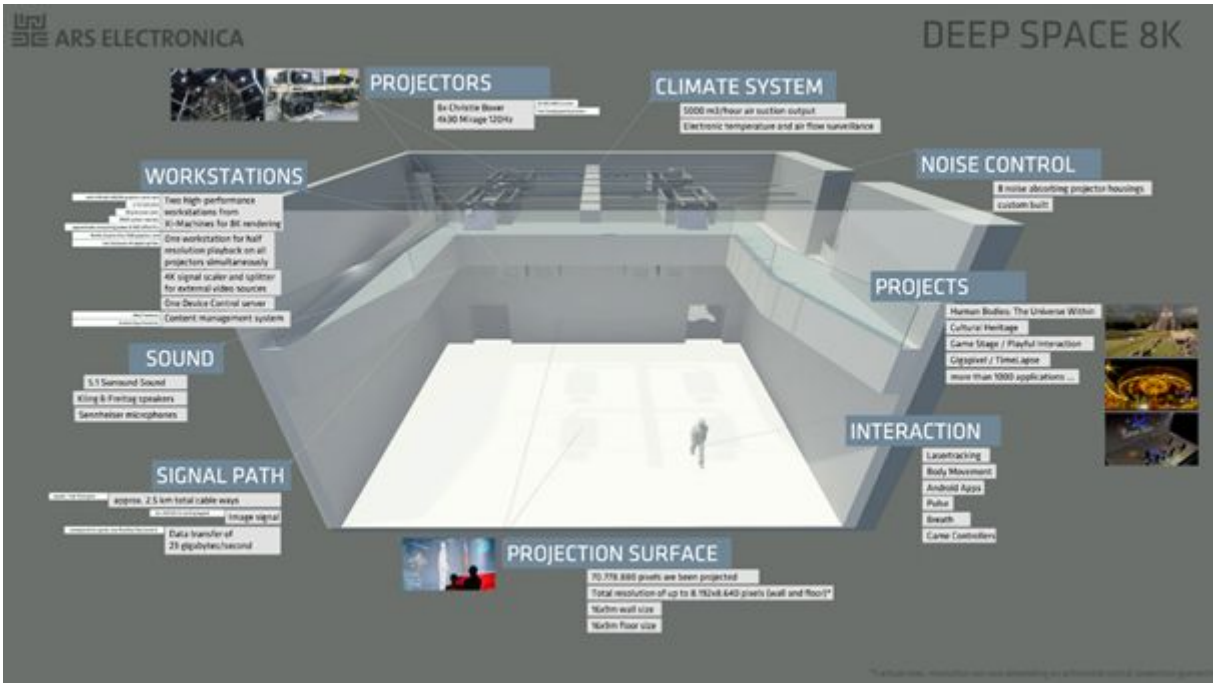
Second Signal Way: One machine for Wall and Floor together (4K resolution only)

[1] https://unity.com/
[2] https://ars.electronica.art/center/en/exhibitions/deepspace/
[3] https://github.com/ArsElectronicaFuturelab/DeepSpaceDevKit

Positional Tracking: On the floor within the projected area

1.2 System Overview

Below is a general overview infographic, summing up all important subcomponents, about the Deep Space 8K.



Additional information can be found on the Ars Electronica blog[4] and on the Ars Electronica Futurelab's website[5].

1.3 Interaction

There are several ways to interact with an application. Some are for our guides ("Infotrainer") only others are made for visitors.

VRController (Guides only)

The VRController is an Android application developed at the Ars Electronica Futurelab to start, stop and control Deep Space applications. It shows all available applications in Deep Space as a list where a specific application can be started. Furthermore, it can send predefined control messages (such as button triggers or position and rotation data) to the currently running application. It implements the VRPN protocol and uses the built-in VRPN server at the Deep Space. Buttons can be defined in the Deep Space CMS and will appear at the VRController screen when the corresponding application is running. For more information about the usage of VRController please see the VRPN Plugin section.

Mobile Control (Guides only)

[4] https://ars.electronica.art/aeblog/en/tag/deep-space/
[5] https://ars.electronica.art/futurelab/en/project/deepspace8k/

Each Highlight Guide has access to a mobile device to have basic control over the Deep Space 8K. The Mobile Control is an Android app consisting of generic views to interact with a Deep Space 8K application. Therefore, a Unity Application can tell the Mobile Control, which views and content shall be presented on the screen and the Mobile Control can tell the Unity Application, what option has been chosen on the screen (e.g. button presses or list item selections).

The Mobile Control was originally developed to control the "Human Body – A Universe Within" Application, one of the main attractions in the Deep Space 8K. The app was kept generic so that it can be used manifold. The main advantage of this mobile app lies in its dynamic approach. The generic views of the Mobile Control can be named and used in diverse and changing contexts. Another advantage is that the Info Trainer does not need another device for the presentation, because the mobile device is in use anyway. However, if you feel that you can control your application easier with the Xbox Controller or another way, do not use the Mobile Control. Implementing controls and testing your application is much more effort with the Mobile Control than it might be with other interfaces.

Pharus / Laser Tracking

Six Laser Rangers track movement on the floor. Our own Tracking Software (Pharus) is processing the measured data and provides the results (people moving in the Deep Space 8K in real-time) as UDP Stream. A Unity application can get access to this Tracking Stream and use the Tracks as interaction possibility.

Xbox Controller

A standard controller can be used to control an application.

Leap Motion (experimental)

A Leap Motion can be connected to the Deep Space 8K System and used as input for a Unity application. However, using the Leap Motion is currently experimental and can only be used in mono (2D projected, 60 Hz) applications. The Leap Motion plugin is not part of the Deep Space 8K DevKit. Feel free to add the official plugin[6] to your project.

Myo (experimental)

A Myo can be connected to the Deep Space 8K System and used as input for a Unity application. However, using Myo is currently experimental, because gesture recognition only works well with predefined profiles that cannot be created or switched fast. On the other hand, accelerometer and gyroscope can be used easily without using a profile. If you are planning to do a performance (Deep Space Live) this might still be an option for you. The Myo plugin is not part of the Deep Space 8K DevKit. Feel free to add the official plugin[7] to your project.

[6] https://developer.leapmotion.com/unity
[7] http://developerblog.myo.com/setting-myo-package-unity/

1.4 General Concepts

This documentation concentrates on explaining how to build an application for the cluster, using Wall and Floor together. Therefore, a network connection is needed for most of the projects.

Typically, the wall is used as the server and the floor acts as the client. The main reason for this is that all interfaces are only connected to the wall-machine. For example, the Audio-System (input and output) is only connected to the wall machine, so playing sound in an application that runs on the floor will have no effect. Also input interfaces like the Xbox controller or the Mobile Control App are directly connected to the wall. If you want the floor to interact on input, you have to receive the input on the wall and pass it on to the floor via network.

2 Implementation Details

In this section, everything that is provided by the DevKit is explained in detail.

2.1 Structure

Everything that belongs to the SDK has been put in the Assets Subfolder “DeepSpace”. The directory names are self-explaining.

All classes are packed into the “DeepSpace” namespace or sub namespaces.

2.2 Project Settings

Depending on what you want to do in your Unity application, there need to set up some settings to achieve these aims. All suggested settings can be done via an integrated editor window, which can be found in the menu: Tools -> DeepSpace -> Settings...

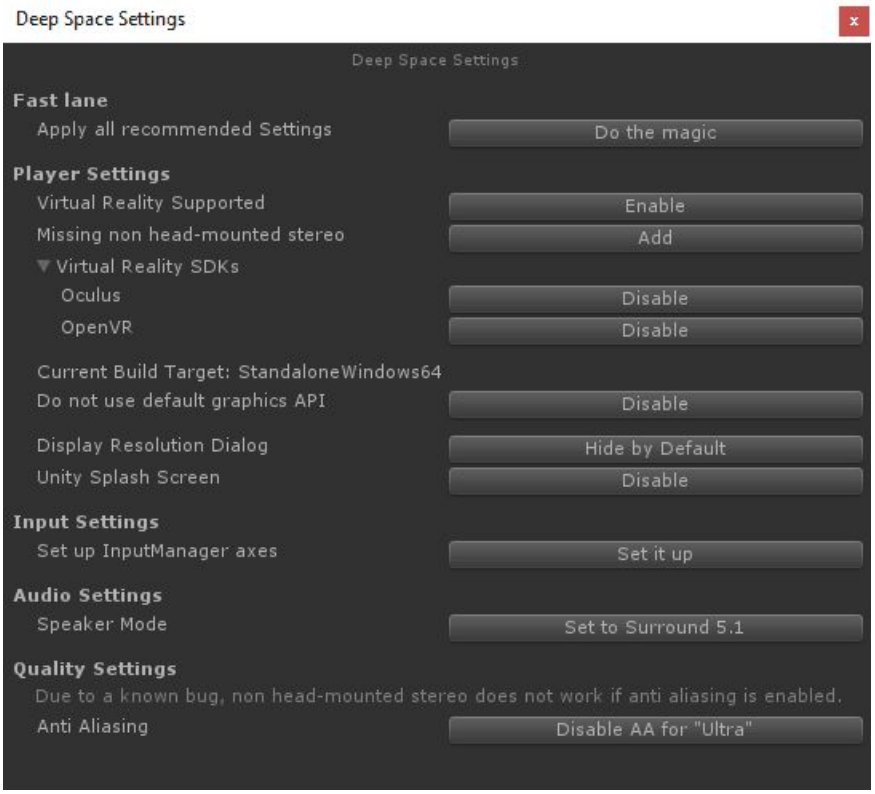


Figure 1: The Deep Space Settings Window to enable all recommended settings.

You can do all recommended settings by pressing the “Do the magic” button on the very top, or only do selected settings by clicking the available buttons. The descriptions should be self-explaining. Once a button was pressed and the setting was changed, the button vanishes. If there are no buttons left to press, everything is set up as recommended.

All Settings can of course also be done manually by using Edit -> Project Settings... and setting up the things you want.

Some settings are default values in Unity 2018.3.x that had other default values in earlier versions. If you are using the DevKit in older versions of Unity (which is not supported anymore, but should be possible in principle) you might want to follow these additional recommendations:

- Editor Settings: Set Asset Serialization Mode to “Force Text”.
- Player Settings: Activate “Run In Background” (under Resolution and Presentation).
- Player Settings: Activate “Visible In Background” (under Resolution and Presentation).

2.3 Provided Functionality

Depending on what you plan to do there are some scripts and prefabs for standard situations that are provided by this SDK.

Command Line Configuration

In general, we are using command line arguments to configure dynamic parameters. For example, where the application runs (on the wall or on the floor), IP addresses and ports (to know how to connect to each other or to any other hardware), or if there shall be shown a debug UI.

You can use your own command line arguments by deriving from the CmdConfigManager class and overriding the ParseArgument Method. If you need a callback, after all parameters have been parsed, you can implement the virtual method FinishedParsingArguments. Have a look at the UdpCmdConfigMgr to learn more about this.

Arguments are passed to the application by the key=value scheme. Do not use spaces around the equal sign between key and value. All keys are processed as lowercase, so it does not matter if you write “-mode”, “-Mode”, or “-MODE”. The result might look like this:

YourDeepSpaceApp.exe -mode=Wall -udpAddress=192.168.0.1 -udpPort=1234

If you want to have more control over your ports, use “-udpSendingPort” and “-udpReceivingPort” instead of “-udpPort”. You will then do something like this:

YourDeepSpaceApp.exe -mode=Wall -udpAddress=127.0.0.1 -udpSendingPort=4444 -udpReceivingPort=8888

YourDeepSpaceApp.exe -mode=Floor -udpAddress=127.0.0.1 -udpSendingPort=8888 -udpReceivingPort=4444

What “-udpPort” does is depending on the “-mode”. On the wall, it will set the sendingPort to the udpPort and the receivingPort to udpPort+1. On the floor, it will set the sendingPort to udpPort+1 and the receivingPort to udpPort.

Wall-Floor-Camera

An often wanted camera view is a connected Wall-Floor-Camera, which means that a user who stands in the so called sweet spot can look at Wall and Floor without seeing the edge between these two projection planes. This effect is created by an off-axis projection.

This setup can be taken directly from the WallFloorCombinedScene which can be found in Assets/DeepSpace/Demo/Scenes. Alternatively you can use the prepared prefabs and scripts to build the setup yourself as described in the following paragraphs.

In the Prefabs directory is a prefab called “CameraSetup” which represents a connected Wall-Floor Camera. This Camera Setup needs to be used together with the WallFloorStarter (or a derived class) Script. The Camera Setup is using two cameras, one for the Wall and another one for the Floor. They are configured to be both visible at the same time in the GameView. The wall camera is rendered in the upper half of the view, the floor camera is rendered in the lower half. If you have set the GameView resolution to 16:9, both camera views are squeezed together. If you want, you can set the Resolution to 16:18 (1600 x 1800). You can ignore the Message in the GameView that tells you that the “Scene is missing a fullscreen camera” because the WallFloorStarter Script will (depending on the configuration) disable one of the cameras and set the other one back to fullscreen. This makes it possible to build one application and copy it to both cluster machines.

If you want to translate or rotate the camera, do not change the Unity Camera itself. Move and rotate the Camera Setup instead. Of course, as soon as you are moving the Camera Setup dynamically (by some kind of input), you need to send the translation over the network. Therefore, you should not move the Camera Setup directly, but the NetworkTransformTarget. For details, have a look at the *Networking* Section.

If you want to change the SweetSpot, translate the Observer GameObject, which is a child of the CameraSetup. The Observer represents the head of a visitor in the Deep Space 8K and is responsible for the view. To see, where the Observer stands in the Deep Space, activate the WallProjectionPlane

and the FloorProjectionPlane objects. If you do not want to see the view frustum anymore, you can tick off the “Draw Debug” value from the Observers CameraOffAxisProjection Script or change the wall color and floor color values.

Rotating the Observer is currently not implemented. If you want to do so, feel free to extend the CameraOffAxisProjection Script.

Pharus / Laser Tracking

One of the nicest ways to interact with an application in the Deep Space 8K is by using the Laser Tracking System, because it is a multi-user-input-system that can handle up to 30 users at the same time.

Pharus, a software that receives the data from our laser rangers and calculates current positions of people in the room, is developed by Otto Naderer, a Futurelab Programmer. The network interface on the Unity side was originally implemented and open sourced by Andreas Friedl from the UAS Upper Austria / Hagenberg and can be found in a GitHub[8] git repository. This DevKit used this implementation in earlier versions to interact with Pharus. After a lot of refactoring, there are only little intersections left.

Basic Setup

Two protocols can be used to interact with Pharus. Its own protocol “TrackLink” or the well-known “TUIO” protocol. We recommend using the TUIO protocol, because it is light, easy to handle and well documented. Only use TrackLink if you want to make use of advanced features like echoes (foot step recognition).

Getting tracking data from Pharus is easy. Two Scenes (PharusTracklinkScene and PharusTuioScene) are prepared to show a simple sample how to use the tracking. (Find more information about this in the Demonstration section.) Depending on the protocol you want to use, you only have to set up your scene slightly different.

Working with Tracking Data

To use TUIO tracking, you need a TuioReceiveHandler Component (this is receiving and gathering the TUIO data from the network) in the scene. To use TrackLink, you need an UdpReceiver Component (that receives the data) and a TracklinkReceiveHandler Component (to gather the TrackLink data) in the scene. For both setups, a TrackingEntityManager Component can be used to make something with the data. The TrackingEntityManager is independent from the selected tracking method and therefore works with TUIO and TrackLink. If you want to extend the functionality, the easiest way is to create a class and derive from TrackingEntityManager. In this case, you can overwrite the following methods:

- TrackAdded
  is called when a new Track was added.
- TrackUpdated
  is called once per Unity Update, to give you updates about existing tracks.
- TrackRemoved
  is called, when a Track has left the tracked area (Deep Space floor) or got lost (e.g. because the person was jumping).

A Track is a person (or tracked object) standing or moving in the Deep Space. The TrackAdded and TrackUpdated methods get a TrackRecord instance passed as parameter. A TrackRecord is holding all

[8] Unity Tracking Client: https://github.com/Playful-Interactive-Environments/UnityTrackingClient

data of a Track (e.g. position, speed, rotation, etc.). The TrackRemoved method get a trackID as parameter to know which track was removed.

```
// Extend from TrackingEntityManager if you want to manage the tracking data yourself.
public class YourTrackingEntityManager : TrackingEntityManager
{
	public override void TrackAdded(TrackRecord trackRecord)
	{
		// Implement this method to get informed about new tracks.
	}

	public override void TrackUpdated(TrackRecord trackRecord)
	{
		// Implement this method to get informed about existing tracks.
	}

	public override void TrackRemoved(int trackID)
	{
		// Implement this method to get informed about removed or lost tracks.
	}
}
```

Code Snippet 1: Getting Track data from Pharus.

In Code Snippet 1 can be seen, how to access the tracking data. The TrackRecord class that you can find in the SDK, is well documented and will answer questions about what a Track is in detail.

If you do not want or can derive from TrackingEntityManager, you can have a look at the implementation of this class to have a look at how to register for these callback methods by registering at an instance of TrackingReceiveHandler and by implementing the ITrackingReceiver interface. All classes can be found in the DeepSpace.Lasertracking namespace.

To simulate activity in the Deep Space 8K at your place, you can use the Pharus-ReplayOnly application. Therefore, start one of the batch files (Start*.bat) in the pharus-replay folder or take one of the ".rec" Files and drop it onto Pharus.exe. Pharus will open and the chosen record file (recorded movement data in the Deep Space) will play in an endless loop.

To get a first impression of how these recorded data can be used in a Unity application, just open the PharusTUIOScene or PharusTracklinkScene. Please have a look at the Demonstration section for more details.

Configuration

To work with TrackLink and Pharus in different setups (e.g. on a development machine and in the Deep Space directly), there are existing two config files in json format. In Assets/StreamingAssets/DeepSpaceConfig the files tracklinkConfig.json and tuioConfig.json can be found. Both contain the basic network information and the virtual size ("screenWidthPixel" and "screenHeightPixel" in pixel) and real size ("stageWidth" and "stageHeight" in centimetres) of the tracking area. This is required to map a real person in the Deep Space onto the display (projection).

Xbox Controller

The Deep Space SDK provides a controller input manager that wraps the Unity input system for controllers. All functionality is packed into the ControllerInputManager class. It is a singleton and can be accessed easily from all other script components. To make the wrapper work, the Unity Input Settings have been already set to support all controller keys. If you export the

ControllerInputManager Script to another project, do not forget to copy the InputManager.asset file from the ProjectSettings directory too.

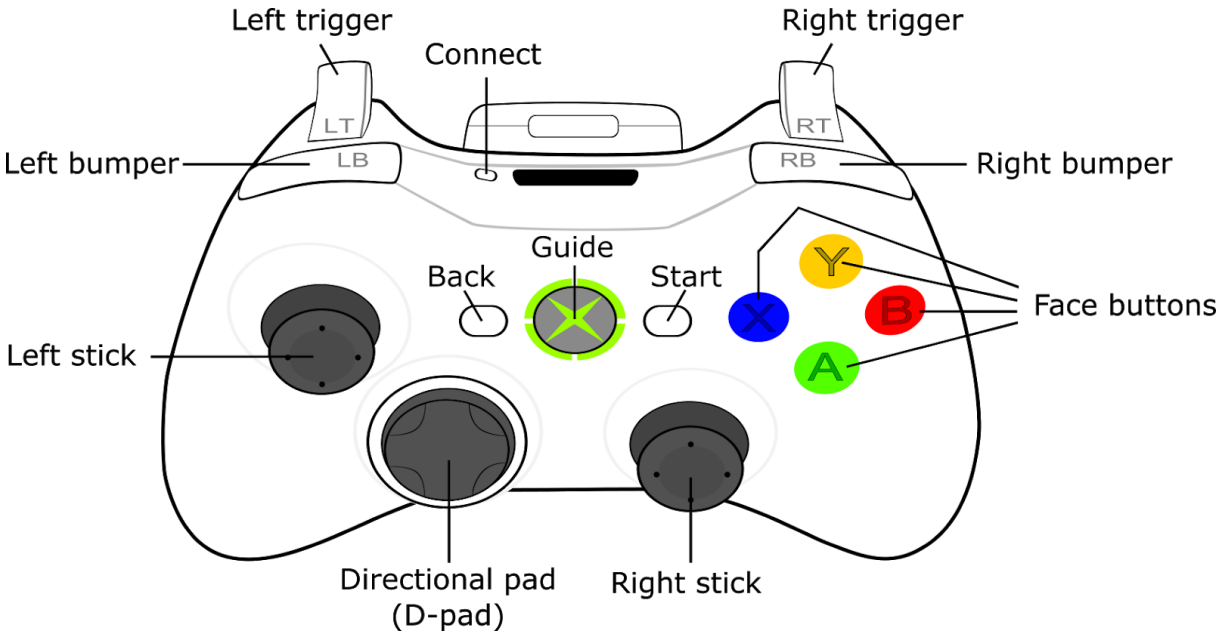


Figure 2: Xbox Controller Layout

For an easier understanding, Figure 2 will help to talk about the various buttons and axes.

Accessing Axes

All axes are defined in a range between -1 (Left or Down) and +1 (Right or Up). To access them, just read out the now described class attributes:

- Left stick: Accessible via Vector2 LeftJoystick
- Right stick: Accessible via Vector2 RightJoystick
- Directional pad: Accessible via Vector2 DPad
    - x value: left / right
    - y value: up / down
- Left trigger / Right trigger: Accessible via float ShoulderTrigger

Accessing Buttons

Interacting with buttons can end in three states:

1. ButtonDown: Pressing the button down (called once)
2. ButtonHold: Keeping the button down (called each frame the button is pressed)
3. ButtonUp: Releasing the button (called once)

These states can be accessed in callback events (see Code Snippet 2). All callbacks receive a Button enum type as parameter that defines the button that was interacted with. The enum values are named intuitive, but to sum it up:

- Face buttons: Button.A, Button.B, Button.X, Button.Y
- Right bumper / Left bumper: Button.R1, Button.L1
- Back / Start: Button.BACK, Button.START
- Pressing down Right stick or Left stick: Button.R3, Button.L3

```
private void Update()
{
	// Getting the current value of axes in this frame

	Vector2 leftJoystick = ControllerInputManager.Instance.LeftJoystick;
	// leftJoystick.x -> -1 / +1 -> Left / Right
	// leftJoystick.y -> -1 / +1 -> Down / Up

	float shoulderValue = ControllerInputManager.Instance.ShoulderTrigger;
	// shoulderValue -> -1 / +1 -> L2 / R2
}

private void SubscribeButtonEvents()
{
	ControllerInputManager.Instance.RegisterButtonDownCallback(OnButtonDown);
	ControllerInputManager.Instance.RegisterButtonHoldCallback(OnButtonHold);
	ControllerInputManager.Instance.RegisterButtonUpCallback(OnButtonRelease);
}

private void OnButtonDown(ControllerInputManager.Button button)
{
	// React on pressing down a button
}

private void OnButtonHold(ControllerInputManager.Button button)
{
	// React on holding down a button
}

private void OnButtonRelease(ControllerInputManager.Button button)
{
	// React on releasing a buton
}
```

Code Snippet 2: Looking at Axes and Registering for Controller Button Events

Please have a look at the CameraController.cs Script to see how the Controller can be used.

Note that the Xbox gamepad is only connected to the wall machine. Therefore, the wall application is the only one that recognizes the input. If you want to send button or axis events to the floor machine, you have to receive the events on the wall and send it over to the floor via networking. This can be easily done by Extending the Networking.

Mobile Control

The Mobile Control is our own proprietary Android App (client) that can be connected with a server (the Unity application on the wall) via TCP. It was developed by Martin Gösweiner, a former Futurelab App Developer. The basic communication concept is that the server tells the client what view shall be shown and the client tells the server what buttons have been pressed in the currently visible view. You do not need to care about the specific protocol between these two because all messages that can be sent and received are wrapped in the MobileControlMessages namespace classes.

To make the beginning of using the MobileControl easier, the DevKit provides the MobileControlManager. This class handles the communication between server and client. The public interface for this class handles Registering and Unregistering of active and passive views, getting a current or a specified view, and activating a specific or the next or last in order view.

All this might sound unfamiliar or complicated to you, but I will explain everything in detail and you will see that it is not at all. Please be sympathetic about the fact that it is not possible to explain everything in a short-kept documentation, so having a look in the source and trying out things is highly recommended.

Introducing Views

A view is a screen with specified items that can be shown to a user. In general, the screen is split up into two parts: The header on the top and the view, the main part of the screen, below the header. The header has a left and right navigation arrow and a home button in the middle. If there is only one view available, these buttons are not available. If there are more views available, a user can switch between them by the left and right button on the top. The home button in the middle is a special request that can be implemented to navigate to a specified view. The view itself is the part that can be set by the server. The Android action bar shows several options (on the left side: open the menu, the name of the current panel; on the right side: unlock the orientation, connect to the server, open the connection-settings) and the name of the currently shown view that can be set by the server.

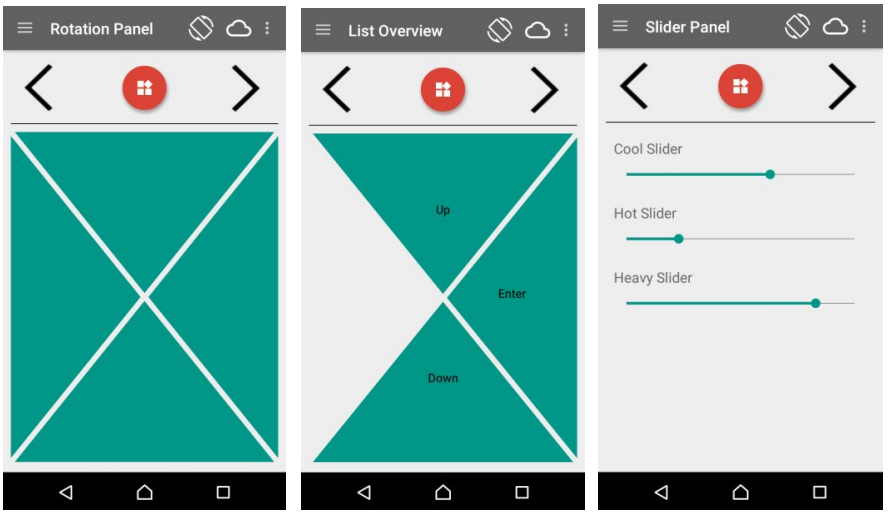


Figure 3: a) A default navigation view. b) A named navigation view with disabled left button. c) A slider view.

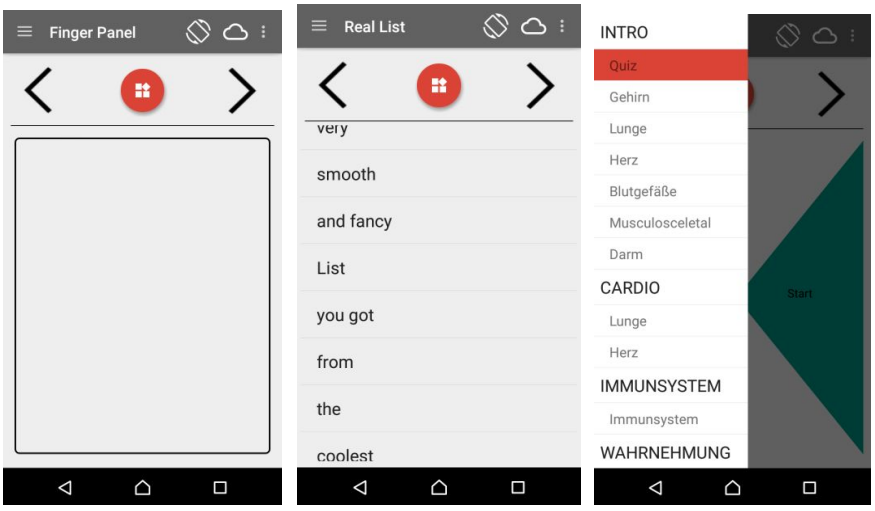


Figure 4: a) A touch view. b) A list view. c) The side menu.

Figure 3: a) A default navigation view. b) A named navigation view with disabled left button. c) A slider view.Figure 3 and Figure 4 show all generic views that are available.

General View Handling

What you can see in the screenshots in Figure 3 and Figure 4 is a screen on the mobile device in the mobile control app. As already said, everything below the top menu (left, home and right button) and above the android navigation bar is what we call a view. You can set the views and receive pressed buttons in this view by the server (unity application). The communication package system is completely handled in the MobileControlMessages namespace classes. If you open the MobileMessages.cs file, you will find everything you need to communicate with the mobile control.

The first thing that is important is the Command enum. Comments are telling describing, how these commands are used. You do not have to care about the Sending Message Types because they are handled internally. But you will make use of the Receiving and Bidirectional Message Types.

The basic workflow to send a message is to create a new instance of a sending (or bidirectional) message (depending on what you want the mobile control to do) and initialize all the members to generate your wanted result out of this generic view. To send this message over the network, use the Message.GetMessageBytes method. It returns the message byte array that can be sent over the network. The sending itself can be done by your own TCP code or by using the MobileConrolManager and MobileControlTcpManager classes.

To receive a message, go through all received messages and check, if there is a message of type you want to receive. For example, if you are waiting for a PressOptionMessage (when a button was pressed), check a message for the Command type PRESS_OPTION. (All commands are named intuitively like the messages they belong to. If you are not sure about a command, have a look for it in the classes and you will easily figure out the connections.) After knowing the message type, create the right (concrete) Message (like PressOptionMessage) and initialize it by the message bytes (by using the constructor). All this is explained in detail and with examples in the following sections.

SimpleView / NavigationView

Figure 3a and 2b are views of the same type. 2a is a navigation view with no specified extras. 2b specifies the left button as disabled and gives the other buttons names (Up, Down, Enter). Code Snippet 3 shows how to create a navigation view.

```
// This shows a navigation view on the mobile device like 2a does:
ShowSimpleViewMessage navigationViewMsg =
new ShowSimpleViewMessage(Command.SHOW_NAVIGATION_VIEW);

// The following lines define options to specify the view (similar to 2b):
// Set button names (Left and Right)
navigationViewMsg.namedButtonList.Add(
new KeyValuePair<uint, string>((uint)ChoseOptionMessage.Select.RIGHT, "Right"));
navigationViewMsg.namedButtonList.Add(
new KeyValuePair<uint, string>((uint)ChoseOptionMessage.Select.LEFT, "Left"));
// Disable the up and down button
navigationViewMsg.disabledButtonList.Add((uint)ChoseOptionMessage.Select.UP);
navigationViewMsg.disabledButtonList.Add((uint)ChoseOptionMessage.Select.DOWN);
```

Code Snippet 3: Creating a navigation view with additional options

If you neither define a name for a button nor disable it, it will be shown as a blank button.

If the user presses a button, a ChoseOptionMessage is sent as an answer from MobileControl.

```
// A message (Type Message) came from the mobile control:
if (message.Command == Command.CHOSE_OPTION) // Check if it is a button press
{
	// Create the message with the expected type:
	ChoseOptionMessage choseMessage = new ChoseOptionMessage(message.Bytes);
	// Unpack the message (this fills the attributes with their values):
	choseMessage.UnpackReceivedMessage();
	// Check which option (button) was pressed:
	if (choseMessage.ChosenOption == (uint)ChoseOptionMessage.Select.RIGHT)
	{
		// Right button was pressed...
	}
}
```

Code Snippet 4: Receiving a button press from the current view

As you can see in Code Snippet 4, a ChoseOptionMessage was received from the client, containing the information which button was pressed. By the way, the message variable is coming from the MobileControlManager and is passed on to the current view, which will be explained later. In this case, only the Right button was checked, but all other directions are working the same way. Of course, if a button in this view was disabled, you do not have to wait for a button with the disabled direction, because it will never happen. Besides using the ChoseOptionMessage to ask if a button was tapped on (shortly touched and released right afterwards), you can also ask if the button is being held down (PressOptionMessage) or released (ReleaseOptionMessage). These two message types are derived from ChoseOptionMessage and therefore behave the same way. To know when you can use these types, just ask if the message.Command is Command.PRESS_OPTION or Command.RELEASE_OPTION.

Sending a view message and Receiving an answer message is basically always the same. All messages, that the Mobile Control can display, are classes that end with the phrase “ViewMessage” and can be found in the DeepSpace.MobileControlMessages namespace.

SliderView

Figure 3c shows a SliderView. This View can show up to 255 sliders. Let’s have a brief look, how to create this ViewMessage and how to receive an answer.

ShowSliderViewMessage contains an inner class called SliderDetails. For each Slider you want to show, create a SliderDetail (parameters should be self-explaining) and add it to the ShowSliderViewMessage.sliders list. Keep in mind, in which order you put in the sliders, because the answer will refer to the sliders index.

If a user changes a slider, the server will receive a SetSliderValueMessage. Again, check the type of a received message (Command.SET_SLIDER_VALUE), create a new SetSliderValueMessage with the received message.Bytes and unpack the received message. After that, you can access the two parameters: sliderIndex is the index of the slider that you have added when creating the view message; curSliderValue is the new / changed value for this slider.

TouchView

Figure 4a shows a TouchView. To show a touch view, you have to create a ShowTouchViewMessage. No parameters, no members, it’s easy as that.

Reading out a touch view answer is therefore a little trickier. You will receive a FingerPositionsMessage for every touch and finger movement. After checking the type (Command.FINGER_POSITIONS), creating and unpacking it, you can access the member list fingerPositions of type FingerPosition. Each FingerPosition contains a fingerID, xPos and yPos. The ID

stays the same as long as the finger stays on the display. xPos (horizontal) and yPos (vertical) are scaled from 0 to 1, meaning from left / down to right / up.

A basic touch handling is implemented in the class TouchView that handles up to five fingers on the screen. You can derive from this base class and implement the virtual [x]FingersChanged methods. For demonstration, you can have a look at DemoTouchView.cs which is doing exactly this.

ListView

In Figure 4b, a ListView can be seen. To create such a view, you have to instantiate a ShowListViewMessage. The constructor takes a List<string> as argument, which contains the elements of the list.

When a list entry is pressed on the mobile control, you will receive a ChoseListEntryMessage. After unpacking the message, you can read out the chosenIndex member. This integer contains the index from the chosen element in the list, you used to initialize this view.

Side Menu

The mobile control will automatically send a request (Command.GET_SCENE_MENU) for the menu when the app connects to the server. However, you can set and change its content at any time. To open the Side Menu on the mobile control, just swipe from the left border to the right.

```
// Create the message:
SceneMenuMessage sceneMenuMessage = new SceneMenuMessage();
// Create a category:
SceneMenuMessage.Category category0 = new SceneMenuMessage.Category();
category0.categoryID = 0;
category0.categoryName = "Demo Category 1";
// Create a Section (Choseable Menu Entry belonging to a category)
SceneMenuMessage.Category.Section category0Section0;
category0Section0 = new SceneMenuMessage.Category.Section();
category0Section0.sectionID = 0;
category0Section0.sectionName = "Demo Section 1";
// Add Section to Category. You can add as many sections as you want.
category0.sectionList.Add(category0Section0);
// Create another Section
SceneMenuMessage.Category.Section category0Section1;
category0Section1 = new SceneMenuMessage.Category.Section();
category0Section1.sectionID = 1;
category0Section1.sectionName = "Demo Section 2";
// Add Section to Category
category0.sectionList.Add(category0Section1);
// Add Category to SceneMenu. You can add as many categories to the menu as you want.
sceneMenuMessage.categorySectionList.Add(category0);
```

Code Snippet 5: Creating a Side Menu

Code Snippet 5 shows how to create SceneMenuMessage to define the values of the side menu. As you can see in Figure 4c, the Side Menu consists of categories (bold font, not clickable) and sections (normal font, clickable). When a Side Menu entry has been chosen, the side menu closes automatically and the server receives a message of Command CHOSE_SCENE. This bidirectional message (ChoseSceneMessage) contains the index of the chosen category and section. If you send a ChosenSceneMessage to the mobile control, the app will highlight the requested entry.

Server Client Communication

Now that you know about the possible messages and views that can be sent, let's talk about how to communicate these messages between the devices. In this case we are using the MobileControlTcpManager class. You can change or exchange both classes so that they fit your

needs. The MobileControlTcpManager is a simple TCP Connector that sends and receives packages (in this case between Unity and the Mobile Control). If you want to use your own TCP classes or library, you just have to configure the port, where the mobile control connects to the unity application. (The same port needs to be set in the mobile control settings.) The unity application has to listen on this port and wait for a connection. The TcpManager also cares about keeping up the connection. The mobile control will send a heartbeat message from time to time. If no other message is sent, the unity application also needs to send a heartbeat back. If no message is sent for 10-15 seconds, the connection will be terminated by the mobile control.

View Handling

To handle our views, we are using the MobileControlManager together with specialized classes derived from MobileControlView and MobileControlPassiveView. Again, if you have other needs, you can change these or write other classes. Anyway, the prepared classes together with some demonstration cases will help you understand how to work with the mobile control.

The MobileControlManager is connected to the MobileControlTcpManager and therefore responsible for sending and receiving messages to and from the mobile control. A MobileControlView is the server representation of a view that can be seen on the mobile device. If a view is active (only one view can be active at a time) it receives all messages from the mobile control and can communicate with it. A MobileControlPassiveView is a server representation of a view that cannot be seen on the mobile control but receives all messages that are sent from the mobile control. (It does not have an active state. All passive views receive all messages.) You can register (and unregister) views and passive views in the MobileControlManager with (Un)Register(Passive)View methods. If you register more than one MobileControlViews, you can switch between them with the mobile control top navigation buttons (left, home and right). As said, only one MobileControlView can be active at once, so switching from one to another will disable the last one and enable the next one. In the LateUpdate method, the MobileControlManager handles the basic communication to the Mobile Control by processing general requests (like switching views) and passes on special requests to the currently active view. The MobileControlManager handles the GET_WINDOWS Command (tells the mobile control, how many views are available), GET_WINDOW_BY_NUMBER Command (disables the current view and activates the requested one (by index number)), GET_SCENE_MENU Command (Creates the Side Menu as described above), CHOSE_SCENE Command (when an entry from the side menu has been chosen), USE_MICRO_STORY_CONTROL Command (when the home button in the middle of the top menu was pressed – A view has to be enabled) and the UNKNOWN Command (Error handling). All other Commands (Messages) are passed on to the views. In any case (not depending on the type / Command) the message is passed on to every registered MobileControlPassiveView.

When you implement your own view, you simply have to derive from MobileControlView. The three most important methods in this class are EnableView, DisableView and ProcessMessage. Enable view is called, when the view shall be displayed on the mobile control. This is where you create and send your view messages. You can use the DisableView method to clean up your states. It is called when this view gets disabled and another one gets enabled instead. If you override these two methods, do not forget to call the base method. The ProcessMessage is called when the view is active and gets the message as a parameter. The message will be something like a button press or a chosen list entry depending on the view. To register or unregister a view, the (Un)RegisterThisView methods can be used. You can do this from inside the class or from any other class that has a reference to this instance. Please have a look at the demo views (e.g. DemoNavigationView.cs) for a better understanding of views.

Implementing a MobileControlPassiveView is similar to the normal view. Instead of a ProcessMessage, it has a GotMessage method. It cannot be made active because it receives all messages anyway as soon as it is registered.

To send a view request to the mobile control (e.g. inside the EnableView method), create the message as described in the sections above and send it like the message in Code Snippet 6.

```
// viewMsg is a previously created message (subtype of Message)
_mobileControlMgr.tcpManager.SetViewMessage(viewMsg);
```

Code Snippet 6: Sending a View Message

The _mobileControlMgr is a reference to the MobileControlManager that is available in any MobileControlView. (Basically the MobileControlManager is available everywhere because it is a singleton.) The tcpManager is a reference to the MobileControlTcpManager, and the SetViewMethod is explicitly for creating new views (Messages with Commands that end with VIEW) on the mobile control. For Non-View-Messages use the AddMessage(msg) method.

Mobile Control Configuration

To configure the mobile control app to connect with your computer, just open the settings (you can do this by tapping the three dots in the top bar and choosing settings). Under Unity Server IP, enter the IP of your machine (wherever the Unity Editor or build, you want to connect to, is running). Under Unity Server Port enter the same port, you already set in the MobileControlTcpManager. Use the Android back button to navigate back to the main screen, your settings are saved automatically. Make sure that the mobile device and your machine is in the same Network. Then press the connection button in the middle of the view to start using the mobile control.

VRPN Plugin

In addition to the aforementioned ways of communication, it is also possible to use the built-in VRPN server in the Deep Space. The VRPN server is mainly used by VRController. As mentioned before, it is an Android application to start, stop and control Deep Space applications. Using VRController brings additional control possibilities such as interacting with buttons or using its virtual touchpad that sends touch positions to Unity3d applications through VRPN protocol.

Implementing the VRPN plugin[9] in Unity is straightforward. All you need is to put the attached TrackerHostSettings script into your scene and to set the IP (or hostname) of your VRPN server in that script.

Generally speaking, VRPN can be used to track three different types of data:
The *Tracker* type delivers information about position and orientation, the *Analog* type contains axis information of any type (e.g. joystick) and finally, the *Button* type is a binary type that delivers information about the state of a button and whether it is on or off.

The VRPN_HOST script retrieves the current states of all these values from the server. It provides an interface of four methods:

[9] The attached implementation based on UnityVRPN (https://github.com/arviceblot/unityVRPN)

```
public Vector3 GetPosition(string tracker, int channel)
{
	//get position from tracker by channel
}

public double GetAnalog(string tracker, int channel)
{
	//get analog value from tracker by channel
}

public Quaternion GetRotation(string tracker, int channel)
{
	//get quaternion from tracker by channel
}

public bool GetButton(string tracker, int channel)
{
	//get button state from tracker by channel
}
```

Code Snippet 7: Retrieving VRPN tracking data

The first argument "tracker" describes the name of the VRPN device. In case you are using the aforementioned VRController in Deep Space, it has to be "Tracker0". The second argument "channel" corresponds to the channel of the data (since VRPN devices usually support several channels for different hardware inputs), which is in the case of Deep Space for both methods GetPosition and GetOrientation "0". Buttons are linked to different channels since you have to use one channel per button. A nice feature of the VRController is that you can define all your buttons in the Deep Space CMS and link them to different ID's (channels). They will appear at the VRController screen when your application is running.

Networking

In general: You are free to use any networking library (UNET or 3rd party) to synchronize the state between the Wall and Floor application. The reason why we decided to implement some networking messages ourselves was because of performance and control. Using UDP directly is slightly faster than using UNET. You buy this little performance by a big loss of features and comfort. Nevertheless, you can extend and use this feature if you prefer a simple byte or JSON packaged based network communication. Furthermore, if you are using an external server application, you can easily communicate between your server and the unity applications via JSON packages.

You can find all related classes for this implementation in Assets/DeepSpace/Scripts/ in the subdirectories UDP and JsonProtocol. Basically, you need one UdpReceiver in your scene to receive JSON packages (no matter from where the packages are coming) and one UdpSender for each other machine, you want to send data to. The receiver and sender needs to be initialized by calling ActivateReceiver(port) and ActivateSender(receiverIP, port). If these methods are not called, the socket will not be initialized and therefore no data will be received or sent. For the regular DeepSpace Setup, there is a class called UdpManager, which wants to get references to two senders (one sending to the Wall and the other one sending to the Floor) and one receiver. The manager activates the right sender and the receiver based on the configuration (UdpCmdConfigMgr) and disables the not needed sender based on the application type (Wall does not need to send data to the Wall).

Actually, we are using this implementation only for our Wall-Floor-Camera synchronization. See the demonstration chapter (WallFloorCombinedScene) for a use case of the UdpTransform.

Please have a look at Command Line Configuration to understand how to configure and start the application properly when using the networking system.

Json Protocol

The DeepSpace.JsonProtocol namespace contains simple classes that can be serialized to JSON and back to class instances by the built in UnityEngine.JsonUtility static class. Please have a look at the Unity documentation if you want to use it but do not know it yet. The small protocol consists of:

- EventType: Used to identify what kind of message was received to cast it to the right class. Feel free to add your own event types to the EventEnum.
- AssetIdType: Derives from EventType. Used to give a message an ID (string) which can be connected to a remote object with the same ID.
- TransformAsset: Derives from AssetIdType, containing transform information (position, rotation, scale). Used to change remote transforms.
- SpawnAsset: Derives from TransformAsset, contains the name of the gameObject that shall be spawned. This is not yet implemented.

Extending the Networking

If you want to extend the networking to use it for your own purposes, there are several things you need to have an eye on.

First, write your POD serializable class[10]. You can of course extend this class from EventType and add additional items to the EventEnum, but keep in mind that when this DevKit gets updated and you replace the old version by a newer one, you will lose all the changes you have done within the existing DeepSpace classes. Or you will need to spend a lot of time when looking through diffs and merge everything carefully. The better approach is to create your own serializable class (hierarchy). (Do not extend this class from MonoBehaviour.)

Then have a look at the JsonConverter class, which is responsible for receiving a Json string and creating a data instance. You will need to write two classes that should not extend directly from JsonConverter, but from its child classes JsonConverterWall or JsonConverterFloor, which are responsible for their application type (Wall or Floor). You can now override and implement the method ReceivedPlainMessage(string jsonString, UdpReceiver.ReceivedMessage.Sender sender). In this method, you can use the JsonUtility.FromJson Method to bring the received Json data back into a class instance. If you cannot use the received data, call the base class to pass through the internally used Json messages. Have a look at Assets/DeepSpace/Demo/Scripts/Misc/UsageJsonConverterWall.cs to see the base structure of such a class. You only need to have one JsonConverter per instance, so you can use the WallFloorStart Class (or something similar) to disable the converter that is not needed (e.g. the FloorConverter is not needed on the Wall instance).

The workflow to send and receive a JSON message looks like this:

1. On the sender side, create an instance of your POD serializable class and fill it with the data you want to send over the network.
2. Gain a Json string out of your data, by using JsonUtility.ToJson(yourDataInstance).
3. Get a reference to the sender you want to use and call AddMessage with your string.
4. On the receiver side, the JsonConverter will get the string data and will call your implemented version of ReceivedPlainMessage().

[10] Unity Documentation about serializable fields: https://docs.unity3d.com/ScriptReference/SerializeField.html

5. In your implementation, you will need to detect what kind of data you received and (if the data is, what you expect) put it into a data instance by using JsonUtility.FromJson<YourDataType>(yourJsonString).
6. Pass on the data to your game logic.

If you do not want to use Json, you can of course use the system to send and receive pure string data (e.g. a comma separated list, simple commands, etc.). Then your workflow will only make use of Point 2 to 4.
Please keep in mind that the system expects to receive json data, so make sure that your message starts with '{' and ends with '}'. So, if you want to send a comma separated list, make it like this:
{"my","comma","separated","list",1}
If you want to send you own plain string, send it like this:
{My own plain string.}
All received strings are expected to be UTF-8 encoded.

The JsonConverterWall and JsonConverterFloor class want to have a WallManager and FloorManager as reference. Currently the JsonConverter and the Wall- and FloorManager classes are pretty empty, they will be filled as needs are coming up. However, the idea behind the Manager classes is, to implement basic routines that are needed locally and provide this functionality at the same time to be executed by the receiving JsonConverter classes (similar, but of course not as handy as, RPCs).

3 Demonstration

For easier understanding, some demonstration Scenes have been prepared and will be explained in the following Sections. You can find these scenes in Assets/DeepSpace/Demo/Scenes.

3.1 MobileControlScene

The MobileControlScene shows how to use the Mobile Control to interact with a Unity application. Therefore, the scene contains the following objects:

- MobileControlManager (Prefab): See View Handling.
  - MobileControlTcpManager: See Server Client Communication.
- UsageDemoMobileControl: Containing three Scripts: DemoNavigationView, DemoTouchView and DemoLogPassiveView.
- MoveCube: A simple object.
- Canvas: For UI
  - Panel: Containing a DemoEnableDisableViews script.

The UsageDemoMobileControl contains two active views and one passive view. DemoNavigationView.cs shows how to create a SimpleView with two named and two disabled buttons. The DemoTouchView derives from TouchView and shows how to handle the MobileControl Touch View. With these views, you can move and rotate the MoveCube. The DemoLogPassiveView is a demonstration for passive views and just logs every received message Command.

To test this scene, just press the play button, start the Mobile Control on your Android device and connect to the unity application. (Details, how to do this, can be read in Mobile Control

Configuration.) You can then play around with the Mobile Control. You can register and unregister each view with the UI Buttons. Please note that the MobileControlManager expects at least one non-passive view to be registered (else, the Mobile Control has no view that can be shown to its user).

3.2 PharusTracklinkScene & PharusTuioScene

In the Pharus Demo Scenes (PharusTUIOScene and PharusTracklinkScene) you can see the basic interaction between Unity and the Laser-Tracking-System / Pharus. The scene contains two important items (depending on the scene, the objects are prenamed with Pharus or Tuio):

- TracklinkReceiveHandler / TuioReceiveHandler: A script that takes care about the received data (TrackLink or TUIO) and sends out callback methods for Added, Updated and Removed tracks. The TracklinkReceiveHandler needs an additional instance of UdpReceiver to work. (The TuioReceiveHandler does not need this, as it is instancing TUIO and OSC internally.)
- TrackingEntityManager: This script cares about everything, what is needed for the Unity-Pharus interaction. If required, it is possible to extend from this script to make your own usage of the tracking data. The base version instantiates an object (prefab) of type TrackingEntity to visualize, where a person (or tracked object) on the Deep Space floor is virtually.

To test this scene, start Pharus with a playback track of your choice. (Use one of the prepared batch files or drop a ".rec" file on the Pharus.exe) Then press the unity play button. You will see black circles on the same positions as they are tracked in the room. Basically this is all you need to understand the communication between Pharus and Unity to build your own application with Laser-Tracking interaction.

More details about overwriting the TrackingEntityManager and implementing the Pharus callback methods are described in the Pharus / Laser Tracking section.

3.3 WallFloorCombinedScene

To get the best out of this demonstration scene, you have to change the Aspect ratio in your game view. Therefore, create a new one (in the top left of the game view, expand the aspect ratio and press the "+" button), label it "16:18", set type to aspect ratio, and enter 1600 for width and 1800 for height. Just ignore the visual "Scene is missing a fullscreen camera" warning in the game view. We only use this setup in the editor to get an impression of the whole view. In play mode (or in a build), only one main camera is set active and fullscreen.

This scene contains a lot of SDK Components. Everything is bundled in the form of DeepSpaceSetup child GameObjects:

- CameraSetup: A camera setup with Off-Axis Projections for the Deep Space 8K in combination with a controller script (to move the camera with an xBox controller) and an UdpTransform class to move the camera over network. Actually, not the cameras are moved, but the NetworkTransformTarget, which is a proxy object that waits some frames after sending the translation over network, to hide the delay between a local movement and a remote movement.
- ControllerInput: To handle the xBox controller.
- UdpCmdConfigManager: For command line parameters and scene configuration.

- WallFloorStarter: Disabling the unused camera and predefined objects for the Wall or Floor application.
- UdpManager: To handle the UDP Json package based communication. It contains a UdpSenderToWall, UdpSenderToFloor and a UdpReceiver. The UdpManager disables the Sender, which is not needed (e.g. the UdpSenderToWall if a Wall instance is started).
- WallManager: Containing functionality that can be used locally and remotely. There are two scripts on this GameObject. A WallManager and a JsonConverterWall, which handles received UDP Json messages. This instance will be disabled by the WallFloorStarter if the application is started in FloorMode.
- FloorManager: Same as WallManager, but for the floor. It will be disabled if the application is started as WallMode.

The CameraSetup has many child components. To see a representation of the Wall and Floor in the DeepSpace, activate the Wall/FloorProjectionPlane GameObjects. Never delete these objects, just disable them again, because they are needed for projection matrix calculations. The off-axis projection is calculated in the CameraOffAxisProjection Script on the Observer child object. You can translate the observer to achieve a different effect. The default setting of the observer (sweet spot) is set in the middle of the back on the floor. You can see the view frustum of the wall and floor cameras in different colors in the editor. You can disable this unchecking the “Draw Debug” option or change the line colors. If you want to resize the view frustum, change the Near and Far clipping plane of the WallCamera. The FloorCamera will be resized the same way if the NearFarSync (on the CamereaOffAxisProjection Script) is set to Equalize, which is the recommended setting.

The controller input is a singleton that needs to exist in the scene. It is used as a wrapper for the unity input system especially for an xBox style controller. This object is used in the CameraController Script, which is set on the CameraSetup GameObject. You can have a look at this script to see how to handle the controller input.

The UdpCmdConfigMgr Object has two scripts on it. The UdpCmdConfigMgr script and a usage script. This child class of ConfigManager has additional options for a UDP address and port. You can set the default settings directly in the editor. If you use the command line parameters, this options will be overwritten. The UsageUdpCmdConfigMgr script just shows how to access the configuration values and how they behave by logging the runtime values.

In praxis, we want to copy the Unity release binary and data folder on the Wall and Floor Server and integrate it in the Deep Space starter system. Therefore, we need to know what parameters can be set. In this case, you could call the application by: application.exe -mode=Wall -udpAddress=127.0.0.1 -udpPort=1212
As already mentioned, please keep in mind that we use the string.ToLower method in the configuration scripts to lower keys and values, which makes the command line arguments case-**in**sensitive.

The WallFloorStarter has a little script on it to disable GameObjects that are not needed for a specified instance of the application (Wall or Floor). In this case, if you start the Wall (by setting the default value in the UdpCmdConfigMgr ApplicationType to Wall), the Floor Camera will be disabled, the Wall Camera will be changed to full screen, and the UdpReceiver will be disabled. If you start the application for the Floor, the Wall Camera will be disabled, the Floor Camera will be set to fullscreen and the UdpSender will be disabled. This is just for demonstration. You can implement your own WallFloorStarter (derive from this class) to do whatever you need.

Making the server (Wall) to a sender and the client (Floor) to a receiver is important for the basic implementation of the UdpTransform, which is put on the CameraSetup. This simple implementation of UdpTransform checks if the UdpSender is enabled or not. If it can send, it sends the position. If it can receive (UdpReceiver is existing and enabled), it sets the transform as requested by the message. You can do your own implementation and check for the Wall or Floor ApplicationType from the ConfigManager. However, the NetworkId of the UdpTransform is set to 1, so if you want to use this script on other objects, do not forget to set another ID.

3.4 DebugStereoCanvas Prefab

Finding the right Stereo Settings in the Deep Space 8K may require some trial and error. Therefore, a prefab with some uGUI elements has been prepared to change the stereo settings at runtime. Just drop it into a scene to make the final tests in the Deep Space 8K easier. Press space to enable or hide the UI.

You will need to create a UI EventSystem GameObject, if you do not already have one, to make the interaction with the GUI elements possible.

3.5 VRPN Demo

The included VRPN example scene “VRPNTransformation” contains a simple application that works with the aforementioned VRController app in the Deep Space. It consists of a cube whose color and position can be changed through VRPN data coming from the VRController.

The “VRPNTransformationDemo” script contains the actual implementation. It continuously retrieves the state of three buttons as well as the tracking values (position and rotation) for channel 0. When a button gets activated, it draws the corresponding color on the cube or it respectively changes the position and the rotation of the cube when the user touches the virtual touchpad area of the VRController or changes the orientation of the Android device.

4 Integrate the DevKit into an existing project

If you do not want to use the project, that comes along with this SDK, as your starting point, because you already have an existing project and just want to integrate the Deep Space DevKit functionality in your project, there is an easy workflow to do so.

4.1 Preparation

At first you need to change your Editor Project Settings (Edit – Project Settings – Editor). Under Asset Serialization, change Mode from Mixed to Force Text, if you did not already do so.
This is something we recommend in general, because git (and other version control systems) are then able to compare and merge binary assets like settings and scene files.

Now you can import the DevKit package. If you did not get a *.unitypackage file but a ZIP file with the project instead, you can just copy the complete Assets/DeepSpace folder from the DevKit project into your Assets directory.

4.2 Project Settings

To create a build that can use all benefits of the Deep Space 8K, you need to modify some of the project settings. To make it easy for you, we created a Deep Space Settings Editor, that is explained in detail in section Project Settings.

4.3 Use DevKit Assets

Once you have imported the files from the DevKit, it is pretty easy to use them.

On the one hand, you can just use the prepared scripts and prefabs. On the other hand, you can use a little bit more complex combinations of prefabs, as you can see them in the WallFloorCombinedScene. In this Scene, everything is prepared to have an off-axis projected camera for the Deep Space Wall and Floor, where the camera is controllable via a game pad.
If you want to use the prefab combination from WallFloorCombinedScene as your camera, you can do so by dragging prefab by prefab into your scene and connecting all attributes correctly, or you can do a simple move: Open your existing scene, then open the WallFloorCombinedScene (to stick with this example) additionally, by right clicking on the Scene Asset and selecting “Open Scene Additive”. Now you can drag and drop the (or copy and paste) the wanted objects from the DevKit Scene to your scene. Afterwards, just close (right click on the Scene Header and select “Remove Scene”) the WallFloorCombinedScene without saving, so that you can do this again if needed. Afterwards you can remove your original camera by the DeepSpaceSetup, you just copied, and save your own scene.

For details, how to use these assets in general, please have a look at Chapter 3 Demonstration.

5 Known Unity Bugs

Some Unity bugs are really nasty and prevent the Deep Space 8K from showing its full potential. This information might be out of date if you are using a newer Unity Version than defined above.

5.1 Unity UI is not working in Stereo

In Unity 5.4 and earlier versions, the UGUI (which was introduced with Unity 4.6) is only visible on one eye when using a Screen Space Canvas while rendering in stereo. This got fixed with Unity 5.5, please consider to update your project.

If you want to stick with your older versions: World Space Canvases are rendered correctly. The Unity 2D Sprite System is working fine too. We have not yet tested nGUI or other UI Plugins in the Deep Space 8K. You can reproduce this issue on any stereo screen or projector.

5.2 Stereo with off-axis projection is not working with Unity 5.6.x & 2017.x

Short answer: Enable HDR rendering on all your cameras. Afterwards it will work.

Long answer:

A lot of Unity Versions are having issues with displaying non head-mounted stereo correctly. Unity fixed this within some versions and broke it again in later ones. In version 5.6.2p3 (Patch Release) and ongoing 2017.x versions, stereo is working in general but it is not working together with the off-axis projection, which is used in the Demo WallFloorCombined Scene.

Please enable HDR rendering on your cameras. This will make the stereo work with the off-axis projection again. If this brings up other issues for your application, please use the older Unity versions 5.4.x or 5.5.x without HDR rendering.

5.3 Stereo is not working with Unity 2017.2

Unity released Unity 2017.2 and 2017.2p1 without the option to build with "Stereo (non head-mounted display)" by accident, so these versions cannot be used to create stereo builds for the Deep Space. The non head-mounted stereo option was added back to Unity in version 2017.2p2 and will be available in future versions.

5.4 Builds cannot be started with Unity 2018.3.6 and above (Unity 2018.3.x)

We realized that builds that are created with Unity 2018.3.6 and above (within the 2018.3 series) cannot be started in Deep Space. The application crashes immediately after starting it independent of its content. If you want to work with 2018.3, we recommend using 2018.3.5. The latest 2018.4.x versions (tested with 2018.4.22) can be started, but stereo does not work. However, we have successfully used version 2019.3.13 to create a build that works and supports stereo. We have also seen 2019.2.x versions successfully running in the Deep Space with stereo. Please note, that with 2019.2.x, stereo with OpenGL is declared deprecated, but we plan to upgrade the Deep Space 8K to Windows 10, where it is possible to use DirectX 11 stereo, and we are keen that this will be the case before Unity drops the OpenGL support completely.

6 Q&A

If you have any questions, this chapter with Questions and Answers might help you along. If you want to contact us, please keep in mind that we cannot guarantee to answer every email we get about this DevKit and the Deep Space 8K immediately. There is no official support as long as we (you and us) do not have a contract that says otherwise.

If you want to write an email, you can reach us under DeepSpaceDev@ars.electronica.art

Q: I have built an application and want to test it in the Deep Space 8K.
A: Sounds good. We would love to learn more about your project and if feasible see a preview. Please write an email to get in touch with us. For testing it is always a good idea to implement some short keys, debug UI or command line parameters to change some settings like stereo options, the camera aspect ratio or connection IPs and ports. For changing the stereo settings, some UI elements have been prepared in the SDK (see DebugStereoCanvas Prefab).

Q: I have Deep Space 8K related questions that are not answered in this or referenced documents.

A: Do not hesitate to write us an email at DeepSpaceDev@ars.electronica.art

Q: I am working on a project using the laser tracking system. How can I get some tracking input to test my application?
A: You can download the Pharus Player Application from
https://files.aec.at/index.php/s/W85pKmkTxgkS3JF

Q: Where can I find the APK to install the Mobile Control App on my Android device?
A: You can get the Mobile Control APK on request. Please contact DeepSpaceDev@ars.electronica.art

Q: I installed the Mobile Control on my Android device, but it does not work.
A: This software is optimized for the Xperia Z1 Compact and build for Android 5.0. If you have another device or Android Version, you might have problems or features might not work as expected. I am sorry to say that we are not planning to open source the android application and will not optimize this app for other devices as long as it works on the devices in the Deep Space 8K.

Q: I found a bug in the DevKit or incorrectness in the documentation.
A: Do not hesitate to write us a detailed email about the bug. We will do our best to fix all issues.

Q: I implemented functionality that might be useful for others.
A: We are always happy if you want to contribute. If you have some scripts or assets you think are helpful for future DeepSpace applications, feel free to share it with us and future DevKit users. By the way: the original Unity Pharus integration has been implemented by third party developers.

Q: I find some of the Unity scripts very useful. Can I use them in other (private / commercial) projects beside the ones for the Deep Space 8K?
A: For all code and assets in the Deep Space 8K SDK, except parts that belong to 3rd parties, please refer to the licensing section.

Q: I like the Software that comes together with this SDK (Pharus, MobileControl). May I use this software (private / commercial) beside projects for the Deep Space 8K?
A: Pharus and the Mobile Control are only client programs and the versions you can download only have reduced functionality. If you are thinking about creating your own laser tracking system and using Pharus as Software, this will not work with the client you have. The Mobile Control is only optimized for one phone and definitely not thought for public use. However, if you are interested in the full version of Pharus and our Tracking System, for using it for different purposes than the DeepSpace, please contact us: DeepSpaceDev@ars.electronica.art

7 Licensing & Support Terms and Conditions

7.1 License

Our software is provided without any warranty under the MIT license.
Parts of this project have not been created by us or are based on other projects and are therefore published without any warranty under the specific license of their owners.

7.2 Third Party License

The following parts of the Deep Space Dev Kit are based on or are provided by third party projects and are published under their own license, if specified by their owners.

Unity Pharus Tracking

The Unity Pharus Tracking implementation (Source in DeepSpace/Scripts/LaserTracking) is based on a project by Andreas Friedl[11], who was at the time studying at the UAS Upper Austria. The open source repository does not specify a license, but we think it is fair to mention this attribution to the Deep Space Dev Kit.

TUIO & OSN.NET

The original TUIO implementation (Source in DeepSpace/Scripts/3rdParty/TUIO) and the dependency for this project, OSN.NET (Source in DeepSpace/Scripts/3rdParty/OSC.NET), can be found in Martin Kaltenbrunners[12] github repository. A license file (GNU Lesser General Public License v3.0) is provided within his git repository.

VRPN

The VRPN implementation (Source in DeepSpace/Scripts/3rdParty/VRPN) has been created by Scott Redig[13]. A license file (MIT License) is provided within his git repository.

7.3 Support

We currently do not offer premium or priority support for developers using the Deep Space 8K SDK or developing for the Deep Space. Nevertheless, it is possible to contract Ars Electronica Futurelab for commercial development and/or support.

Any non-commercial project is supported for free per email depending on our resource availability.

The support email is DeepSpaceDev@ars.electronica.art

8 Release Notes

This SDK and Documentation Version was released in May 2020. It contains a collection of scripts and prefabs that have been found to be very useful when developing an application for the Deep Space 8K.

The version number of this document always corresponds to the version of the Deep Space Dev Kit.

Version 1.4.1
Initial Release.
Minor rewrites and reformatting.
Specified licensing.

[11] https://bitbucket.org/rohschinken/unitytrackingclient/src/master/
[12] https://github.com/mkalten/TUIO11_NET
[13] https://github.com/arviceblot/unityVRPN