RevSim

Revolution Simulator

Jörg J. Buchholz

November 16, 2019

1 Manual

1.1 Introduction

RevSim (**Rev**olution **Sim**ulator) is a real-time, pilot-in-the-loop, virtual reality simulator of a REVOLUTION [1] kite. RevSim has been programmed in UNITY [2] in C#, compiled for OCULUS [3] headsets (RIFT, ...), and is available in the OCULUS Store with an OCULUS key or via sideloading [4].

A REVOLUTION kite basically[1] consists of two connected triangles (figure 2.6) that are controlled via four lines by two handles. By symmetrically and/or asymmetrically rotating the handles, you can control the angles of attack and the angles of sideslip of both triangles independently. As a result, you can pin the kite at every[2] position and attitude in the sky.

1.1.1 25 years later

A quarter century ago, we created the first ancient RevSim version (figure 1.1).

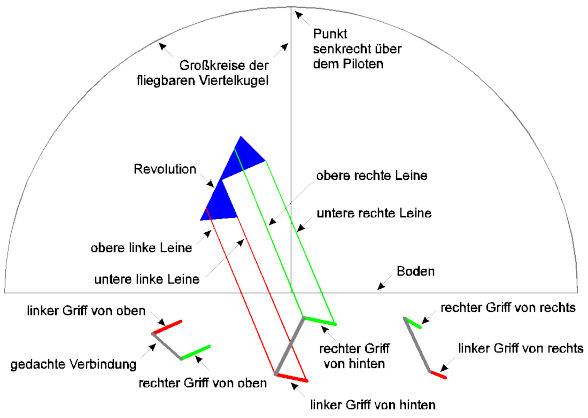

Figure 1.1: 1993 RevSim (original screenshot in German)

It used EGA-graphics (320 × 200 pixels) and the mouse to control the kite.

[1] Yes, real-world Revolutions have flexible spars, bulging sails, wind-permeable mesh grids, stabilizing bridles, ... This first RevSim version has two triangles.

[2] RevSim even allows you to alter the line length with a thumbstick on your controller, offering you a full quarter cube of flying space, instead of the quarter spherical surface of a real-world kite.

A few things have improved since then: Today, we use 3D modeling and simulation tools like UNITY and virtual reality headsets like the OCULUS (RIFT, ...) to create realistic 3D images and sounds, allowing us to deeply immerse into the scenery with an adequate perception of depth. But most of all, we hold two virtual reality controllers in our hands that are precisely tracked with respect to their position and attitude in 3D space. Thus, for the first time, we are offered a nearly perfect[3] way to emulate the handles of a quad line kite.

1.2 Installation

You can download RevSim directly from [4] and sideload it by allowing the VR environment to install and run apps that are not available at the OCULUS store. Alternatively, you might ask us (buchholz@hs-bremen.de) to send you an OCULUS key you can use to download and install RevSim officially from the OCULUS store.

1.3 Operation

Just put on the headset, take both controllers in your hands and you are ready to fly the kite. If you look down at your hands in VR, you see two handles (figure 2.19) moving in sync with the controllers in your hands and two help plates explaining the basic button and thumbstick functions:

Right controller button A: Reset kite position

Right controller button B: Reset all

Right controller thumbstick forward: Increase wind speed

Right controller thumbstick right: Increase air drag

Left controller button X: Tutorial (constant drag, ...)

Left controller button Y: Help/data/off

Left controller thumbstick forward: Increase line length

Keyboard key 1: Music and sound on/off

Keyboard key 2: Lines on/off

Keyboard key 3: Controller help on/off

You can reset the game, alter the wind speed, the line length, and the air drag and step through four tutorials that might help you control the kite.

[3] The only thing missing is a true feedback of the forces the kite exercises on the lines and thus on the pilot; but that's another story ...

2 Objects

This chapter[1] explains the modeling and the hierarchy of the objects we use in UNITY (figure 2.1).

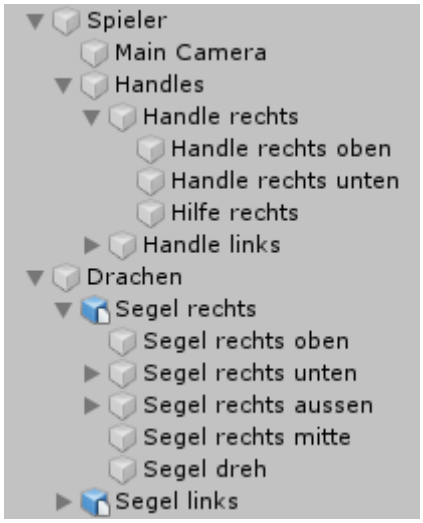

Figure 2.1: Objects hierarchy

2.1 Coordinate system

UNITY uses a left-handed coordinate system:

- The thumb of your left hand points to your right (red X-axis in figure 2.2).
- The index finger of your left hand points up (green Y-axis in figure 2.2).
- The middle finger of your left hand points away from you (blue Z-axis in figure 2.2).

[1] If you are just a simple user of RevSim, you might not want to read any further; the rest of this document is just boring modeling and programming stuff. And, we have to apologize for the fact that we use German names for objects, functions and variables for ... reasons.

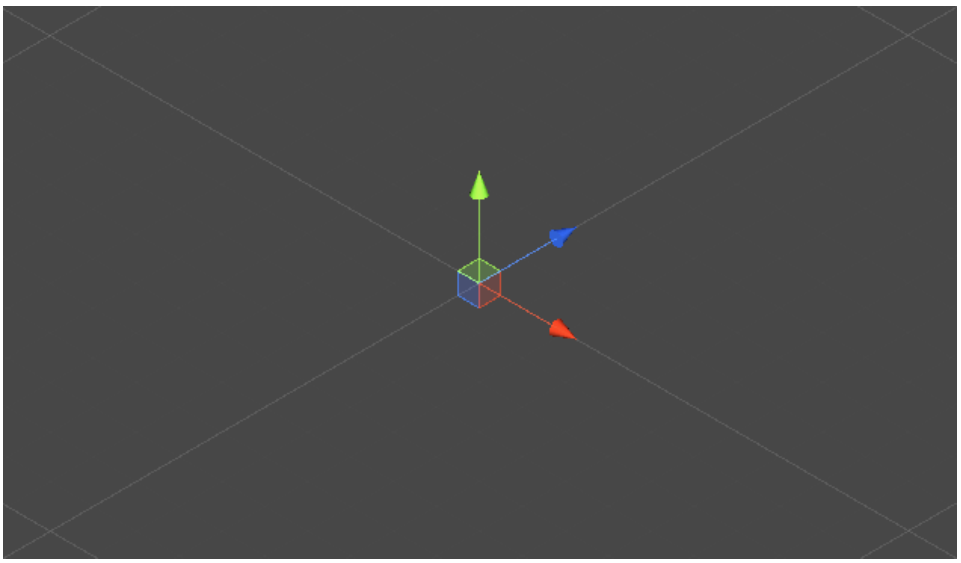

Figure 2.2: Coordinate system

2.2 Kite

The kite consist of two isosceles triangles[2] connected to each other by a hinge joint and to the handles by four configurable joints (representing the lines). If you zoom in at figure 2.4 you can recognize the joints as small brown and gray arrows indicating the free rotational axes of the joints.

Figure 2.3: Kite with joints

[2]To be precise, we model the kite as a true three-dimensional object by extruding each triangle into a triangular prism with a thickness of a few millimeters.

Since we will position the player with the handles at the origin of the X-Z-plane in section 2.3, we initialize the Z-component of the kite's `Position` at 10 (figure 2.4) in order to locate the kite 10 m in front of the player[3] .

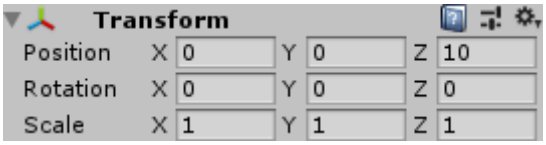

Figure 2.4: Kite: `Transform`

2.2.1 Right sail

UNITY can only create a few 3D primitives by itself:

- `Cube`
- `Sphere`
- `Capsule`
- `Cylinder`
- `Plane`
- `Quad`

For the sails we need two triangular prisms (figure 2.3) that we cannot directly derive from UNITY'S primitives. Fortunately, it is very easy to model a "complicated" 3D object in another 3D modeling program and then import it as a new object into UNITY. Therefore, we use SKETCHUP [5] to extrude a triangle into a triangular prism (figure 2.5).

[3] UNITY'S physics engines use the International System of Units (m, kg, s, ...); one length unit corresponds to one meter, gravity is $9.81 \frac{m}{s^2}$ ...

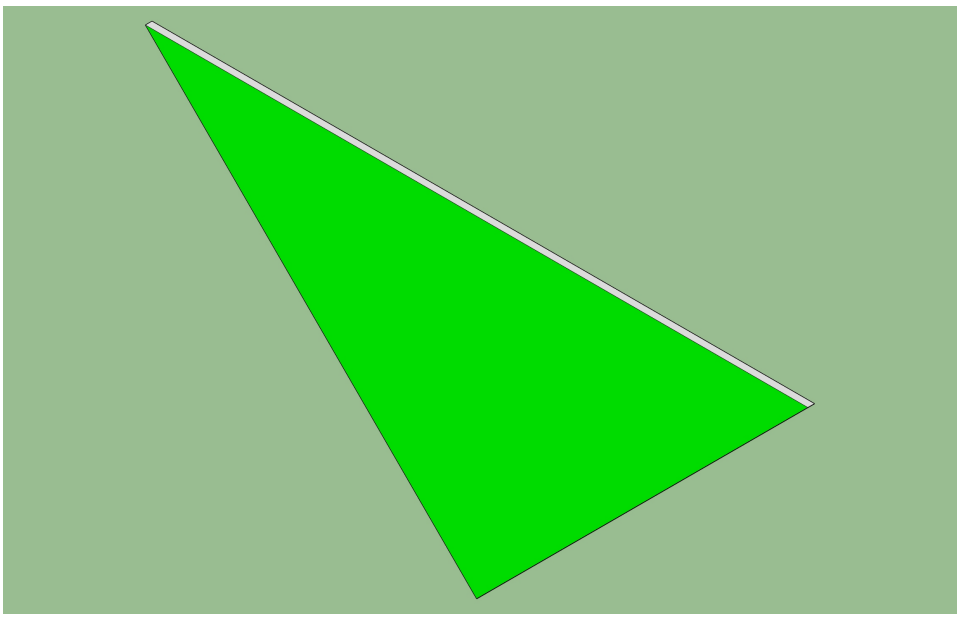

Figure 2.5: Sail in SKETCHUP [5]

Actually, we do not even need to explicitly import the SKETCHUP file; we just use the mouse to drag and drop the .skp file into UNITY'S asset window and then copy the asset into the scene twice to come up with the sail in figure 2.5.

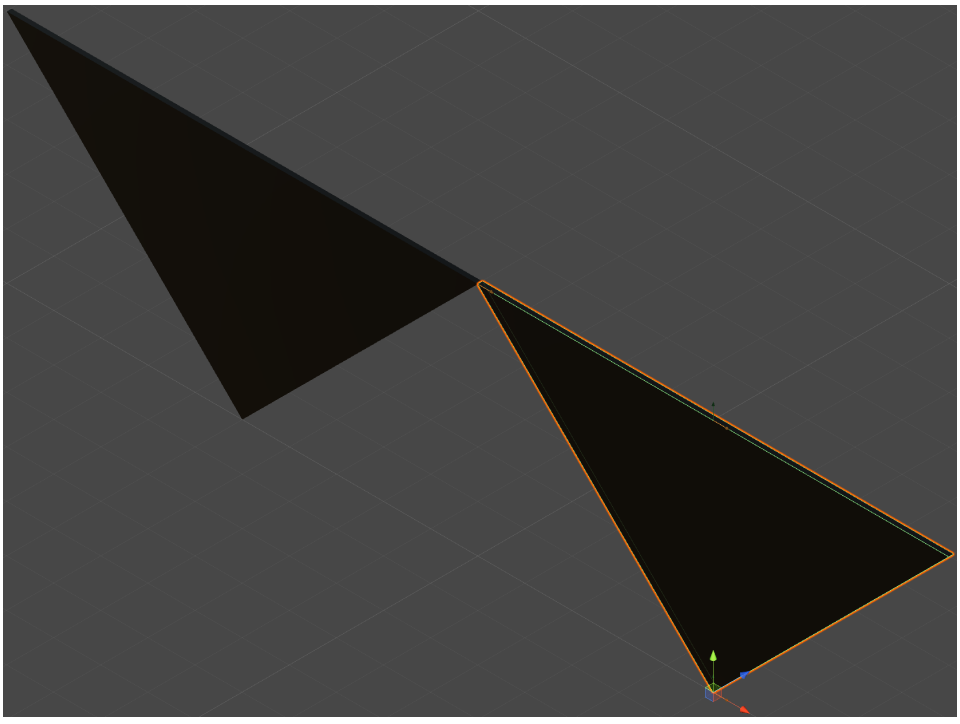

Figure 2.6: Right sail in UNITY

As indicated in figure 2.1, the right sail (`Segel rechts`) is a child of the kite (`Drachen`) in the object's hierarchy. Its pivot point is at its bottom vertex which therefore has to have an X-position of half a meter (to the right) with respect to the kite itself (figure 2.7), assuming the sail has a span of one meter. Accordingly, the pivot point of the left sail has an X-position of -0.5.

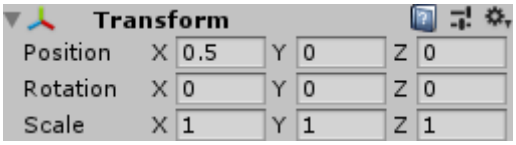

Figure 2.7: Right sail: Transform

Unity's powerful physics engines perform collision detection and solve the differential equations of motion of all objects numerically; all we have to do is to add a `Rigidbody` component (figure 2.8) to the object and define the aerodynamic forces acting on the object in section 3.1. In the `Rigidbody` component, we give the sail a `Mass` of 0.2, define the `Linear` and `Angular Drag` coefficients as 1 and switch on standard gravity ($g_Y = -9.81$).

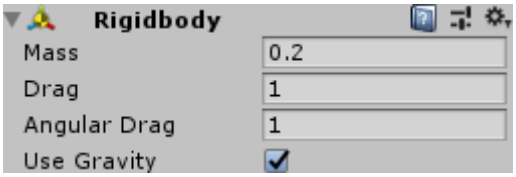

Figure 2.8: Right sail: Rigidbody

The right sail is connected to the left sail by a `Hinge Joint` which allows the right sail to rotate about its X-axis with respect to the left sail. Therefore, we add a `Hinge Joint` component to the right sail and define in figure 2.9 that

- the `Connected Body` is the left sail (`Segel links`)
- the right sail `Anchor` is located in the upper left corner of the right sail
- the hinge joint allows a free rotation about the X-axis
- the `Connected Anchor` is located in the upper right corner of the left sail

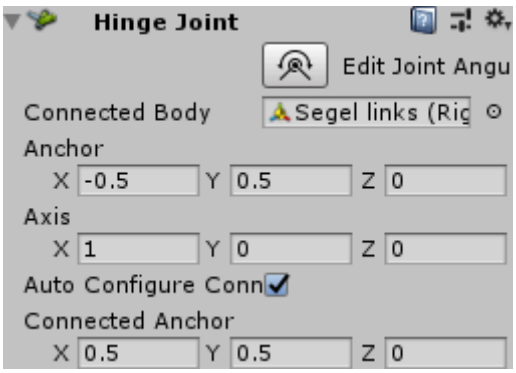

Figure 2.9: Right sail: `Hinge Joint`

The right sail is connected to the right handle by two lines. Unfortunately, lines are still quite difficult to simulate. Lines are flexible but non-stretchable and have a very small mass. The correct way to simulate a line would be to use a huge number of very small, light cylinders connected via very stiff springs. Since the natural frequency ω_0 of a mass spring system is

$$\omega_0 = \sqrt{\frac{c}{m}}$$

where c is the spring constant and m is the mass, such a stiff system would have to be integrated with a very high sampling rate which is inconsistent with the fixed sampling rates of current virtual reality systems (e. g. 90 Hz).

Therefore, we add two `Configurable Joints` (figure 2.9, figure 2.11) to the right sail that connect the kite to the right handle (`Handle rechts`). The trick is to limit the `X Motion`, the `Y Motion`, and the `Z Motion` in figure 2.10 to a `Linear Limit` of 10 which is the initial line length. This allows the kite to float freely at any position in 3D space as long as its distance to the handles is less than 10 m. The graphical representation of the (possibly sagging) lines is later done in section 3.11.

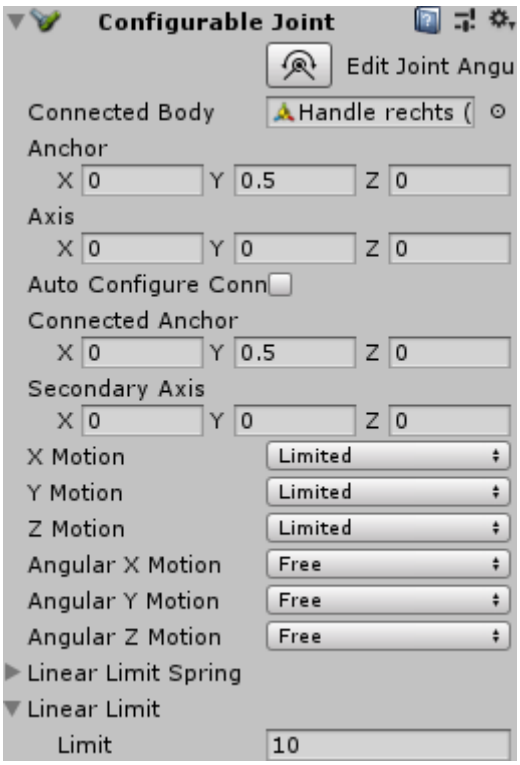

Figure 2.10: Right sail: Upper `Configurable Joint`

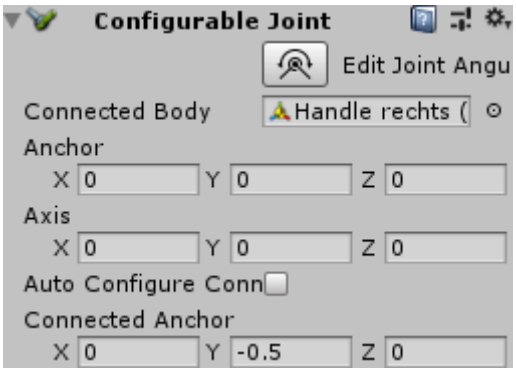

Figure 2.11: Right sail: Lower `Configurable Joint`

The left sail has corresponding mirrored properties.

2.3 Player

As indicated in figure 2.1, the player (`Spieler`) is the parent of the camera (section 2.3.1) and the handles (section 2.3.1). We give the player a height of 1.5 (figure 2.12), which roughly corresponds to the eye height of the author.

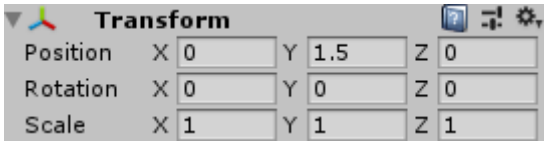

Figure 2.12: Player: `Transform`

2.3.1 Camera

During the game, the positions and attitudes of both the camera and the handles will be defined by the actual positions and attitudes of the headset and the real-world controllers (section 3.7 and section 3.8). Therefore we can simply define the initial `Position` of the camera at the origin of its parent (figure 2.13).

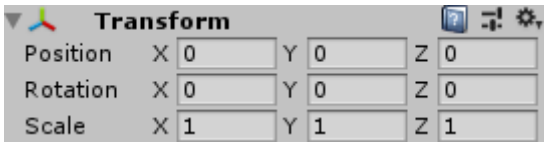

Figure 2.13: Camera: `Transform`

The camera object has a predefined camera component (figure 2.14) that we use out-of-the-box with the slight adjustment of decreasing the `Near Clipping Plane` value from 0.3 to 0.1, in order to prevent the handles from disappearing if we hold them closer to our face.

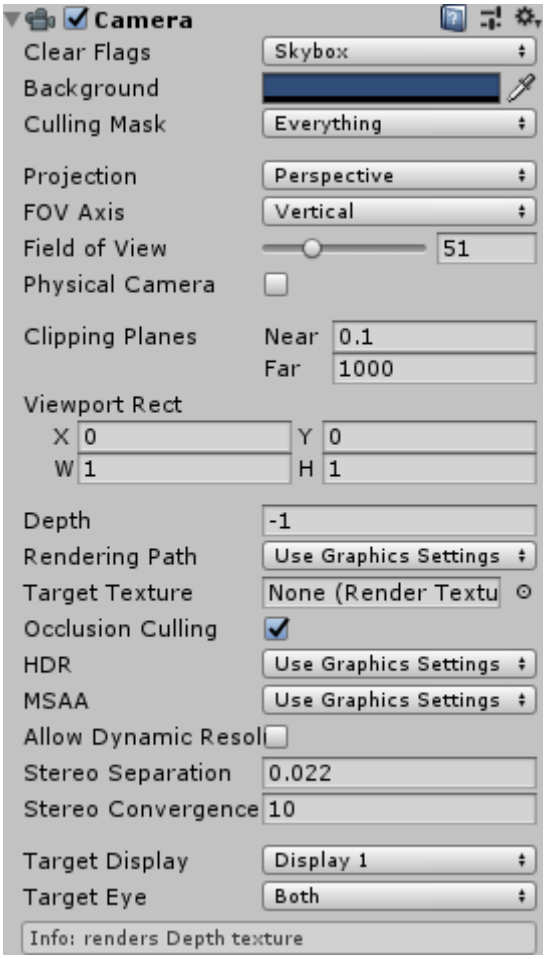

Figure 2.14: Camera: `Camera`

2.3.2 Handles

We create a Handles object (figure 2.15) as a parent for the right handle (`Handle rechts` in figure 2.1) and the left handle.

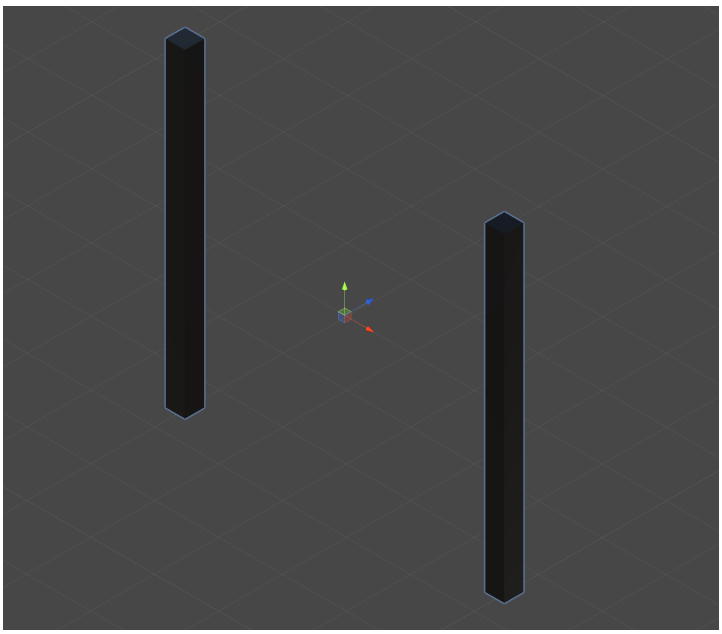

Figure 2.15: Handles

During game development, we can change the `Rotation` angles (figure 2.16) of the Handles object in order to pitch, roll, and yaw both handles simultaneously about the pivot point in figure 2.15.

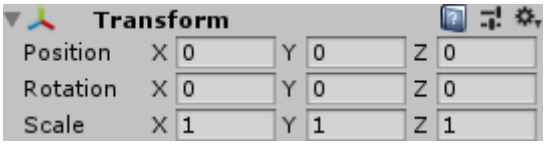

Figure 2.16: Handles: `Transform`

Both handles are scaled (figure 2.17) `Cubes`[4] leading to their elongated shape in figure 2.15. The initial `Position` of the right handle in figure 2.17 is only useful during game development; in the game, the handle objects always synced with the real-world controllers (section 3.7).

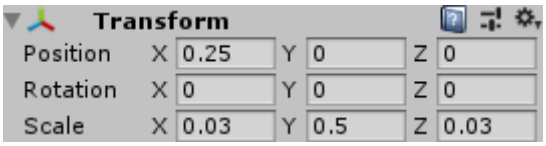

Figure 2.17: Right handle: `Transform`

[4]Yes, real Revolution handles are curved cylinders resulting in a nonlinear control characteristic. Maybe in one of the next RevSim versions ...

The handle has to be a `Rigid Body`; the joint (figure 2.10) between the sail and the handle can only connect two rigid bodies.

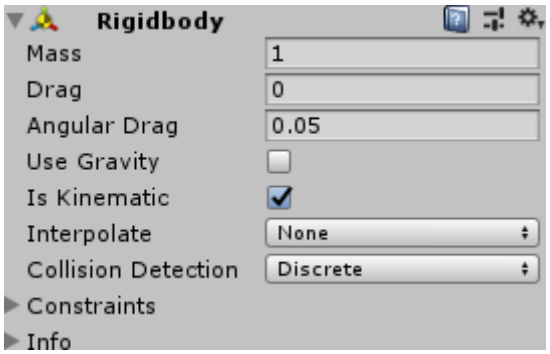

Figure 2.18: Right handle: `Rigidbody`

On the other hand – as already stated – the handle is directly synced with the real-world controllers. The parameters in figure 2.18 (`Mass`, `Drag`, ...) are therefore completely irrelevant; we just leave them at their initial values.

2.3.2.1 End caps and help plates

Figure 2.19 shows white end caps at the top (and the bottom) of the handles and two help plates glued to the top outer sides of the handles.

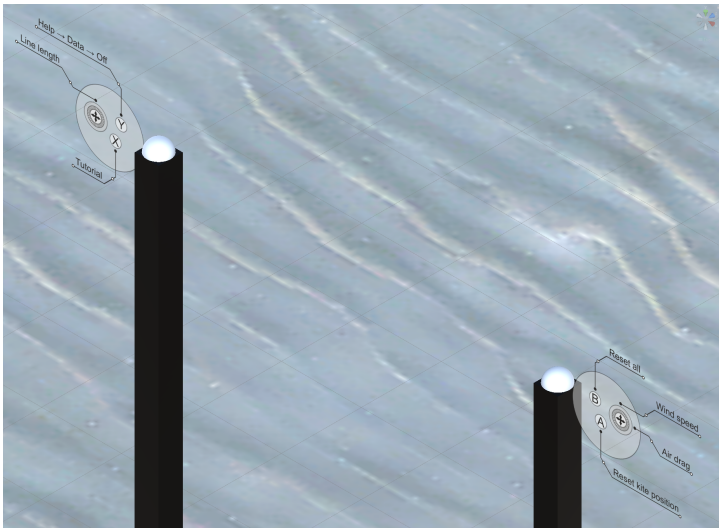

Figure 2.19: End caps and help plates

End caps The end caps are primitive `Spheres`, are children (`Handle rechts oben` and `Handle rechts unten` in figure 2.1) of the handle and do not really have physical

functions; they merely exist for decorative purposes. Since the upper end cap of the right handle is a child of the right handle it inherits the `Scale` of its parent (figure 2.17). Therefore, its `Position` in figure 2.20 has to move up 50 % with respect to the handle length. For the same reason, the `Scale` of the end cap has to invert the `Scale` of the handle in its Y-component (figure 2.20).

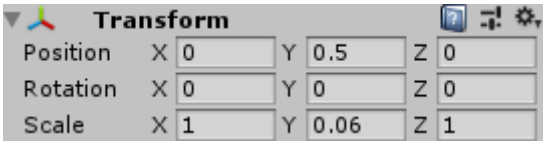

Figure 2.20: Upper right end cap: `Transform`

Help plates The help plates in figure 2.19 inform the inexperienced user about functions of the buttons and thumbsticks on the controllers. They are children (`Hilfe rechts` in figure 2.1) of the handles too and therefore have to be positioned and scaled considering the Scale of their parents (figure 2.17) too (figure 2.21).

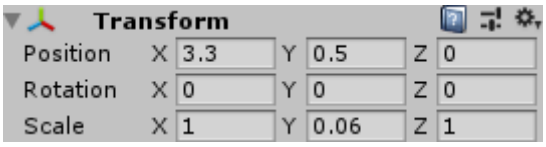

Figure 2.21: Right help: `Transform`

We create the help plates as objects with 50 % transparency in CorelDraw [6], export them as .png files, import them into Unity's Assets folder, and change the Texture Type in the Import Settings to Sprite (figure 2.22).

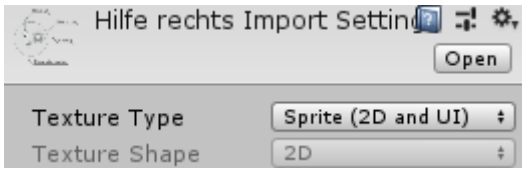

Figure 2.22: Right help: `Import Settings`

2.4 Ground

The ground (`Boden`) is just a square with an edge length of 500 m (figure 2.23).

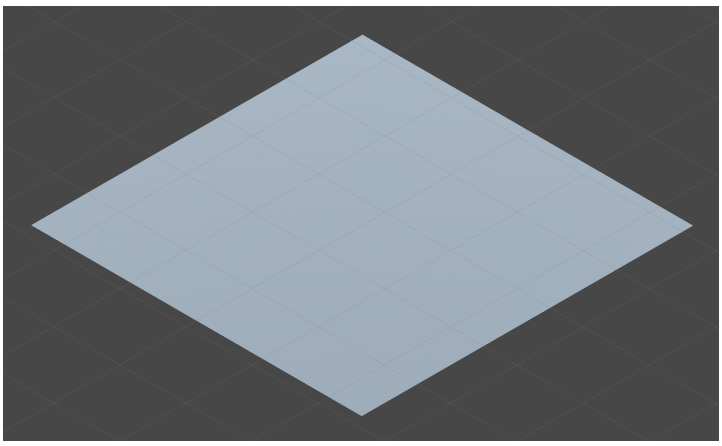

Figure 2.23: Ground

Since UNITY'S `Plane` primitive is a 10×10 square, we have to apply a Scaling factor of 50 (figure 2.24). As depicted in figure 2.31, we want the kite to be 10 m in front of the player ($Z = 0$) and the coastline to be 5 m in front of the kite. The center of the ground therefore has to be at a Z-position of -235 (figure 2.24)

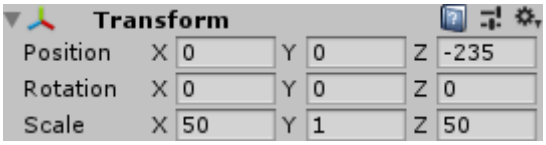

Figure 2.24: Ground: `Transform`

Since the ground itself does not move, it does not have to be a rigid body with Mass, Drag, ... However, if we want the kite to stay above the ground we have to give it a `Mesh Collider` (figure 2.25) that applies an upward force to the kite if it touches the ground.

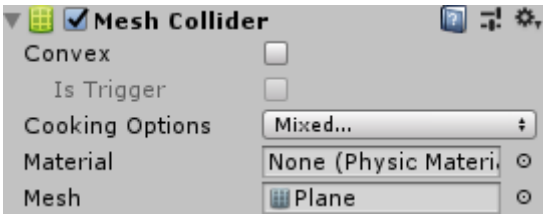

Figure 2.25: Ground: `Mesh Collider`

We want the ground to look like a sandy beach. One of the great pleasures of working with UNITY is the fact that you can download user generated packages of about every imaginable kind from UNITY'S Asset Store; most of them for a small amount of money,

some of them even for free. We download a free Sand textures pack and use its Sand pattern 01 as the material for the `Mesh Renderer` (figure 2.26).

Figure 2.26: Ground: `Mesh Renderer`

Even though the material is just a 2D texture, it allows the renderer to create very natural looking ground waves (figure 2.27).

Figure 2.27: `Ground` detailed

2.4.1 Wind Zone

We use a `Wind Zone` object (figure 2.28) to make the palms (to be implemented in section 2.8) bend in a realistic animated fashion.[5]

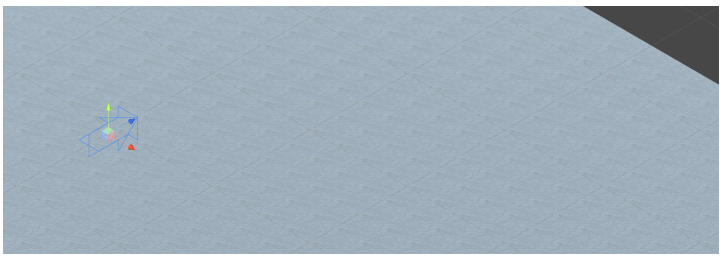

Figure 2.28: `Wind Zone`

[5]Unfortunately, a `Wind Zone` can only move particles, trees, grass and other terrain objects but does not have aerodynamic effects on rigid bodies like the kite. Therefore, we have to compute the aerodynamic forces on the kite by ourselves in section 3.1. We expect UNITY to implement aerodynamic forces on all objects in a future release ...

Since we implemented the `Wind Zone` as a child of the ground, UNITY automatically adjusts the `Position` and Scale values of the `Wind Zone` with respect to its parent in figure 2.29.

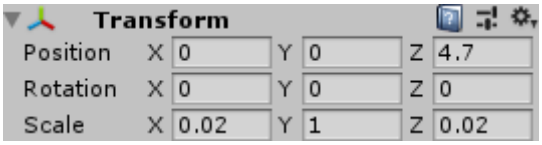

Figure 2.29: Wind Zone: `Transform`

On the other hand, the `Position` and `Scale` values of the `Wind Zone` are completely irrelevant as long as we use the `Directional Mode` of the `Wind Zone`. The initial parameters in figure 2.30 are out-of-the-box; we dynamically adapt them in section 3.13 if the user changes the wind speed. The constant direction of the wind is defined by the `Rotation` value of the `Wind Zone`.

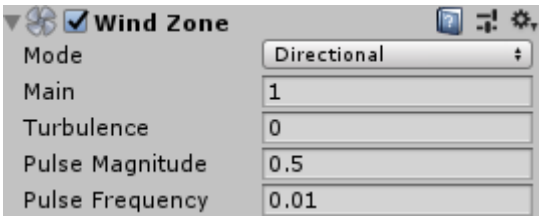

Figure 2.30: Wind Zone: `Wind Zone`

2.5 Sea

UNITY offers a great animated `Water Prefab` in its `Standard Assets` package. You can play around with over 50 parameters (`Color`, `Texture`, `Reflection`, `Lighting`, `Amplitude`, `Frequency`, `Speed`, ...) to adapt the "pounding waves" to your needs.

As indicated in figure 2.31 the coastline between ground and the sea surface is at a Z-coordinate of 15.

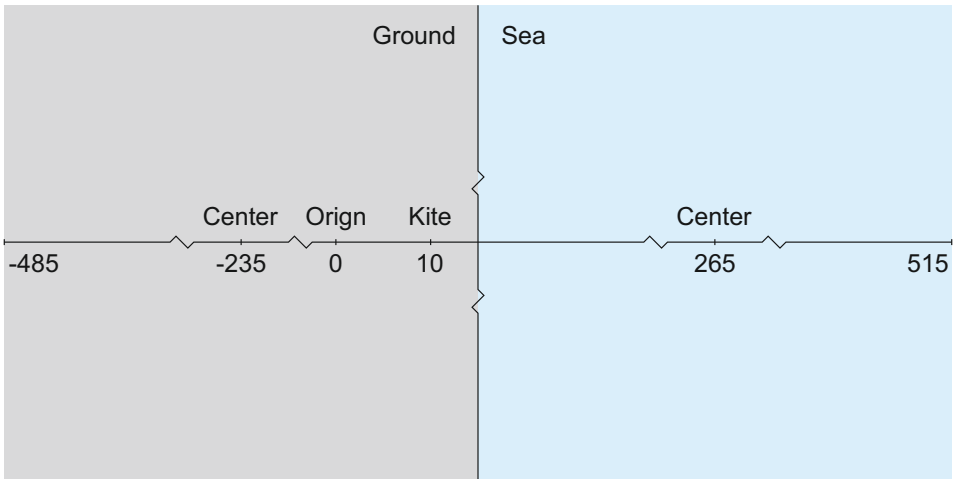

Figure 2.31: Ground and sea position from above

We want the sea to be a 500 × 500 square – just like the ground. To make it adjacent to the ground, we have to center it at a Z-position of 265 (figure 2.32) and scale it accordingly.

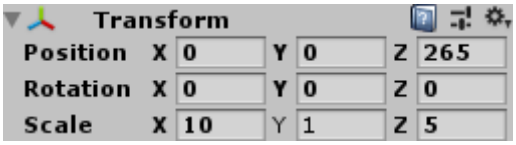

Figure 2.32: Sea: `Transform`

2.5.1 Sea bottom

We want to allow the user to dip the kite into the sea. Therefore, the visible sea surface does not have a collider. On the other hand, we want to simulate an invisible solid sea bottom with a 3° slope (figure 2.33) that makes the kite partly visible when the user submerges it into the shallow waters near the coastline.

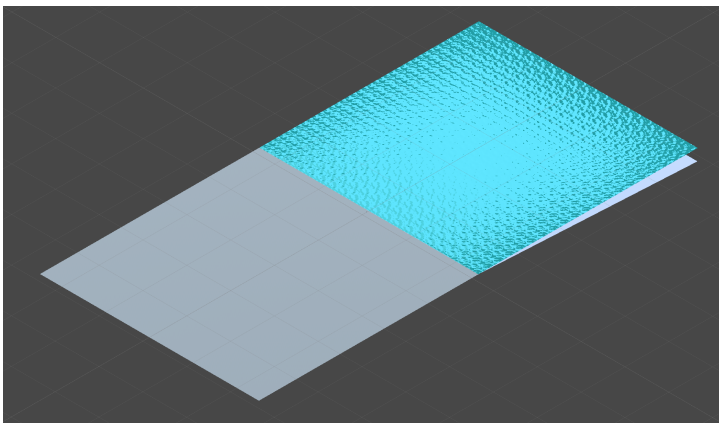

Figure 2.33: Ground, sea surface, and sea bottom

figure 2.35 shows that the sea bottom is just another 500×500 plane with a rotation about the X-axis of 3°. Since the rotation is done about the center (pivot point) of the plane ($\begin{bmatrix} 0 & 250 & 250 \end{bmatrix}$ in the local reference frame), we have to calculate the position of the pivot point using elementary trigonometry[6].

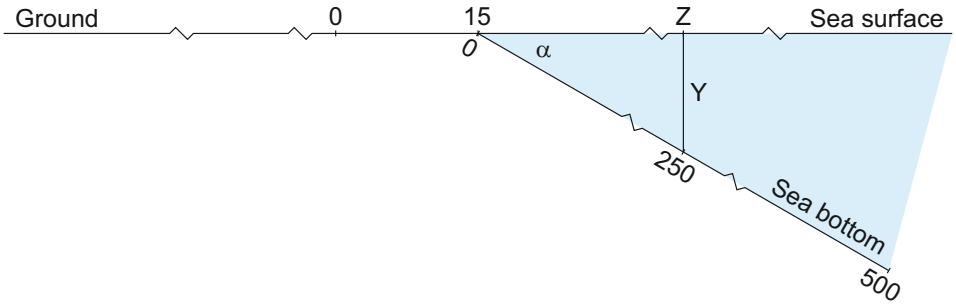

Figure 2.34: Sea bottom position and rotation looking along the coastline

In figure 2.34 we can see the relation between the sea bottom slope angle α and the Y-component of the sea bottom's pivot point:

$$\sin \alpha = \frac{Y}{250}$$

With a small sea bottom rise angle of $\alpha = 3°$, we come up with the resulting Y-component:

$$Y = 250 \cdot \sin 3° = 13.08399$$

[6]If we did not exactly calculate the position of the sea ground's pivot point, the left edges of sea surface and sea bottom in figure 2.34 would not exactly match leading to unrealistic jumps of the kite when the user drags the kite across the coastline.

For the Z-component we have to take into account that the coastline has a Z-position of 15:

$$\cos\alpha = \frac{Z - 15}{250}$$

With $\alpha = 3°$ we get:

$$Z = 250 \cdot \cos 3° + 15 = 264.6574$$

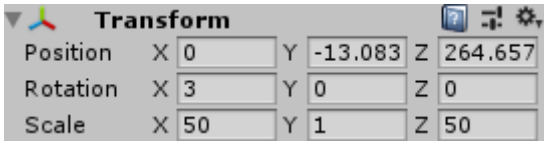

Figure 2.35: Sea bottom: Transform

2.6 Display

In order to display information (Wind speed, line length, help, ...) to the user, we import a[7] `Canvas` (`Hilfe` and `Leinwand`) object from Unity's `GameObject/UI` menu into the scene (figure 2.36).

[7] Actually, we use two canvases; one for the help and one for the data.

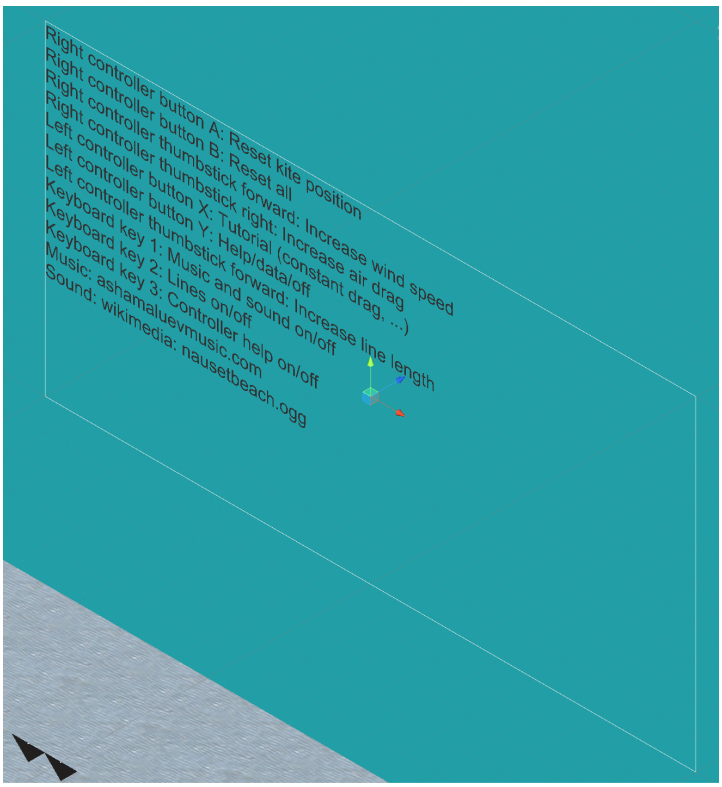

Figure 2.36: Display: Data, help, ...

We change the `Render Mode` of the `Canvas` to `World Space` (figure 2.37), fixing[8] the `Canvas` at a certain position in space.

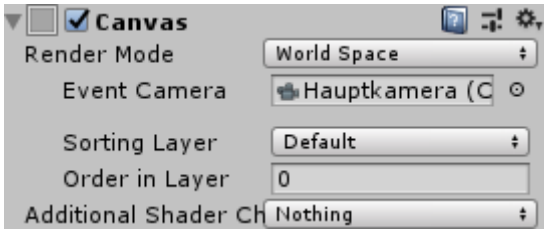

Figure 2.37: Display: `Canvas`

We can then define the canvas's center position 20 in front of the player, up 5 into the sky and give it a `Width` of 20 and a `Height` of 10 (figure 2.38).

[8] Basically, this means that we can look away from the canvas. The canvas does not follow our field of view if we move our head.

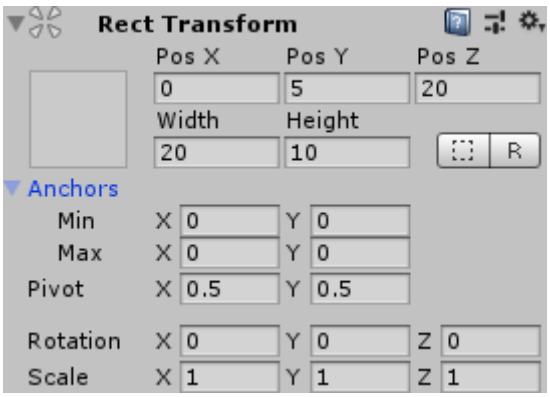

Figure 2.38: Display: `Rect Transform`

Values of 1000 for the number of `Dynamic Pixels Per Unit` and `Reference Pixels Per Unit` in the `Canvas Scaler` (figure 2.39) lead[9] to text with a proper size and resolution.

Figure 2.39: Display: `Canvas Scaler`

2.6.1 Help

For the actual help text we use a `Text` object from UNITY'S `GameObject/UI` menu as a child of the canvas. We position the `Text` in the center of the canvas and give it the same `Width` and a `Height` as the canvas (figure 2.40).

[9] To be honest, we did not really bother to figure out the exact function of the `Canvas Scaler`; the used values simply seem to do the job.

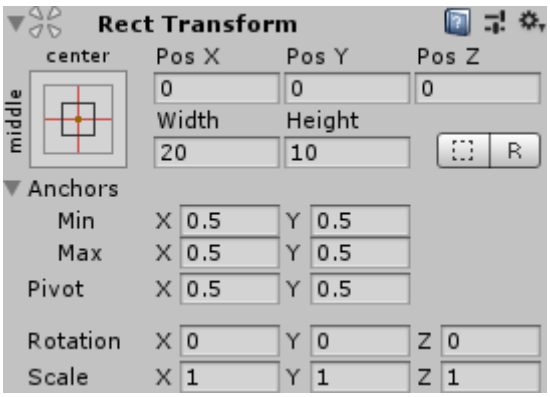

Figure 2.40: Help: `Rect Transform`

In the `Text` component of the `Text` object we can directly define the initial text to be displayed and choose the `Font Size`, ... of the text (figure 2.41).

Figure 2.41: Help: `Text`

We use this feature for the static help text; the data text object on the other hand is not static but displays the current values of wind speed, line length, ... during the

simulation. Therefore, we dynamically adapt the text property of the data display in every simulation step in section 3.2.

2.7 Start square

If the user resets the game, the kite respawns at its initial position ($\begin{bmatrix} 0 & 0 & 10 \end{bmatrix}$ in the global reference frame) and all four lines are reset to a length of 10. If the user has drastically changed her position before the reset – which is not a problem as long as the kite is in the air – unwanted rapid movements or falling over of the kite might occur in the first simulation step after the reset. The start square depicted in figure 2.42 that the user can see if she looks down can help her to step back to the origin before a reset[10].

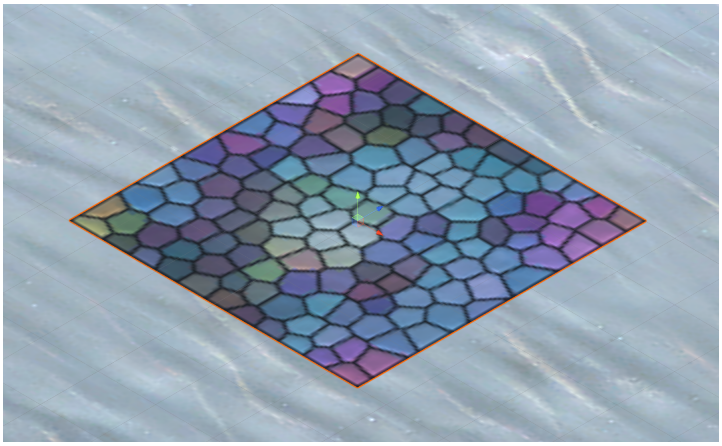

Figure 2.42: Start square

The start square has a size of 0.5×0.5 (figure 2.43). We set the Z-position of the start square center to -0.25. Therefore, if the user stands in the center of the square, her hands and handles are roughly at the origin of the coordinate system. The Y-position of the square is slightly positive to avoid z-fighting [7] and to make it visible in the "sand".

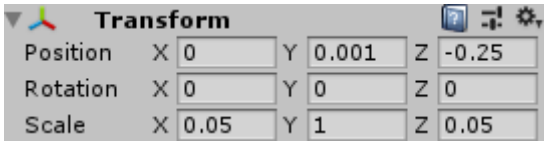

Figure 2.43: Start square: `Transform`

[10] Alternatively, we could always reset the kite's position 10 m in front of the player. While that would avoid the jerky kite jumps after a reset, the user might quickly leave the real-world play area or orientation after a few resets. And there will be lots of resets ...

The nice Tiffany glass material from UNITY's `Standard Assets` package in figure 2.44 might occur a bit unexpected on a Caribbean beach, but is too beautiful to not use ...

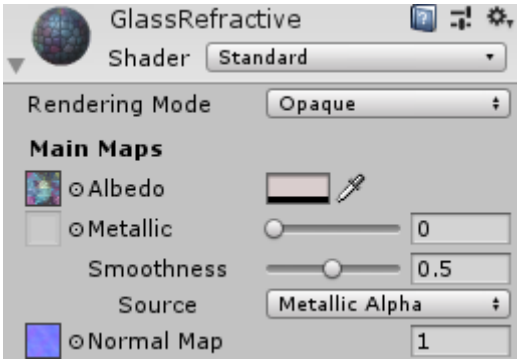

Figure 2.44: Start square: `GlasRefractive Material`

2.8 Palm trees

The palm trees in figure 2.45 are there just to create a little bit of Caribbean atmosphere in the game. We imported them from the free `Coconut Palm Tree Pack` in UNITY's Asset Store. As you can see in figure 2.45 we arbitrarily positioned their lowest point a few centimeters below the ground to give them a more natural look. You can also see in figure 2.45 that all palm trees have simple `Capsule Colliders` that barely cover the lower parts of the tree trunks. We should therefore not expect a too realistic interaction of the kite with the trees; you can easily fly the kite through the fronds and the upper parts of the trunks.

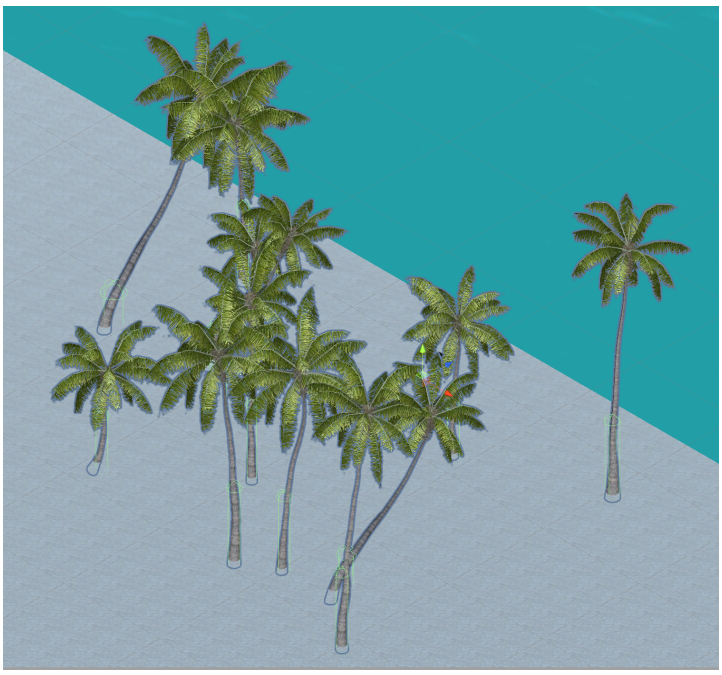

Figure 2.45: Palm trees

2.9 Rocks

Unlike the palm trees, the rocks (`Free_Rocks` package) depicted in figure 2.46 use `Mesh Colliders` instead of `Capsule Colliders` allowing the kite to interact with the rocks quite realistically; we can make the kite touch a rock with its tip, lean it against a rock, rest it on top of a rock, ...

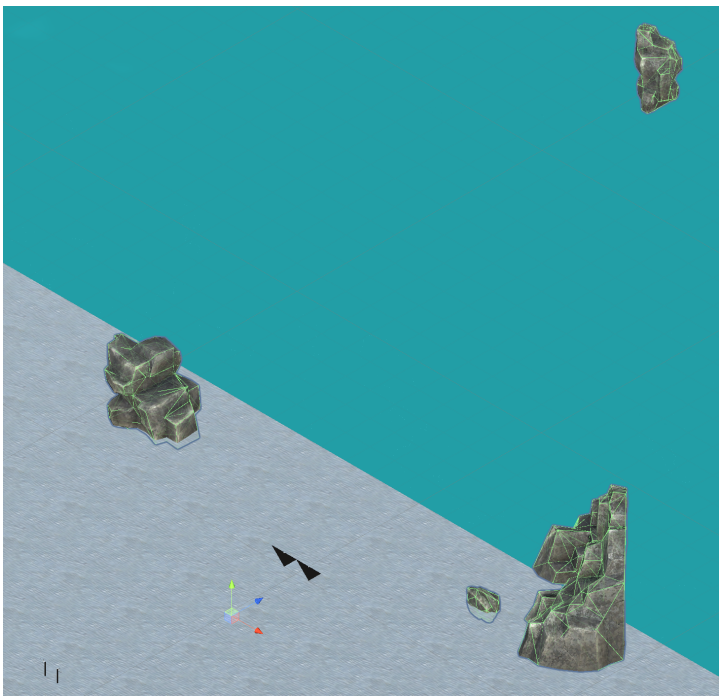

Figure 2.46: Rocks

2.10 Lines

As already stated in on page 9, stiff, lightweight lines are extremely difficult to model and simulate physically. Instead, we use `Configurable Joints` between kite and handles and draw "fake" lines with `GameObjects` from UNITY'S `Effects/Line` menu.

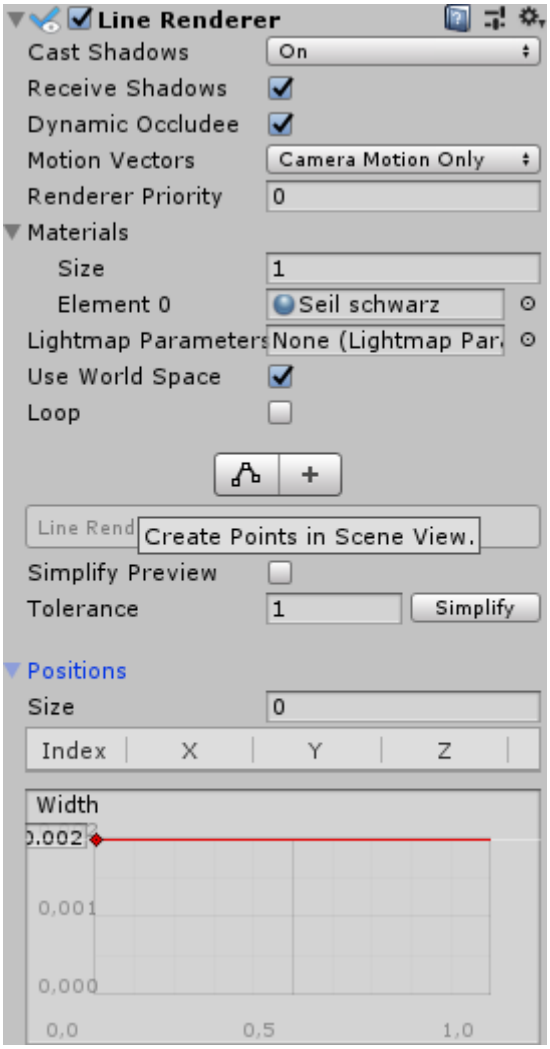

Figure 2.47: Line: `Line Renderer`

Since we compute and display the lines dynamically with the script in section 3.11, we can initialize the `Positions` attribute of the `Line Renderer` as an empty array in figure 2.47. We use the same script for all four lines and therefore have to define the begin (`Anfang`) and end (`Ende`) of every line in figure 2.48.

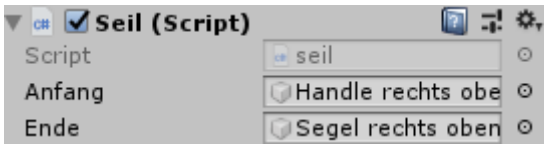

Figure 2.48: Line: Script

2.11 Sounds

We use three different sound sources for music, ocean waves, and ground collision.

2.11.1 Music

We imported the `Sad Piano and Strings` soundtrack (figure 2.50) from the Ukrainian composer and musician Aleksandr Shamaluev who offers fantastic royalty free music on his website [8]. The music just contributes to the calm, melancholic atmosphere of the scenery as a background soundtrack. Initially, we position it directly in the head of the user (figure 2.49).

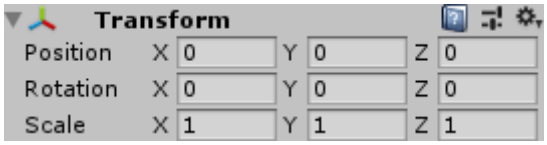

Figure 2.49: Music: `Transform`

By using a fast `Logarithmic Rolloff` (figure 2.50) the user can even use the sound position and intensity to acoustically find her way back to the origin of the coordinate system (section 2.7).

Figure 2.50: Music: `Audio Source`

2.11.2 Ocean waves

Wikimedia Commons is a great source of sound audio files licensed under the Creative Commons license. We use a track of "Sound of waves on Nauset Beach after sunset" [9] to position it at the coastline in front of the player (figure 2.51).

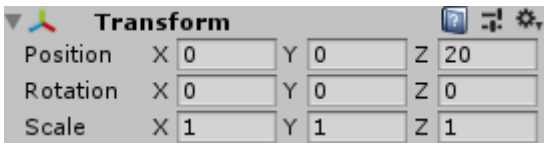

Figure 2.51: Ocean waves: `Transform`

Using a slow Linear Rolloff, we give the user an acoustic indication of her head's orientation[11] with respect to the sea.

[11] If the user yaws her head to the right, she can distinctly hear the ocean waves in her left ear.

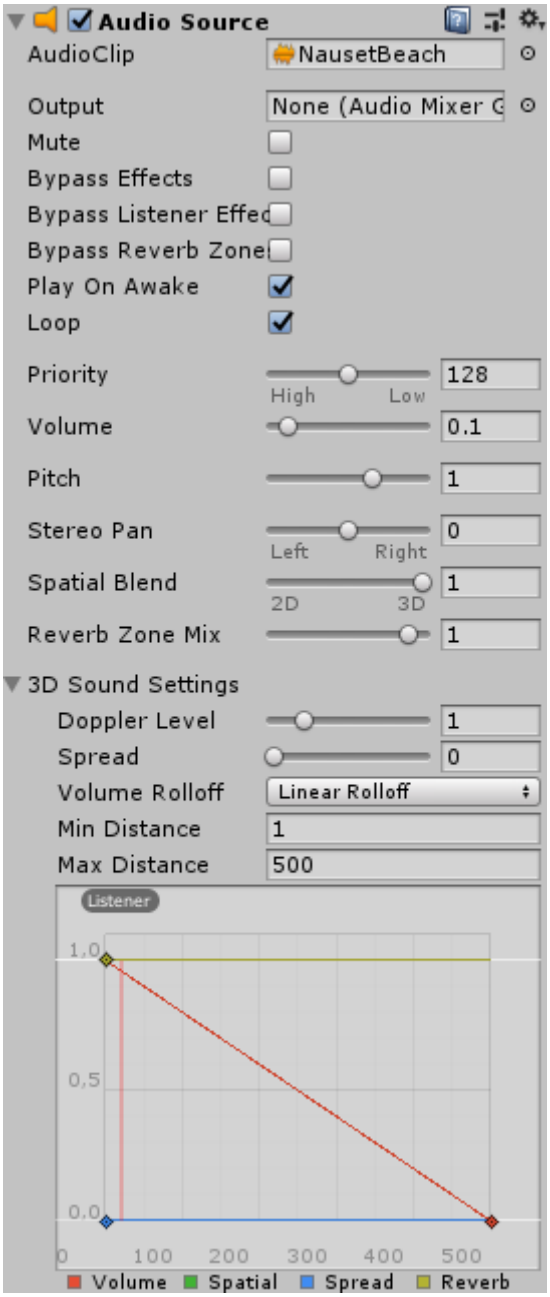

Figure 2.52: Ocean waves: `Audio Source`

2.11.3 Ground collision

We want the handles to vibrate and some sound to be played if one of the kite's vertices hits the ground. For that purpose, we mount four bumpers at the outer vertices of the kite (figure 2.53). The attached script (section 3.6) plays a sound we took from UNITY'S `Standard Assets` package at the position of the corresponding bumper and vibrates

the controllers.

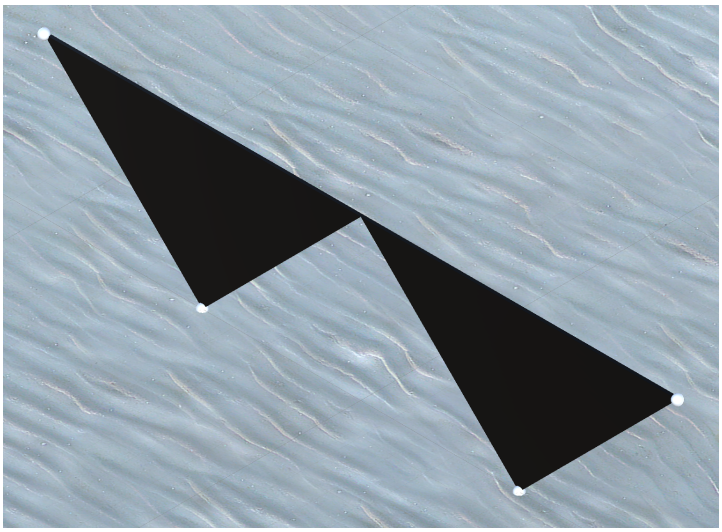

Figure 2.53: Bumper

2.12 Target kite

The target kite is the red transparent kite behind the black user controlled kite in figure 2.54. It is controlled by the script in section 3.9 and activated by the user. It performs sinusoidal pitch, yaw, and roll motions that the user can try to follow in the tutorial.

Figure 2.54: Target kite (red)

As indicated in figure 2.54 and figure 2.55, the pivot point of the target kite is in the origin at a height of 0.75 which is about the position where the user holds the handles.

Defining the pivot point in the origin makes it quite easy to have the target kite perform its motions on a sphere[12] around the user; we just have to alter the target kite's `Rotation` angles.

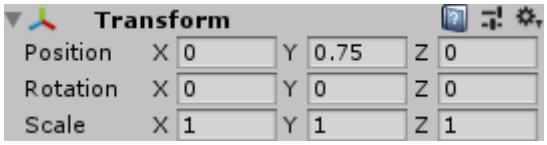

Figure 2.55: Target kite: `Transform`

2.12.1 Target kite right sail

The right sail of the target kite is a child of the target kite.

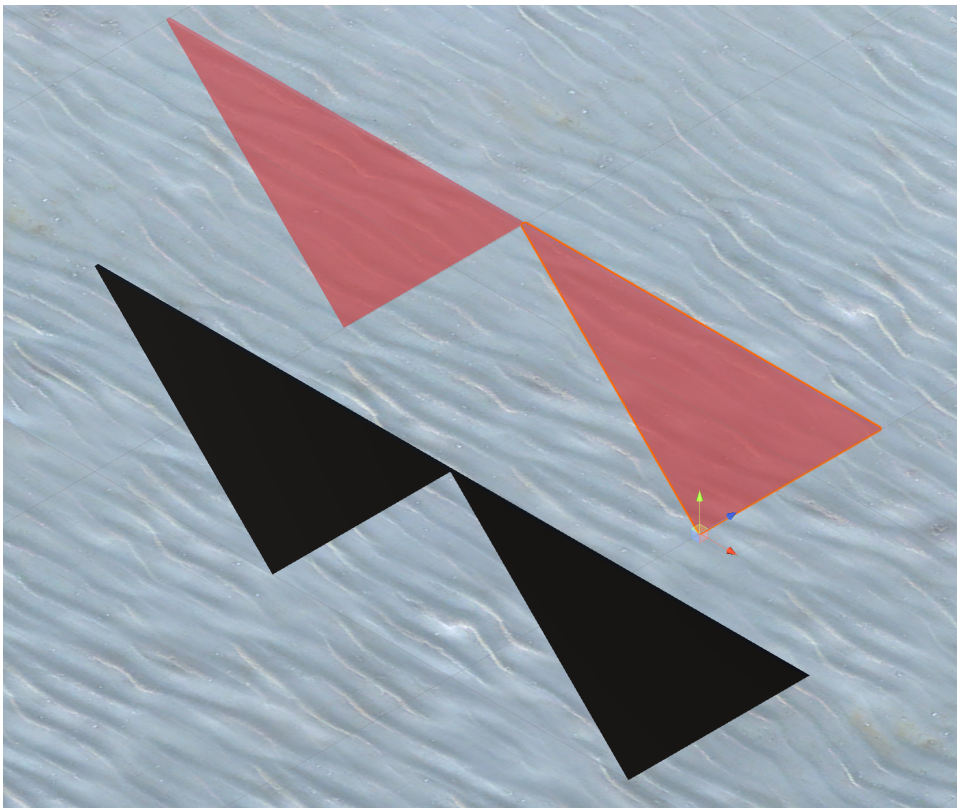

Figure 2.56: Target kite right sail

By fixing its Z-position at 10.2 in figure 2.57, we ensure the target kite to move behind the user controlled kite.

[12] We keep the line length constant at 10 m during the tutorial.

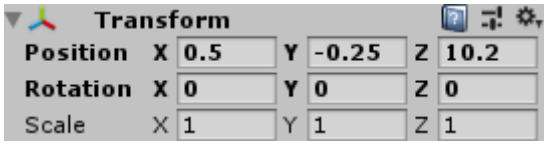

Figure 2.57: Target kite right sail: `Transform`

2.13 Target handles

During the tutorial, the target handles act similar to a flight director [10] in an aircraft. They tell the user the attitude (rotation angles) she should rotate her handles to. As shown in figure 2.58 and explained in section 2.13.2 we additionally add a target handle traverse that rotates with the target handles and might make it a bit easier for the user to detect the actual attitude of the target handles object.

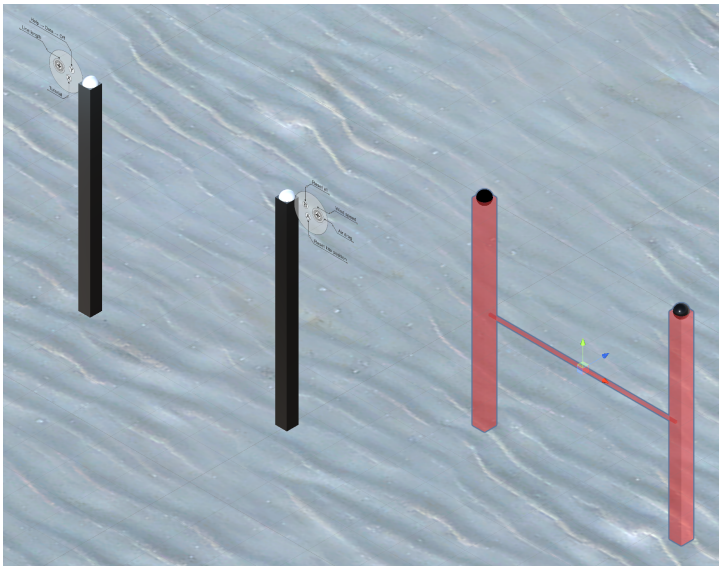

Figure 2.58: Target handles (red)

We fix the Z-position of the target handles at 1 m in front of the user, giving her a chance to always see the target handles, even if she does not see[13] her own handles.

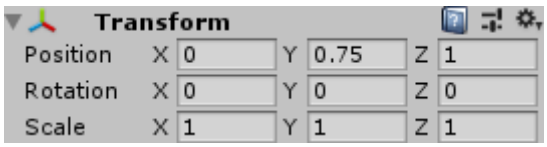

Figure 2.59: Target handles: `Transform`

[13] New kite pilots tend to hold up the handles in front of them; many experienced pilots keep them at a more relaxed lower position closer to their hips.

2.13.1 Right target handle

The right target handle (figure 2.60) has the same `Position` and `Scale` (figure 2.61) with respect to its parent as the actual handle (figure 2.17).

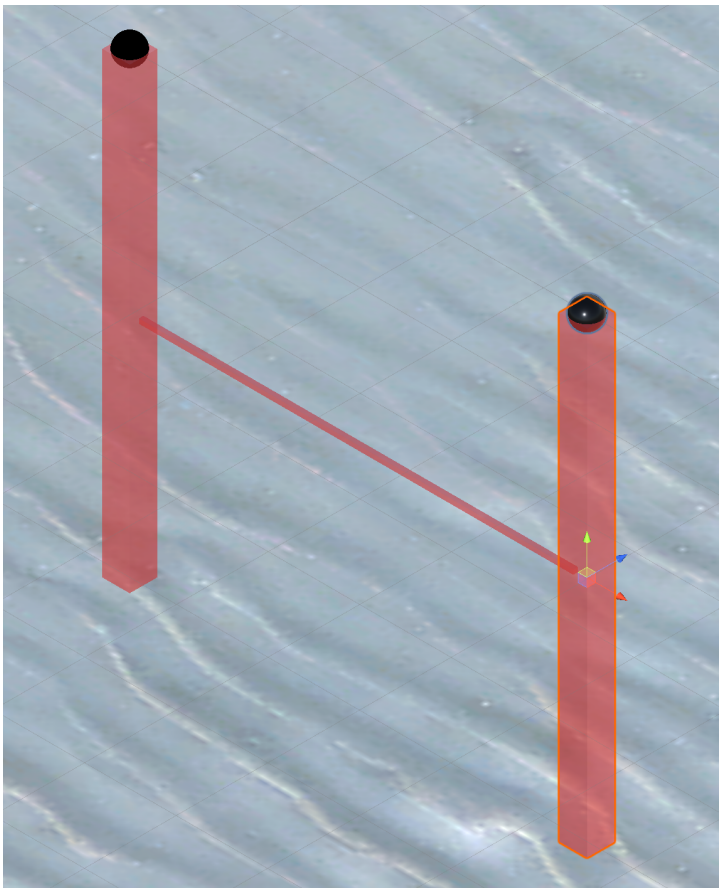

Figure 2.60: Right target handle

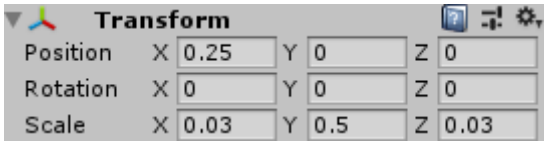

Figure 2.61: Right target handle: `Transform`

2.13.2 Target handle traverse

The target handle traverse (figure 2.62) is another child of the target handles object.

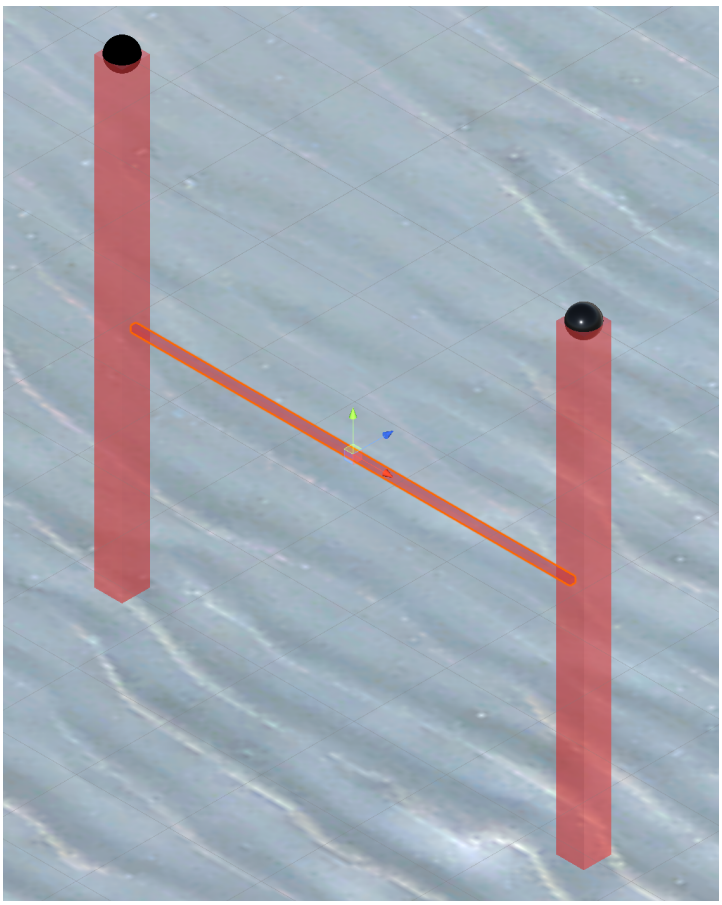

Figure 2.62: Target handle traverse

It is just a cuboid at the pivot point (`Position` in figure 2.63) of the target handles object with an X-length[14] of nearly 0.5.

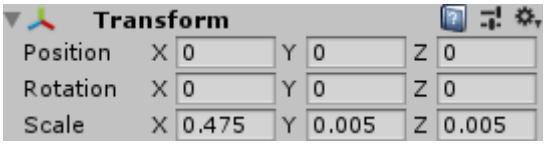

Figure 2.63: Target handle traverse: `Transform`

2.14 Entitlement and Focus Check

In order to offer it in their store, Oculus requires an app to pass a bunch of Virtual Reality Checks. Most of the checks are automatically passed since we create RevSim

[14] We have to subtract the thickness of a handle from the traverse's X-length to prevent the traverse to penetrate the handles. Such a penetration would not be visible with opaque handles, but since we have transparent target handles ...

with UNITY; for two tests – the Entitlement Check and the Focus Check – we have to write[15] two small scripts (section 3.4 and section 3.5) to ensure, that the user is allowed to use the app and that the app stops writing frames if the user presses the OCULUS Home button or quits the application. In order to run the scripts immediately at the start of the app, we create an invisible `Entitlement and Focus Check` object in the scene and add both scripts (section 3.4) to the object.

[15] To be honest, we found some useful code snippets in the OCULUS documentation and adapted them to our needs.

3 Functions

In this chapter, we discuss every function we used in UNITY in detail.

3.1 Aerodynamics

In the `Aerodynamik` class we compute the aerodynamic forces generated by the relative velocity of the kite with respect to the air. We add the script to both sails of the kite allowing each sail to independently compute its own forces.

UNITY scripts automatically import some standard types from predefined namespaces:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

Every UNITY script derives from the `MonoBehaviour` base class:

```
public class Aerodynamik : MonoBehaviour
{
```

Before we define the `Start` function, we declare some (global) objects. We will use the player `GameObject spieler` to access the wind speed the user defines with her right controller:

```
GameObject spieler;
```

We will use `Rigidbody festkoerper` to read the current drag, angular drag, and velocity properties of each sail

```
private Rigidbody festkoerper;
```

and

```
private Transform zustand;
```

for the current attitude of the sail. The joint attributes

```
private ConfigurableJoint[] feder;
private SoftJointLimit linear_limit;
```

are necessary to come up with the current line length and

```
public Vector3 V_W;
public Vector3 V_K;
public Vector3 V_A;
```

stand for

`V_W`	Wind speed vector (velocity of the air with respect to the ground)
`V_K`	Ground speed vector (velocity of the sail with respect to the ground)
`V_A`	Air speed vector (velocity of the sail with respect to the air)

The `start` function is called before the first simulation step:

```
void Start()
{
```

In the function, we address the player object in the scene (section 2.3)

```
spieler = GameObject.Find("Spieler");
```

the array of the configurable joints (figure 2.10)

```
feder = GetComponents<ConfigurableJoint>();
```

and the `Rigidbody` component of the sail (section 2.2.1):

```
festkoerper = GetComponent<Rigidbody>();
```

We initialize the wind speed vector

```
V_W = new Vector3(0, 0, 10);
```

and finally reference the state (Position, Rotation, and Scale) of the sail:

```
zustand = transform;
}
```

For physical simulations we utilize the `FixedUpdate` function:

```
void FixedUpdate()
{
```

It is called at a predefined constant rate independent of the variable graphical frame rate. In every simulation step, we acquire (section 3.3) the current line length the user defines via her right controller

```
float seillaenge =
spieler.GetComponent<Controllereingaben>().seillaenge;
```

and start a loop over both lines connected to the sail:

```
for (int i = 0; i < 2; i++)
{
```

The next three[1] lines of code adapt the `LinearLimit` property of the `ConfigurableJoint` representing the line to the current line length.[2] We buffer the current `LinearLimit` of the joint into the predeclared `SoftJointLimit`

```
  linear_limit = feder[i].linearLimit;
```

copy the current line length into the Limit property of the `SoftJointLimit`

```
  linear_limit.limit = seillaenge;
```

and write back the `SoftJointLimit` into the `LinearLimit` property of the joint:

```
  feder[i].linearLimit = linear_limit;
}
```

The user can customize the air drag with her right controller. For the sake of simplicity, we use the same value for the translational (linear) and rotational (angular) drag coefficients. We read the current user defined drag value

```
float daempfung =
spieler.GetComponent<Controllereingaben>().daempfung;
```

and copy it into the linear

```
festkoerper.drag = daempfung;
```

and angular drag property of the sail:

```
festkoerper.angularDrag = daempfung;
```

The next few lines compute the force vector that is exerted on the kite by the air speed of the sail.

The wind speed (velocity of the air with respect to the ground) can be modified by the user too. We assume the wind always[3] blowing into the global Z-direction (from land to sea). Therefore, we must copy the scalar wind speed into the Z-component of the three-dimensional wind speed vector `V_W` only:

```
V_W[2] =
spieler.GetComponent<Controllereingaben>().windgeschwindigkeit;
```

Unity kindly provides us with the velocity vector of every `Rigidbody` with respect to the ground. Therefore, we can directly determine the ground speed (of each sail) as a vector:

```
V_K = festkoerper.velocity;
```

[1] This lengthy way to alter the joint's `LinearLimit` seems to be quite awkward; nevertheless, we could not find a working shortcut. Any ideas?

[2] To be honest, this line length adaptation does not really have to do too much with the computation of aerodynamic forces. We should outsource it into a different script; but since we are a bit lazy ...

[3] It would not a big deal to introduce different wind directions or even turbulence here; we just figured out it would not add too much more fun to the overall flying experience.

figure 3.1 explains that there is a vectorial relation between

$\boldsymbol{V_W}$ Velocity of the air with respect to the ground

$\boldsymbol{V_K}$ Velocity of a flying object with respect to the ground

$\boldsymbol{V_A}$ Velocity of the object with respect to the air

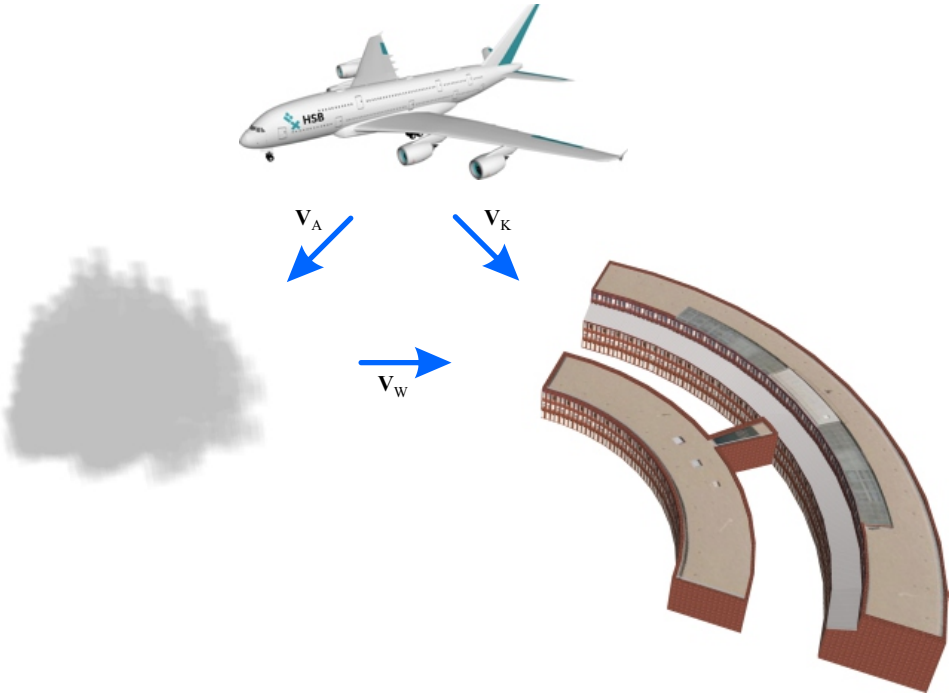

Figure 3.1: Relation between wind speed $\boldsymbol{V_W}$, ground speed $\boldsymbol{V_K}$, and air speed $\boldsymbol{V_A}$ [11]

Thanks to Unity's great vectorial capabilities we can directly express the relation between the speed vectors

$$\boldsymbol{V_K} = \boldsymbol{V_A} + \boldsymbol{V_W}$$

or

$$\boldsymbol{V_A} = \boldsymbol{V_K} - \boldsymbol{V_W} \tag{3.1}$$

in code:

```
V_A = V_K - V_W;
```

The air speed `V_A` is responsible for the aerodynamic force[4] on the sail.[5]

We assume the aerodynamic force vector $\mathbf{R_A}$ to act perpendicular to the sail (figure 3.2) which is not exactly true but produces quite realistic results. The `forward` property of a three-dimensional vector is UNITY'S shorthand notation for writing `Vector3(0, 0, 1)`, which returns a unit vector pointing into the Z-direction of the local sail coordinate system. Therefore,

```
Vector3 normalenvektor = zustand.forward;
```

provides us with a unit normal vector $\mathbf{n}$ perpendicular on the sail (figure 3.2).

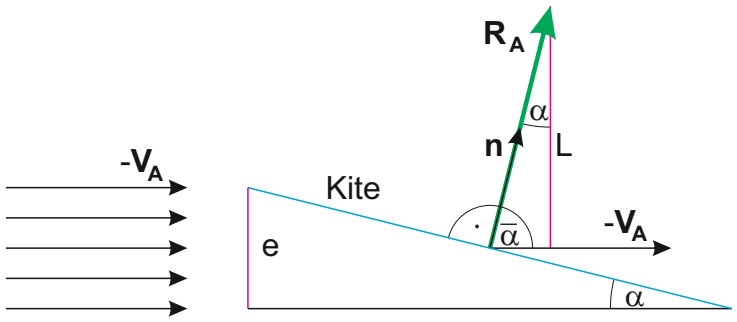

Figure 3.2: Effective area e

The dot product s of the negative air speed vector $-\boldsymbol{V_A}$ and the unit[6] normal vector $\mathbf{n}$

```
float s = Vector3.Dot(-V_A, normalenvektor);
```

becomes proportional to the magnitude (norm) V_A of the airspeed vector and to the sine[7] of the angle of attack α:

$$\begin{aligned} s &= -\boldsymbol{V_A} \cdot \mathbf{n} \\ &= V_A \cdot n \cdot \cos\bar{\alpha} \\ &= V_A \cdot \cos\bar{\alpha} \\ &= V_A \cdot \cos\left(\frac{\pi}{2} - \alpha\right) \\ &= V_A \cdot \sin\alpha \end{aligned} \tag{3.2}$$

[4]There is lift if the wind blows but the kite does not move; but there is also lift if we drag the kite in calm air (e. g. indoors). It is the relative speed between the kite and the air that produces the lift.

[5]It seems to be a bit contra-intuitive to define the air speed $\boldsymbol{V_A}$ pointing away from the kite into the direction of the incoming air flow (note that we use $-\boldsymbol{V_A}$ in figure 3.2). This definition is very useful in flight mechanics of aircraft because according to equation (3.1) the direction of the air speed vector$\boldsymbol{V_A}$ is equal to the direction of the ground speed vector $\boldsymbol{V_K}$ if there is no wind ($\boldsymbol{V_W} = \mathbf{0}$). For a kite however, one might be tempted to define $\boldsymbol{V_A} = \boldsymbol{V_W} - \boldsymbol{V_K}$, because with a kite, the air speed $\boldsymbol{V_A}$ is usually produced by wind ($\boldsymbol{V_W}$) and not by motion of the kite ($\boldsymbol{V_K}$). Nevertheless, for the sake of continuity, we stick to the well established definition according to equation (3.1).

[6]Keep in mind the trivial fact that the norm of a unit vector equals 1: $|\mathbf{n}| = n = 1$

[7]Seen from another physical angle in figure 3.2, $\sin\alpha$ in equation (3.2) assumes that the aerodynamic force depends on the area e that defines how much of the sail area is effected by the inflow.

Physics tells us that the dynamic pressure $\bar{q}$ does not linearly dependent on the airspeed but is proportional to the **square** of the airspeed (and the air density ρ)

$$\bar{q} = \frac{\rho}{2} V_A^2 \tag{3.3}$$

while the aerodynamic force R_A depends on the surface area S, the dynamic pressure $\bar{q}$, and the drag (or lift, respectively) coefficient C:

$$R_A = S \cdot \bar{q} \cdot C \tag{3.4}$$

Using equation (3.2) and equation (3.3) in equation (3.4) we come up with:

$$\begin{aligned} R_A &= 0.1 \cdot V_A \cdot s \\ &= 0.1 \cdot V_A^2 \cdot \sin\alpha \end{aligned}$$

```
float R = 0.1f * V_A.magnitude * s;
```

where the empirical factor of 0.1 represents the influences of surface area and drag coefficient.

We can then use R_A as the magnitude of the aerodynamic force vector $\mathbf{R_A}$

$$\mathbf{R_A} = \mathbf{n} \cdot R_A$$

and finally make $\mathbf{R_A}$ the force vector acting on the sail:

```
festkoerper.AddForce(normalenvektor * R);
}
}
```

Please note that we model the aerodynamics in a very rudimentary way; nevertheless, the kite behaves quite realistically: The lift force L compensating the weight of the kite is the vertical component of the aerodynamic force vector $\mathbf{R_A}$ in figure 3.2

$$L = R_A \cdot \cos\alpha$$

making the lift proportional to

$$L \sim \sin\alpha \cos\alpha$$

The corresponding graph is depicted in figure 3.3.

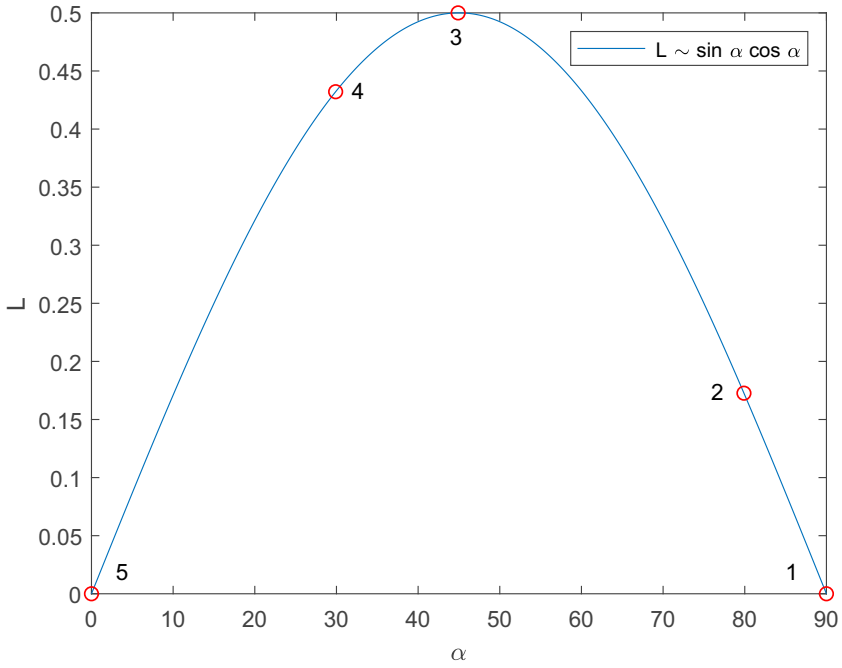

Figure 3.3: Lift L over angle of attack α

1. If the kite is vertically standing on the ground, its angle of attack is 90° and figure 3.3 returns a lift force of zero. This is physically correct: The aerodynamic force vector $\mathbf{R_A}$ in figure 3.2 is horizontal, creating maximum drag that pulls the kite away from the pilot but no lift that could compensate the weight of the kite.
2. If the pilot decreases the angle of attack by rotating the kite's trailing edge closer to herself, we move backwards on the graph in figure 3.3. The aerodynamic force vector rotates up, the lift increases until it equals the weight of the kite, and the kite leaves the ground for the first time.
3. If the angle of attack reaches 45°, the lift is it at its maximum; the kite reaches its maximum height.
4. If the pilot decreases the angle of attack any further, the lift decreases too and the kite loses height. The aerodynamic force vector points upwards more and more, but the effective sail area becomes smaller and smaller.
5. If the pilot has decreased the angle of attack to zero, the kite floats horizontally in the air. Theoretically, the aerodynamic force vector is now perfectly vertical but its magnitude is zero because the horizontal sail cannot produce any lift.[8]

[8] You can actually fly this maneuver in RevSim. If you rapidly rotate (the upper ends of) both handles towards you, the kite starts floating towards the ground and the lines pile up on the ground. Maybe,

3.2 Data display

We use the `Anzeige` class to dynamically display the current wind speed, line length, and drag on the `Text` (`Anzeige`) object which is a child of the `Canvas` (`Leinwand`) object described in section 2.6.

Additionally to the standard types (section 3.1) we import the `UnityEngine.UI` type:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
```

In the class

```
public class Anzeige : MonoBehaviour
{
```

we declare the `Text` component of the `Text` object

```
Text anzeige_text;
```

and the `GameObject spieler` to access the current wind speed, line length, and drag the user defines with her controllers:

```
GameObject spieler;
```

During the initialization

```
void Start()
{
```

we address the `Text` component of the `Text` object

```
anzeige_text = GetComponent<Text>();
```

and the player object in the scene (section 2.3):

```
spieler = GameObject.Find("Spieler");
}
```

In every (graphical) simulation step

```
void Update()
{
```

we read the wind speed (`windgeschwindigkeit`), line length (*`seillaenge`*), and drag (`daempfung`) properties from the Controller input (`Controllereingaben`) component of the player (`spieler`) object and display the data in the `text` property of the `Text` component of the `Text` object:

one of the next RevSim versions might even offer you the ability to catch the floating kite with your hands ...

```
anzeige_text.text =
"Press left controller button Y for help." +
"\n" +
"\n" +
"Wind: " +
spieler.GetComponent<Controllereingaben>().
windgeschwindigkeit.ToString("F1") +
" m/s" +
"\n" +
"Line: " +
spieler.GetComponent<Controllereingaben>().
seillaenge.ToString("F1") +
" m" +
"\n" +
"Drag: " +
spieler.GetComponent<Controllereingaben>().
daempfung.ToString("F1") +
" m";
}
}
```

3.3 Controller input

In the class `Controllereingaben`

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Controllereingaben : MonoBehaviour
{
```

we use the inputs the user has made with her controllers to modify the corresponding game parameters (line length, wind speed, ...).

We declare the relevant parameters

```
public float seillaenge;
public float windgeschwindigkeit;
public float daempfung;
public float t_start;
public int solldrachen_zustand;
public int anzeige_zustand;
```

and the objects to be addressed and modified

```
GameObject segel_rechts;
GameObject segel_links;
GameObject solldrachen;
GameObject sollhandles;
GameObject anzeige;
GameObject hilfetext;
GameObject musik;
GameObject meeresrauschen;
GameObject seile;
GameObject hilfe_rechts;
GameObject hilfe_links;
```

and use the initialization function to find the objects

```
void Start()
{
segel_rechts = GameObject.Find("Segel rechts");
segel_links = GameObject.Find("Segel links");
solldrachen = GameObject.Find("Solldrachen");
sollhandles = GameObject.Find("Sollhandles");
anzeige = GameObject.Find("Anzeige");
hilfetext = GameObject.Find("Hilfetext");
musik = GameObject.Find("Musik");
meeresrauschen = GameObject.Find("Meeresrauschen");
seile = GameObject.Find("Seile");
hilfe_rechts = GameObject.Find("Hilfe rechts");
hilfe_links = GameObject.Find("Hilfe links");
```

and initialize the parameters

```
seillaenge = 10f;
windgeschwindigkeit = 10f;
daempfung = 1f;
t_start = 0f;
solldrachen_zustand = 0;
solldrachen.SetActive(false);
sollhandles.SetActive(false);
hilfetext.SetActive(false);
musik.SetActive(true);
anzeige_zustand = 0;
}
```

In every simulation step

```
void Update()
{
```

we modify the parameters according to the controller inputs. As long as the user pushes

the left thumbstick forward, the line length (`seillaenge`) is increased[9] with a maximum of 10 cm per simulation step:

```
seillaenge +=
0.1f * OVRInput.Get(OVRInput.Axis2D.PrimaryThumbstick)[1];
```

While we do not limit[10] the maximum line length, we do not allow line lengths less than 50 cm:

```
if (seillaenge < 0.5f)
{
  seillaenge = 0.5f;
}
```

Pushing the right thumbstick forward increases the wind speed (`windgeschwindigkeit`)

```
windgeschwindigkeit +=
0.1f * OVRInput.Get(OVRInput.Axis2D.SecondaryThumbstick)[1];
```

and pushing the right thumbstick to the right increases the drag (`daempfung`):

```
daempfung +=
0.1f * OVRInput.Get(OVRInput.Axis2D.SecondaryThumbstick)[0];
```

Negative drag would generate an unrealistic force acting **into** the direction of the current velocity vector; we limit the minimum drag to zero:

```
if (daempfung < 0.0f)
{
  daempfung = 0.0f;
}
```

Pressing the A button on the right controller (figure 2.19) resets (section 3.3.1) the attitude and the position of the kite using the current[11] line length:

```
if (OVRInput.Get(
OVRInput.Button.One, OVRInput.Controller.RTouch))
{
  Reset();
}
```

The B button on the right controller (figure 2.19) hard resets the whole simulation to its initial state. Additionally to the grounding and attitude reset of the kite in section 3.3.1, we also reset the line length, the wind speed, the drag, and the state[12] of the target kite

[9] Assuming a graphical sampling rate of 90 Hz, you can increase the line length up to 9 meters per second. Obviously, pulling the thumbstick backwards pulls the kite back towards to you.

[10] There is always a natural resolution limit: If the graphic representation of the distant kite falls short of two pixel you can no more recognize the kite's bank angle making it harder and harder to control its attitude and position.

[11] Think of this as a soft reset if you momentarily lost control of the kite. Your currently chosen line length, wind speed, and drag are kept untouched but the kite is reset to the ground with an upright attitude.

[12] You can always terminate the tutorial by pressing the B button.

(`solldrachen_zustand`):

```
if (OVRInput.Get(
OVRInput.Button.Two, OVRInput.Controller.RTouch))
{
  seillaenge = 10;
  windgeschwindigkeit = 10;
  daempfung = 1;
  solldrachen_zustand = 0;

  Reset();
}
```

The X button on the left controller (figure 2.19) cycles through the steps of the tutorial. Whenever the button is pressed

```
if (OVRInput.GetDown(
OVRInput.Button.One, OVRInput.Controller.LTouch))
{
```

we buffer the current simulation time offset in order to start every tutorial step with an initial time of zero in section 3.9

```
  t_start = Time.time;
```

and increment the state of the target kite:

```
  solldrachen_zustand += 1;
```

Since we have four tutorial steps, the fifth press of the Y button cycles back out of the tutorial:

```
  solldrachen_zustand %= 5;
```

Every X button press also hard resets[13] the line length, the wind speed, the drag, and the kite:

```
  seillaenge = 10;
  windgeschwindigkeit = 10;
  daempfung = 1;

  Reset();
}
```

If we are not doing the tutorial, we do neither want to see the target kite (*solldrachen*) nor the target handles (`sollhandles`):

```
if (solldrachen_zustand == 0)
{
```

[13]Yes, we should write an extra hard reset function, but well ...

```
  solldrachen.SetActive(false);
  sollhandles.SetActive(false);
}
```

If we are in the tutorial, we make the target kite and the target handles visible and keep the line length, the wind speed, and the drag[14] constant:

```
else
{
  solldrachen.SetActive(true);
  sollhandles.SetActive(true);

  seillaenge = 10;
  windgeschwindigkeit = 10;
  daempfung = 10;
}
```

We use the keyboard[15] keys to toggle sound and object visibility: If the user presses the "1" key, we toggle the audibility of the music (`musik`) and the ocean waves (`meeresrauschen`):

```
if (Input.GetKeyDown(KeyCode.Alpha1))
{
  if (musik.activeSelf)
  {
    musik.SetActive(false);
    meeresrauschen.SetActive(false);
  }
  else
  {
    musik.SetActive(true);
    meeresrauschen.SetActive(true);
  }
}
```

Pressing the "2" key, toggles the visibility of the lines (`seile`)

```
if (Input.GetKeyDown(KeyCode.Alpha2))
{
  if (seile.activeSelf)
  {
    seile.SetActive(false);
  }
  else
  {
```

[14] Using a constant drag of 10 during the tutorial instead of the default value of 1, helps the inexperienced pilot a lot: The kite moves in slow motion as if the air has become smooth liquid honey.

[15] Yes, we should not use the keyboard in VR applications. And yes, we should save the users choice and respect her will the next time she plays RevSim ...

```
    seile.SetActive(true);
  }
}
```

and the "3" key toggles the visibility of the right (`hilfe_rechts`) and the left (`hilfe_links`) help plates (figure 2.19):

```
if (Input.GetKeyDown(KeyCode.Alpha3))
{
  if (hilfe_rechts.activeSelf)
  {
    hilfe_rechts.SetActive(false);
    hilfe_links.SetActive(false);
  }
  else
  {
    hilfe_rechts.SetActive(true);
    hilfe_links.SetActive(true);
  }
}
```

Every Y button press on the left controller (figure 2.19)

```
if (OVRInput.GetDown(
OVRInput.Button.Two, OVRInput.Controller.LTouch))
{
```

cycles through three different display (section 2.6) states (`anzeige_zustand`)

- Display of dynamic data
- Static help display
- No display

```
  anzeige_zustand += 1;
  anzeige_zustand %= 3;
```

Depending on the display state

```
  switch (anzeige_zustand)
  {
```

we display dynamic data (line length, wind speed, and drag)

```
    case 0:
      anzeige.SetActive(true);
      hilfetext.SetActive(false);
      break;
```

the static help

```
    case 1:
      anzeige.SetActive(false);
      hilfetext.SetActive(true);
      break;
```

or nothing at all:

```
    case 2:
      anzeige.SetActive(false);
      hilfetext.SetActive(false);
      break;

    default:
      break;
    }
}
}
```

3.3.1 Reset

The (soft) `Reset` function of the controller input class

```
void Reset()
{
```

is called, whenever the user presses the A button on the right controller. It resets the right sail (*segel_rechts*) to its initial position using the current line length (`seillaenge`)

```
segel_rechts.transform.position =
new Vector3(0.5f, 0, seillaenge);
```

and resets all Euler angles (pitch angle, yaw angle, and bank angle) of the right sail to zero:

```
segel_rechts.transform.rotation = Quaternion.Euler(0, 0, 0);
```

Obviously, the same reset has to be done for the left sail (`segel_links`) :

```
segel_links.transform.position =
new Vector3(-0.5f, 0, seillaenge);

segel_links.transform.rotation = Quaternion.Euler(0, 0, 0);
```

Finally, we provide the user with a light (`0.1f`) haptic right controller vibration feedback by calling the corresponding coroutine (section 3.3.2):

```
StartCoroutine(Vibriere(0.1f));
}
```

3.3.2 Right controller vibrate

The right controller vibration (`Vibriere`) coroutine is called with the required vibration strength (*staerke*)

```
IEnumerator Vibriere(float staerke)
{
```

and uses the parameter to switch on the vibration of the right controller with maximum frequency (`1f`) and the required strength:

```
OVRInput.SetControllerVibration(
1f, staerke, OVRInput.Controller.RTouch);
```

We want the vibration to last 100 ms

```
yield return new WaitForSeconds(0.1f);
```

and then switch it off again:

```
OVRInput.SetControllerVibration(
0f, 0f, OVRInput.Controller.RTouch);
}
}
```

3.4 Entitlement check

The `EntitlementCheck` class

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Oculus.Platform;

public class EntitlementCheck : MonoBehaviour
{
```

ensures, that the user is allowed to use the app (section 2.14).

The `awake` function of a game object is called as soon as the object has been initialized by UNITY.

```
void Awake()
{
```

The recommended way to perform the entitlement check is to wrap it in a `try catch` block

```
try
{
```

initialize the platform SDK

```
  Core.AsyncInitialize();
```

perform the entitlement check, and call a function if the check is complete:

```
  Entitlements.IsUserEntitledToApplication().
  OnComplete(EntitlementCallback);
}
```

If the platform SDK initialization was not successful

```
catch (UnityException e)
{
```

we inform the user[16]

```
  Debug.LogError(
  "Platform failed to initialize due to exception.");

  Debug.LogException(e);
```

and quit the application immediately:

```
  UnityEngine.Application.Quit();
}
}
```

If the check is complete, the corresponding function is called:

```
void EntitlementCallback(Message msg)
{
```

If the check was not successful

```
if (msg.IsError)
{
```

we inform the user and quit the application

```
  Debug.LogError("You are NOT entitled to use this app.");

  UnityEngine.Application.Quit();
}
```

If the check was successful, we inform the user and the application can proceed:

```
else
{
  Debug.Log("You are entitled to use this app.");
}
}
}
```

[16]Obviously, debug outputs are only useful during the development phase.

3.5 Focus Check

The `FocusCheck` class

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR;

public class FocusCheck : MonoBehaviour
{
```

ensures, that the app stops writing frames if the user presses the OCULUS Home button or quits the application.We declare the main camera (`hauptkamera`) object and its camera component (`kamera`)

```
GameObject hauptkamera;

private Camera kamera;
```

and access the object and the component in the initialization function:

```
void Start()
{
hauptkamera = GameObject.Find("Hauptkamera");

kamera = hauptkamera.GetComponent<Camera>();
```

In Unity we add an event handler (i. e. the function called when the event is raised) by using the overloaded assignment operator. We add handlers for four events:

`InputFocusLost`	The input focus is lost because the game pauses because the user pressed the OCULUS Home button.
`InputFocusAcquired`	The input focus is regained because the game resumes because the user returned from OCULUS Home.
`VrFocusLost`	The VR focus is lost because the user set the headset down.
`VrFocusAcquired`	The VR focus is regained because the user put the headset back on again.

```
OVRManager.InputFocusLost += GamePause;
OVRManager.InputFocusAcquired += GameResume;
OVRManager.VrFocusLost += VRFocusLost;
OVRManager.VrFocusAcquired += VRFocusFound;
}
```

3.5.1 Game pause

If the game is paused

```
void GamePause()
{
```

we turn off the sounds

```
AudioListener.pause = true;
```

and stop the simulation:

```
Time.timeScale = 0.0f;
}
```

3.5.2 Game resume

If the game is resumed

```
void GameResume()
{
```

we switch the sounds back on again

```
AudioListener.pause = false;
```

and resume the simulation

```
Time.timeScale = 1.0f;
```

3.5.3 VRFocusLost

If the user sets the headset down

```
void VRFocusLost()
{
```

we pause the sounds and stop the simulation:

```
AudioListener.pause = true;
Time.timeScale = 0.0f;
```

Additionally, we stop rendering to the headset

```
kamera.stereoTargetEye = StereoTargetEyeMask.None;
```

and disable positional tracking of the headset:

```
UnityEngine.XR.InputTracking.disablePositionalTracking = true;
}
```

3.5.4 VRFocusFound

If the user puts the headset back on again

```
void VRFocusFound()
{
```

we restart sounds, simulation, and rendering and reenable the headset tracking

```
AudioListener.pause = false;
Time.timeScale = 1.0f;
kamera.stereoTargetEye = StereoTargetEyeMask.Both;
UnityEngine.XR.InputTracking.disablePositionalTracking = false;
}
}
```

3.6 Ground collision

We want the controllers to vibrate and the kite to make some noise if it hits the ground (section 2.11.3). In the `Explosion` class

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Explosion : MonoBehaviour
{
```

we declare the audio source (`quelle`) component we attached to every ground contact sensor object

```
public AudioSource quelle;
```

and the sail (`segel`)

```
Rigidbody Segel;
```

During the initialization

```
void Start()
{
```

we access the sound component

```
quelle = GetComponent<AudioSource>();
```

and the sail as the parent of the sensor object:

```
Segel = gameObject.GetComponentInParent<Rigidbody>();
}
```

In every simulation step

```
void Update()
{
```

we check if the kite touches[17] the ground with one of its sensors

```
if (transform.position[1] < 0.01)
{
```

and vibrate both controllers. The strength of the vibration is proportional to the ground impact velocity[18]:

```
  StartCoroutine(Vibriere(-Segel.velocity[1] / 50));
}
```

Three conditions

1. The previous sound has finished.
2. The kite is close enough to the ground.
3. The ground impact velocity[19] is greater than $3\frac{\mathrm{m}}{\mathrm{s}}$.

```
if (
  !quelle.isPlaying &&
  transform.position[1] < 0.01 &&
  Segel.velocity[1] < -3
  )
{
```

have to be met for the ground collision sound to be played:

```
  quelle.Play();
}
}
```

3.6.1 Both controllers vibrate

The coroutine to vibrate both controllers is identical to the one described in section 3.3.2, except for the fact that now the left controller vibrates too:

```
IEnumerator Vibriere(float staerke)
{
OVRInput.SetControllerVibration(
```

[17] Due to numerical issues, we have to detect "ground collision" even if the contact sensor is a few millimeters above the ground. Users do not recognize the difference.
[18] The maximum vibration strength is reached at a devastating crash speed of $50\frac{\mathrm{m}}{\mathrm{s}}$.
[19] Yes, we could make the sound volume proportional to the crash speed ...

```
1f, staerke, OVRInput.Controller.RTouch);

OVRInput.SetControllerVibration(
1f, staerke, OVRInput.Controller.LTouch);

yield return new WaitForSeconds(0.1f);

OVRInput.SetControllerVibration(
0f, 0f, OVRInput.Controller.RTouch);

OVRInput.SetControllerVibration(
0f, 0f, OVRInput.Controller.LTouch);
}
}
```

3.7 Left handle

The `Handle_links` class

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Handle_links : MonoBehaviour
{
```

is attached to the left handle and transfers the motion of the real world left controller to the left handle in virtual reality. In every fixed rate simulation step

```
void FixedUpdate()
{
```

we read the position of the left controller and copy it to the position of the left handle:

```
transform.localPosition =
OVRInput.GetLocalControllerPosition(
OVRInput.Controller.LTouch);
```

Just like that we copy the attitude of the left controller to the attitude of the left handle

```
transform.localRotation =
OVRInput.GetLocalControllerRotation(
OVRInput.Controller.LTouch);
}
}
```

3.8 Right handle

Obviously, we have to copy the position and attitude of the right controller to the corresponding properties of the right handle:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class Handle_rechts : MonoBehaviour
{
void FixedUpdate()
{
transform.localPosition =
OVRInput.GetLocalControllerPosition(
OVRInput.Controller.RTouch);

transform.localRotation =
OVRInput.GetLocalControllerRotation(
OVRInput.Controller.RTouch);
}
}
```

3.9 Setpoint generator

In the tutorial, we use the setpoint generator (`Sollwertgenerator`) class

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Sollwertgenerator : MonoBehaviour
{
```

to generate setpoint sequences for elevation, azimuth, and roll angles for the target kite (section 2.12). We declare the player (`spieler`) object and the setpoint angles

```
GameObject spieler;

public float elevation;
public float azimut;
public float rollwinkel;
```

and access the player during the initialization:

```
void Start()
{
spieler = GameObject.Find("Spieler");
}
```

In every simulation step

```
void FixedUpdate()
{
```

we read the simulation time offset `t_start`

```
float t_start =
spieler.GetComponent<Controllereingaben>().t_start;
```

we buffer in section 3.3 whenever the user starts a new tutorial step by pressing the X button. Additionally, we read the current state of the target kite (`solldrachen_zustand`) indicating the current tutorial step:

```
int solldrachen_zustand =
spieler.GetComponent<Controllereingaben>().solldrachen_zustand;
```

In order to start every tutorial step with an initial time of zero, we subtract the tutorial step time offset from the current time:

```
float t = Time.time - t_start;
```

We chose the maximum setpoint elevation angle ϵ (in degrees)

```
float elevation_max = 2.8114f;
```

according to figure 3.4.

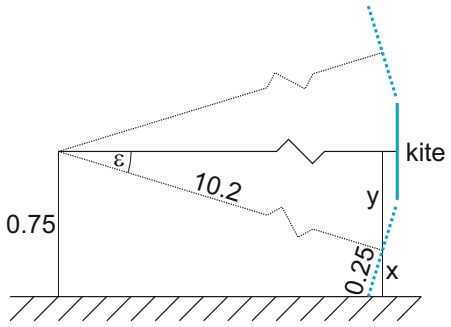

Figure 3.4: Maximum setpoint elevation angle ϵ

In the elevation tutorial, the target kite starts 75 cm above the ground (figure 3.4) and

reaches its maximum[20] elevation angle when the kite touches the ground with its lower[21] vertices. In order to compute the maximum elevation angle, we can[22] find three equations for the three unknowns x, y, and ϵ in figure 3.4. The first equation describes the trivial fact that x and y add up to 75 cm:

$$x + y = 0.75$$

The second equation can be found in the rectangular triangle involving y, the virtual line length of the target kite (10.2), and ϵ:

$$\sin \epsilon = \frac{y}{10.2}$$

For the last equation, we use x, half the kite's height (0.25), and ϵ in the small lower triangle:

$$\cos \epsilon = \frac{x}{0.25}$$

We use MATLAB's Symbolic Math Toolbox to declare the unknowns

```
syms x y epsilon
```

express the equations

```
g_1 = x + y == 0.75;
g_2 = sin (epsilon) == y / 10.2;
g_3 = cos (epsilon) == x / 0.25;
```

and solve the equation system for the maximum elevation angle

```
erg = solve (g_1, g_2, g_3);
erg.epsilon
```

$$\begin{pmatrix} -2\,\mathrm{atan}\left(\frac{\sqrt{10354}}{10} - \frac{51}{5}\right) \\ 2\,\mathrm{atan}\left(\frac{\sqrt{10354}}{10} + \frac{51}{5}\right) \end{pmatrix}$$

which corresponds to a floating point value of:

```
rad2deg (double (erg.epsilon))

ans =

    2.8114
  174.3805
```

[20] The left-hand rotation rule says: If you point your left thumb into the positive direction of the X-axis (i. e. the red vector to your right in figure 2.2) your bended fingers indicate the positive direction of rotation, which is down.

[21] We happily neglect the fact that the kite has a tiny thickness and therefore touches the ground with its lower **back** vertices first.

[22] Alternatively, we could assume that ϵ is very small, that the kite touches the ground approximately vertically and therefore, that $y \approx 0.5$. The approximate maximum elevation angle would then be $\epsilon \approx \arcsin \frac{0.5}{10.2} = 2.8097$.

Depending on the current tutorial (`solldrachen_zustand`)

```
switch (solldrachen_zustand)
{
```

we compute sinusoidal sequences for the corresponding setpoint angles.

If the user has chosen the elevation tutorial

```
case 1:
```

we compute the current elevation angle[23] setpoint:

```
  elevation = elevation_max * (Mathf.Cos(1f * t));
```

and imprint it on the target kite:

```
  transform.rotation =
  Quaternion.Euler(new Vector3(elevation, 0, 0));

  break;
```

In case of the azimuth tutorial

```
case 2:
```

we start at a height of 75 cm above the ground, slowly[24] move the kite 5° to the right, then 5° to the left, and so on:

```
  azimut = 5f * (Mathf.Sin(0.5f * t));

  transform.rotation =
  Quaternion.Euler(new Vector3(0, azimut, 0));

  break;
```

We offer two roll tutorials. In the first (small) roll tutorial

```
case 3:
```

we let the target kite roll ±20° starting at its standard position 75 cm above the ground with a period of $T \approx 6$ sec:

```
  rollwinkel = 20f * (Mathf.Sin(1f * t));

  transform.rotation =
  Quaternion.Euler(new Vector3(0, 0, rollwinkel));

  break;
```

[23] Note that – because of the cosine and the left-hand rule – the up and down motion of the target kite starts at $t = 0$ on the ground ($\epsilon = \epsilon_{max}$).

[24] The angular frequency $\omega = 0.5$ corresponds to a period of $T = \frac{2\pi}{\omega} = 4\pi\,\text{sec} \approx 13\,\text{sec}$.

In the second (large) roll tutorial

```
case 4:
```

we need more space beneath the kite for a roll angle of 70°. Therefore, we lift the kite 2° up[25]

```
elevation = -2;
```

and very slowly ($T \approx 21\,\text{sec}$) roll the target kite ±70°:

```
  rollwinkel = 70f * (Mathf.Sin(0.3f * t));

  transform.rotation =
  Quaternion.Euler(new Vector3(elevation, 0, rollwinkel));

  break;

default:

  break;
}
}
}
```

3.10 Controller

The `Regler` class

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Regler : MonoBehaviour
{
```

could be attached to the handles and used as a closed-loop controller to make the kite follow external commands. Alternatively, it can be attached to the target handles open-loop style in order to act as a flight director [10]. In RevSim's tutorial we want the user to control the kite with her hands. Therefore, we do not activate the `Regler` class on the handles but only on the target handles.

The block diagram in figure 3.5 displays a standard feedback control loop in which the state of the kite is "measured" and compared to the setpoint[26] resulting in an error if

[25] Remember, the kite has a span of 2 m; the standard 75 cm about the ground might now become a little tight. $10.2 \cdot \sin 2° \approx 0.36$ gives us the necessary additional ground clearance.

[26] During the tutorial, the setpoint is generated in the setpoint generator class in section 3.9 and used to position the target kite directly (figure 3.5).

both values are not identical. The error is then fed into the feedback controller that – in an automatic control application – would use the handles to make the kite follow the target kite (dotted line in figure 3.5). In RevSim, we use the signal for the target handles (flight director) **showing** the user how to rotate her own handles if she wanted the kite to follow the target kite.

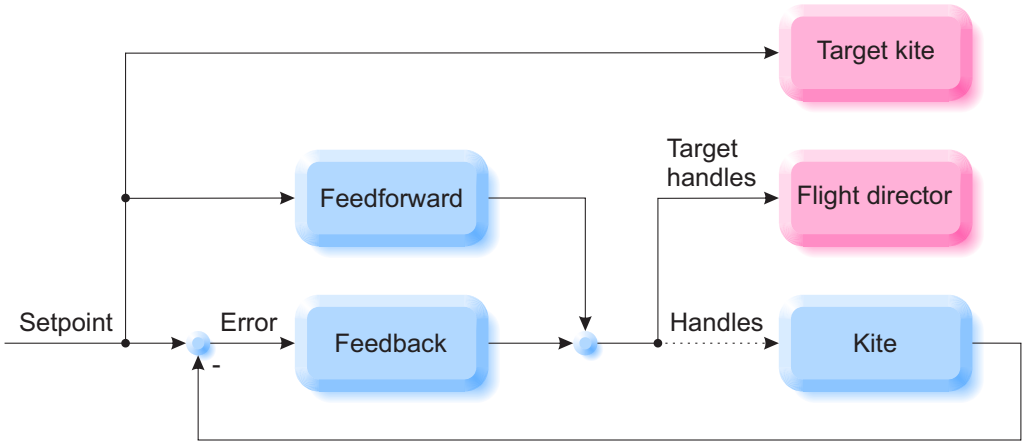

Figure 3.5: Control block diagram

Additionally to the classic feedback loop, we use a feedforward controller (figure 3.5) that computes an additional handle signal directly from the setpoint. For example, we know we have to pitch both handles simultaneously if we want the kite to lift off. We know from experience that we need a pitch offset of about 12° to compensate the weight of the kite by the aerodynamic lift force. We also know that – as long as the kite is not too high above the ground – its elevation angle is linearly depending on the handle pitch angle. Therefore, the feedforward controller in figure 3.5 could look like figure 3.6.

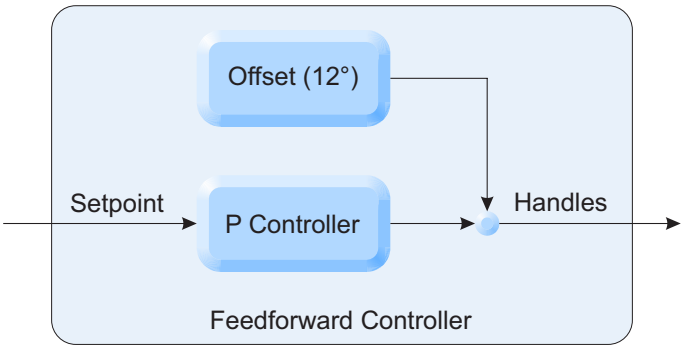

Figure 3.6: Elevation (pitch) feedforward controller

In the class, we define a few objects

```
GameObject solldrachen;
```

```
GameObject segel_rechts;
GameObject segel_links;
GameObject segel_dreh;
GameObject spieler;
GameObject handle_rechts;
GameObject handle_links;
```

variables, and controller parameters:

```
public float drachen_x;
public float drachen_y;
public float drachen_z;

public float elevation_soll;
public float elevation_ist;
public float elevation_regelfehler;
public float elevation_vorsteuerung_verstaerkung;
public float elevation_vorsteuerung_offset;
public float elevation_regler_verstaerkung;
public float elevation_stellgroesse;

public float rollwinkel_soll;
public float rollwinkel_ist;
public float rollwinkel_regelfehler;
public float rollwinkel_vorsteuerung_verstaerkung;
public float rollwinkel_regler_verstaerkung;
public float rollwinkel_stellgroesse;

public float azimut_soll;
public float azimut_ist;
public float azimut_regelfehler;
public float azimut_vorsteuerung_verstaerkung;
public float azimut_regler_verstaerkung;
public float azimut_stellgroesse;

public float handle_rollen;
```

In the initialization function

```
void Start()
{
```

we access the objects

```
solldrachen = GameObject.Find("Solldrachen");
segel_rechts = GameObject.Find("Segel rechts");
segel_links = GameObject.Find("Segel links");
segel_rechts_aussen = GameObject.Find("Segel rechts aussen");
segel_links_aussen = GameObject.Find("Segel links aussen");
```

```
segel_dreh = GameObject.Find("Segel dreh");
spieler = GameObject.Find("Spieler");
handle_rechts = GameObject.Find("Handle rechts");
handle_links = GameObject.Find("Handle links");
```

and initialize the controller parameters.

```
elevation_vorsteuerung_offset = -11.63f;
elevation_vorsteuerung_verstaerkung = 1f;
elevation_regler_verstaerkung = 10f;

rollwinkel_vorsteuerung_verstaerkung = 0f;
rollwinkel_regler_verstaerkung = 1f;

azimut_regler_verstaerkung = 10f;
azimut_vorsteuerung_verstaerkung = 1f;

handle_rollen = 1f;
}
```

Please note the already discussed negative feedforward offset (`elevation_vorsteuerung_offset`) of -11.63 because of the left-hand rule. The feedforward P controller gain (`elevation_vorsteuerung_verstaerkung`) has a value of 1, assuming 1° of kite elevation increase with every 1° additional handle pitch. The feedback controller gain (`elevation_regler_verstaerkung`) of 10 ensures a fast control reaction with a small stationary error.

During the roll tutorial, we lift the leading edge of the target kite to 1 m and then roll it about a point in the "middle" of the kite, half way down[27] (25 cm) from the leading edge (figure 3.7).

[27] Yes, the center of mass of this isosceles triangle is at $\frac{1}{3} \cdot 50\,\text{cm} = 16.\bar{6}\,\text{cm}$ down from the leading edge, but we roll the (target) handles and correspondingly the (target) kite about the middle of the handle object (axis tripod in figure 2.58).

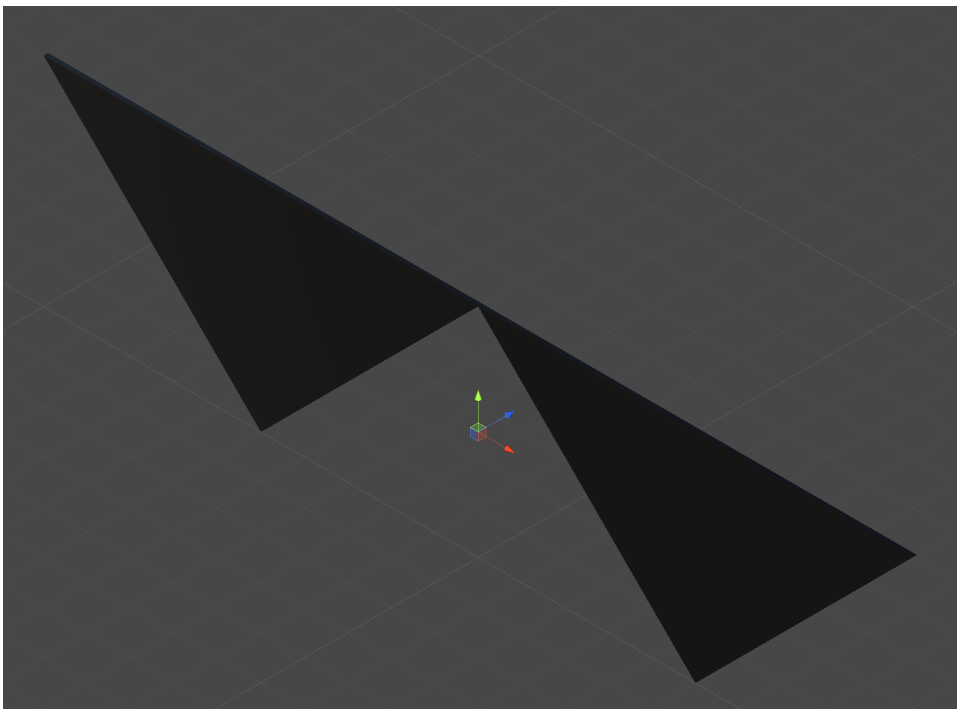

Figure 3.7: Point of rotation

Therefore, we define a new invisible game object `segel_dreh` (axis tripod in figure 3.7) as the kite's point of rotation. In every simulation step

```
void FixedUpdate()
{
```

we obtain its position, with its origin raised up 75 cm

```
drachen_x = segel_dreh.transform.position.x;
drachen_y = segel_dreh.transform.position.y - 0.75f;
drachen_z = segel_dreh.transform.position.z;
```

and read the current step of the tutorial:

```
int solldrachen_zustand =
spieler.GetComponent<Controllereingaben>().solldrachen_zustand;
```

If we are in the large roll angle tutorial

```
if (solldrachen_zustand == 4)
{
```

we switch off the roll feedforward

```
  rollwinkel_vorsteuerung_verstaerkung = 0f;
```

and indicate that we want to roll the complete handle object:

```
  handle_rollen = 1f;
}
```

In any other tutorial step we use an appropriate roll feedforward gain and switch off the handle roll:

```
else
{
  rollwinkel_vorsteuerung_verstaerkung = 0.062f;
  handle_rollen = 0f;
}
```

Elevation angle According to figure 3.5, we compute the necessary handle pitch angle η to achieve a certain elevation setpoint angle ϵ_s by taking the control error between the the elevation setpoint angle ϵ_s and the actual elevation angle ϵ, amplify this error by the feedback controller gain K_{FB}, and add the feedforward path according to figure 3.6:

$$\eta = K_{FB} \cdot (\epsilon_s - \epsilon) + K_{FF} \cdot \epsilon_s + \mathit{offset} \tag{3.5}$$

We now run into a little problem: Unity does not know about negative angles and returns the elevation setpoint angle ϵ_s (`elevation_soll`)

```
elevation_soll = solldrachen.transform.rotation.eulerAngles.x;
```

in the range

$$0° \leq \epsilon_s < 360°$$

while the arcsine function we use to compute the actual elevation angle ϵ has a range of

$$-90° \leq \epsilon < 90°$$

In order to make $\epsilon_s = 350°$ comparable to $\epsilon = -10°$, we express setpoint angles in the third[28] and fourth quadrant as negative numbers:

```
if (elevation_soll > 180)
{
  elevation_soll -= 360;
}
```

We compute the actual elevation angle ϵ according to figure 3.4 with a line length of 10 m:

```
elevation_ist = -Mathf.Rad2Deg * Mathf.Asin(drachen_y / 10f);
```

The elevation control error is the difference between setpoint ϵ_s and actual angle ϵ:

```
elevation_regelfehler = elevation_soll - elevation_ist;
```

[28] In the tutorial, we assume the kite always to be in front of (and not behind) the pilot; the $|\epsilon| \leq 90°$ of the arcsine function should therefore be sufficient.

Finally, we compute the necessary handle pitch angle η (`elevation_stellgroesse`) to achieve the elevation setpoint angle according to equation (3.5):

```
elevation_stellgroesse =
elevation_regler_verstaerkung * elevation_regelfehler +
elevation_vorsteuerung_verstaerkung * elevation_soll +
elevation_vorsteuerung_offset;
```

Roll angle The user has two different ways to make the kite roll: For a small, fast roll, she can pitch the handles asymmetrically (upper end of one handle towards her, upper end of the other handle away from her). For larger roll angles, she might want to slowly roll both handles as a whole, synchronously[29] about the origin of the Handles object in figure 2.15. We use this second way to roll the kite in the fourth tutorial step, described on page 66.

The computation of the necessary handle roll angle to achieve a certain kite roll angle is similar to the elevation control in the previous paragraph: We retrieve the roll angle of the target kite in order to use it as a setpoint (`rollwinkel_soll`)

```
rollwinkel_soll = solldrachen.transform.rotation.eulerAngles.z;
```

and map it to the appropriate range:

```
if (rollwinkel_soll > 180)
{
  rollwinkel_soll -= 360;
}
```

For the computation of the actual kite's roll angle we retrieve the roll angles of the right

```
float segel_rechts_rollwinkel =
segel_rechts.transform.eulerAngles.z;

if (segel_rechts_rollwinkel > 180)
{
  segel_rechts_rollwinkel -= 360;
}
```

and the left sail (both mapped into the correct range)

```
float segel_links_rollwinkel =
segel_links.transform.eulerAngles.z;

if (segel_links_rollwinkel > 180)
{
  segel_links_rollwinkel -= 360;
}
```

[29] Just like you would use a steering wheel of a car.

and compute their mean[30]:

```
rollwinkel_ist =
(segel_rechts_rollwinkel + segel_links_rollwinkel) / 2f;

if (rollwinkel_ist > 180)
{
  rollwinkel_ist -= 360;
}
```

As in the elevation case, we compute the roll control error

```
rollwinkel_regelfehler = rollwinkel_soll - rollwinkel_ist;
```

and the commanded handle roll angle utilizing feedback (`rollwinkel_regler_verstaerkung`) and feedforward[31] (`rollwinkel_vorsteuerung_verstaerkung`) control:

```
rollwinkel_stellgroesse =
rollwinkel_regler_verstaerkung * rollwinkel_regelfehler +
rollwinkel_vorsteuerung_verstaerkung * rollwinkel_soll;
```

In the tutorial, very large target handle asymmetrical roll angles might become difficult to interpret. Therefore, we limit the handle roll angle magnitude to 40°:

```
float rollwinkel_stellgroesse_max = 40;

if (rollwinkel_stellgroesse > rollwinkel_stellgroesse_max)
{
  rollwinkel_stellgroesse = rollwinkel_stellgroesse_max;
}

else if (rollwinkel_stellgroesse < -rollwinkel_stellgroesse_max)
{
  rollwinkel_stellgroesse = -rollwinkel_stellgroesse_max;
}
```

For the asymmetrical roll, one handle has to pitch forward and the other handle has to pitch backwards. Therefore, we have to access each handle separately by using the `GetChild` method:

```
transform.GetChild(0).localRotation =
Quaternion.Euler(new Vector3(-rollwinkel_stellgroesse, 0, 0));

transform.GetChild(1).localRotation =
Quaternion.Euler(new Vector3(rollwinkel_stellgroesse, 0, 0));
```

[30] Even though the roll angles of both sails should always be identical since they are connected via a inflexible hinge, we found the simulation to be a bit more stable if we used the average of both sails, possibly for numerical reasons.

[31] We do not need an offset in the roll feedforward controller because the roll degree of freedom is symmetrical. We need a handle pitch offset to float the kite above the ground with a zero elevation angle but we do not need a handle roll offset to achieve a steady kite roll angle of zero.

Azimuth angle To move the kite to the left or the right, we rotate the whole handle object around its vertical axis. The computation of the necessary handle azimuth angle is similar to the already discussed handle roll angle computation. We read the azimuth angle setpoint

```
azimut_soll = solldrachen.transform.rotation.eulerAngles.y;
```

and transform it into a useful range:

```
if (azimut_soll > 180)
{
  azimut_soll -= 360;
}
```

We compute the current azimuth angle of the kite using its position in the X-Z-plane

```
azimut_ist = Mathf.Rad2Deg * Mathf.Atan2(drachen_x, drachen_z);
```

and compute the error as the difference of setpoint and actual value:

```
azimut_regelfehler = azimut_soll - azimut_ist;
```

Finally, we compute the necessary handle azimuth angle (`azimut_stellgroesse`):

```
azimut_stellgroesse =
azimut_vorsteuerung_verstaerkung * azimut_soll +
azimut_regler_verstaerkung * azimut_regelfehler;
```

Handle setting Now that we computed all necessary handle angles, we can finally set the attitude of the handles. We have already set the binary flag `handle_rollen` to 1 or 0 depending on the current tutorial step on page 71; we can now use it as a gain in order to decide whether or not we want the handles to roll as a whole:

```
transform.localRotation = Quaternion.Euler(new Vector3(
elevation_stellgroesse,
azimut_stellgroesse,
handle_rollen * rollwinkel_soll));
```

The user can hold her handles at an arbitrary height. We want the target handles to automatically adjust their height to the user defined handle height. We read the position of the target handles[32]

```
Vector3 sollhandles_position = transform.position;
```

compute the height of the target handles as the mean of the heights of both actual handles

[32]Again, we did not find a way to adjust the Y-component of the position vector directly. Any idea?

```
sollhandles_position.y =
(
handle_rechts.transform.position.y +
handle_links.transform.position.y
) / 2;
```

and transfer the adjusted position vector back to the target handles

```
transform.position = sollhandles_position;
}
}
```

3.11 Lines

As already discussed on page 9 and in section 2.10, we "fake" the lines between the handles and the kite by four `Line Renderer` objects that simply draw polygons without any physical properties.

In the line (`seil`) function

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class seil : MonoBehaviour
{
```

we declare the line renderer object

```
LineRenderer line_renderer;
```

and since we use the same script for all four lines, we also have to declare the begin (`Anfang`) and end (`Ende`) of every line and define them manually (figure 2.48):

```
public GameObject anfang;
public GameObject ende;
```

Additionally, we declare the player (`spieler`) object in order to retrieve the current line length:

```
GameObject spieler;
```

We use 101 vertices (100 segments) for each line

```
static readonly int n_punkte = 101;
```

and declare the corresponding polygon vector:

```
private Vector3[] neue_position = new Vector3[n_punkte];
```

Since we want that part of the lines potentially piling up the ground to look a bit more[33] "random", every line gets its own superimposed sine phase:

```
private float phase;
```

During the initialization

```
void Start()
{
```

we access the player

```
spieler = GameObject.Find("Spieler");
```

and tell the polygon that it has 101 vertices:

```
line_renderer = GetComponent<LineRenderer>();
line_renderer.positionCount = n_punkte;
```

Using the name[34] of the sail the current line is attached to, we give each line its own phase:

```
if (ende.name == "Segel rechts oben")
{
  phase = 0;
}
else if (ende.name == "Segel rechts unten")
{
  phase = 20;
}
else if (ende.name == "Segel links oben")
{
  phase = 40;
}
else if (ende.name == "Segel links unten")
{
  phase = 60;
}
}
```

In every graphical simulation step

```
void Update()
{
```

[33] We can not utilize an actual random number generator, because these random numbers would be totally different in every simulation step, making the lines on the ground jiggle around absolutely unrealistically. Yes, we could compute an array of random numbers in advance and than use sections of that array ...

[34] This seems to be more "quick-and-dirty" than good programming style. What would be the proper way to address the specific current line?

we read the current line length (`seillaenge`)

```
float seillaenge =
spieler.GetComponent<Controllereingaben>().seillaenge;
```

and the current distance (`abstand`) between the handle and the kite

```
  float abstand = Vector3.Distance(
  ende.transform.position,
  anfang.transform.position);
```

and start a loop over every polygon vertex:

```
for (int i = 0; i < n_punkte; i++)
{
```

We need a floating point parameter `t` (running from 0.0 to 1.0) to compute the position of each vertex:

```
  float t = (float)i / (n_punkte - 1);
```

According to figure 3.8, we want the line to sag down (blue curve) if the distance between the handles and the kite (bottom left part of the red line) is less[35] than the line length (entire red line).

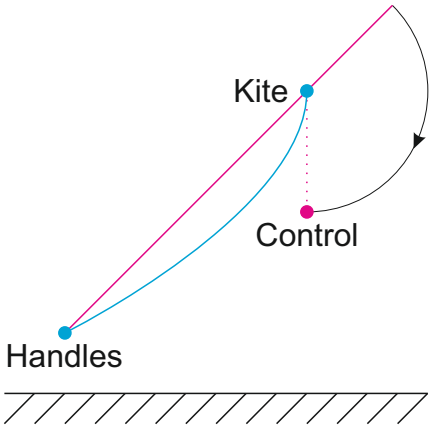

Figure 3.8: Bézier line emulation

One way to achieve this is to utilize a Bézier curve [12]. A quadratic Bézier curve uses three points (P_1: start, P_2: control, P_3: end) to compute intermediate points $P(t)$ via

$$P(t) = (1-t)^2 \cdot P_1 + 2 \cdot (1-t) \cdot t \cdot P_2 + t^2 \cdot P_3, \quad 0 \leq t \leq 1 \tag{3.6}$$

Naturally, we chose P_1 at the handle and P_3 at the kite, leaving us with a free choice of the control point P_2. In a perfect world, we would now use the control point to ensure

[35] This happens if the user rapidly increases the line length or if she quickly decreases the angle of attack until the kite loses its lift and floats downwards (figure 3.10).

that the line integral (or at least the sum of the polygon segment lengths) of the blue curve in figure 3.8 equaled the line length and that its curvature was shaped by the wind and the own weight of the line. In RevSim, we simply rotate the excess part of the line length that sticks out past the kite in figure 3.8 down below the kite and thus obtain the control point P_2.

There is absolutely no mathematical or physical justification for this simplified approach but it ensures that

1. The line is straight if the distance between the handles and the kite equals the line length; the control point P_2 is identical with the kite P_3 in that case.
2. The sagging becomes more pronounced if the difference between the handle-kite-distance and the line length increases.
3. It looks quite natural, imitating the wind induced shape of the blue curve in figure 3.8.

We are now ready to pour equation (3.6) into code:

```
neue_position[i] =
(1 - t) * (1 - t) * anfang.transform.position +
2 * (1 - t) * t * (
ende.transform.position +
new Vector3(0, abstand - seillaenge, 0)) +
t * t * ende.transform.position;
```

On the ground If the kite has a low altitude and is close enough to the user, part of the Bézier curve extends below ground level. Since the curve does not have a rigid body, we have to take care of the shape of the lines on the ground by ourselves. If the current Bézier vertex is below the ground

```
if (neue_position[i].y < 0)
{
```

we superimpose three sinusoidal functions with different frequencies (figure 3.9)

$$\Delta x = 0.02(\sin(9z + \phi) + \sin(5z + \phi) + \sin(3z + \phi))$$

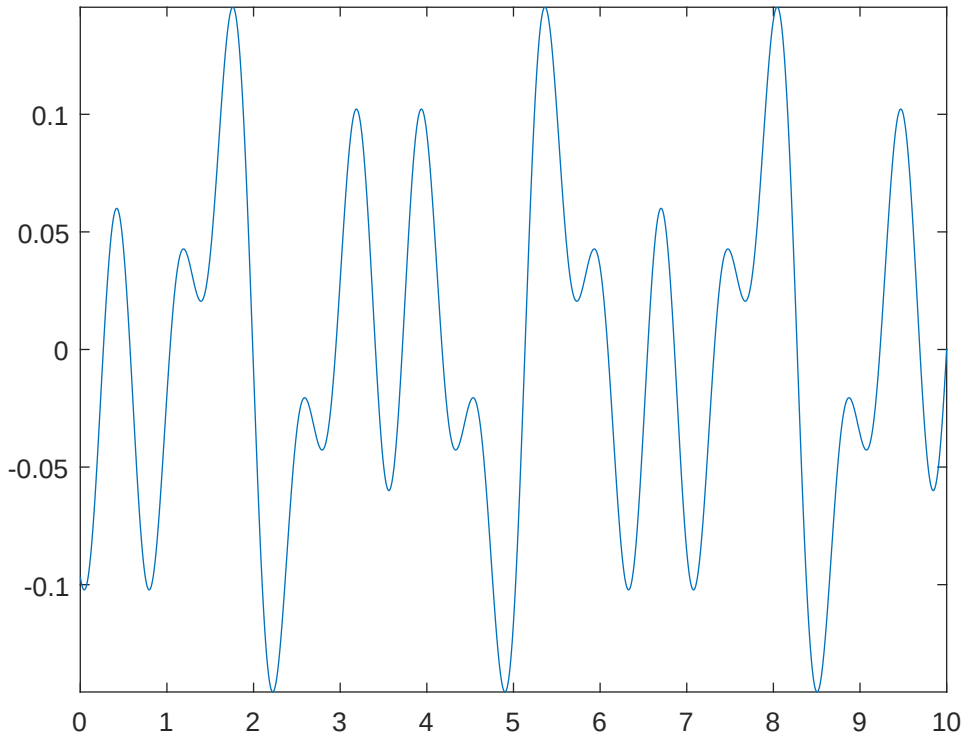

Figure 3.9: Superimposed sinusoidal functions (Δx over z)

and add the value to the X-component (left-right) of the vertex with an amplitude of 2 cm:

```
    neue_position[i].x += 0.02f * (
    Mathf.Sin(9 * (neue_position[i].z + phase)) +
    Mathf.Sin(5 * (neue_position[i].z + phase)) +
    Mathf.Sin(3 * (neue_position[i].z + phase)));
```

In order to make the vertex visible, we move it one millimeter above the ground

```
    neue_position[i].y = 0.001f;
  }
}
```

Finally, we hand all computed vertices over to the polygon:

```
line_renderer.SetPositions(neue_position);
}
}
```

Since every line has its own phase offset (picking a different section of the graph in figure 3.9) the lines on the ground in figure 3.10 look adequately[36] random.

[36] One drawback of the simple way we model the lines on the ground is the fact that every motion of the handles also moves the whole line on the ground and even that section from the ground to the kite (figure 3.10).

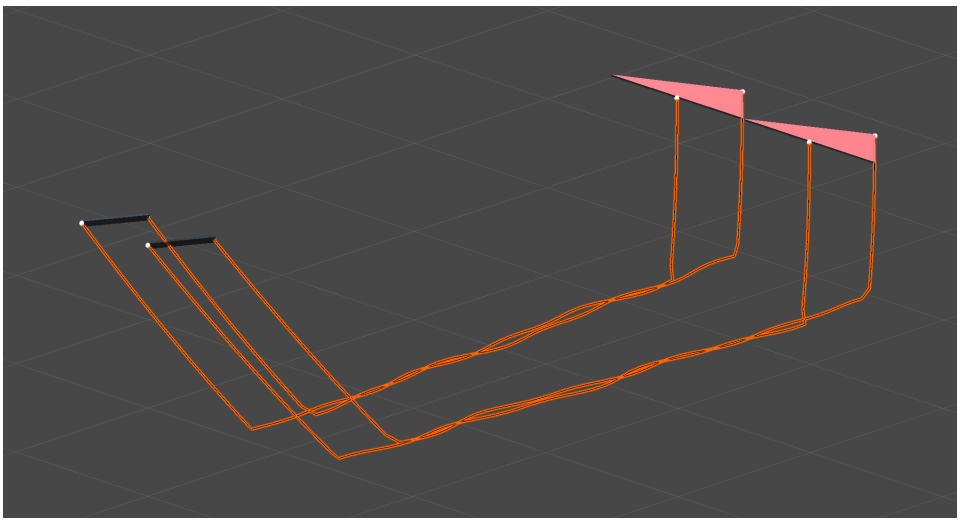

Figure 3.10: Lines on the ground (lines selected for better recognizability)

3.12 Water drag

Water generates more drag than air. Therefore, we want a part of the kite that is underwater to move slower than the rest of the kite: If e. g. the kite moves fast, horizontally just above the sea surface and (accidentally) one of its tips dips into the water, a significant rolling moment should occur. Additionally, we want to give the user a chance to slowly "fly" the kite completely[37] in the ocean; e. g. to relaunch it from underwater.

Unity makes it very comfortable to model this behavior: We just have to position child game objects at every vertex of the kite (partly visible in figure 2.53) and introduce an additional force vector to the corresponding sail at the vertex location if this vertex[38] is below the ocean surface. The corresponding moment vector is than automatically computed and applied by Unity's friendly physics engines.

In the water drag (`Wasserkraft`) function, attached to every kite vertex object

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Wasserkraft : MonoBehaviour
{
```

[37] Since the `Water Prefab` makes the water semitransparent you can still see the kite if it is not too deep below the surface.

[38] Yes, we could precisely compute the affected underwater areas and their angles with respect to the velocity vector in order to come up with the exact magnitudes and locations of the additional water forces. On the other hand, simple single vertex forces seem to create a satisfactory illusion of kite water interactions.

we declare

```
Rigidbody Segel;
```

and initially access the sail (`Segel`) as the parent of the current vertex child:

```
void Start()
{
Segel = gameObject.GetComponentInParent<Rigidbody>();
}
```

In every (physical) simulation step

```
void FixedUpdate()
{
```

we test if the current vertex is below sea [39] level

```
if (transform.position[1] < 0f)
{
```

and add a force[40] to the sail at the location of the vertex acting in the opposite direction of the current velocity vector

```
  Segel.AddForceAtPosition(
  -Segel.velocity, transform.position);
}
}
}
```

3.13 Wind velocity

As already stated in section 2.4.1, we use the `Wind Zone` object just to make the palms bend in a realistic animated fashion; the much more complex aerodynamic force computation is done in section 3.1. We add the wind velocity (`Windgeschwindigkeit`) function to the `Wind Zone` object

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Windgeschwindigkeit : MonoBehaviour
{
```

[39] As long as the kite flies above ground (and not water), the collision detection algorithms of the physics engines ensure that no part of the kite can dunk into the ground.

[40] Empirically, we set the value of the additional water force coefficient to 1N per $1\frac{\mathrm{m}}{\mathrm{s}}$. Fun fact: If you increase this coefficient significantly, you can create cute comic-like bouncy effects if the user tries to ram the kite into the water with high velocity.

and declare the wind zone and the player:

```
WindZone wind;
GameObject spieler;
```

In the initialization, we address the wind zone and the player (`spieler`):

```
void Start()
{
wind = gameObject.GetComponent<WindZone>();

spieler = GameObject.Find("Spieler");
}
```

In every simulation step

```
void FixedUpdate()
{
```

we read the user controlled wind speed setting (page 50) and use it for the turbulent

```
wind.windTurbulence =
0.1f * spieler.GetComponent<Controllereingaben>().
windgeschwindigkeit;
```

and the linear wind zone parameter:

```
wind.windMain = wind.windTurbulence;
}
}
```

Bibliography

[1] Revolution Kites. (2019). [Online]. Available: https://revkites.com/

[2] Unity. (2019). [Online]. Available: https://unity.com

[3] Oculus. (2019). [Online]. Available: https://www.oculus.com/

[4] J. J. Buchholz. (2019) RevSim. [Online]. Available: https://m-server.fk5.hs-bremen.de/revsim/revsim.html

[5] Sketchup. (2019). [Online]. Available: https://www.sketchup.com/

[6] CorelDraw. (2019). [Online]. Available: https://www.coreldraw.com/de/

[7] Wikipedia. (2019) Z-fighting. [Online]. Available: https://en.wikipedia.org/wiki/Z-fighting

[8] A. Shamaluev. (2019) AShamaluevMusic. [Online]. Available: https://www.ashamaluevmusic.com/

[9] W. Commons. (2019) Sound of waves on Nauset Beach after sunset. [Online]. Available: https://commons.wikimedia.org/wiki/File:NausetBeach.ogg

[10] Wikipedia. (2019) Flight director (aeronautics). [Online]. Available: https://en.wikipedia.org/wiki/Flight_director_(aeronautics)

[11] J. J. Buchholz. (2019) Regelungstechnik und Flugregler. [Online]. Available: https://m-server.fk5.hs-bremen.de/rtfr/skript/skript10.pdf

[12] Wikipedia. (2019) Bézier curve. [Online]. Available: https://en.wikipedia.org/wiki/Bezier_curve