Installation

This Unity 3D plugin is intended to be used with the Pupil Capture/Service applications, the related eye tracking hardware and Virtual/Augmented Reality headsets (Head Mounted Displays). You can get the latest versions here

https://github.com/pupil-labs/pupil/releases/latest

The plugin is developed and tested with the latest versions of Unity 3D (2017 releases), publicly available at

https://store.unity.com/

Much like the Pupil software, this plugin is developed in the open and made available on GitHub

https://github.com/pupil-labs/hmd-eyes

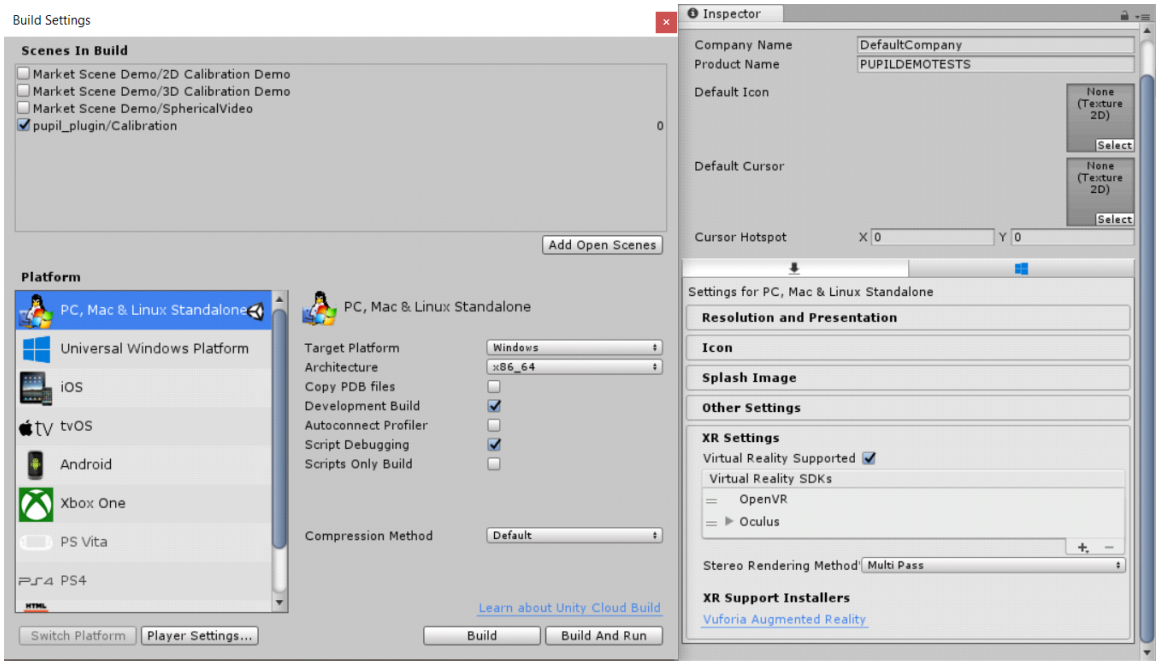
Once downloaded, three different projects are included in the main directory

- python_reference_client
- unity_pupil_plugin_hololens
- unity_pupil_plugin_vr

This document will describe the use of the latter two in more detail. Both share a common code basis but also contain implementations for their respective platforms. Here, VR is the main focus with the most features and demos, while the HoloLens implementation had to be stripped down due to the UWP environment.

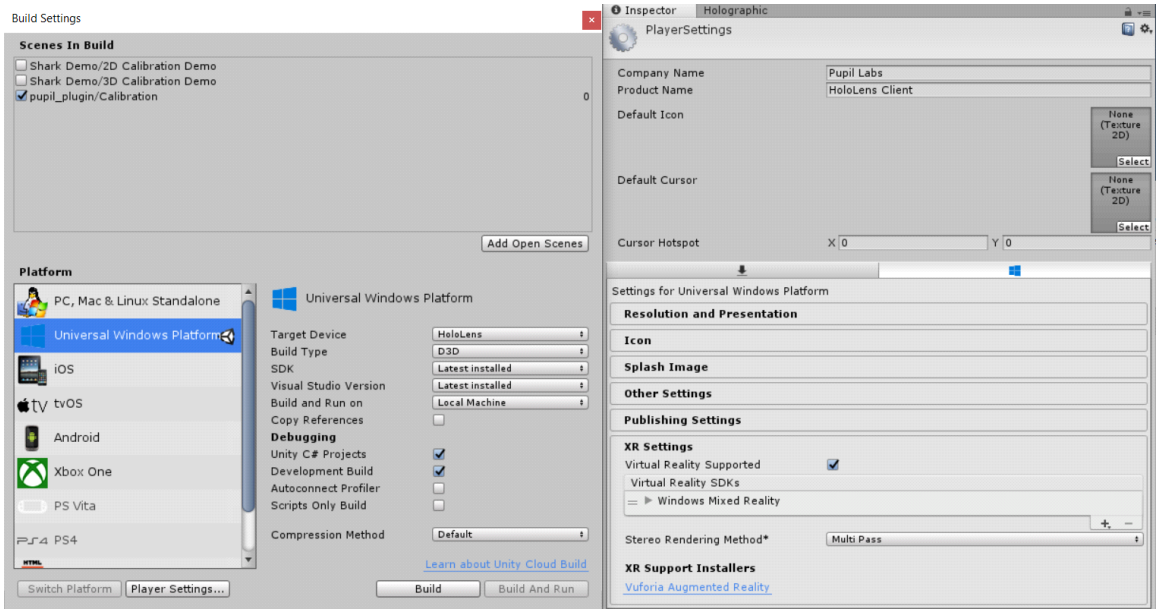
To open either project, start Unity and select "unity_pupil_plugin_hololens" or "unity_pupil_plugin_vr" as source folder.

The VR Build and Player Settings should look like this



The software has been tested for both Oculus and OpenVR SDKs. Please make sure to select the correct one for your type of headset.

As for HoloLens



The "Build Settings" debugging option "Unity C# Projects" will allow you to debug the code at runtime while deploying to HoloLens from Visual Studio. As an alternative debugging option, Unity's "Holographic Emulation" is also supported.

As for the Player Settings, make sure the following "Capabilities" are enabled under "Publishing Settings"

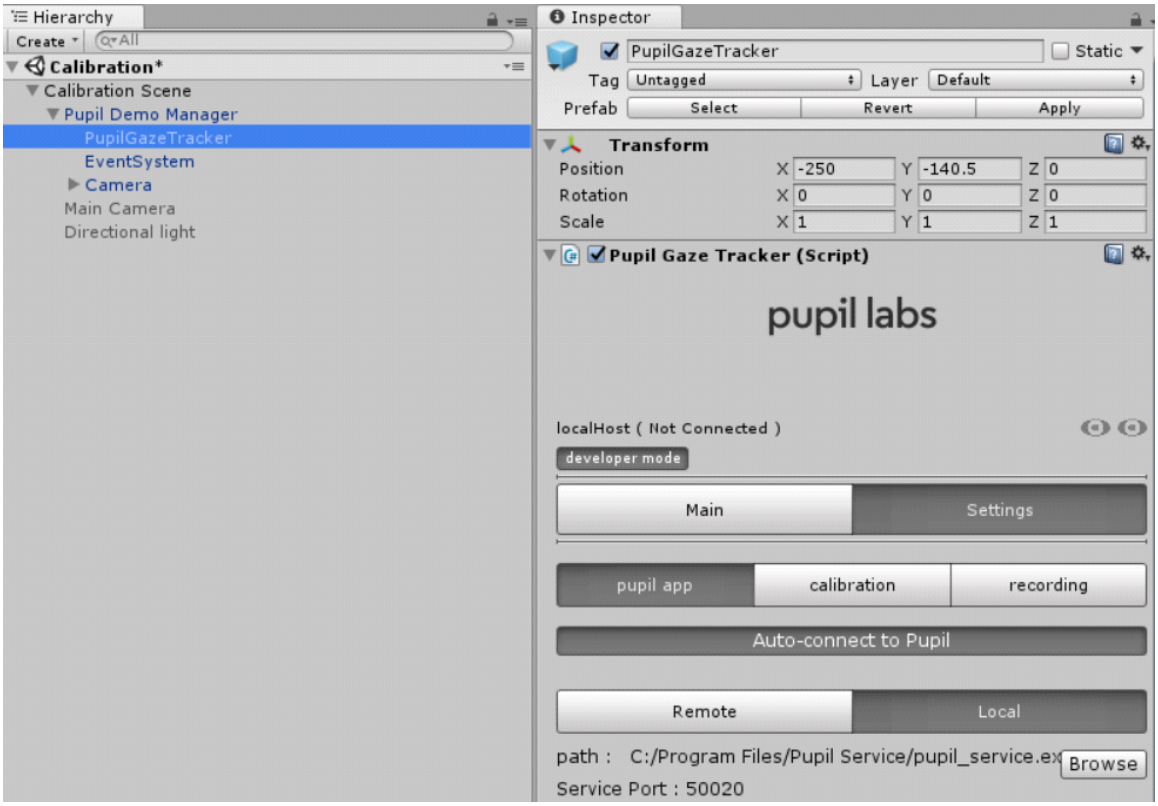
InternetClient, PrivateNetworkClientServer, Microphone, SpatialPerception

General Setup

The Unity scene "pupil_plugin/Calibration.unity", which is included in both projects, serves as a good starting point for VR and HoloLens development.

In case of VR, two things are important for the initial steps

- The "PupilGazeTracker" gameobject offers an Inspector GUI to set how to communicate with Pupil Capture/Service. Either can run on the same "Local" computer or on a "Remote" PC. On local PCs, you have the option to let Unity start Pupil Capture/Service by setting the "Path" to the executable (click the "Browse" button).

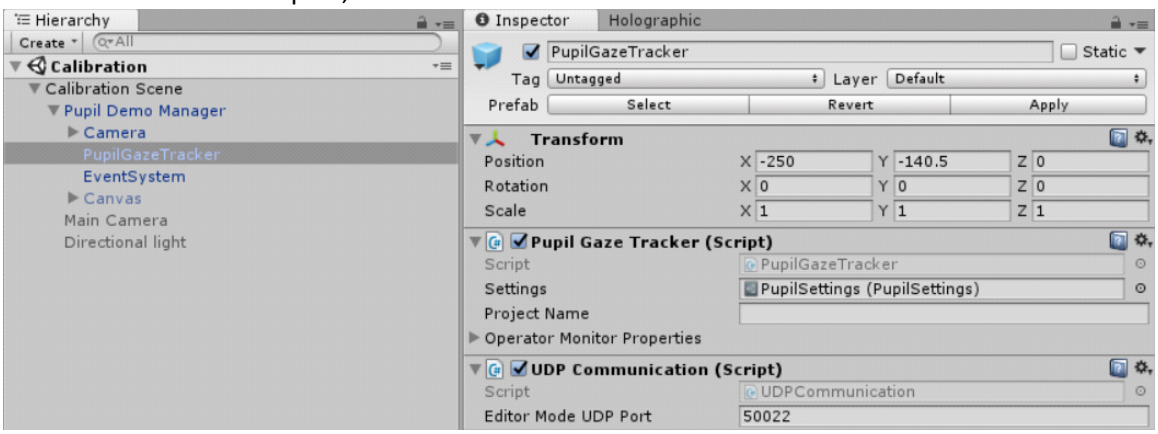


- "PupilSettings" is located in "pupil_plugin/Resources" and is used to save global settings that are not specific to a scene. Important for the initial steps is the "Connection", which lets you set both the IP (in case of Pupil running remotely) as well as the port over which to communicate. If the standard port of 50020 does not work for you, please set an alternative here and also make sure that the same port is set in Pupil Capture (more on that, later).

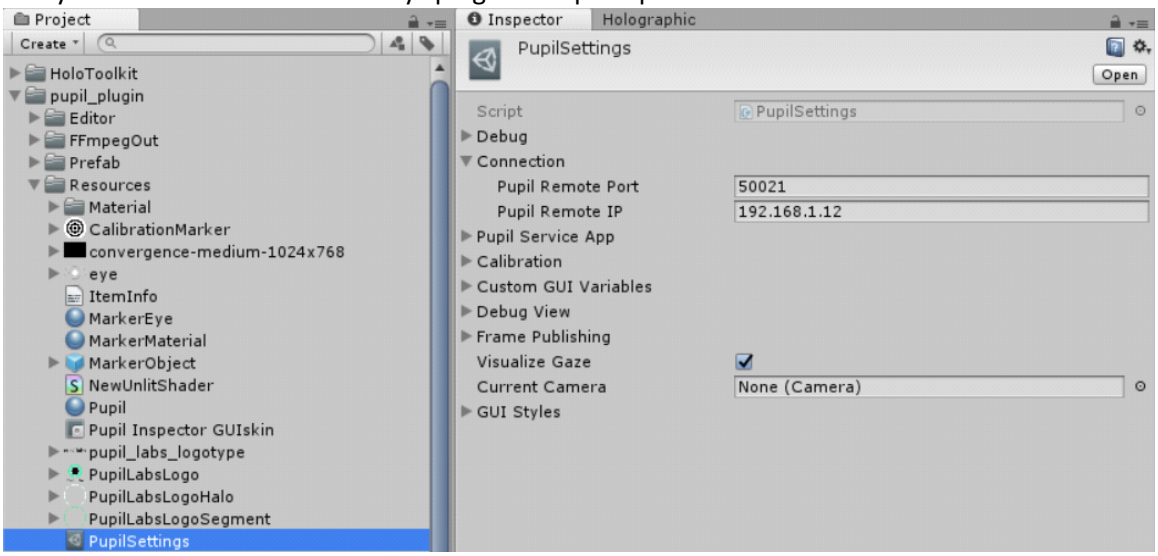
Differences for HoloLens

- As Pupil Capture/Service does not support native execution on HoloLens/UWP but has to run on a remote PC, "PupilGazeTracker" does not need to offer any options to change this. In contrast to its VR pendant, the gameobject includes an additional component called "UDP Communication". Here, an additional port (named "Editor Mode UDP Port") can be set. It is needed, if you use "Holographic Emulation" and (at least on Windows machines) needs to be

different from the main port, discussed next.

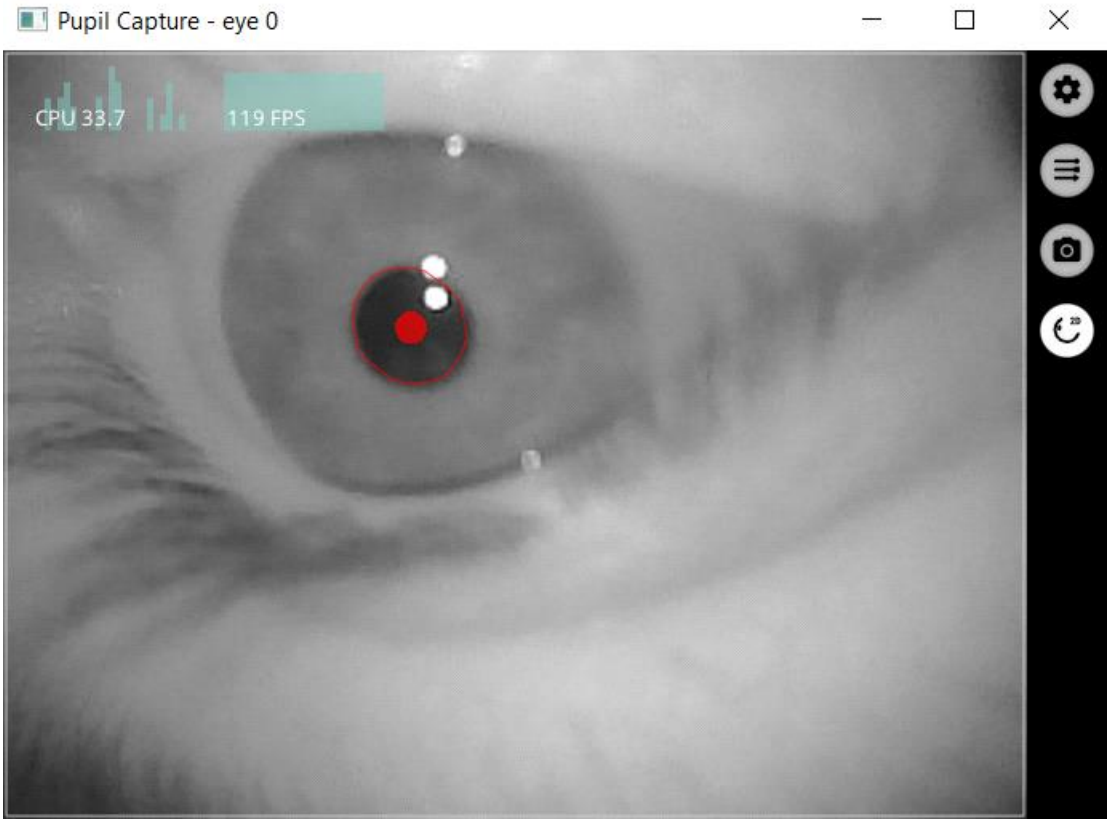


- As mentioned before, the HoloLens implementation needed to be customized to be able to run it in the UWP environment. Our solution relies on UDP to communicate data between the Unity plugin and Pupil. Select "PupilSettings" in the "Project" tab and set the IP of the PC Pupil is running on under "Connection". Please also make sure the port set here corresponds with the one you set in the "HoloLens Relay" plugin in Pupil Capture.



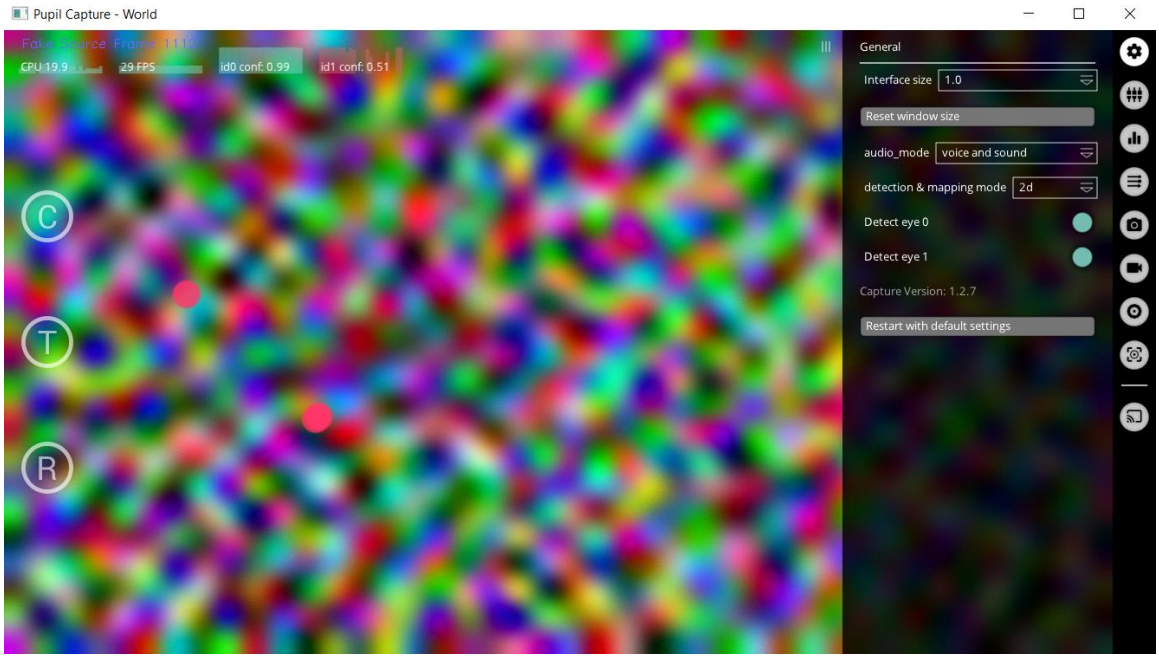
Pupil Capture/Service setup

My personal recommendation is to start development with Pupil Capture, as it gives the most options for feedback if things are not set correctly or if something goes wrong. Once everything is working as intended, Pupil Service is a good solution to minimize the windows on screen. In our case, one or two eye windows, depending on your setup

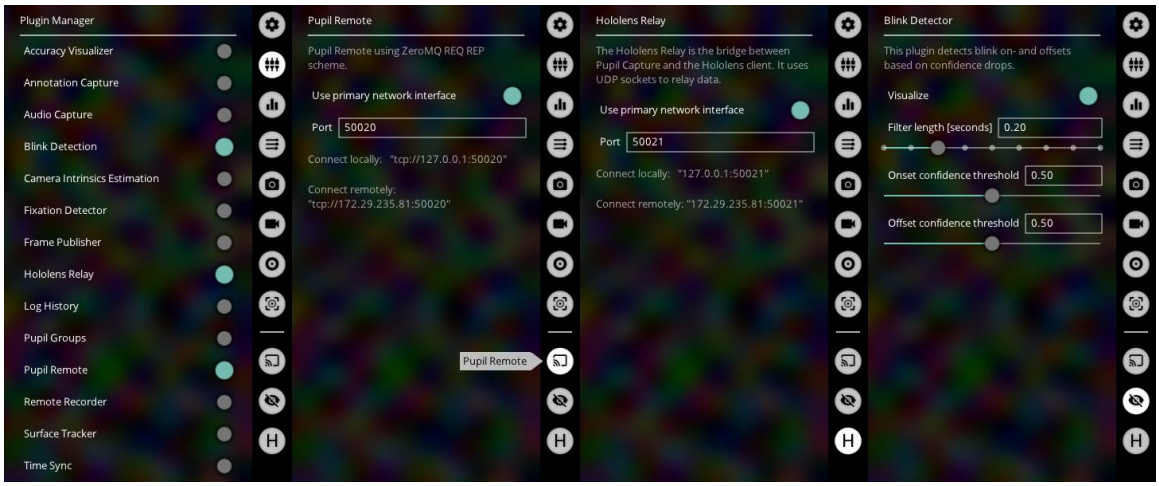


During the tracking process, one instance of these windows should always be open per eye camera.

In its main GUI, Pupil Capture gives real time feedback on how the confidence level is doing

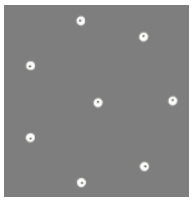


One of the most common problems is the standard communication port "50020" being blocked. In that case, Pupil Capture will chose an alternate one. To be able to check this, activate "Pupil Remote" in Plugin Manager

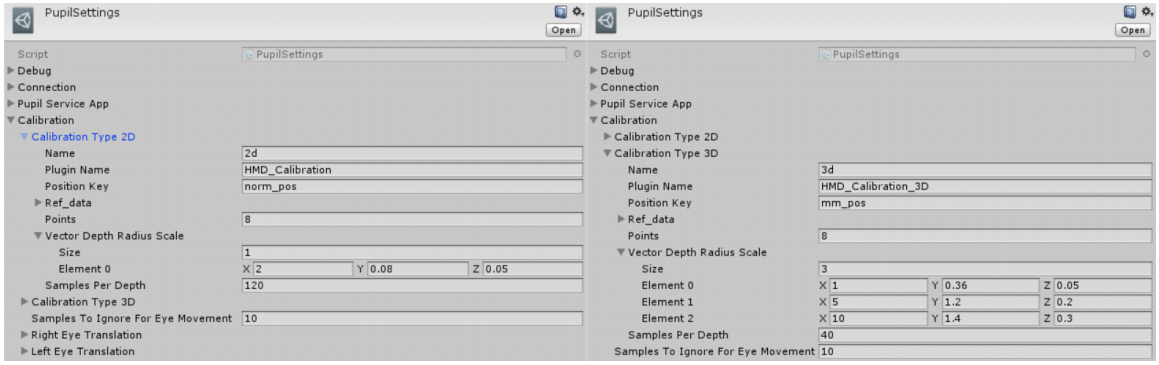


Also included in the image above are screenshots of the "HoloLens Relay" and the "Blink Detector" plugin. The former allowing you to adapt the port for HoloLens communication, the latter to change blink detection variables.

Running a Calibration



Once a connection has been established, you are presented with an option to start the calibration. The process shows a marker in varying positions for the user to follow. The standard pattern is a circle on which the marker is placed (2D case displayed on the left). Based on the headset you are using (our setup is optimized for HTC Vive), this pattern might need to be adapted. To do so, go to "pupil_plugin/PupilSettings" and select "Calibration"



"Points" defines the number of points per circle, the first being in the center of it.

"Vector Depth Radius Scale" can be adapted to define the dimension of the circle

- 2D calibration: We place the markers based on viewport coordinates, starting at the center position (0.5,0.5). Depth translates to the distance from the camera at which marker is placed. Radius defines the distance from the center point (in viewport space). Scale determines the size of the marker.
- 3D calibration: We place the markers based on XYZ coordinates in local camera space. As we need to calibrate for depth, as well, we have multiple distances at which the circle pattern is applied. In the standard case at a distance of '1', '5' and 10'. Radius defines the XY distance from the current center point (0,0,Depth). Scale determines the size of the marker at that depth.

"Samples Per Depth" defines the amount of calibration points recorder for each marker position. We reduced this number for 3D calibration to a third of the samples of 2D calibration so it takes the same duration of time.

"Samples To Ignore For Eye Movements" defines how much time the user is expected to need to move from one calibration marker position to the next. During this period, the gaze positions will not be recorded.

During calibration, the marker color will indicate whether there is a problem with the Pupil confidence level. While white corresponds to perfect levels, red indicates the opposite with confidence for both eyes being close to zero. If only eye ID 0 values are bad, the marker color shifts towards pink and if eye ID 1 values are close to zero, the color shifts to yellow. Every color in between points to something not being optimal, but the closer to white the marker is, the better the confidence levels are.

Once the calibration is finished and all reference points are sent to Pupil, the software will respond with either a "success" or "failure" message.

- In case of success, the PupilDemoManager with activate the gameobjects for the actual scene and display the respective gaze visualizations
- In case of a failure, please use Pupil Capture to see if the confidence levels are good for both eyes when the user is wearing the headset. Unity accepts gaze data with a confidence level greater than 0.6, but values above 0.9 should be targeted. Also have a look at the console, as it might give you a hint at what is going wrong.

Pupil Demo Manager

Excluding the Blink Unity scene, every other demo contains the PupilDemoManager gameobject. Its corresponding script is listening for Pupil events such as the connection being established/quit and the calibration being started, stopped or failing. The Demo Manager is also available as Unity prefab (to be found in "pupil_plugin/Prefabs"), so that it can easily be added to existing scenes. It brings its own camera and a simple GUI to guide the user through the connection and calibration process. A successful calibration results in activating the actual scene objects, so for example the 3D market scene. If you use the prefab for your own scene, please specify the gameobject to activate by adding them to the "Game Objects To Enable" list, exposed in the Unity Inspector.

Accessing Data

After a successful calibration, you need to call PupilTools.Subscribe(string topic) to receive messages for the 'topic' you specify. Most of the demos included in this project are based on subscribing to "gaze" and its implementation shall serve as an example on how to do it for other topics. Message interpretation is handled inside the code block starting in line 127 of Connection.cs. Pupil messages for the "gaze" topic contain dictionaries, which are deserialized using the MessagePackSerializer classes (line 167) and stored to PupilTools.gazeDictionary. PupilTools.UpdateGaze() goes through the data, storing relevant information (e.g. the gaze positions) through PupilData.AddGazeToEyeData(string key, float[] position). Based on the chosen calibration type, this can either be 2D or 3D. To access the data, use

- PupilData._2D.GetEyeGaze (PupilData.GazeSource s), which will provide the current viewport coordinates in camera space (used e.g. for the three colored markers)
- PupilData._2D.GetEyePosition (Camera sceneCamera, GazeSource gazeSource), which will apply an additional frustum center offset for each eye (used e.g. for the shader implementations)
- PupilData._3D.GazePosition, which contains the camera relative gaze position

As the 3D calibration is currently still under active development, it is recommended to rely on 2D calibration, for now. Unity provides methods to convert 2D viewport coordinates to 3D, which is e.g. used for marker positioning

- PupilMarker.UpdatePosition(Vector2 newPosition), line 56

  gameObject.transform.position = camera.ViewportToWorldPoint(position);

An alternative is used by the laster pointer implementation in the "2D Calibration Demo" scene

- MarketWith2DCalibration.Update(), line 63

  Ray ray = sceneCamera.ViewportPointToRay(viewportPoint);

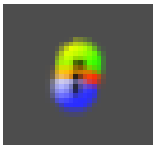
This solutions requires the use of Unity Colliders, though, which, when hit by the above defined ray, return the 3D hit point position.

Demo Scenes

Here a short description of the included examples and what should be visible on screen, once the calibration (if needed) is finished.

Calibration

This scene will display three colored markers representing the left (green) and right (blue) eye plus the center point (red). If everything calibrated as intended, these markers should be very close to each other.



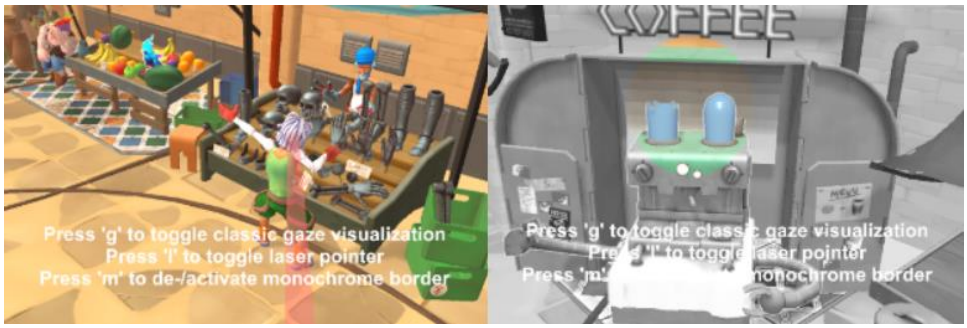
Blink

A barebones implementation for users who do not need gaze data or want a simple example on how to subscribe to a topic and read data from the socket. As suggested by its name, this demo utilizes the Blink_Detection Pupil plugin. It does not require to run through the calibration process. While dictionary setup is usually kept within PupilTools, BlinkDemoManager contains all blink-specific variants to give a better overview of what is involved. This includes starting/stopping the plugin, un-/subscribing to "blinks" and receiving the dictionary packages from Pupil.

2D/3D Calibration Demo (VR)

These scenes will display a 3D market scene, based on Unity assets available for free on the Asset Store. While the 3D calibration scene only includes a simple white marker visualization, the 2D calibration pendant includes the three colored variant used in the Calibration demo scene and two additional visualizations

- A laser pointer going to the center of your gaze (depicted on the left)
- A shader-based implementation that grays out the area around each of the eyes position (right)



2D/3D Calibration Demo (HoloLens)

As it is not a common use-case for HoloLens to visualize complete scenes, we reduced the market scene to a single object (the 'sharkman') for the user to look at. The laser pointer visualization has also been removed.

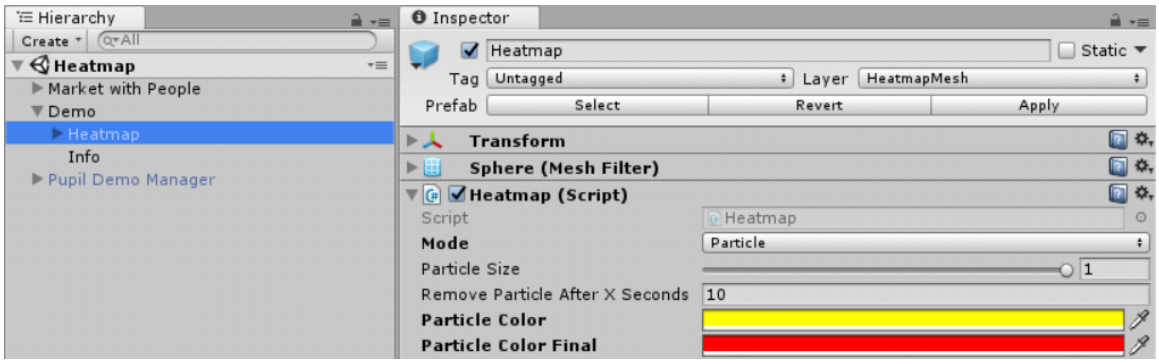
Spherical Video demo

As many users asked for this feature, we now include a demo that can load and display a 360 degree video. Combined with Pupil, this allows to visualize what the user is looking at. There are two ways to load a spherical video, select the `VideoSphere` gameobject to toggle between them

1. Using Unity's internal player, for which the video has to be part of the project files. The included example is based on the market demo scene
2. Using ffmpeg to load external videos and access videoframes at runtime. Depending on the machine it is running on and if it is running in Unity Editor or standalone, this can be slow

Heatmap demo

The heatmap demo allows to export gaze visualization to a spherical video or image. Eye tracking positions are translated to particles on a spherical texture, which is overlayed on the 3d scene. The heatmap is available as Prefab, as well, and can be added to existing scenes by dragging it onto the main camera of that scene. After calibration, press `h` to start recording the output video or to capture the current view to an image. The output path is the same as the one defined by the settings for PupilGazeTracker recordings. A few variables can change how the heatmap behaves



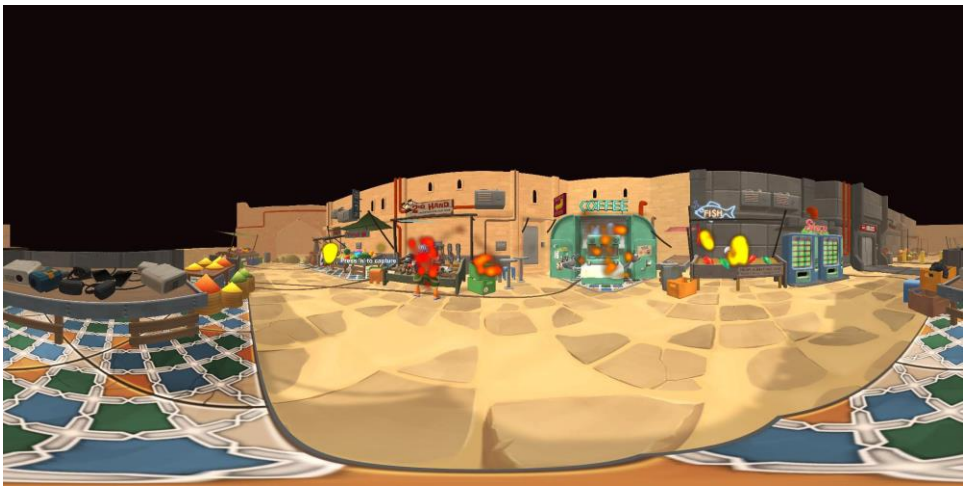
- Mode

  'Particle' will color the area the user is looking at

  'ParticleDebug' will show it for the user, as well

  'Highlight' will only fill-in the area looked at

'Image' will keep all particles and color code them based on the time of creation



- `Particle Size` - The size of a single gaze visualizing particle
- `Remove Particle After X Seconds` - Set how many seconds a particle should be visualized (not used for Image mode)
- `Particle Color` - The color a particle should have.
- `Particle Color Final` - Color for oldest particle in Image mode. Every color in between will be interpolated