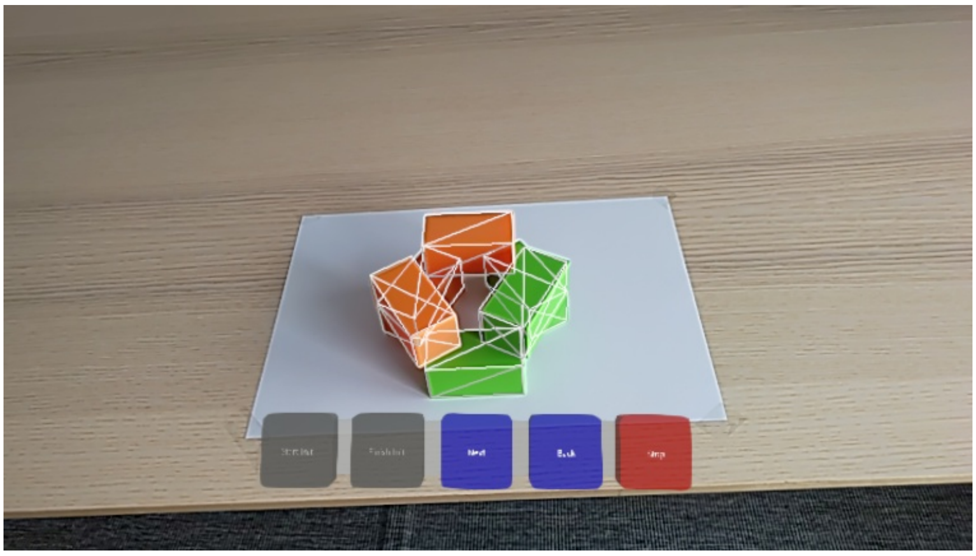
Object Tracking on the HoloLens



Sven Kellenberger

Master Thesis
April 2021

Supervisors:
Dr. Roi Poranne
Dr. Timothy Sandy
Prof. Dr. Stelian Coros

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CRL

Abstract

Using computer vision algorithms to track objects in variable three-dimensional environments and using the found poses to guide complex construction tasks is an interesting area of research which opens up a lot of exciting possibilities. If the construction is performed by a human using a mixed reality headset promises great flexibility an easy of use.

In this thesis I integrate a preexisting object tracking library into the HoloLens 2 ecosystem and build a small application with a simple user interface. This application enables the user to use object tracking to accurately build a simple brick tower by hand. To accomplish this, I make use of various Windows APIs and the new Research Mode for HoloLens 2 to provide the Object Tracker with the data streams it needs.

Zusammenfassung

Das Tracking von Objekten in variablen drei-dimensionalen Umgebungen zur Unterstützung von Bauvorhaben von komplexen Konstruktionen ist ein aktives Forschungsgebiet, welches viele spannende Möglichkeiten bietet. Für Fälle, wo die Konstruktion von einem Menschen per Hand ausgeführt wird, ist es für die höchstmögliche Flexibilität vielversprechend, ein Mixed-Reality Headset einzusetzen.

In dieser Arbeit integriere ich eine bereits existierende Objekt-Tracking Library in das Ökosystem der HoloLens 2 und baue eine kleine Applikation mit einem einfachen User Interface. Diese Applikation kann genutzt werden, um einen kleinen Turm aus Bauklötzen mit grosser Präzision per Hand zu konstruieren. Um dieses Zeil zu erreichen verwende ich verschiedene Windows APIs und den neuen Research Mode für die HoloLens 2, um den Objekt-Tracker mit den nötigen Datenströmen zu versorgen.

Contents

List of Figures

List of Tables

1 Introduction

In our world today we rely more and more on algorithms to help us achieve our goals. We use auto-correction to avoid typos in our text messages, we let Netflix suggest the best matching movies dependent on our tastes to us and we rely on robots to run fully automated assembly lines to produce everyday products. However, all these examples have in common that they run in well defined ecosystems where most external parameters can be controlled and the remaining ones can be accounted for. As soon as we try to tackle problems where it is not possible to control most external factors, we need more robust algorithms which are naturally harder to develop.

One such problem is the construction of complex structures in unknown environments such as building sites. In [Sandy and Buchli 2018] the authors propose a framework which enables real-time tracking of a sensor-head relative to multiple objects and use it to help constructing complex brick structures by hand. In this thesis, I will build on this work and incorporate the existing code into a mixed reality application which can be run on a mixed reality headset. This allows for easy use without the need of holding a camera and other sensors.

1.1. Mixed Reality

The field of computer vision concerns itself with algorithms analysing images and videos to extract as much information as possible about the depicted three-dimensional environment. Within computer vision, mixed reality is a very active area of research. While virtual reality creates a virtual three-dimensional space in which the user can move around and interact with objects, mixed reality incorporates virtual elements into the real world and lets virtual and real objects interact with each other and the user. Its applications range from video games to semi-virtual meetings to supporting surgeons by visualizing CT scans.

Specifically developed headsets are used to provide the most immersive mixed reality experience. One such headset is the HoloLens which was first launched in 2016 by Microsoft and its successor the HoloLens 2, launched in 2019. Equipped with state of the art sensors to allow for spatial orientation as well as fully articulated hand tracking and eye gaze tracking, the HoloLens 2 is able to run complex programs, helping the user perform various tasks [HlH].

In this thesis I make use of the low level sensor streams of the HoloLens 2 available via Resarch Mode [Ungureanu et al. 2020] to provide the aforementioned code with the sensor data streams it requires. I also build a small application which visualizes the state of the Object Tracker and enables the user to interact with it.

2

Related Work

2.1. HoloLens 2

The HoloLens 2 (see fig. 2.1) is a mixed reality headset developed by Microsoft and launched in 2019. It's the successor to the original HoloLens which was launched in 2016. Among the improvements are a larger field-of-view, fully articulated hand tracking, eye gaze tracking and a custom deep neural network core [Ungureanu et al. 2020]. The HoloLens 2 runs Windows 10 and developers can create mixed reality apps by placing holograms in 3D space. See-through holographic lenses allow for an immersive mixed reality experience [HlH].

Figure 2.1.: *Side view of the HoloLens 2. Directly taken from [HlO].*

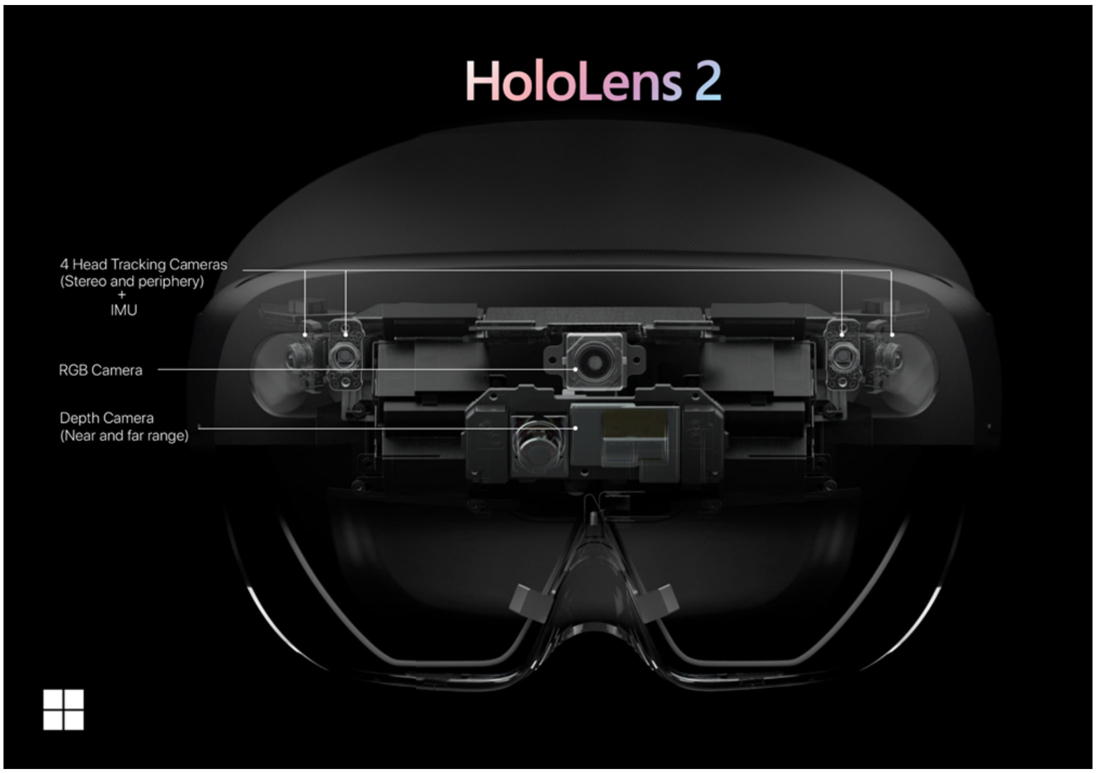


Figure 2.2.: *Image depicting the HoloLens 2 and some of its sensors, directly taken from Microsoft Docs [HlH].*

2.1.1. Hardware

The HoloLens 2 is designed to enable the creation of immersive mixed reality applications. It therefore is equipped with a lot of different sensors [Ungureanu et al. 2020, HlH], some of them are:

- RGB camera: Runs at up to 30fps with a resolution of up to 1080p. Able to take 8-MP photos.
- Depth camera: Is able to operate in two modes: Long Throw, which is a low-framerate ($\leq$ 5fps), far-depth mode and AHAT (Articulated HAnd Tracking) which is a high-framerate (45fps), near-depth mode.
- Four visible-light tracking (VLC) cameras: Placed at the left and right side of the headset and used for head tracking. Have a very wide field-of-view and produce greyscale images.
- Inertial Measurement Unit (IMU): Consists of an accelerometer, a gyroscope and a magnetometer. The accelerometer measures the linear acceleration, the gyroscope the rotation and the magnetometer the absolute orientation of the headset.

The placement of these sensors can be seen in fig. 2.2.

The headset uses its various sensors to provide articulated hand tracking, eye gaze tracking, head tracking as well as spatial mapping out of the box. These tasks are performed by a special holographic processing unit with a deep neural network core which runs all native computer vision algorithms of the device. An additional Qualcomm Snapdragon 850 CPU is provided to run third-party applications.

2.1.2. Research Mode

The Windows Mixed Reality APIs [MRD b] provide access to a lot of functionality of the HoloLens 2 such as the capturing of camera frames and tracking of the the current head pose. However, these APIs don't allow for the developer to access the raw data streams of most sensors but only to the results of the processing of said streams for example the current head pose which is calculated using ineratial measurements. To address this, Microsoft introduced the Research Mode for HoloLens 2 [Ungureanu et al. 2020] with the goal of encouraging contributions in the field of mixed reality by industrial and academic researchers. Research Mode needs to be specifically enabled in the Device Portal of the HoloLens 2 [Dev] and is not meant for applications targeted at end-users since they won't be able to run it. When enabled, Research Mode provides a set of C++ APIs to access various sensor streams of the HoloLens 2. Sensors exposed by Research Mode are the following:

- VLC camera sensor: Provides access to the greyscale images produced by these cameras. Each camera can be targeted individually.
- Depth camera sensor: Provides access to the computed depth data for both modes described in sec. 2.1.1 individually while also providing the raw infrared stream used to compute the depth map for both modes.
- IMU sensors: Provides access to measurement samples for each indiviual sensor. The API provides batches of measurement samples at a frequency between 12Hz and 22Hz.

Camera sensors and IMU sensors expose different functions. For example IMU sensors expose functions to either retrieve a single measurement sample per frame or a batch of measurement samples while camera sensors expose functions to map points from the three-dimensional camera space into the two-dimensional image space.

The API also provides extrinsic transformation matrices for each sensor relative to a static rigid node on the headset which represents the device origin and corresponds to the left front VLC camera. Each sensor returns a 4x4 transformation matrix to the this rigid node. To locate the sensors relative to other coordinate systems Research Mode allows for the retrieval of the GUID of the rigid node which can then be located relative to different coordinate systems using the HoloLens Perception API [Per].

The authors of the Research Mode paper also provided a Github repository [Hl2] which contains the full documentation for Research Mode as well as different example programs showcasing the use of Research Mode and code samples which can be used to develop with Research Mode.

2.1.3. Development

There are multiple options to develop mixed reality applications for HoloLens 2 [MRD a]. The most convenient ways are using one of the game engines Unity [Uni] or Unreal [Unr]. Unity is especially well supported and runs the Mixed Reality Toolkit (MRTK) [MRT] which is an open-source, cross-platform development kit for developing mixed reality applications. However there are multiple problems, when developing with Unity.

For one, Unity runs on C# code while the used preexisting code for object tracking (see sec. 2.2) is written in C++ . While it is possible to run DLLs (dynamic link libraries) compiled from C++ in Unity, the HoloLens 2 needs those DLLs to be compiled for UWP (Universal Windows Platform) and the ARM64 CPU architecture. Unity can run neither of those things in its built-in editor. Therefore we would need to compile those DLLs for different configurations, however the MRTK also does not support Research Mode at the current moment in time and Unity therefore is unable to create a correct build which we can then run on the HoloLens 2 without any modifications necessary. We can also not run any Research Mode code in the Unity editor. Couple this with long build times and the advantages of Unity for easily developing three-dimensional applications with nice user interfaces are outweighed by its disadvantages when working with C++ code and Research Mode.

Because of this, I decided to use Native OpenXR development for this application. It requires to handle the positioning and drawing of three-dimensional objects by oneself through the access of low level Mixed Reality Windows APIs but it is able to do this in pure C++ code. To make life a little easier I used the Cannon library for mixed reality development provided in [Hl2] which uses DirectX and provides helper functions for perception and rendering.

2.2. Object Tracker

Robots have been used on assembly lines for a long time to great effect. They are usually cheaper, faster and more precise than human workers and don't need any breaks. However, these systems normally work in very constrained environments and cannot be used in more challenging scenarios like outside construction sites.

The first challenge for such tasks is the tracking of objects in various environments. Early works used image gradients to detect edges and match them to objects [Lowe 1991] and were then improved to allow for real-time object tracking [Drummond and Cipolla 2002]. There is also work which improves tracking performance further, for example by online-learning object specific detectors [Kalal et al. 2009] or by incorporating color features [Petit et al. 2014].

Visual-inertial object tracking denotes object tracking with the addition of inertial measurements to improve tracking in cases where the sensor-head can be moving fast. For example in [Klein and Drummond 2002] the authors use inertial data to tune the parameters of feature detectors while also using camera images to calibrate the IMU to prevent drift (accumulated error over time).

This thesis uses the work of [Sandy and Buchli 2018] which provides a system to track a sensor-head relative to multiple objects in real-time. It uses a probabilistic moving horizon estimator to

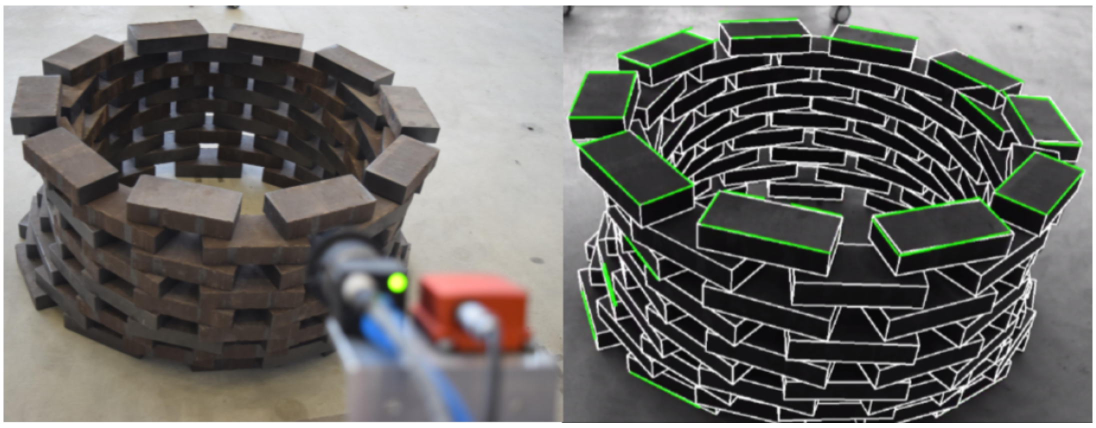


Figure 2.3.: *A brick tower built with the help of a visual-inertial tracking system. The left image shows the brick tower and the sensor-head while the right image shows the camera image overlaid with the rendered geometry. Directly taken from [Sandy and Buchli 2018].*

fuse IMU measurements and camera frames to generate accurate pose estimates. The presented framework also provides functionalities for scene rendering and automatic tracking recovery and was showcased by accurately constructing a complex brick structure by hand as can be seen in fig. 2.3.

This work has since been further developed and currently consists of platform-independent **C++** code and an application for Android phones. For the rest of this thesis we will refer to this code simply as the Object Tracker. The sensor fusion library used has since been open-sourced [Sandy et al. 2019] and the code further uses the ceres-solver [Agarwal et al.], Eigen [Guennebaud et al. 2010], OpenCV [Bradski 2000] and Boost [Boo] libraries.

3 Results

In thesis I create a port for the HoloLens 2 using the aforementioned Object Tracker. I use Research Mode [Ungureanu et al. 2020] to provide the Object Tracker with the sensor data it needs and create a working application around it which runs on the HoloLens 2. In this chapter I will take a closer look at the indiviual components of my work.

3.1. Sensor API

At the heart of the application lies the Sensor API. It unifies several sensors from different APIs to provide raw data streams. The `SensorApi` class serves as the interface between the Sensor API and the application that retrieves its data. It initializes the individual sensor interfaces with their respective APIs and manages them.

Each sensor is represented in the code by a sensor interface. Each sensor interface is running a separate thread in which it continuously loads new data. The `SensorApi` class then provides functions for extracting this data by the application processing it.

In the case of the IMU sensors, `SensorApi` only exposes a single function which returns the merged IMU measurements instead of independent measurements for each individual sensor. The measurements from the IMU sensors usually don't arrive at the same time, let alone at the same frame rate. The `SensorApi` therefore stores the measurements it receives in separate buffers and returns only complete IMU measurements (consisting of both accelerometer and gyroscope measurements) where data from both sensors has arrived.

Furthermore the `SensorApi` class exposes functions to retrieve extrinsic transformation matrices for all sensors. With these it is possible to create transformation matrices for transformations between any pair of sensors. As mentioned in sec. 2.1.2, all sensors are located relative to a rigid node which is determined by Research Mode. For example `GetT_gyroscope_rigid()` would return a 4x4 transformation matrix which can be used to transform a vector from the coordinate system of the rigid node into the coordinate system of the gyroscope. For readers unfamiliar with this naming convention, I refer to the appendix sec. A.3.

The whole Sensor API only uses Mixed Reality Windows APIs and Research mode. The conversion to Eigen, OpenCV and similar libraries used by the Object Tracker is done outside of the Sensor API. It is therefore possible to use this API as a plug and play module which can be integrated into projects with similar use cases.

A UML diagram of all the important classes, properties and methods can be found in fig. 3.1. I have implemented interfaces for four different kinds of sensors which I describe briefly in the following subsections.

3.1.1. Camera Interface

The `CameraInterface` class uses only Mixed Reality Windows APIs to retrive frames of the RGB camera of the HoloLens 2. It retrieves these frames at 30fps with a resolution of 960x540 but the resolution can be configured to be up to 1920x1080. It uses the GUID of the Research Mode rigid node provided by `SensorApi` to locate the RGB camera relative to this rigid node. This transformation matrix has the form T_{RC}.

The returned `CameraFrame` struct looks as follows:

```
struct CameraFrame {
  uint64_t ts;
  SoftwareBitmap bitmap;
  float2 principalPoint;
  float2 focalLength;
  SpatialCoordinateSystem coordinateSystem;
};
```

The `SoftwareBitmap` class defines all properties of the image, for example data, image width, image height and format. For convenience `bitmap` is already converted to BGRA 8-Bit. `coordinateSyste` is later used to calculate a transformation between the world coordinate system of the Object Tracker and the world coordinate system of the HoloLens.

3.1.2. IMU Interfaces

The `AccelInterface` and `GyroInterface` classes represent their respective Research Mode sensors. They share most of their code and therefore inherit it from an abstract super class `ImuInterface`. The sensors themselves work at different frame rates and at each frame the Research Mode API returns a buffer of measurement samples. The interfaces then down sample the incoming measurements to a desired frequency, in my case 200 measurements per second. The measurements can than be retrieved as a vector of `ImuSensorDataStruct` structs:

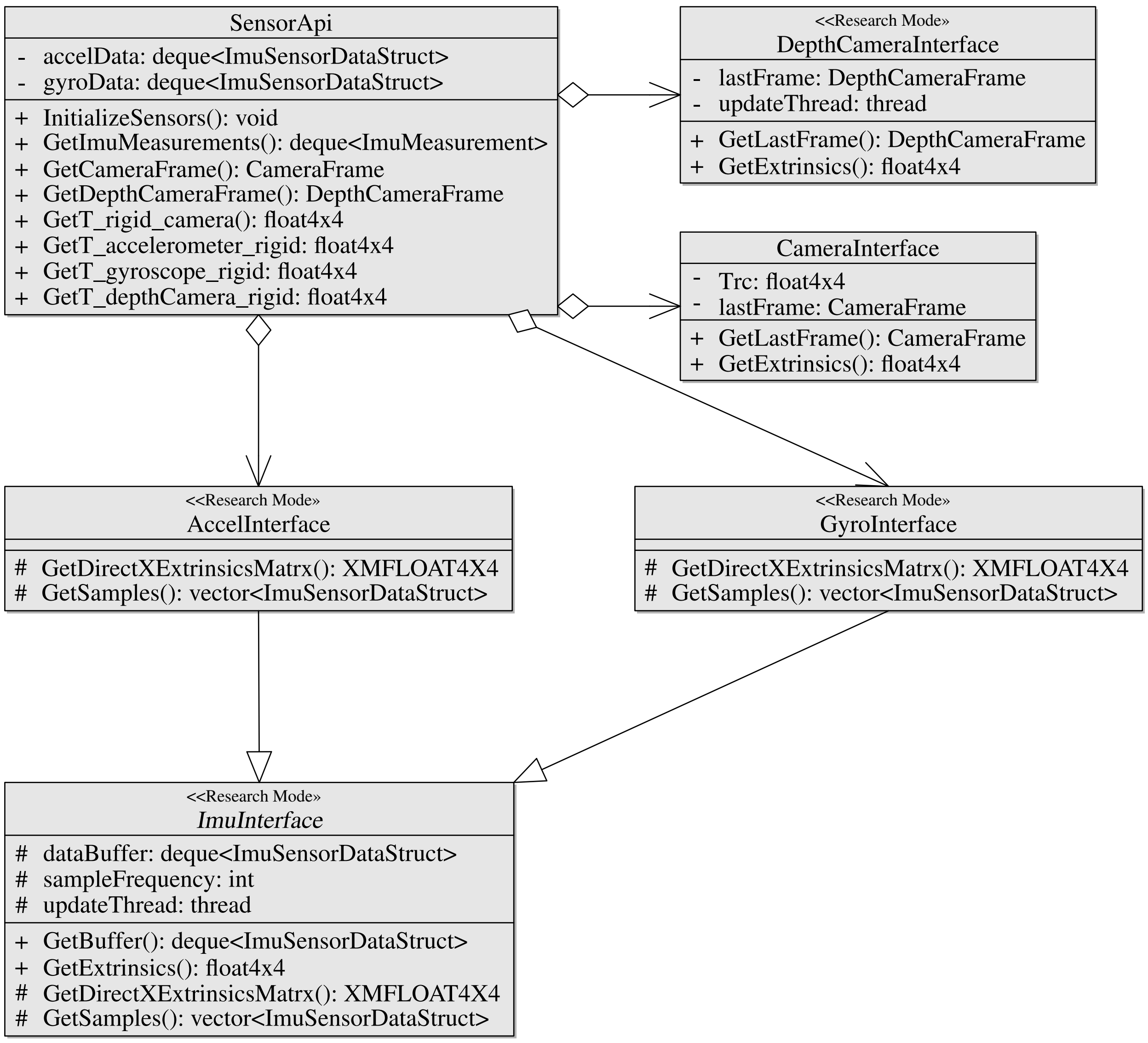

SensorApi
- accelData: deque<ImuSensorDataStruct>
- gyroData: deque<ImuSensorDataStruct>
+ InitializeSensors(): void
+ GetImuMeasurements(): deque<ImuMeasurement>
+ GetCameraFrame(): CameraFrame
+ GetDepthCameraFrame(): DepthCameraFrame
+ GetT_rigid_camera(): float4x4
+ GetT_accelerometer_rigid: float4x4
+ GetT_gyroscope_rigid: float4x4
+ GetT_depthCamera_rigid: float4x4
<<Research Mode»
DepthCameraInterface
- lastFrame: DepthCameraFrame
- updateThread: thread
+ GetLastFrame(): DepthCameraFrame
+ GetExtrinsics(): float4x4
CameraInterface
- Trc: float4x4
- lastFrame: CameraFrame
+ GetLastFrame(): CameraFrame
+ GetExtrinsics(): float4x4
<<Research Mode»
AccelInterface
# GetDirectXExtrinsicsMatrx(): XMFLOAT4X4
# GetSamples(): vector<ImuSensorDataStruct>
<<Research Mode»
GyroInterface
# GetDirectXExtrinsicsMatrx(): XMFLOAT4X4
# GetSamples(): vector<ImuSensorDataStruct>
<<Research Mode»
ImuInterface
# dataBuffer: deque<ImuSensorDataStruct>
# sampleFrequency: int
# updateThread: thread
+ GetBuffer(): deque<ImuSensorDataStruct>
+ GetExtrinsics(): float4x4
# GetDirectXExtrinsicsMatrx(): XMFLOAT4X4
# GetSamples(): vector<ImuSensorDataStruct>

Figure 3.1.: *UML Diagram of the Sensor API: The `SensorApi` class manages all sensor interfaces and exposes functions to retrieve data and extrinsic transformation matrices from them. The `CameraInterface` class collects its data from Mixed Reality Windows APIs, all other sensor interfaces retrieve their data via Research Mode. `ImuInterface` is an abstract class implementing all of the shared code for the sensor interfaces of the different IMU sensors.*

```
struct ImuSensorDataStruct {
  uint64_t ts;
  float ImuValues[3];
};
```

These measurements are then merged by `SensorApi` into `ImuMeasurement` structs of the form:

```
struct ImuMeasurement {
  uint64_t ts;
  float AccelValues[3];
  float GyroValues[3];
};
```

Extrinsic transformation matrices are retrieved directly from research mode and take the form T_{AR} and T_{GR} respectively.

While I didn't implement a sensor interface for the magnetometer since it is not needed for the Object Tracker, it should be trivial to do so since its Research Mode sensor shares most of its behaviour with the other IMU sensors.

3.1.3. Depth Camera Interface

As mentiond in sec. 2.1.1, `DepthCameraInterface` has two possible modes of operation, AHAT or Long Throw. Which one to use is defined by `SensorApi`. Independent of the mode, the interface can be queried for the `DepthCameraFrame` struct:

```
struct DepthCameraFrame {
  uint64_t ts;
  uint32_t width;
  uint32_t height;
  std::vector<uint16_t> depthData;
  std::vector<uint16_t> activeBrightnessImage;
};
```

Here `depthData` is a vector of size `width * height` containing the depth values for each pixel. Invalid pixels contain a value of 0. `activeBrightnessImage` is a vector of the same size and contains the original infrared image used to calculate the depth map.

At the moment, this sensor is not used by the Object Tracker algorithm and therefore disabled in `SensorApi`. However it is ready to be used at either a later point in time or by a different application.

3.2. Object Tracker Integration

I now use the Sensor API to retrieve the raw sensor data and feed it into the Object Tracker. The main interface for the Object Tracker is the `ObjectTracker` class which exposes various functions to interact with the library. To connect the `ObjectTracker` and `SensorApi` classes I have created the `ViotApplication` class. This class first initiates an instance of `SensorApi` and then waits for all sensors to be initialized and ready. Once that is the case, it initializes the `ObjectTracker` by loading a configuration file. This file allows for the specification of various configurations such as brick size, which building plan to use and if a log file should be created.

Next the $T_{CameraIMU}$ transformation matrix needs to be set so the `ObjectTracker` can transform IMU measurements to the camera space and vice-versa. Since the object tracker doesn't distinguish between different IMU sensors but treats them as a single sensor, we choose to define the IMU position as the position of the accelerometer. To calculate $T_{CameraIMU}$, we retrieve $T_{RigCamera}$ and $T_{AccelRig}$ from the Sensor API and set:

$$T_{CameraIMU} := T_{RigCamera}^{-1} * T_{AccelRig}^{-1}$$

The camera calibration also needs to be set in the `ObjectTracker`. These parameters can be read directly from the current `CameraFrame` provided by `SensorApi`.

`ViotApplication` runs a separate thread for the RGB camera, the depth camera and the IMU sensor each. Again, the depth camera thread is disabled because it is not needed at the current time. Whenever new data arrives, it is retrieved from the `SensorApi` instance, transformed into the format `ObjectTracker` expects, and fed into it. For example are IMU measurements converted to Eigen vectors and images are converted to OpenCV matrices. This process is shown in fig. 3.2.

Since the position of the accelerometer is treated as the IMU position, gyroscope measurements need to be transformed into the accelerometer space. For this the rotation matrices $R_{AccelRig}$ and $R_{GyroRig}$ are extracted from $T_{AccelRig}$ and $T_{GyroRig}$ respectively which are retrieved from the `SensorApi`. All gyroscope measurements are then rotated by:

$$R_{AccelGyro} = R_{AccelRig} * R_{GyroRig}^{-1}$$

It is not necessary to perform use the full transformation matrix (i.e. include the translation) because gyroscope measurements are translation invariant.

Because IMU measurements can lag behind camera frames by up to 0.3 seconds, camera frames are stored in a buffer and only fed into the `ObjectTracker` when the corresponding IMU measurements have arrived. This way `ObjectTracker` doesn't wait idly for IMU measurements to arrive but optimizes the last available state.

After feeding camera frames into the `ObjectTracker`, `ViotApplication` retrieves the latest estimation for $T_{WorldIMU}$ by the `ObjectTracker`. The pose of the camera at the same time is then used to calculate the estimated transformation matrix from the HoloLens world to the Object Tracker world. This is necessary to be able to transform objects from the world coordinate system of the Object Tracker in which they are tracked into the world coordinate system of the HoloLens in which they are drawn.

The user interface thread of the application calls the `Update()` and `Draw()` functions of `ViotApplication` once every iteration. During `Update()` the poses of all objects are

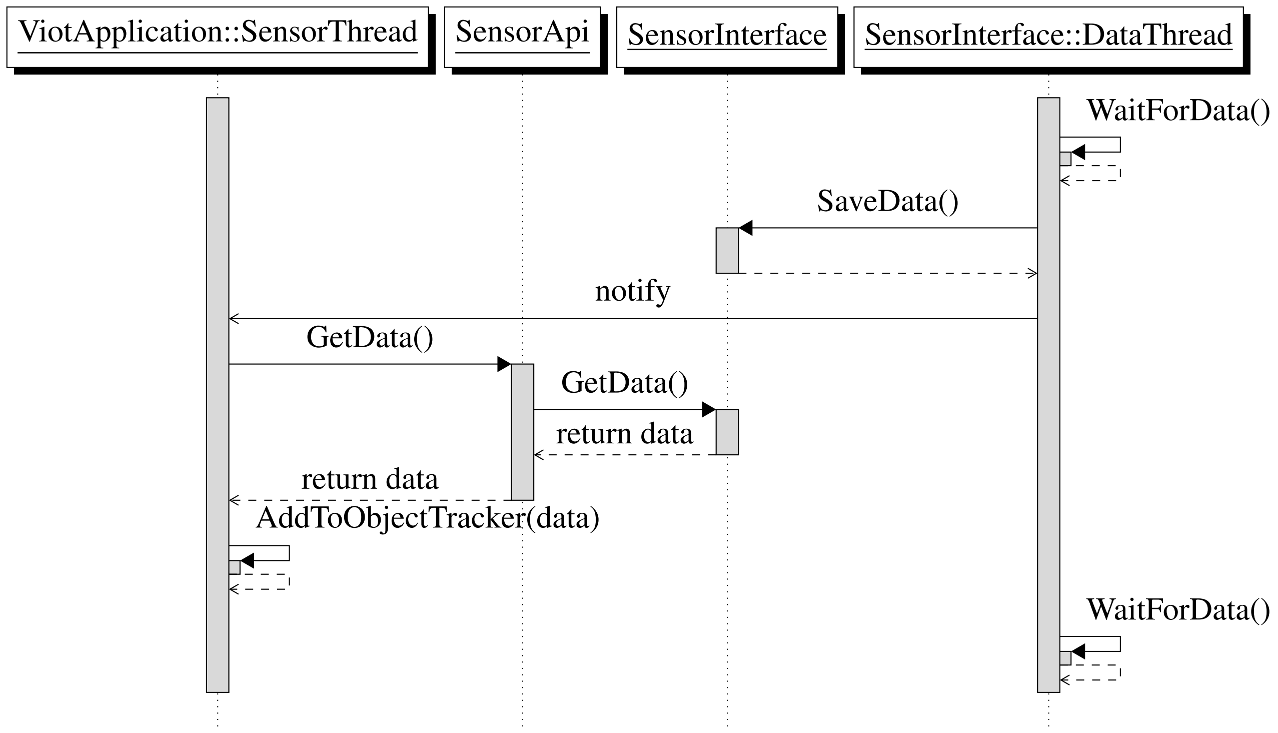


Figure 3.2.: *Abstracted sequence diagram of data thread interactions: Each sensor interface runs a thread to continuously fetch data from its sensor which is then saved in the sensor interface instance. After that, the respective thread of the* `ViotApplication` *instance is notified which then retrieves the data from the sensor interface via the* `SensorApi` *and feeds it into the* `ObjectTracker`*.* `ViotApplication` *runs one thread for each sensor type.*

	Start Init	**Done Init**	**Next**	**Back**	**Stop**
HOLD	✓	X	X	X	X
INIT	X	✓	X	X	✓
TRACKING	X	X	✓	✓	✓

Table 3.1.: *The state of each button of the user interface depending on the current state of the Object Tracker. A checkmark means that the button is enabled in this state, while an X means it is disabled.*

updated with the estimates provided by the `ObjectTracker` while `Draw()` draws them into the holographic space.

An UML diagram of the this part of the application can be seen in fig. 3.3.

3.2.1. User Interface

The user interface consists of five buttons which are either enabled or disabled depending on the current state of the Object Tracker. The Object Tracker can be in one of three states:

- HOLD: The Object Tracker is idle and waits for tracking to start.
- INIT: The Object Tracker is in the process of being initialized. In this state the user can place the first brick and start tracking when he or she is finished.
- TRACKING: The Object Tracker is in the process of tracking.

The five buttons are:

- Start Init: This button starts the pose initialization of the Object Tracker. This means, the first brick will be drawn and move with the users head.
- Done Init: Once the brick moving with the users head is correctly positioned, this button may be pressed to lock it in place. This starts the tracking of the brick.
- Next: This button goes one step forward. This means that if all active objects have been placed, a new pending object is added and if a pending object is ready to be tracked, it changes this object's state from pending to active.
- Back: This button does the reverse of the Next button.
- Stop: This button stops the tracking.

The state of each button depending on the state of the Object Tracker can be found in tab. 3.1.

The buttons are placed in a row on the lower part of the users viewport. A screenshot can be seen in fig. 3.4. The state transitions in the object tracker triggered by the buttons can be seen in fig. 3.5.

When a brick is not positioned correctly the Object Tracker provides poses for arrows which

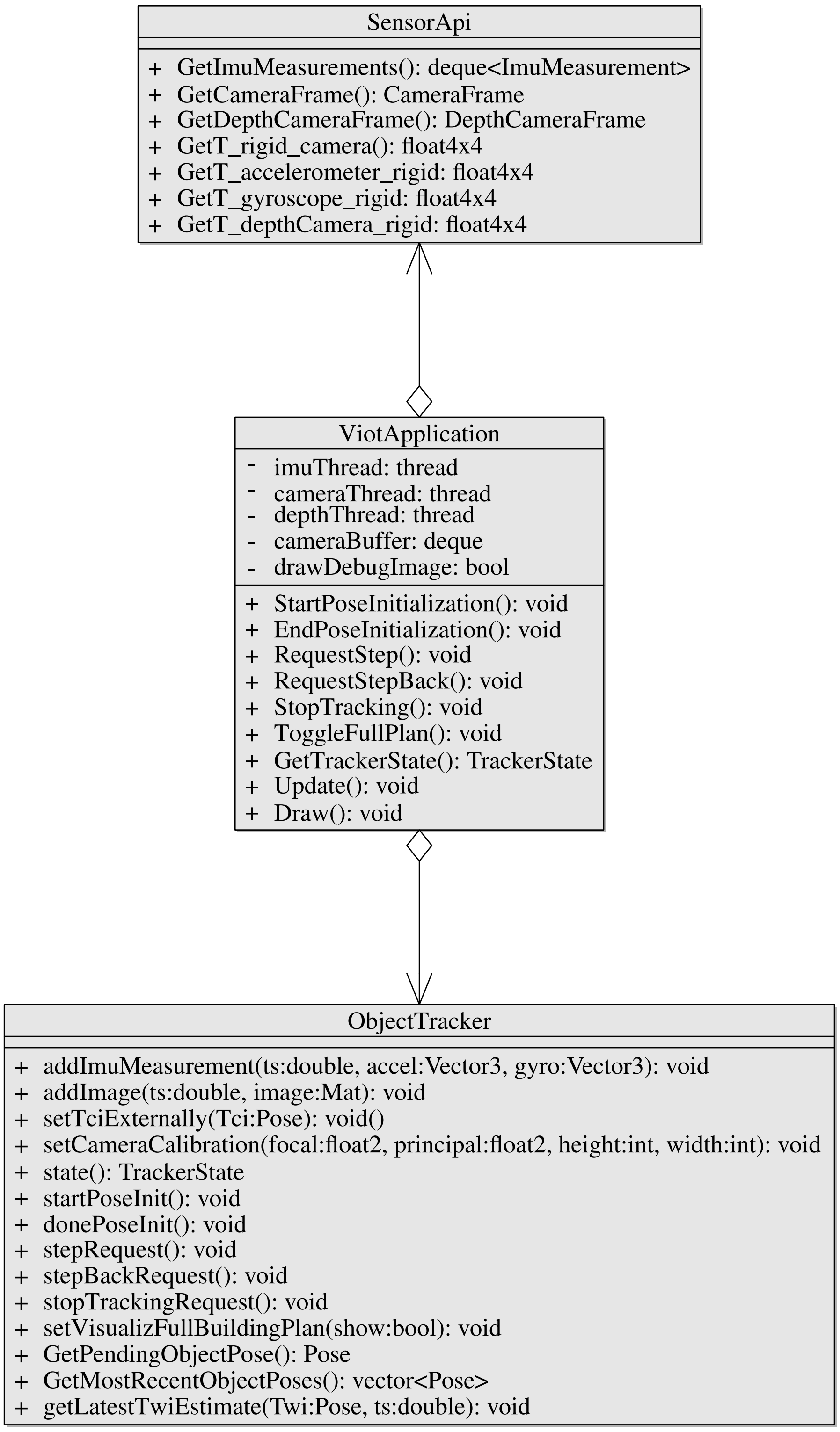

SensorApi
+ GetImuMeasurements(): deque<ImuMeasurement>
+ GetCameraFrame(): CameraFrame
+ GetDepthCameraFrame(): DepthCameraFrame
+ GetT_rigid_camera(): float4x4
+ GetT_accelerometer_rigid: float4x4
+ GetT_gyroscope_rigid: float4x4
+ GetT_depthCamera_rigid: float4x4
ViotApplication
- imuThread: thread
- cameraThread: thread
- depthThread: thread
- cameraBuffer: deque
- drawDebugImage: bool
+ StartPoseInitialization(): void
+ EndPoseInitialization(): void
+ RequestStep(): void
+ RequestStepBack(): void
+ StopTracking(): void
+ ToggleFullPlan(): void
+ GetTrackerState(): TrackerState
+ Update(): void
+ Draw(): void
ObjectTracker
+ addImuMeasurement(ts:double, accel:Vector3, gyro:Vector3): void
+ addImage(ts:double, image:Mat): void
+ setTciExternally(Tci:Pose): void()
+ setCameraCalibration(focal:float2, principal:float2, height:int, width:int): void
+ state(): TrackerState
+ startPoseInit(): void
+ donePoseInit(): void
+ stepRequest(): void
+ stepBackRequest(): void
+ stopTrackingRequest(): void
+ setVisualizFullBuildingPlan(show:bool): void
+ GetPendingObjectPose(): Pose
+ GetMostRecentObjectPoses(): vector<Pose>
+ getLatestTwiEstimate(Twi:Pose, ts:double): void

Figure 3.3.: *UML Diagram of the Object Tracker integration: The `ViotApplication` class retrieves data from the `SensorApi`, transforms it into the correct format and feeds it into the `ObjectTracker`. It is also responsible for updating and drawing tracked objects retrieved from `ObjectTracker`.*

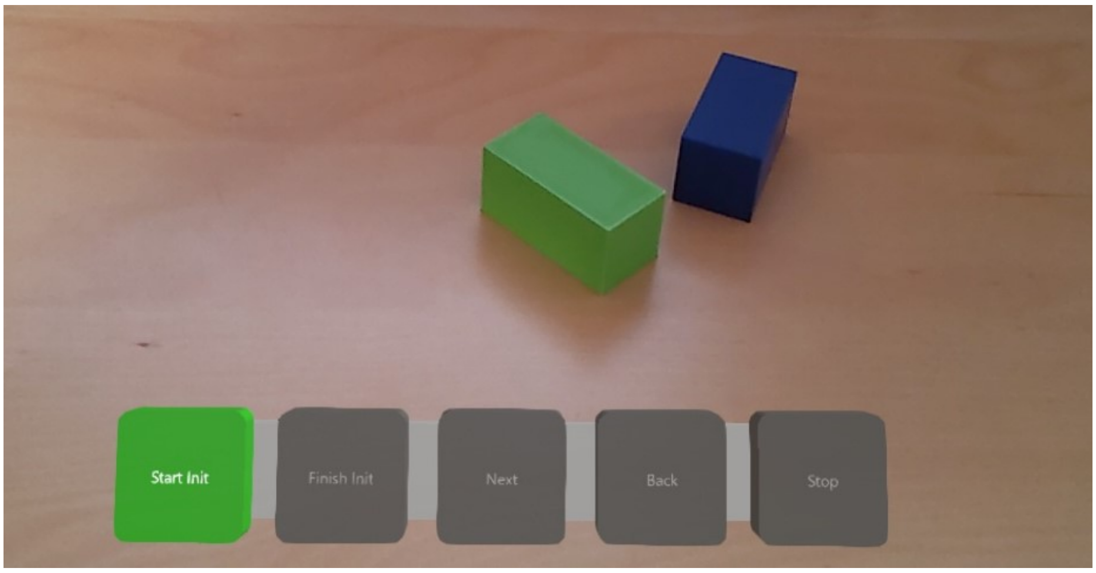


Figure 3.4.: *Screenshot of the user interface: The buttons are lined up on the bottom half of the viewport and enabled or disabled depending on the state of the object tracker. In this case, only the first button is enabled.*

$$\begin{tikzcd}
\text{HOLD} \arrow[r, "\text{StartInit}"] & \text{INIT} \arrow[r, "\text{DoneInit}"] \arrow[l, bend left, "\text{Stop}"] & \text{TRACKING} \arrow[ll, bend right, "\text{Stop}"'] \arrow[loop below, "\text{Next}\\\text{Back}"]
\end{tikzcd}$$

Figure 3.5.: *State transition diagram for the object tracker UI: Boxes represent the possible states the object tracker can be in while transitions represent button presses in the user interface.*

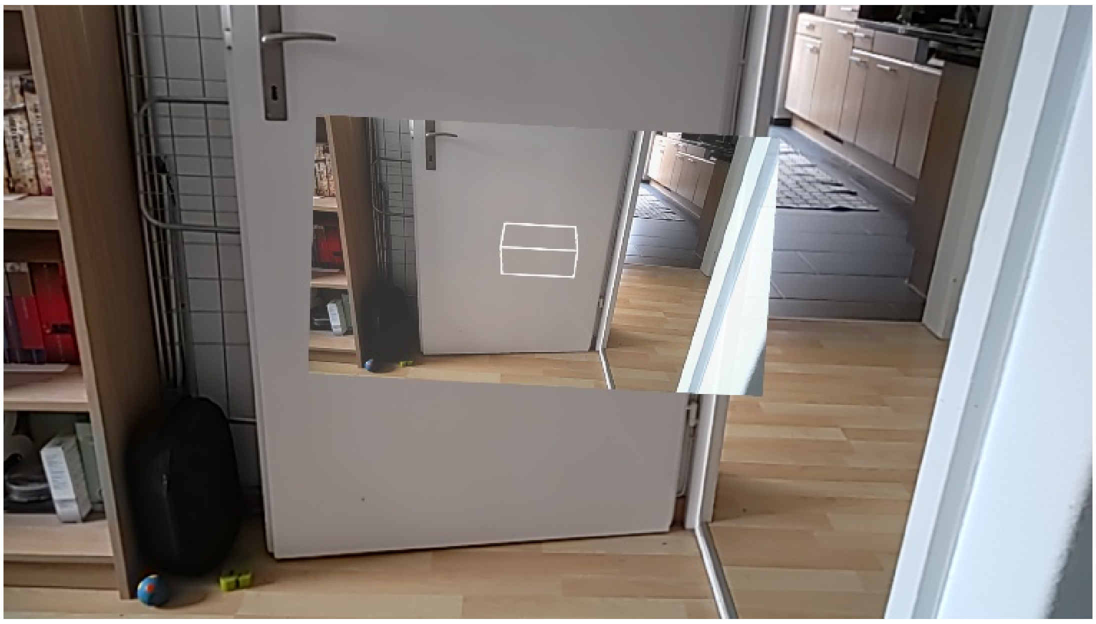


Figure 3.6.: This image shows the debug screen which can be enabled in `ViotApplication` *debugging and easier development*

are then drawn at the corners of the brick. This helps the user to correct the position.

3.2.2. Debug Screen

To help with development and debugging, I also provide a debug screen which can be enabled in `ViotApplication`. It draws a virtual screen which moves with the user's head and can display any image required. It is especially useful for showing what the Object Tracker would render but it can be configured to show any other image stream.

3.2.3. Showcase

Fig. 3.7 shows a brick tower built with application presented in this thesis. A video of the complete build process is linked in appendix sec. A.1.

3.3. Limitations

As already mentioned in sec. 3.2 the IMU measurements provided by Research Mode generally lag up to 0.3 seconds behind the camera frame. Because of this, camera frames have to be held back until the corresponding IMU measurements have arrived. This, of course leads to noticeable delays in the information retrieved from the Object Tracker. Furthermore, while it is possible to use large numbers of IMU measurements per second (I use 200 for this application but larger numbers are possible), these measurements arrive in large batches but at a low frame

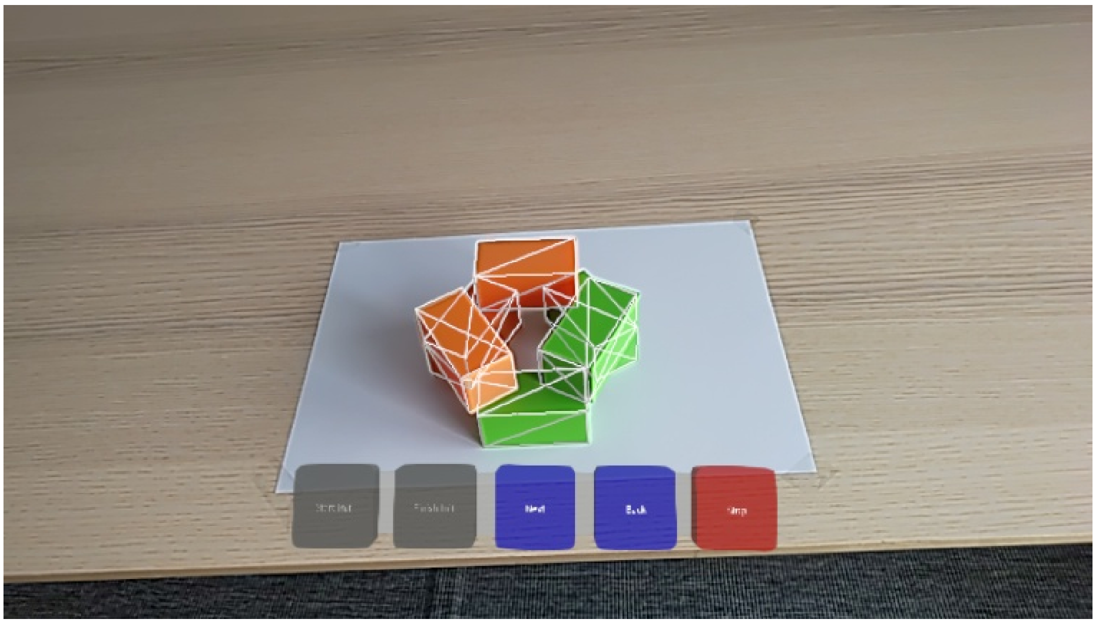


Figure 3.7.: *Brick tower built with the Object Tracker on the HoloLens 2. The image shows a screenshot of the HoloLens 2.*

rate of between 12Hz to 22Hz which leads to the input of a lot of measurements simultaneously. It would be preferred to provide these measurements in a more continuous stream.

The most noticeable limitation imposed by the delayed IMU measurements occurs during the initialisation of the first object pose. It is decidedly more challenging because the pose which should move with the head of the user has a noticeable delay. Once the first pose is initialized it gets better but from time to time one can still notice delayed readjustments of the brick poses to head movement.

Another limitation is the computing power of the HoloLens 2 in general. Because of this, the Object Tracker often needs longer to optimize than it would on a modern phone. This can lead to even more delayed adjustments, general instability or even loss of tracking.

4 Conclusion and Outlook

In this thesis I presented an integration of the Object Tracker for the HoloLens 2 ecosystem. I presented the Sensor API, an interface to get data streams from various HoloLens 2 sensors via sensor interfaces which use standard Windows APIs and Research Mode. Because I don't use any other libraries than standard Windows APIs and Research Mode in the Sensor API, it can easily be used for different projects with the similar requirements.

I then used these raw data streams to incorporate the Object Tracker into a mixed reality application which provides a simple user interface to interact with the Object Tracker. I also used the Cannon library to display bricks tracked by the Object Tracker in the holographic space which allows for the user to accurately build a simple brick tower by hand.

The current application can be improved further in a couple of different ways. For one, the Object Tracker could make use of the depth camera of the HoloLens 2. The Sensor API already provides an interface to retrieve this data however it is not yet used by the Object Tracker.

Another point of improvement is Research Mode itself. While it grants access to low-level sensor streams which are otherwise not available, the API is not yet optimized for applications working in real-time. For one, IMU measurements retrieved via Research Mode always lag a couple of tenths of a second behind real-time. This introduces lag into the application which makes it more difficult to perform real-time taks. While it is possible to retrieve a lot of measurement samples per second for every IMU sensor, these measurement samples arrive in batches at low frame rates which reduces some of the advantages of having many samples per second available. If Microsoft decides to improve Research Mode in the future, such changes would have a large impact on real-time object tracking.

Finally, the performance of the Object Tracker on the HoloLens 2 is not as good as on other platforms because of limited computational power. Some of the optimization steps take a lot of time which can result in delayed adjustments and general instability. This could be improved by looking at ways to optimize the Object Tracker further, especially in regard of the ecosytem of the HoloLens 2. It is also to be expected that future generations of mixed reality headsets will have more computational power which will of course diminish this effect.

A

Appendix

A.1. Showcase Video

I recorded a short video showcasing my work. I uploaded the video to YouTube and it can be found at `https://youtu.be/wArK9bmHatM`.

A.2. Source Code

The code written for this thesis is included within the Object Tracker code repository. It includes a section within the Readme with instructions on how to set it up and how to build and run it.

Since the code of the Object Tracker is proprietary, its Github repository is private and the code will not be released to the public. However the Sensor API part of the code (see sec. 3.1) is designed to be compatible with any HoloLens 2 application using Research Mode and can be provided to interested parties.

A.3. Transformation Matrix Naming Convention

The transformation matrices in this thesis follow the simple convention that T_{AB} is a transformation matrix which transforms a vector from the coordinate system B into the coordinate system A, i.e.:

$$x_A = T_{AB} * x_B$$

where x_B is a vector in coordinate system B and x_A is the corresponding vector in coordinate system A.

Similarly R_{AB} denotes a rotation matrix from coordinate system B into the coordinate system A.

Bibliography

AGARWAL, S., MIERLE, K., AND OTHERS. Ceres solver. `http://ceres-solver.org`.

Boost C++ libraries. `http://www.boost.org/`. [Online; accessed April 2021].

BRADSKI, G. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Hololens device portal. `https://docs.microsoft.com/en-us/windows/mixed-reality/develop/platform-capabilities-and-apis/using-the-windows-device-portal`. [Online; accessed April 2021].

DRUMMOND, T., AND CIPOLLA, R. 2002. Real-time visual tracking of complex structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 7, 932–946.

GUENNEBAUD, G., JACOB, B., ET AL., 2010. Eigen v3. http://eigen.tuxfamily.org.

Microsoft HoloLens2ForCV Github repository. `https://github.com/microsoft/HoloLens2ForCV`. [Online; accessed April 2021].

HoloLens 2 hardware. `https://docs.microsoft.com/en-us/hololens/hololens2-hardware`. [Online; accessed April 2021].

HoloLens 2 overview. `https://www.microsoft.com/en-us/hololens/hardware`. [Online; accessed April 2021].

KALAL, Z., MATAS, J., AND MIKOLAJCZYK, K. 2009. Online learning of robust object detectors during unstable tracking. In *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, 1417–1424.

KLEIN, G., AND DRUMMOND, T. 2002. Tightly integrated sensor fusion for robust visual tracking. In *In Proc. British Machine Vision Conference (BMVC'02*, 787–796.

LOWE, D. G. 1991. Fitting parameterized three-dimensional models to images. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*, 5, 441–450.

Introduction to mixed reality development. `https://docs.microsoft.com/en-gb/windows/mixed-reality/develop/development`. [Online; accessed April 2021].

Mixed reality documentation. `https://docs.microsoft.com/en-us/windows/mixed-reality/`. [Online; accessed April 2021].

Mixed Reality Toolkit for Unity. `https://docs.microsoft.com/en-gb/windows/mixed-reality/develop/unity/mrtk-getting-started`. [Online; accessed April 2021].

HoloLens perception API. `https://docs.microsoft.com/en-us/uwp/api/windows.perception.spatial?view=winrt-19041`. [Online; accessed April 2021].

PETIT, A., MARCHAND, E., AND KANANI, K. 2014. Combining complementary edge, keypoint and color features in model-based tracking for highly dynamic scenes. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 4115–4120.

SANDY, T., AND BUCHLI, J. 2018. Object-based visual-inertial tracking for additive fabrication. *IEEE Robotics and Automation Letters 3*, 3, 1370–1377.

SANDY, T., STADELMANN, L., KERSCHER, S., AND BUCHLI, J. 2019. Confusion: Sensor fusion for complex robotic systems using nonlinear optimization. *IEEE Robotics and Automation Letters 4*, 2, 1093–1100.

UNGUREANU, D., BOGO, F., GALLIANI, S., SAMA, P., DUAN, X., MEEKHOF, C., STÜHMER, J., CASHMAN, T. J., TEKIN, B., SCHÖNBERGER, J. L., OLSZTA, P., AND POLLEFEYS, M., 2020. Hololens 2 research mode as a tool for computer vision research.

Unity real-time development platform. `https://unity.com/`. [Online; accessed April 2021].

Unreal engine. `https://www.unrealengine.com/`. [Online; accessed April 2021].

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Object Tracking for the HoloLens

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Kellenberger	Sven

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Wil, 14.4.2021	Kellenberger

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.