massless

Massless Pen API

Developers Guide

We provide an API for developers to include input from the Massless Pen in their own applications. This takes the form of a C-API in a native DLL built for 64bit Windows. We also include the corresponding header (.h) and library (.lib) files to conveniently link against this API.

This guide will provide information on how to get started using the API, and what exactly it will output.

Structure of the API

The API is simple, containing only the following functions:

```
int PenStart(uint8_t IntegrationKey[16]);
int PenStop();
int PenSetUnits(float units);
int PenAttachListener(PoseUpdate updateClient);
int PenAttachStateListener(InputStateUpdate updateClient);
int PenAttachFullPoseListener(FullPoseUpdate updateClient);
int PenAttachFullStateListener(FullStateUpdate updateClient);
int PenAttachNotificationListener(NotificationCallback updateClient);
int PenGetTrackerPose(int typeInt, float* x, float* y, float* z,
float* qr, float* qx, float* qy, float* qz);
int PenSimpleBuzz(uint16_t DurationMs = 208);
int32_t PenGetSelectedTrackerID(uint8_t pointingType = 0); (reserved
unused)
int PenSetOriginTracker(int32_t ID); (reserved unused)
uint64_t PenGetDllVersionNumber();
```

The pose listener that you attach must be a function of the following type:

```
typedef int(__stdcall *PoseUpdate)(uint32_t length, uint8_t*
PenPose);
```

This will be the callback function that is called each time the API has a new pose for the pen. The function details are given at the end of this document.

The input state listener that you attach must be a function of the following type:

```
typedef int(__stdcall *InputStateUpdate)(uint32_t length, uint8_t*
State);
```

This will be the callback function that is called each time the API has a new set of inputs for the pen. The function details are given at the end of this document.

The full pose listener that you attach must be a function of the following type:

```
typedef int(__stdcall *FullPoseUpdate)(uint32_t length, uint8_t*
Data);
```

This will be the callback function that is called each time the API has a new pose for the pen. It is designed to be both backward and forward compatible. The function details are given at the end of this document.

The full state listener that you attach must be a function of the following type:

```
typedef int(__stdcall *FullStateUpdate)(uint32_t length, uint8_t*
Data);
```

This will be the callback function that is called each time the API has a new set of inputs for the pen. It is designed to be both backward and forward compatible. The function details are given at the end of this document.

The notification callback that you attach must be a function of the following type:

```
typedef int(__stdcall *NotificationCallback)(uint32_t length,
uint8_t* Data);
```

This will be the callback function that is called each time the API has a notification to inform the client about. The function details are given at the end of this document.

Basic Usage

To use the API, it is recommended to first call `PenAttachNotificationListener(function)` with whichever function you have defined to interpret the notifications. This will enable you to receive any error notifications that may occur during `PenStart`.

To start tracking you would need to call `PenStart(integrationKey)` with your assigned integration key, which would start the camera and tracking algorithms running, and create the data structures to hold all the settings. If you want the API to return the pose with length units other than the default mm, call the `PenSetUnits(unitLength)` function.

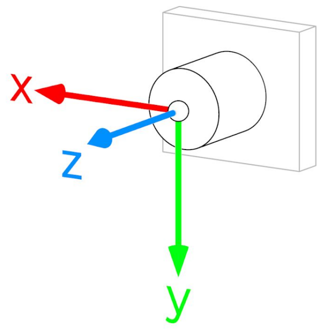
You need to get the tracker's pose relative to the attached headset system's tracker. This is found by calling `PenGetTrackerPose( typeInt, &x, &y, &z, &qr, &qx, &qy, &qz)` specifying the correct typeInt that corresponds to the type of headset system you are running, this provides you with the pose filled into the variables.

Then you would register the callback by calling `PenAttachFullPoseListener(function)` with whichever function you have defined to be your full pose callback. If you want information about the pen's capacitive sensor and surface sensor, you would register the callback `PenAttachFullStateListener(function)` with whichever function you have defined to be your full state callback.

`PenSimpleBuzz(duration)` can be called at any time to vibrate the pen's haptic feedback motor for the specified duration in ms.

When you have finished you would call `PenStop()` for the program to release its resources and close. The `PenStop()` also deregisters your notification callback in preparation for closing down the whole API. However, if you want to start it up again without unloading the DLL it is fine to just call `PenAttachNotificationListener(function)` again and then `PenStart(integrationKey)` to start the tracking up again.

Coordinate System



Massless Trackers

The Massless API follows the computer vision convention for coordinate systems, having the right-handed coordinate system defined by the camera.

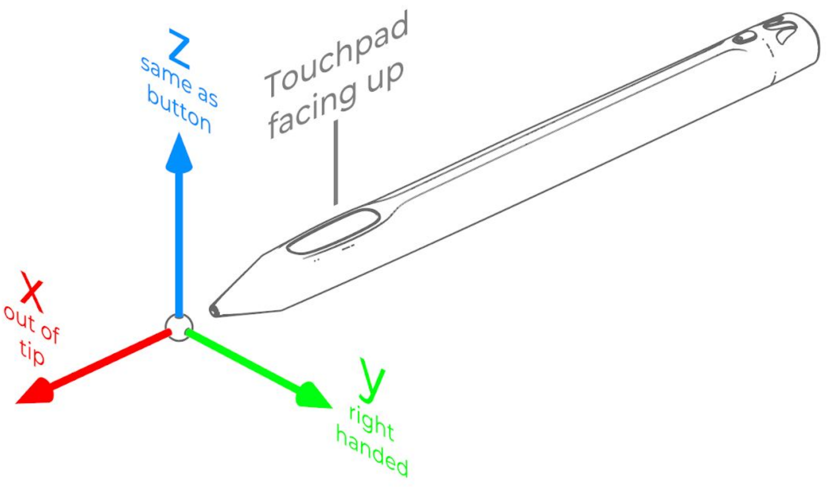
The figure on the left shows the convention that we follow for the cameras' coordinate systems.

Origin

The origin of the Massless coordinate system is that of our tracker. Typically this tracker must be attached to one of the headset system's trackers. We then provide the transformation between the coordinate system of our tracker and that of an Oculus Tracker, or a Vive Lighthouse that is attached to our tracker via one of the brackets we supply. You must decide which headset system you are using and call the function to get the tracker's pose offset from this system. Currently, we only support the Oculus Rift and the HTC Vive systems (including Vive Pro).

Massless Pen

The Massless Pen has a coordinate system itself. Its pose is represented as a transformation between this coordinate system and the Massless Tracker's coordinate system. The origin of the Massless Pen's coordinate system is the tip, with the x-axis extending along the pen's central axis, and the z-axis opposite the power button. The y-axis is then fully defined by those two and the fact that it is a right-handed system.



Transformations

We provide the “pose” as a transformation, comprised of a displacement vector and a quaternion. Both of these are actually object-transformations rather than straight descriptions of position and orientation. This is illustrated with an example.

Take a model of the Massless Pen including a marker on it. Apply the pen pose transformation according to the following equation, to find the position of the marker in the Massless Tracker’s coordinate system:

$$\underline{x}_t = P_{tp}\underline{x}_p$$

where $\underline{x}_p$ is the position of the marker in the Massless Pen’s coordinate system and the pose transformation returned by the API is

$$P_{tp}\underline{x} = R_{tp}\underline{x} + T_{tp}$$

the rotation is actually provided as a quaternion, so the rotation operator here is not a matrix but represents the application of the unit quaternion’s rotation to the vector.

Use of the Coordinate systems in Unity

Unity uses a left-handed coordinate system, we take Y as the axis being flipped, which brings our cameras into line with how their camera coordinate systems function.

On bringing our pose into Unity, we recommend the following transformations.

$y_{unity} = -y_{Massless}$

then divide the whole thing by 1000 to go from mm to m, for the translation part of the pose. The quaternion needs the following,

$qx_{unity} = -qx_{Massless}$ and $qz_{unity} = -qx_{Massless}$

This will then be in proper Unity units, so we can just set the resulting quaternion and position to the transform of a game object.

Full Function Documentation for the API

Callback Versioning Format

The newer callbacks are designed to be backward and forward compatible. The first byte is a version number, in future versions we will only ever append data onto the end of the message. Therefore, you can use it by checking the version is above a certain value before reading the parts of the data. If you're using an older version of the API then you will need to generate default values for the data that isn't supplied, or produce an error. If you are using a newer version of the API that you can handle, then you will just ignore the newer data that you don't know about.

PenStart

Starts the camera, loads the calibration files, loads the model of the pen, finds and opens the serial port and sets up the tracking. Needs an integration key to be passed in, contact Massless support if you do not have one yet.

Arguments

- `uint8_t integrationKey[16]` // Integration key that you have been provided.

Returns

- `int status` // Error code, 0 for success.

PenStop

Stops the camera, closes the serial port, unloads everything, and releases the resources.

Returns

- `int status` // Error code, 0 for success.

PenSetUnits

Sets the units of length for reporting to the units that have the provided length in SI units (metres). For example, if you want lengths reported in inches, provide the unitLength parameter as 0.0254, for millimetres, use 0.001.

Arguments

- `float unitLength` // Length of the required unit in metres.

Returns

- `int status` // Error code, 0 for success.

PenAttachListener

Registers the callback to be used to inform of an updated pose for the Massless Pen.

PoseUpdate functions are described below.

Arguments

- `PoseUpdate updateClient` // Pointer to the callback function.

Returns

- **`int`** `status` // Error code, 0 for success.

PoseUpdate

Function to be defined in the program using the Massless Pen API. This function will be provided with the pose of the massless pen in the format of an `int32` for the length, and a pointer to the **`uint8`** start of the message array. The message will be 3 floats describing the position, followed by 4 describing the orientation quaternion.

0. **`float`** `x` // X coordinate of position.
1. **`float`** `y` // Y coordinate of position.
2. **`float`** `z` // Z coordinate of position.
3. **`float`** `q_w` // W element of orientation quaternion.
4. **`float`** `q_x` // X element of orientation quaternion.
5. **`float`** `q_y` // Y element of orientation quaternion.
6. **`float`** `q_z` // Z element of orientation quaternion.

It expects an integer return, for the possibility of communicating an error state. However, this is currently ignored.

Arguments

- **`uint32_t length`** // Number of bytes in the message data (normally 28 = 4*7).
- **`uint8_t*`** **`pPoseData`** // Pointer to the pose data

Returns

- **`int`** `status` // Error code, 0 for success. Currently ignored.

PenAttachStateListener

Registers the callback to be used to inform of an updated input state for the Massless Pen.

InputStateUpdate functions are described below.

Arguments

- `InputStateUpdate updateClient` // Pointer to the callback function.

Returns

- **`int`** `status` // Error code, 0 for success.

InputStateUpdate

Function to be defined in the program using the Massless Pen API. This function will be provided with the input state of the massless pen in the format of an **`int32`** for the length, and a pointer to the **`uint8`** start of the message array. The message will be 1 float describing the surface proximity, followed by 1 **`uint8`** describing the capacitive sensor state. The surface proximity value will always be between 0 (free space) and 1 (touching a surface). The capacitive sensor has a value between 0 (touched near the tip) to 254 (touched near the back) as a linear slider. When the capacitive sensor is not touched, it has the value 255.

`Byte 0 float` `surface` // Proximity between 0 and 1 of a surface.

`Byte 4 uint8_t` `capSense` // Value for the capacitive touch sensor.

It expects an integer return, for the possibility of communicating an error state. However, this is currently ignored.

Arguments

- **`uint32_t`** `length` // Number of bytes in the message data (normally 5 = 4 + 1).
- **`uint8_t*`** `pInputStateData` // Pointer to the input state data

Returns

- **`int`** `status` // Error code, 0 for success. Currently ignored.

PenAttachFullPoseListener

Registers the callback to be used to inform of an updated pose for the Massless Pen.

FullPoseUpdate functions are described below.

Arguments

- `FullPoseUpdate updateClient` // Pointer to the callback function.

Returns

- **`int`** `status` // Error code, 0 for success.

FullPoseUpdate

Function to be defined in the program using the Massless Pen API. This is a backwards and forwards compatible callback. This function will be provided with the pose of the massless pen in the format of an **`int32`** for the length, and a pointer to the **`uint8`** start of the message array. The data will have the following format.

- **`Uint8_t`** `version` // version of this callback.
- **`float`** `x` // X coordinate of position.
- **`float`** `y` // Y coordinate of position.
- **`float`** `z` // Z coordinate of position.
- **`float`** `q_w` // W element of orientation quaternion.
- **`float`** `q_x` // X element of orientation quaternion.
- **`float`** `q_y` // Y element of orientation quaternion.
- **`float`** `q_z` // Z element of orientation quaternion.
- **`uint8_t`** `StatusFlags` // Byte containing bitfield status flags.

It expects an integer return, for the possibility of communicating an error state. However, this is currently ignored.

Arguments

- **`uint32_t`** `length` // Number of bytes in the message data (normally 28 = 4*7).
- **`uint8_t*`** `pPoseData` // Pointer to the pose data

Returns

- **`int`** `status` // Error code, 0 for success. Currently ignored.

PenAttachStateListener

Registers the callback to be used to inform of an updated input state for the Massless Pen.

InputStateUpdate functions are described below.

Arguments

- `InputStateUpdate updateClient` // Pointer to the callback function.

Returns

- `int status` // Error code, 0 for success.

InputStateUpdate

Function to be defined in the program using the Massless Pen API. This function will be provided with the input state of the massless pen in the format of an `int32` for the length, and a pointer to the `uint8` start of the message array. The message will be 1 float describing the surface proximity, followed by 1 `uint8` describing the capacitive sensor state. The surface proximity value will always be between 0 (free space) and 1 (touching a surface). The capacitive sensor has a value between 0 (touched near the tip) to 254 (touched near the back) as a linear slider. When the capacitive sensor is not touched, it has the value 255.

`Byte 0 float surface` // Proximity between 0 and 1 of a surface.

`Byte 4 uint8_t capSense` // Value for the capacitive touch sensor.

It expects an integer return, for the possibility of communicating an error state. However, this is currently ignored.

Arguments

- `uint32_t length` // Number of bytes in the message data (normally 5 = 4 + 1).
- `uint8_t* pInputStateData` // Pointer to the input state data

Returns

- `int status` // Error code, 0 for success. Currently ignored.

PenAttachListener

Registers the callback to be used to inform of an updated pose for the Massless Pen.

PoseUpdate functions are described below.

Arguments

- `PoseUpdate updateClient` // Pointer to the callback function.

Returns

- `int status` // Error code, 0 for success.

PoseUpdate

Function to be defined in the program using the Massless Pen API. This function will be provided with the pose of the massless pen in the format of an `int32` for the length, and a pointer to the `uint8` start of the message array. The message will be 3 floats describing the position, followed by 4 describing the orientation quaternion.

- `float x` // X coordinate of position.
- `float y` // Y coordinate of position.
- `float z` // Z coordinate of position.

- `float q_w` // W element of orientation quaternion.
- `float q_x` // X element of orientation quaternion.
- `float q_y` // Y element of orientation quaternion.
- `float q_z` // Z element of orientation quaternion.

It expects an integer return, for the possibility of communicating an error state. However, this is currently ignored.

Arguments

- `uint32_t length` // Number of bytes in the message data (normally 28 = 4*7).
- `uint8_t* pPoseData` // Pointer to the pose data

Returns

- `int status` // Error code, 0 for success. Currently ignored.

PenGetTrackerPose

This will return the pose offset between the tracking reference (Oculus Tracker, or Vive Lighthouse) that you are using, and the Massless coordinate system, defined by the Massless Tracker. It takes the input argument of the type (currently 0 for OculusTracker, and 1 for ViveLighthouse) and then fills the rest of the output arguments with the pose.

`Byte 0 float surface` // Proximity between 0 and 1 of a surface.

`Byte 4 uint8_t capSense` // Value for the capacitive touch sensor.

It expects an integer return, for the possibility of communicating an error state. However, this is currently ignored.

Arguments

- `int typeInt` // Type of tracker we are using.
- `float* x` // Pointer to be filled with x value
- `float* y` // Pointer to be filled with y value
- `float* z` // Pointer to be filled with z value
- `float* qr` // Pointer to be filled with quaternion's real value
- `float* qx` // Pointer to be filled with quaternion's x value
- `float* qy` // Pointer to be filled with quaternion's y value
- `float* qz` // Pointer to be filled with quaternion's z value

Returns

- `int status` // Error code, 0 for success.

PenSimpleBuzz

Activates the Massless Pen's haptic feedback motor for the specified length of time, in integer units of milliseconds. If no duration is provided, the default of 200ms is used.

Arguments

- `uint16_t DurationMs = 200` // Duration of the buzz in milliseconds.

Returns

- `int status` // Error code, 0 for success.

PenGetSelectedTrackerID

Not used yet.

PenSetOriginTracker

Not used yet.