# Tobii Game Integration 8.0.0

*Tobii Game Integration(TGI) library provides several high-level eye- and head-tracking features: Extended View, automatic (re)connection to trackers, and automatic remapping of gaze points to game window coordinates.*

- *Current version supports*
  - *Windows 7 - 64 bit only*
  - *Windows 8.1 - 32/64 bit*
  - *Windows 10 - 32/64 bit*
  - *Windows 11*
- *TGI supports all Tobii consumer eye- and head-tracking technologies*
- *C++ interface*
- *C# bindings*

# Linking

TGI consists of 3 main files:

- tobii_gameintegration_x64.dll
- tobii_gameintegration_x64.lib
- tobii_gameintegration.h

TGI supports both dynamic linking and linking via an import library. A reference implementation is distributed together with the library:

- tobii_gameintegration_dynamic_loader.h
- tobii_gameintegration_dynamic_loader.cpp

# Technical behavior

- TGI does not perform any dynamic memory allocations.
- TGI spawns a thread for communication with a tracker.

# Version check

TGI exposes one function that returns an object of type *ITobiiGameIntegrationApi*. Version check is performed whenever a developer requests an ITobiiGameIntegrationApi object. If the requested version does not match library version, *nullptr* is returned. This mechanism is a safeguard against a TGI header/implementation mismatch.

# Functionality

TGI provides the following components:

- Tracker Controller
- Streams Provider
- Interaction Features
- Filters

# Tracker Controller

The tracker controller handles the connection (and potential disconnects) to a tracker. It allows setting a tracking mode to a game window, arbitrary rectangle on a screen or a head-mounted display. On disconnection, it continuously scans for a reconnection. When set to track a game window, the tracker controller automatically switches to the appropriate tracker (if available). It also remaps gaze coordinates from display space to window space.

## Tracker Info

Some functions of the Tracker Controller allows you to get detailed information about any of the system's detected trackers, a specific detected tracker identified by its url, or the tracker that is currently connected to. These functions populate an array of, or a singular, Tracker Info struct.

The Tracker Info struct holds multiple fields of detailed data about a particular device, but one is of particular interest for integrating support for a wide range of Tobii devices in an application, namely the Capabilities field. The Capabilities is a bit-field of the type StreamFlags, and can be bitwise compared against other named constant enumerators of the StreamFlags.

**In order to differentiate between devices that support eye tracking, head tracking or both,** the Capabilities field of the Tracker Info for the device can be compared against the `StreamFlags::Gaze` and `StreamFlags::Head` enumerators.

For more information on acquiring the Tracker Info data, see the functions of the Tracker Controller. For a complete example see the TrackerInfoSample.cpp file included in the Tobii.GameIntegration.Samples folder.

# Streams Provider

This component is an entry point to the raw data retrieved from the tracker. It provides three types of data streams:

## Eye tracking data

2D coordinates of a gaze point. Default unit type for gaze points is signed normalized. It also provides these points in several other coordinate spaces:
*By **tracking area**, we mean a provided rectangle or a game window.*

- Signed normalized
  - **Origin** — a center of the tracking area.
  - Y-axis is pointing up. The x-axis is pointing right.
  - Values **scale** is [-1, 1] inside of the tracking area. Gaze points can have coordinates bigger than 1 or smaller than -1 outside of the tracking area.
- Normalized
  - **Origin** — the left bottom corner of the tracking area.
  - Y-axis is pointing up. The x-axis is pointing right.
  - Values **scale** is [0, 1] inside of the tracking area. Gaze points can have coordinates bigger than 1 or smaller than 0 outside of the tracking area.
- Millimeters
  - **Origin** — the left bottom corner of the tracking area.
  - Y-axis is pointing up. The x-axis is pointing right.
  - Values **scale** is [0, tracking_area_size_mm] inside of the tracking area. Gaze points can have coordinates bigger than tracking_area_size or smaller than 0 outside of the tracking area. tracking_area_size is a size of the tracking area in millimeters.
- Pixels
  - **Origin** — the left bottom corner of the tracking area.
  - Y-axis is pointing up. The x-axis is pointing right.
  - Values **scale** is [0, tracking_area_size_px] inside of the tracking area. Gaze points can have coordinates bigger than tracking_area_size or smaller than 0 outside of the tracking area. tracking_area_size is a size of the tracking area in pixels.

## Head tracking data

*Rotation* is expressed in degrees using right-handed rotations around each axis.

- Yaw is a clockwise rotation about the down vector, the angle increases when turning your head right.
- Pitch is a clockwise rotation about the right vector, the angle increases when turning your head up.
- Roll is a clockwise rotation about the forward vector, the angle increases when tilting your head to the right.

*Position* is measured in millimeters from the center of the screen.

- X-axis is pointing right, origin at the center of the screen.
- Y-axis is pointing up, origin at the center of the screen.
- Z-axis is pointing outward from the screen, origin at the center of the screen.

## Presence data

Boolean indicating if the tracker has positively detected a user looking at the monitor associated with the tracker.

# Interaction Features

## Extended view

Extended view outputs angles suitable for adding onto a game-camera orientation to achieve a smooth camera orientation modifier based on both eye tracking and head tracking data. This effectively extends the users field of view without conventional camera control input. For example, looking at the right edge of the screen and/or a head rotation towards the right causes a rotation of the camera to the right.

# Filters

Filters are used to change the velocity of gaze point, make it smoother and less responsive. One of the features where usage of filters is highly encouraged is Aim At Gaze, so you are welcome to use AimAtGaze filter from TGI in your implementation of Aim At Gaze.

# Statistics

TGI provides a possibility to track features being turned on or off by the user. This data is then used to improve the implementation of features, based on the usage statistics. Collected data is completely anonymous and doesn't contain any user identifying tokens like steam ID, nicknames or others.

# Reference documentation

## Getting the API

```
ITobiiGameIntegrationApi* GetApi(const char* fullGameName.
                                 bool analyticalUse = false);

ITobiiGameIntegrationApi* GetApi(const char* fullGameName,
                                 const uint16_t* license,
                                 uint32_t licenseSize,
                                 bool analyticalUse = false);

ITobiiGameIntegrationApi* GetApiDynamic(const char* fullGameName,
                                        bool analyticalUse = false);

ITobiiGameIntegrationApi* GetApiDynamic(const char* fullGameName,
                                        const uint16_t* license,
                                        uint32_t licenseSize,
                                        bool analyticalUse = false)

ITobiiGameIntegrationApi* GetApiDynamic(const char* fullGameName,
                                        const char* dllPath,
                                        bool analyticalUse = false);

ITobiiGameIntegrationApi* GetApiDynamic(const char* fullGameName,
                                        const char* dllPath,
                                        const uint16_t* license,
                                        uint32_t licenseSize,
                                        bool analyticalUse = false)
```

**Remarks**
These functions are used to get an object that provides all available TGI functionality. **You should call one of them before anything else.**

- In case of dynamic linking of the dll, you need to make sure that files:
    - tobii_gameintegration_dynamic_loader.h
    - tobii_gameintegration_dynamic_loader.cpp

  are included in your project. Use a method **GetApiDynamic** to get a TGI API.
- In case of linking via an import library, you need to make sure that you've added a tobii_gameintegration_xxx.lib to your build system. Use a method **GetApi** to get a TGI API.

**Return value**
A pointer to an object of type ITobiiGameIntegrationApi that should be used for any interaction with TGI.

**Parameters**
*fullGameName* — a full game name. Specify a full, real game name, not a code name for the game. Example: "Assassin's Creed Origins". fullGameName is a **UTF-8** encoded string.

*dllPath* — a path to a TGI dll. Should be a zero-terminated C string.

*analyticalUse* — set to **true** if you are developing an application that stores or transfers Tobii eye tracking, presence or position data. Then you also need to obtain a special license for Analytical Use. Read more at https://analyticaluse.tobii.com/.

*license* — a buffer containing a license provided by Tobii. A license is needed for some features, for instance to access the unfiltered gaze stream on devices older than Tobii Eye Tracker 5.

*licenseSize* — the size of the *license* buffer.

# ITobiiGameIntegrationApi

To get an object of type *ITobiiGameIntegrationApi*, use Getting the API section.
Use these getter functions to get different API modules:

```
ITrackerController* GetTrackerController();
IStreamsProvider* GetStreamsProvider();
IFeatures* GetFeatures();
IStatistics* GetStatistics();
IFilters* GetFilters();
```

Use these functions to perform the general API control:

```
bool IsInitialized();
void Update();
void Shutdown();
```

---

## IsInitialized

```
bool IsInitialized();
```

Determines whether TGI managed to initialize successfully.

**Remarks**
If this function returns false the results of any API call apart from IsInitialized is undefined.

**Return value**
Returns a boolean that indicates whether TGI managed to initialize. It should return true immediately after you get the API object (see Getting the API section). No additional initialization steps required.

---

## Update

```
void Update();
```

Updates internal TGI state.

**Remarks**

Call update every game tick. It is crucial to call **Update** even when there is no tracker connected, to get a proper behavior interaction features and update of all TGI states.
Update should be called from a single thread.

---

## Shutdown

```
void Shutdown();
```

Disables all TGI functionality. Stops the thread spawned for communication with a Tobii tracker and disposes of internal resources.

**Remarks**
This function is meant to be called once when the game stops. If it is called while the game runs by mistake or by purpose, TGI starts a thread again as soon as any other call to start tracking or **Update** is called.

# ITrackerController

```
bool GetTrackerInfo(TrackerInfo& trackerInfo);
bool GetTrackerInfo(const char* url, TrackerInfo& trackerInfo);
void UpdateTrackerInfos();
bool GetTrackerInfos(const TrackerInfo*& trackerInfos, int& numberOfTrackerInfo
bool TrackHMD();
bool TrackRectangle(const Rectangle& rectangle);
bool TrackWindow(void* windowHandle);
void StopTracking();
bool IsConnected() const;
bool IsEnabled() const;
bool IsStreamSupported(const StreamFlags& stream) const;
```

This component is responsible for a tracker selection, connection, reconnection and setting up the type of eye tracking you need.

---

## GetTrackerInfo

```
bool GetTrackerInfo(TrackerInfo& trackerInfo);
```

Returns detailed information for the currently connected tracker.

**Remarks**
This function is safe to call each tick and can be use to manually check for device changes.

**Return value**
Returns true if the call succeeded and a result is available in *trackerInfo*, otherwise false.

**Parameters**
*trackerInfo* — Will be modified to hold the tracker information if the call succeeds. Will remain unmodified on failure.

---

## GetTrackerInfo by url

```
bool GetTrackerInfo(const char* url, TrackerInfo& trackerInfo);
```

Returns detailed information for the tracker specified by *url*.

**Remarks**
Every tracker has a unique URL which can be represented by a string. Url can be treated as a unique identifier for the tracker and is used for establishing a connection and getting the tracker info. This function is safe to call each tick and can be use to manually check for device changes.

**Return value**
Returns true if the call succeeded and a result is available in *trackerInfo*, otherwise false.

**Parameters**
*url* — a unique identifier for the eye tracker. Represented as an ASCII zero-terminated string. *trackerInfo* — Will be modified to hold the tracker information if the call succeeds. Will remain unmodified on failure.

---

## UpdateTrackerInfos

```
void UpdateTrackerInfos();
```

Starts a scan for all Tobii trackers physically attached to the computer. After the scan has finished the result can be retrieved by calling **GetTrackerInfos**.

**Remarks**

This is an asynchronous operation running on a thread dedicated to scanning for attached trackers.

---

## GetTrackerInfos

```
bool GetTrackerInfos(const TrackerInfo*& trackerInfos, int& numberOfTrackerInf
```

Returns true if the scan triggered by **UpdateTrackerInfos** has finished and a result is available, otherwise false.

**Remarks**
If **GetTrackerInfos** is called without first calling **UpdateTrackerInfos** then trackerInfos will point to an outdated TrackerInfo-array.

The user needs to supply a pointer to a TrackerInfo which **GetTrackerInfos** will set to point to a TrackerInfo-array. The length of this array is assigned to

numberOfTrackerInfos.

**Parameters**
*trackerInfos* — a reference to a pointer that will be set to point to an array of TrackerInfo.
*numberOfTrackerInfos* — the number of TrackerInfo items in the array.

---

## TrackHMD

```
bool TrackHMD();
```

Starts tracking of a head-mounted display.

**Remarks**
Supports no more than one head mounted display attached to a computer. In case of several head-mounted displays attached to a computer, the specific one that is chosen for connection is undefined.

**Return value**
Returns *true* if TGI manages to connect to a head-mounted eye tracker and start tracking, *false* otherwise.

---

## TrackRectangle

```
bool TrackRectangle(const Rectangle& rectangle);
```

Connects to a tracker that is mounted to a monitor that contains **rectangle**. Gaze point coordinates are remapped to a given rectangle.

**Remarks**
If there is no tracker mounted to a display that contains *rectangle*, TGI will disconnect from any previously connected trackers.
The user gets gaze point coordinates from -1.0 to 1.0 inside the *rectangle* and (-∞, -1) and (1, ∞) outside of it. Coordinates are not clamped which means you can get values that are bigger than 1 and smaller than -1. If rectangle bounds change, call *TrackRectangle* again with an updated *rectangle*. If there are several trackers mounted on a display which contains given rectangle, it's undefined which tracker is connected to.

**Return value**

Returns *true* if TGI manages to connect to a tracker and start tracking, *false* otherwise.

**Parameters**
*rectangle* — an object of type *Rectangle* that represents a rectangle in an operating system coordinate system that the user wants to track.

---

## TrackWindow

```
bool TrackWindow(void* windowHandle);
```

Connects to a tracker that is mounted to a monitor that contains a window with a given *windowHandle*. Gaze point coordinates are remapped to this window.

**Remarks**
If there is no tracker mounted to a display that contains a given window, TGI disconnects from any previously connected trackers.
The user gets gaze point coordinates from -1.0 to 1.0 inside the window and (-∞, -1) and (1, ∞) outside of it. Coordinates are not clamped which means you can get values that are bigger than 1 and smaller than -1. If window bounds change, TGI automatically detects that and adjusts gaze remapping. If the window moves to another monitor, TGI disconnects from the current tracker and connect to a one, mounted on a "new" monitor. If there are several trackers mounted on a display which contains given window, TGI connects to one of them in a random order.

**Return value**
Returns *true* if TGI manages to connect to a tracker and start tracking, *false* otherwise.

**Parameters**
*windowHandle* — game window handle in an operating system.

---

## StopTracking

```
void StopTracking();
```

Stops tracking and the device thread.

**Remarks**
Similar to *Shutdown*, but this function does not dispose of the internal state of TGI,

and that's why is more lightweight than *Shutdown*. There are no common scenarios that involve a call to *StopTracking*; the frequent interaction is to keep tracking active until the game stops.

---

## IsConnected

```
bool IsConnected() const;
```

Indicates whether TGI is currently connected to a tracker.

**Remarks**
Be aware that by "connected" is meant "established connection and ready to get data streams" rather than "physically attached to a computer". If *Is Connected* returns true, it means that the user is ready to get data streams or already is getting them. If it returns false, it means that TGI didn't establish a connection to any tracker.

**Return value**
Returns *true* if TGI is currently connected to a tracker, *false* otherwise.

---

## IsEnabled

```
bool IsEnabled() const;
```

Indicates whether a tracker is enabled in Tobii software.

**Remarks**
When a tracker is disabled in Tobii software by the user, it is possible to connect to the tracker, but it provides no data. Common behavior for the game is to disable all Tobii settings in the menu and turn off all features.

**Return value**
Returns *true* if a tracker is currently enabled in Tobii software, *false* otherwise.

---

## IsStreamSupported

```
bool IsStreamSupported(const StreamFlags& stream) const;
```

Indicates whether a connected tracker supports a specific stream or not.

**Remarks**
Please note that this will only ever return true if a connection has been made to a tracker that supports the StreamFlags. Use IsConnected() to ensure that a connection has been made.

**Return value**
Returns *true* if the specified stream is supported by the currently connected tracker, *false* otherwise.

# IStreamsProvider

```
int GetHeadPoses(const HeadPose*& headPoses);
bool GetLatestHeadPose(HeadPose& headPose);
int GetGazePoints(const GazePoint*& gazePoints);
bool GetLatestGazePoint(GazePoint& gazePoint);
int GetUnfilteredGazePoints(const GazePoint*& gazePoints);
bool GetLatestUnfilteredGazePoint(GazePoint& gazePoint);
int GetHMDGaze(const HMDGaze*& hmdGaze);
bool GetLatestHMDGaze(HMDGaze& latestHMDGaze);
bool IsPresent();
void SetAutoUnsubscribe(StreamType stream, float timeout);
void UnsetAutoUnsubscribe(StreamType stream);
void ConvertGazePoint(const GazePoint& fromGazePoint, GazePoint&
  toGazePoint, UnitType fromUnit, UnitType toUnit);
```

This component is responsible for getting the data streams from the tracker.

---

## GetHeadPoses

```
int GetHeadPoses(const HeadPose*& headPoses);
```

Gets all head pose data between the two latest calls to **Update**.

**Remarks**
This method is usually used for writing filters. For most of the features, it is enough to call **GetLatestHeadPose**.
The user should allocate a pointer to a *HeadPose* array. After a call to **GetHeadPoses** it gets filled with an address of a head poses array. Head pose contains head position and rotation.
*Head position* is measured in **millimeters**. *Head rotation* is measured in **degrees**.

**Return value**
Returns the amount of head poses received between the two latest calls to **Update**.

**Parameters**
*headPoses* — a reference to a pointer that will be directed to an array of head poses.

---

## GetLatestHeadPose

```
bool GetLatestHeadPose(HeadPose& headPose);
```

Gets the latest head pose data.

**Remarks**
Provides access to the latest head tracking data (head position and rotation).
*Head position* is measured in **millimeters**. *Head rotation* is measured in **degrees**.

**Return value**
Returns *true* if there were any new head tracking packets since last call to **Update**, *false* otherwise.

**Parameters**
*headPose* — storage for latest head rotation and position.

---

## GetGazePoints

```
int GetGazePoints(const GazePoint*& gazePoints);
```

Gets all gaze points between the two latest calls to **Update**.

**Remarks**
This method is usually used for writing filters. For most of the features, it is enough to call **GetLatestGazePoint**.
The user should allocate a pointer to a *GazePoint* array. After a call to **GetGazePoints** it gets filled with an address of a gaze points array. Gaze point is a 2D point on a screen in a signed normalized space (from -1 to 1). Gaze coordinates get remapped to a rectangle or a window depending on which type of tracking the user uses.

**Return value**
Returns the amount of gaze points received between the two latest calls to **Update**.

**Parameters**
*gazePoints* — a reference to a pointer that will be directed to an array of gaze points.

---

## GetLatestGazePoint

```
bool GetLatestGazePoint(GazePoint& gazePoint);
```

Gets the latest gaze point.

**Remarks**
Provides access to the latest gaze point, which is a 2D coordinate in signed normalized space (from -1 to 1). Gaze coordinates get remapped to a rectangle or a window depending on which type of tracking the user uses.

**Return value**
Returns *true* if there were any new eye tracking packets since last call to **Update**, *false* otherwise.

**Parameters**
*gazePoint* — storage for latest gaze point.

---

## GetUnfilteredGazePoints

```
int GetUnfilteredGazePoints(const GazePoint*& gazePoints);
```

Gets all unfiltered gaze points between the two latest calls to **Update**.
Note: Unfiltered gaze points are only available on Tobii Eye Tracker 5 and newer devices, alternatively on an older device with a special license provided by Tobii and used in the call to **GetApi** or **GetApiDynamic**.

**Remarks**
This method is usually used for writing filters. For most of the features, it is enough to call **GetLatestUnfilteredGazePoint**. The user should allocate a pointer to a *GazePoint* array. After a call to **GetUnfilteredGazePoints** it gets filled with an address of a gaze points array. Gaze point is a 2D point on a screen in a signed normalized space (from -1 to 1). Gaze coordinates get remapped to a rectangle or a window depending on which type of tracking the user uses.

**Return value**
Returns the amount of gaze points received between the two latest calls to **Update**.

**Parameters**
*gazePoints* — a reference to a pointer that will be directed to an array of gaze points.

## GetLatestUnfilteredGazePoint

```
bool GetLatestUnfilteredGazePoint(GazePoint& gazePoint);
```

Gets the latest unfiltered gaze point.
Note: Unfiltered gaze points are only available on Tobii Eye Tracker 5 and newer devices, alternatively on an older device with a special license provided by Tobii and used in the call to **GetApi** or **GetApiDynamic**.

**Remarks**
Provides access to the latest unfiltered gaze point, which is a 2D coordinate in signed normalized space (from -1 to 1). Gaze coordinates get remapped to a rectangle or a window depending on which type of tracking the user uses.

**Return value**
Returns *true* if there were any new eye tracking packets since last call to **Update**, *false* otherwise.

**Parameters**
*gazePoint* — storage for latest gaze point.

---

## GetHMDGaze

```
int GetHMDGaze(const HMDGaze*& hmdGaze);
```

Gets all VR gaze data packets between the two latest calls to **Update**.

**Remarks**
This method is usually used for writing filters. For most of the features, it is enough to call **GetLatestHMDGaze**.
The user should allocate a pointer to a *HMDGaze* array. After a call to **GetHMDGaze** it gets filled with an address of a VR gaze data packets array.

**Return value** Returns the amount of VR gaze data packets received between the two latest calls to **Update**.

**Parameters**
*HMDGaze* — a reference to a pointer that will be directed to an array of VR gaze data packets.

---

## GetLatestHMDGaze

```
bool GetLatestHMDGaze(HMDGaze& latestHMDGaze);
```

Gets the latest available VR gaze data packet from the head mounted display.

**Return value**
Returns *true* if there were any new gaze packets in between the two latest calls to **Update**.

**Parameters**
*latestHMDGaze* — storage for latest VR gaze data.

---

## IsPresent

```
bool IsPresent();
```

Indicates whether the user is present in front of the computer.

**Remarks**
Presence is defined as a combined relation of gaze and head tracking data.

**Return value**
Returns **true** if the user is present, **false** otherwise.

---

## ConvertGazePoint

```
void ConvertGazePoint(const GazePoint& fromGazePoint,
  GazePoint& toGazePoint, UnitType fromUnit, UnitType toUnit);
```

Converts gaze point from one unit to another.

**Remarks**
Often used in features that require a dead zone in millimeters.

**Parameters**
*fromGazePoint* — gaze point that user wants to convert.
*toGazePoint* — gaze point that is going to be filled with the result of conversion.
*fromUnit* — units of *fromGazePoint*.
*toUnit* — the unit that the user wants to convert to.

## SetAutoUnsubscribe

```
void SetAutoUnsubscribe(StreamType stream, float timeout);
```

Sets a behavior to unsubscribe automatically from a given stream after a timeout.

**Remarks**
The default behavior for all streams but *Gaze* is to unsubscribe if the stream is not consumed for more than 5 seconds. By consuming a stream, we mean a call to *IsPresent* for a presence stream or *GetHeadPoses* or *GetLatestHeadPose* for a head poses stream.
The default behavior for a *Gaze* stream is to be always subscribed after the first call to *GetGazePoints* or *GetLatestGazePoint*.

**Parameters**
*stream* — the stream that should be automatically unsubscribed from after a timeout.
*timeout* — a timeout in seconds that is used to unsubscribe from a stream.

---

## UnsetAutoUnsubscribe

```
void UnsetAutoUnsubscribe(StreamType stream);
```

Sets a behavior to always be subscribed to a given stream after the first consumption.

**Remarks**
The default behavior for all streams but *Gaze* is to unsubscribe if the stream is not consumed for more than 5 seconds. By consuming a stream, we mean a call to *IsPresent* for a presence stream or *GetHeadPoses* or *GetLatestHeadPose* for a head poses stream.
The default behavior for a *Gaze* stream is to be always subscribed after the first call to *GetGazePoints* or *GetLatestGazePoint*.

**Parameters**
*stream* — the stream that you want to always be subscribed to.

# IFeatures

```
IExtendedView* GetExtendedView();
```

This component provides access to all interaction features of TGI.

---

## GetExtendedView

```
IExtendedView* GetExtendedView();
```

Returns an interface to work with *Extended View* feature.

**Remarks**
Extended view converts a user's gaze position and head rotation into a corresponding rotation of the in-game camera.

**Return value**
Returns an object of type *IExtendedView* which provides an interface to setup and use Extended View feature.

# IExtendedView

```
Transformation GetTransformation() const;
bool UpdateSettings(const ExtendedViewSettings& settings);
void ResetDefaultHeadPose();
void Pause(bool reCenter, float transitionDuration = 0.2f);
void UnPause(float transitionDuration = 0.2f);
void IsPaused();
void GetSettings(ExtendedViewSettings& settings) const;
```

This component provides a way to setup and use *ExtendedView* feature.

---

## GetTransformation

```
Transformation GetTransformation() const;
```

Gets the camera transformation that should be applied to an in-game camera to implement Extended View. Angles are returned in **degrees** and are kept inside the angle ranges provided in the settings.

**Return value**
*Transformation* contains rotation and position that should be applied to in-game camera to implement Extended View.

---

## UpdateSettings

```
bool UpdateSettings(const ExtendedViewSettings& settings);
```

Updates settings for Extended View.

To manually prevent Extended View from consuming any head tracking data, set the UseHeadTracking to **false** in the *ExtendedViewSettings* struct you supply to **UpdateSettings**. If head tracking is not consumed elsewhere in your code, and the Capabilities specified in **SetAutoUnsubscribeForCapability** allows for unsubscription of the head tracking stream, all head tracking calculations will be bypassed. For a complete description of the *ExtendedViewSettings* struct, see the specification under **GetSettings** section.

**Remarks**

- **UpdateAdvancedSettings** and **UpdateSimpleSettings** in TGI version 7 has been replaced with **UpdateSettings**.
- **UpdateSettings** in TGI version 6 and earlier has been replaced with **UpdateAdvancedSettings**.
- **UpdateGazeOnlySettings** and **UpdateHeadOnlySettings** have been removed. TGI will no longer switch between settings based on input availability.
- If you want to changes settings based on tracker capabilities, please do so by using the *Capabilities* field in *TrackerInfo* instead.

**Return value**
Returns **true** if all Setting members contained in the *ExtendedViewSettings* were in their allowed ranges, otherwise copies of these values were clamped to their allowed ranges before being accepted and the function returns **false**.

**Parameters**
*settings* — an Extended View settings object.

---

## ResetDefaultHeadPose

```
void ResetDefaultHeadPose();
```

Resets the default head pose for the user to a current position.

**Remarks**
Extended View feature slowly adapts to a comfortable user sitting position. If the user sits in some position for long enough time, Extended View treats this user pose as an origin for the camera rotation(Extended View resets angles to 0,0). If the user changes her sitting position dramatically, it is sometimes beneficial to reset origin to a new sitting position. This can be done using *ResetDefaultHeadPose* function.

---

## Pause

```
void Pause(bool reCenter, float transitionDuration = 0.2f);
```

Pauses the Extended View feature.

**Parameters**
*reCenter* — If *reCenter* is true Extended View will fade out (effectively centering any camera influenced by the Extended View output transform). If *UnPause* is called

before the centering is complete, the centering will be aborted and Extended View will fade in to the normal state again. If *reCenter* is false, Extended View will pause immediately and the transform will stay still, as is, until *UnPause* is called.

*transitionDuration* — The time it will take for Extended View to fade out, when the *reCenter* parameter is true. Default is 0.2 seconds.

---

## UnPause

```
void UnPause(float transitionDuration = 0.2f);
```

Starts the Extended View feature again from a paused (or fading out) state.

**Parameters**
*transitionDuration* - The time it will take for Extended View to fade in. Default is 0.2 seconds.

---

## IsPaused

```
bool IsPaused();
```

Returns the paused state of the Extended View feature

**Return value**
*true* if Extended View is paused (or fading out due to a *Pause* call with *reCenter*=true), *false* otherwise.

---

## GetSettings

```
void GetSettings(ExtendedViewSettings& settings);
```

Get the current Extended View settings

**Parameters**
*settings* — a reference to an Extended View settings object.

# ExtendedViewSettings

```
Setting<bool>  UseHeadTracking;
Setting<bool>  HeadTrackingAutoReset;
Setting<float> HeadRotationResponsiveness;
Setting<float> GazeResponsiveness;
Setting<float> GazeYawLimitDegrees;
Setting<float> GazePitchUpLimitDegrees;
Setting<float> GazePitchDownLimitDegrees;

AxisSettings HeadYawRightDegrees;
AxisSettings HeadYawLeftDegrees;
AxisSettings HeadPitchUpDegrees;
AxisSettings HeadPitchDownDegrees;
AxisSettings HeadRollRightDegrees;
AxisSettings HeadRollLeftDegrees;
AxisSettings HeadXRightMm;
AxisSettings HeadXLeftMm;
AxisSettings HeadYUpMm;
AxisSettings HeadYDownMm;
AxisSettings HeadZBackMm;
AxisSettings HeadZForwardMm;

void SetHeadMultiAxisSettingsValues(
    AxisFlags axes,
    AxisDirectionFlags axisDirections,
    Setting<float> AxisSettings::* pAxisSettingsMember,
    float value,
    bool valueIsNormalizedOnRangeMagnitude);

void SetHeadAllRotationAxisSettingsSensitivity(float sensitivityScaling);

void SetHeadAllRotationAxisSettingsLimitNormalized(
    float limitNormalizedOnRange);

void SetHeadAllPositionAxisSettingsSensitivity(float sensitivityScaling);

void SetHeadAllPositionAxisSettingsLimitNormalized(
    float limitNormalizedOnRange);

void SetEyeHeadTrackingRatio(float eyeHeadTrackingRatio);

void SetCameraMaxAngleYaw(float positiveYawLimitDegrees);

void SetCameraMaxAnglePitchUp(float positivePitchUpLimitDegrees);

void SetCameraMaxAnglePitchDown(float negativePitchDownLimitDegrees);

void SetCenterStabilization(float centerStabilization);
```

## UseHeadTracking

```
Setting<bool>  UseHeadTracking;
```

To manually prevent Extended View from consuming any head tracking data, set the UseHeadTracking to **false** in the ExtendedViewAdvancedSettings structure you supply to **UpdateAdvancedSettings**. If head tracking is not consumed elsewhere in your code, and the Capabilities specified in **SetAutoUnsubscribeForCapability** allows for unsubscription of head tracking stream, all head tracking calculations will be bypassed.

## HeadTrackingAutoReset

```
Setting<bool>  HeadTrackingAutoReset;
```

Subtly adapt the Extended View to small changes in user head pose while playing.

## HeadRotationResponsiveness

```
Setting<float> HeadRotationResponsiveness;
```

The responsiveness of head rotation input.

## GazeResponsiveness

```
Setting<float> GazeResponsiveness;
```

The responsiveness of gaze input.

## GazeYawLimitDegrees

```
Setting<float> GazeYawLimitDegrees;
```

Adjust the maximum yielded yaw angles from gaze input.

## GazePitchUpLimitDegrees

```
Setting<float> GazePitchUpLimitDegrees;
```

Adjust the maximum yielded pitch up angles from gaze input.

## GazePitchDownLimitDegrees

```
Setting<float> GazePitchDownLimitDegrees;
```

Adjust the maximum yielded pitch down angles from gaze input.

## HeadYawRightDegrees

```
AxisSettings HeadYawRightDegrees;
```

Contains the characteristics of the yaw **right** (+) head rotation axis.

## HeadYawLeftDegrees

```
AxisSettings HeadYawLeftDegrees;
```

Contains the characteristics of the yaw **left** (-) head rotation axis.

## HeadPitchUpDegrees

```
AxisSettings HeadPitchUpDegrees;
```

Contains the characteristics of the pitch **up** (+) head rotation axis.

## HeadPitchDownDegrees

```
AxisSettings HeadPitchDownDegrees;
```

Contains the characteristics of the pitch **down** (-) head rotation axis.

## HeadRollRightDegrees

```
AxisSettings HeadRollRightDegrees;
```

Contains the characteristics of the roll **right** (+) head rotation axis.

## HeadRollLeftDegrees

```
AxisSettings HeadRollLeftDegrees;
```

Contains the characteristics of the roll **left** (-) head rotation axis.

## HeadXRightMm

```
AxisSettings HeadXRightMm;
```

Contains the characteristics of the X **right** (+) head position axis.

## HeadXLeftMm

```
AxisSettings HeadXLeftMm;
```

Contains the characteristics of the X **left** (-) head position axis.

## HeadYUpMm

```
AxisSettings HeadYUpMm;
```

Contains the characteristics of the Y **up** (+) head position axis.

## HeadYDownMm

```
AxisSettings HeadYDownMm;
```

Contains the characteristics of the Y **down** (-) head position axis.

## HeadZBackMm

```
AxisSettings HeadZBackMm;
```

Contains the characteristics of the Z **back** (+) head position axis.

## HeadZForwardMm

```
AxisSettings HeadZForwardMm;
```

Contains the characteristics of the Z **forward** (-) head position axis.

## SetHeadMultiAxisSettingsValues

```
void SetHeadMultiAxisSettingsValues(
    AxisFlags axes,
    AxisDirectionFlags axisDirections,
    Setting<float> AxisSettings::* pAxisSettingsMember,
    float value,
    bool valueIsNormalizedOnRangeMagnitude);
```

Sets the Value of a specific member in the AxisSettings of multiple specified head tracking axes.

**Remarks**
This function enables you to specify a set of AxisSettings and set a specified member of all of them in one call. The other helper functions in this struct makes use of it for their specific purposes, but you are free to use this function to cater to some specific functionality not covered by the other functions.

**Parameters**
*axes* - A bit field specifying which axes to operate on.

*axisDirections* - A bit field further specifying which direction(s) of the specified axes to operate on.

*pAxisSettingsMember* - A pointer-to-member specifying which Setting member of AxisSettings of the specified axes to set. Valid usage examples: &AxisSettings::Limit, &AxisSettings::SensitivityScaling, &AxisSettings::SCurveStrengthNorm, &AxisSettings::SCurveMidPointNorm, &AxisSettings::DeadZoneNorm.

*value* - Either, if *valueIsNormalizedOnRangeMagnitude* is false: The float value to assign to the Value of the specified AxisSettings member, or if *valueIsNormalizedOnRangeMagnitude* is true: A normalized float (0.0f to 1.0f), used to interpolate over the specified AxisSettings member's MinMaxRange. For positive axes, a value of 0.0f maps to the Min of the range, and 1.0f maps to the Max of the range. For negative axes, a value of 0.0f maps to the Max of the range, and 1.0f maps to the Min of the range. (i.e. value maps to the magnitude of the range).

*valueIsNormalizedOnRangeMagnitude* - A bool used to specify if the value should be interpreted as normalized on the magnitude of the range as described above.

## SetHeadAllRotationAxisSettingsSensitivity

```
void SetHeadAllRotationAxisSettingsSensitivity(float sensitivityScaling);
```

Sets the SensitivityScaling Value of the AxisSettings of all positive and negative headtracking rotation-axes.

**Parameters**
*sensitivityScaling* - Acts as a multiplier on the head tracking rotation input to produce the ExtendedView rotation output.

## SetHeadAllRotationAxisSettingsLimitNormalized

```
void SetHeadAllRotationAxisSettingsLimitNormalized(
    float limitNormalizedOnRange)
```

Sets the Limit Value of the AxisSettings of all positive and negative head tracking rotation-axes.

**Parameters**
*limitNormalizedOnRange* - a normalized float (0.0f to 1.0f) specifying the magnitude of the rotation angle-limit normalized on the MinMaxRange for each AxisSettings.

## SetHeadAllPositionAxisSettingsSensitivity

```
void SetHeadAllPositionAxisSettingsSensitivity(float sensitivityScaling)
```

Sets the SensitivityScaling Value of the AxisSettings of all positive and negative head tracking position-axes.

**Parameters**
*sensitivityScaling* - Acts as a multiplier on the head tracking position input to produce the ExtendedView position output.

## SetHeadAllPositionAxisSettingsLimitNormalized

```
void SetHeadAllPositionAxisSettingsLimitNormalized(
    float limitNormalizedOnRange)
```

Sets the Limit Value of the AxisSettings of all positive and negative head tracking position-axes.

**Parameters**
*limitNormalizedOnRange* - a normalized float (0.0f to 1.0f) specifying the magnitude of the position-limit normalized on the MinMaxRange for each AxisSettings.

## SetEyeHeadTrackingRatio

```
void SetEyeHeadTrackingRatio(float eyeHeadTrackingRatio);
```

Sets the ratio between eye- and head tracking. This will change the distribution of the maximum gaze and head angles.

**Parameters**
*eyeHeadTrackingRatio* - Range 0.0f to 1.0f. 0.0f = only eye tracking, 1.0f = only head tracking.

## SetCameraMaxAngleYaw

```
void SetCameraMaxAngleYaw(float positiveYawLimitDegrees);
```

Sets the maximum yaw angle extended view can produce in degrees.

**Parameters**
*positiveYawLimitDegrees* - This value limits the maximum output of rotating your head left or right + looking left or right. This value is mirrored for left and right rotations and a value of 0 would effectively turn the feature off. Extended view yaw angle will be capped at this value.

## SetCameraMaxAnglePitchUp

```
void SetCameraMaxAnglePitchUp(float positivePitchUpLimitDegrees);
```

Sets the maximum pitch up angle extended view can produce in degrees,

**Parameters**
*positivePitchUpLimitDegrees* - This value limits the maximum output of rotating your

head upwards + looking up. Extended view pitch angle will be capped at this value.

### SetCameraMaxAnglePitchDown

```
void SetCameraMaxAnglePitchDown(float negativePitchDownLimitDegrees);
```

Sets the maximum pitch down angle extended view can produce in degrees,

**Parameters**
*negativePitchDownLimitDegrees* - This value limits the maximum output of rotating your head downwards + looking down. Extended view pitch angle will be capped at this value.

### SetCenterStabilization

```
void SetCenterStabilization(float centerStabilization);
```

Increase to make head tracking more stable in the center. Operates on the AxisSettings members controlling the headtracking deadzone and S-curve of all rotational axes.

**Parameters**
*centerStabilization* - Range 0.0f to 1.0f

---

## AxisSettings

The **AxisSettings** members defines an S-shaped curve that scales the influence of head rotation and translation. The shape of the curve is defined by **SCurveStrengthNorm**, **SCurveMidPointNorm** and **DeadZoneNorm**. **Limit** sets the range of the curve in degrees for rotations axes and in millimeters for position axes. **SensitivityScaling** scales the input before applying the curve.

```
Setting<float> Limit;
Setting<float> SensitivityScaling;
Setting<float> SCurveStrengthNorm;
Setting<float> SCurveMidPointNorm;
Setting<float> DeadZoneNorm;

int PlotCurvePoints(
    Vector2d allocatedPointsBuf[],
    int maxPoints,
    bool scalePointsToLimit = false) const;
```

## Limit

```
Setting<float> Limit;
```

Defines the limit of the axis, in degrees for rotation axes, and in millimeters for position axes.

## SensitivityScaling

```
Setting<float> SensitivityScaling;
```

A sensitivity scaling of the head pose input. The input will be multiplied by this value before applying the axis curve.

## SCurveStrengthNorm

```
Setting<float> SCurveStrengthNorm;
```

A normalized value that defines how much the curve bends. Set to 0 for a straight line and 1 for maximum bend.

## SCurveMidPointNorm

```
Setting<float> SCurveMidPointNorm;
```

Normalized value defining the mid-point of the curve.

## DeadZoneNorm

```
Setting<float> DeadZoneNorm;
```

A normalized value defining the start point of the axis curve.

## PlotCurvePoints

```
int PlotCurvePoints(
    Vector2d allocatedPointsBuf[],
    int maxPoints, bool
    scalePointsToLimit = false) const;
```

Get a poly-line representation of the curve defined by the members of the **AxisSettings** struct. You may use this function to get a series of points that can be drawn to visualize the shape of those curves.

**Parameters**
*allocatedPointsBuf* — an array of **Vector2d** objecs to be used as output of points.

*maxPoints* — the maximum number of points this function may write to the output array *allocatedPointsBuf*.

**Return value**
Returns the number of points written to the *allocatedPointsBuf*. This will always be less than or equal to the *maxPoints* parameter depending on the shape of the curve.

**Remarks**
The *allocatedPointsBuf* **must** be allocated by the user and big enough to hold *maxPoints* **Vector2d** ojects.

The returned actual number of points written to the *allocatedPointsBuf* **may** be smaller than *maxPoints*.

All points written are positive and normalized coordinates starting with (0,0) and ending with (1,1).

# Setting<T>

```
T Value;
const SettingMetadata<T> Metadata;

bool Clamp();
```

Represents a setting of type T with corresponding metadata. It has operators to handle assignment and casting and can be used just as the templated type T.

## Value

```
T Value;
```

The current value of the setting.

## Metadata

```
const SettingMetadata<T> Metadata;
```

Metadata contains additional information about the setting.

## Clamp

```
bool Clamp();
```

Clamps the setting's *Value* inside the range defined in *Metadata*.

# SettingMetadata<T>

```
const T           Default;
const Range<T>    MinMaxRange;
```

A type to handle additional information about a setting like default value and valid range.

## Default

```
const T           Default;
```

The default value of the setting.

## MinMaxRange

```
const Range<T>    MinMaxRange;
```

The valid range of the setting.

# Range<T>

```
T   Min;
T   Max;
```

Defines a range from Min to Max.

## Min

```
T   Min;
```

The minimum value of the range.

## Max

```
T   Max;
```

The maximum value of the range.

# IFilters

```
const ResponsiveFilterSettings& GetResponsiveFilterSettings() const;
void SetResponsiveFilterSettings(ResponsiveFilterSettings settings);
const AimAtGazeFilterSettings& GetAimAtGazeFilterSettings() const;
void SetAimAtGazeFilterSettings(AimAtGazeFilterSettings settings);
const BilateralFilterSettings& GetBilateralFilterSettings() const;
void SetBilateralFilterSettings(BilateralFilterSettings settings);
void GetResponsiveFilterGazePoint(GazePoint& gazePoint) const;
void GetAimAtGazeFilterGazePoint(GazePoint& gazePoint,
  float &gazePointStability) const;
void GetBilateralFilterGazePoint(GazePoint& gazePoint) const;
```

This component provides several gaze filters that could be used for different features. The Aim at gaze filter is often used for choosing a point that user focused on with a certain stability measurement.
The Responsive filter is used to get a smoother, sticky movement of a gaze point.
The Bilateral filter is originally a non-linear, edge-preserving, and noise-reducing smoothing filter for images that is based on pixel distance and intensity. Here the intensity dimension is replace with time to make an eye tracking signal that is very responsive to larger movements and yet very stable for small.

---

## Filter Settings

```
const ResponsiveFilterSettings& GetResponsiveFilterSettings() const;
void SetResponsiveFilterSettings(ResponsiveFilterSettings settings);
const AimAtGazeFilterSettings& GetAimAtGazeFilterSettings() const;
void SetAimAtGazeFilterSettings(AimAtGazeFilterSettings settings);
const BilateralFilterSettings& GetBilateralFilterSettings() const;
void SetBilateralFilterSettings(BilateralFilterSettings settings);
```

Getters and setters for filters settings.

**Remarks**
Default values for settings could be found in the main header *tobii_gameintegration.h*. The default value in filter settings for **IsEnabled** is *false*, which means that filters would not be updated to save the computational costs. **To enable a filter, set *IsEnabled* in filter settings to *true***.

## Filtered Gaze Points

```
void GetResponsiveFilterGazePoint(GazePoint& gazePoint) const;
void GetAimAtGazeFilterGazePoint(GazePoint& gazePoint,
  float &gazePointStability) const;
void GetBilateralFilterGazePoint(GazePoint& gazePoint) const;
```

Methods to get a filtered gaze point.

**Remarks**
To use filters, the first thing to do is to enable them in filter settings. After the filter is enabled, gaze points could be retrieved using these methods.
Aim at gaze filter provides an additional measurement for the filtered gaze point called *gazePointStability*. The value for stability ranges from 0 to 1, where 0 is a point that the user is completely not focused on, and 1 corresponds to a point that the user focused on for a significant time. The primary use case for aim at gaze filter is to find a point to aim at when the user goes into the aiming mode.

# IStatistics

```
void SetFeatureList(const Feature* gameFeatures, int numberOfFeatures);
const char* GetLiteral(Literal literal) const;
void SendFeatureEnabled(int featureId);
void SendFeatureDisabled(int featureId);
void SendFeaturesState();
void StopAllLogging();
void ResumeAllLogging();
```

This component provides a way to send usage statistics about features being turned on or off.

---

## SetFeatureList

```
void SetFeatureList(const Feature* gameFeatures, int numberOfFeatures);
```

**Remarks**
Sets the feature list to the statistics component. Each feature has an Id, Name and an initial state. **For known features, like Extended View, please use the GetLiteral function to get the consistent name.** See sample for the usage example. The initial state of the feature is the value's from the game settings, whether the feature is initialized or not. After the feature list is set, call to *SendFeaturesState* once.

---

## GetLiteral

```
const char* GetLiteral(Literal literal) const;
```

**Remarks**
Gets the common literals for statics logging. Like "game started", or names of the common features. Use it to name your features in the consistent way. See the sample for the usage example.

**Return value**
String literal that is consistent between the game integrations.

---

## SendFeatureEnabled/Disabled

```
void SendFeatureEnabled(int featureId);
void SendFeatureDisabled(int featureId);
```

**Remarks**
Functions, used to send statistics when user turned the feature on or off in settings. The featureId is the same you provided in the *SetFeatureList*.

---

## SendFeatureState

```
void SendFeaturesState();
```

**Remarks**
Sends the current state of the features, being enabled or disabled. Use it once in the beginning of the game, when settings are read from the save file. During the game play, when the user changes settings, use *SendFeatureEnabled* and *SendFeatureDisabled*.

---

## StopAllLogging

```
void StopAllLogging();
```

**Remarks**
Stops the statistics logging from TGI

---

## ResumeAllLogging

```
void ResumeAllLogging();
```

**Remarks**
Resumes the statistics logging from TGI