Developer's Guide

Tobii EyeX SDK for C/C++

November 19, 2015
Tobii Tech

The Tobii EyeX Software Development Kit (SDK) for C/C++ provides you with everything you need for building games and applications using the Tobii EyeX API and the C or C++ programming languages.

Table of Contents

Introduction

The Tobii EyeX Software Development Kit (SDK) for C/C++ provides you with everything you need for building games and applications using the Tobii EyeX API and the C or C++ programming languages.

Building a great user experience starting from eye-gaze coordinates is a challenging task. The Tobii EyeX Engine takes care of the groundwork for you, so you don't need to worry about hardware configuration, calibration, or data processing. The interaction concepts provided by the Tobii EyeX software give you the benefit of Tobii's extensive experience with eye-gaze interaction, and let you avoid common design pitfalls. The EyeX Engine provides a faster way to create a consistent user experience across the entire operating system.

The EyeX SDK is available on the same platforms as the EyeX Engine: currently Windows 8.1, Windows 8, and Windows 7.

The EyeX SDK is available in three variants, one for C/C++, one for .NET and one for the Unity game engine. More will follow and they are all available for download from the Tobii Developer Zone.

Note: If your application is intended to run in an embedded or single-application[1] environment, or if it needs to run on platforms where the EyeX Engine isn't available, then the low-level Gaze SDK from Tobii might be a better match for your needs. The Gaze SDK is available for download at the Tobii Developer Zone.

[1] Single-application means that the application replaces or hides the Windows shell, so that no other applications are visible. For example, in an information kiosk or ATM.

Getting started

Install and make sure Tobii EyeX is running

As the EyeX SDK builds on the EyeX interaction concepts and the API provided by the EyeX Engine, the first thing that you need to do is to install the Tobii EyeX software, and ensure that it works with your Tobii EyeX Controller or other Tobii eye tracker.

Now, does it track your eyes properly? Good, then you're ready for the next step.

Building and running the code samples

This guide assumes that you have installed Microsoft Visual Studio 2012 or later on your development machine. It should be possible to use the SDK with other editors and compilers as well, but then you'll have to find out how to set include paths, etc, yourself.

The EyeX SDK is distributed as a plain zip file. Extract it to, for example, c:\EyeXSDK. Then browse to the new SDK directory and locate the subdirectory called "samples". There you will find a Visual Studio solution file that includes all the C and C++ code samples:

- The **ActivatableButtons** sample which demonstrates the *activatable behavior*, with two simple activatable buttons. This sample is about as small as it gets for an application using the activatable behavior. It is written in C++ and uses GDI+ for the user interface.
- The **ActivatableBoardGame** sample which demonstrates the *activatable behavior* in the context of a five-in-a-row board game. This sample is slightly more elaborate than the ActivatableButtons sample. It is written in C++ and uses GDI+ for the user interface.
- The **MinimalGazeDataStream** sample which demonstrates the *lightly filtered Gaze point data stream*. This is a console application written in C.
- The **MinimalFixationDataStream** sample which demonstrates the *fixation data stream*. This is a console application written in C.
- The **MinimalStatusNotifications** sample which connects to the EyeX Engine and shows *tracking status, screen settings, current user profile, user presence* and *gaze tracking* information. This is a console application written in C.
- The **MinimalUserProfiles** sample which shows how all available user profiles can be listed and how the current profile can be changed. This is a console application written in C++.

The sample applications are plain Visual Studio projects, which can be built and run from within Visual Studio. So give them a try!

Where to go from here

We strongly recommend browsing through the rest of this Developer's Guide, because it will give you a big picture view of the EyeX Engine and its API. Knowing what the engine can do for you, and how the pieces fit together, will surely be helpful as you move on to create your game or application.

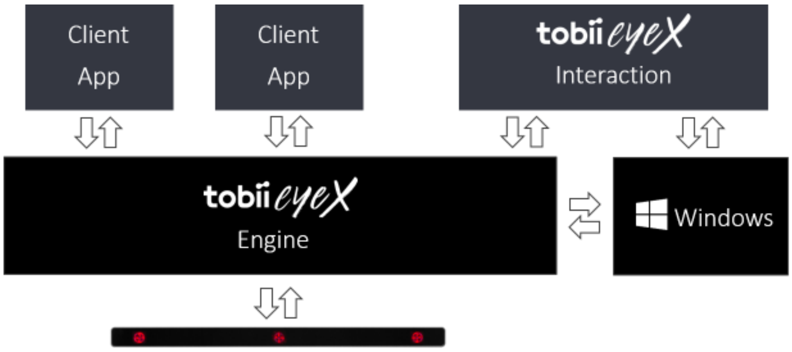
Apart from that, it depends on what you want to do—or how you wish to use the Tobii EyeX interaction concepts in your game or application. Remember that the Tobii Developer Zone is there for you if you need inspiration or if you get stuck.

Introduction to Tobii EyeX

Tobii EyeX is a software package that is used together with compatible eye trackers, such as the Tobii EyeX Controller, to enable new ways to use your eyes for interacting with computers.

The package contains drivers and services for connecting and communicating with the eye tracker as well as the Tobii EyeX Engine and Tobii EyeX Interaction software.

Tobii EyeX Engine is the core software that works like an OS extension for eye tracking. It knows how to configure and talk to the eye tracker, react to system changes, combine the user's eye-gaze and other input and interpret them as user interactions, and it mediates between multiple applications that are using eye tracking simultaneously.



The EyeX Engine also contains a whole lot of eye tracking smartness harnessed over the 10+ years of eye tracking experience at Tobii. This is offered as filtered data streams and built-in heuristics that are specialized in figuring out what the user is actually looking at. Since the EyeX Engine handles all the groundwork with hardware configuration, screen setup, user calibration and so on, you as a game or app developer can focus on creating a great eye-gaze based experience for your players/users.

Tobii EyeX Interaction is a piece of software built on top of the EyeX Engine, and offers a set of basic eye-gaze interactions available out-of-the-box in the Windows environment. The concepts used in EyeX Interaction have matured through beta-testing programs and many iterations of improvements.

The EyeX SDK provides you with access to the EyeX Engine API. It includes code samples, demo scenes, dlls, documentation and code for integration into a selected set of game engines and GUI frameworks

Overview of the Tobii EyeX Engine API

Figure 1 presents an overview of what the EyeX Engine API offers.

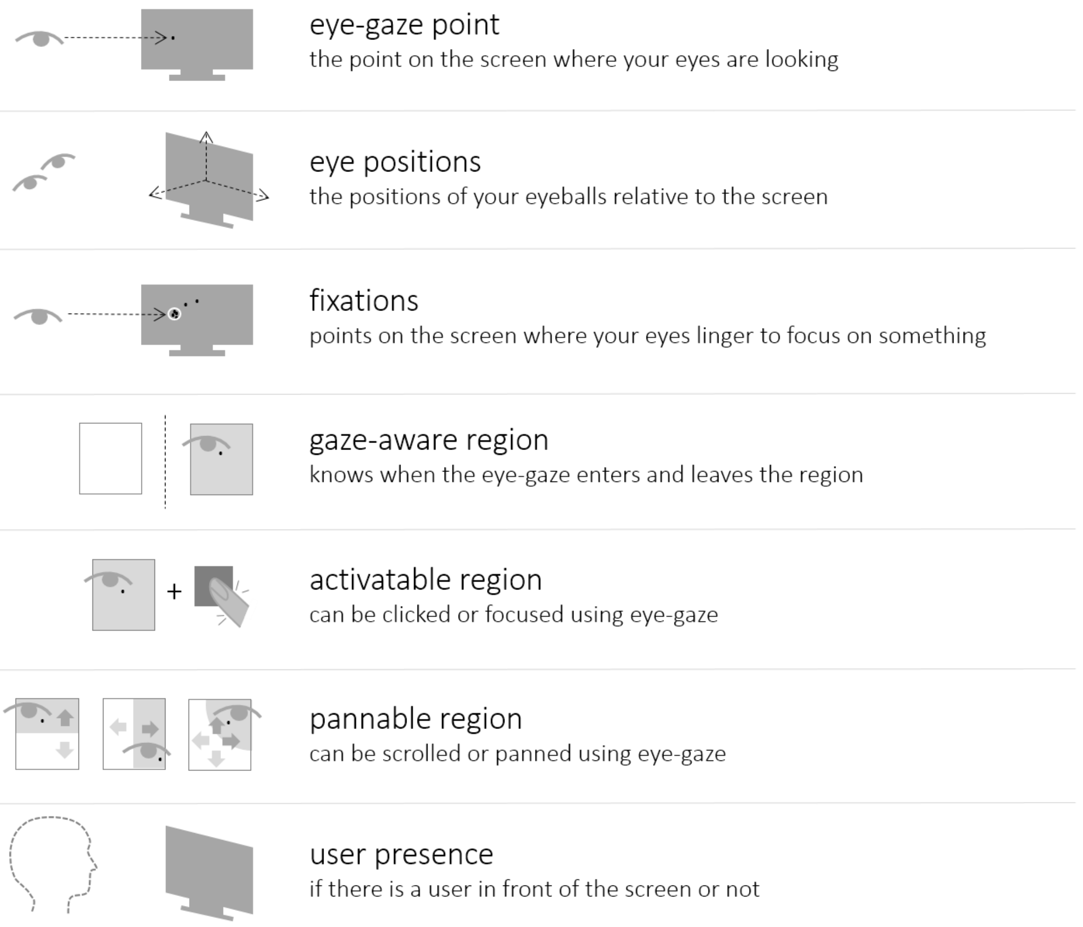


Figure 1 A somewhat simplified view of what the EyeX Engine API can do for you.

The functionality in the EyeX Engine API basically can be grouped in three categories: data streams, states and behaviors. Functionality that falls outside of these categories is there to support and make these three categories possible.

Data Streams

The data streams provide nicely filtered eye-gaze data from the eye tracker transformed to a convenient coordinate system. The point on the screen where your eyes are looking (gaze point), and the points on the screen where your eyes linger to focus on something (fixations) are given as pixel coordinates on the screen. The positions of your eyeballs (eye positions) are given in space coordinates in millimeters relative to the center of the screen.

States

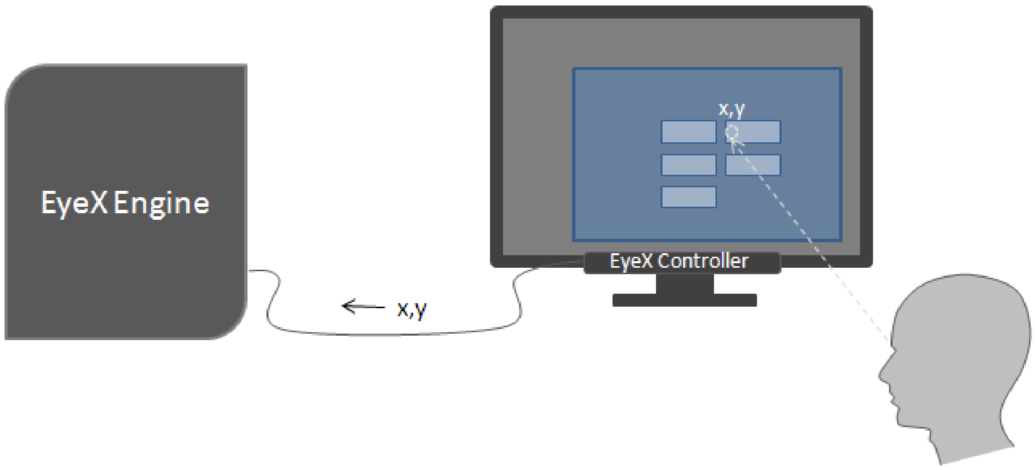
The states provide information about the current state of the EyeX system. There are a couple of more dynamic states related to the user: the user presence state (if there is a user in front of the screen or not) and the gaze tracking state (if the eye tracker is currently able to track the user's eye-gaze or not). There are also states that are related to the system setup, such as the size in millimeters of the display the eye tracker is currently setup for, the currently selected user calibration profile, or if the eye tracker is currently available.

Behaviors

The behaviors are higher level interaction concepts. You can compare them with familiar mouse interactions like click and scroll. The interaction behaviors are related to regions on the screen rather than single points. The regions typically correspond to GUI components like clickable buttons or scrollable panels. The concept of a gaze-aware region is a region that knows if a user is looking at it or not. An activatable region can be clicked by looking at it and simultaneously giving it an activation trigger – such as a tap on a touch pad or click of a keyboard key. The click is direct in the sense that you can click directly on whatever you are looking. The pannable behavior offers a number of different flavors of scrolling and panning a region in different directions. The EyeX Engine uses advanced heuristics and filtering to decide which object a user is trying to interact with and how. This gives a much more stable and precise interaction than one that simply uses the coordinate of the latest gaze point.

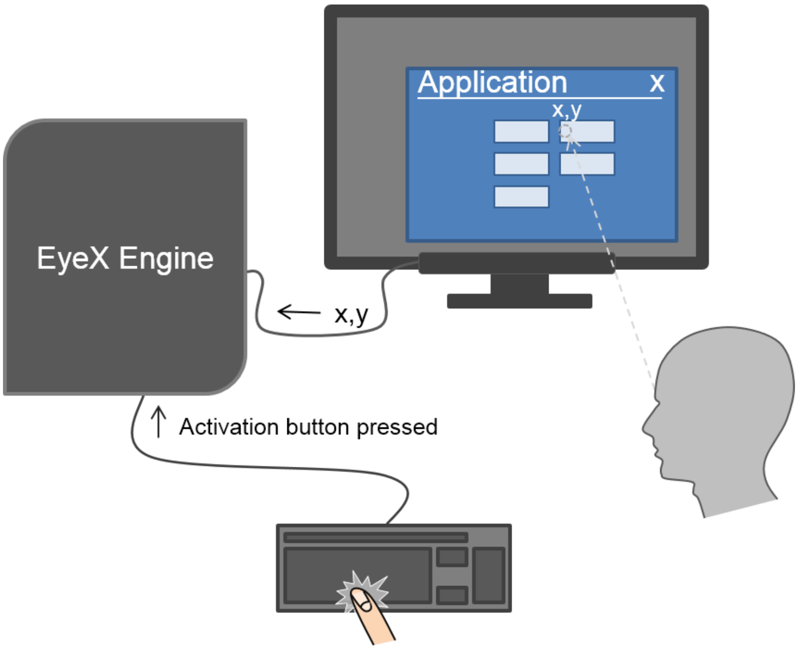
How the engine knows what you are looking at

1. The user looks at the screen. The EyeX Controller calculates the coordinates of the user's gaze point. The EyeX Engine receives the normalized screen coordinates and transforms them to pixel coordinates on the screen.

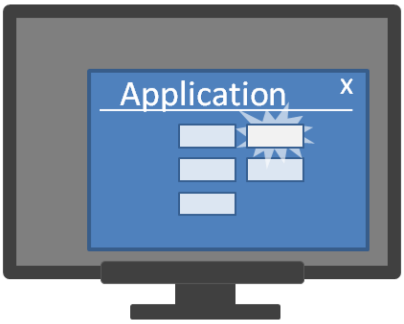


2. The EyeX Engine uses information from the client application to understand what kind of visual object the user is looking at and might want to interact with. Let's say the rectangles on the screen are buttons with the 'Activatable' behavior. That means that they can be clicked using eye gaze. The EyeX Engine now decides which button the user is looking at and then expects the user to indicate that he or she wants to click the button.
3. The client application might offer the user some different ways of actually clicking the button using eye-gaze, but it will always be some combination of looking at the thing you want to click, and giving some secondary input to trigger the click. Let's say that in this example client application the user can trigger a click on the object they are looking at by pressing the space

bar on the keyboard when the client application has input focus.



4. The user presses the space bar while looking at the button he or she wants to click. The client application informs the EyeX Engine that the user has requested to 'Activate' the activatable button currently looked at.
5. The EyeX Engine generates an event that informs the client application which button the user wants to activate.
6. The client application responds to the event and for example gives the user a visual feedback that the button on the screen is clicked, and then performs the action associated with clicking that button.



Using the C API

This section describes how to use the EyeX Engine C API. For a full description of the concepts and terminology used in this section, refer to the *EyeX Engine API reference* section.

Compiling and linking with the C API

The Tobii EyeX Engine C API is made available through a dynamic-link library that your application must be linked with, as well as a set of C header files that describe the API using C syntax.

You will only need to include a single header file—the one called EyeX.h—in your C or C++ program. This include file will in turn include the other header files as needed. All header files for the C API can be found in the include/eyex directory.

The dynamic-link library is available in both 32-bit and 64-bit format and can be found in the lib/x86 and the lib/x64 subdirectories, respectively. The dll must also be copied to the directory where your application's executable file resides, otherwise Windows won't be able to find it at runtime. The code samples use an MSBuild script to copy the correct version to the output directory as part of the linking step, but copying the file manually or by other means works just as well.

Working with interaction objects

One of the most common data types in the C API is called `TX_HANDLE`. It represents some kind of interaction object: an interactor, a snapshot, an event, etc. These are things you absolutely need to know about `TX_HANDLE`s:

- Interaction objects are always associated with a context, and they are reference counted.
- An empty handle is represented by TX_EMPTY_HANDLE.
- You are responsible for releasing *any* `TX_HANDLE`s that the API hands you, using the `txReleaseObject` function. It doesn't matter how you got the handle; if you called a function called `txCreateSomething`, or if one of your callback functions was invoked with a `TX_HANDLE` as a parameter, you still have to release the object.
- When you delete a context, the client library will report any objects that you forgot to release to the log. You should keep an eye on those reports, since resource leaks are a Bad Thing that can cause your application to run gradually slower over time.
- During development it can be quite helpful to be able to peek inside the interaction objects, for example, to see the contents of events that you receive. There is a function called `txDebugObject` that lets you do exactly that; take a look at the samples to see how it can be used. And be sure not to use it in production builds.

Setting up a context

The first thing the application needs to do, to be able to start using the EyeX client library, is to initialize the library using the `txInitializeEyeX` function, and then create an interaction context using the `txCreateContext` function. The context handle is the fundamental point of access to do anything in the library. You need it in order to:

- create a snapshot and the interactors in it,
- register event and query handlers,
- establish a connection to the engine.

The application therefore typically creates the context during initialization, holds on to the context handle through the life span of the application, and deletes the context on shutdown.

Initially, the context's connection to the EyeX Engine is disabled, meaning that there is no communication with the engine. Before enabling the connection, the application uses the context handle to set up the client side of the client–engine interaction. Usually this means registering the message handlers for events, queries and connection state changes. The application could also prepare for data streaming by creating a global interactor snapshot which can then be committed whenever the connection to the engine is established. (It isn't possible to commit a snapshot when disconnected from the engine.)

In some advanced cases the application might also wish to set up a custom allocator and/or threading model. This can be done when the client library is initialized.

Registering handlers

The application needs to register handlers for all types of information from the EyeX Engine it wants to subscribe to.

Handlers for connection state changes, queries, events and messages are registered in the form of callback functions using `txRegisterConnectionStateChangedHandler`, `txRegisterQueryHandler`, `txRegisterEventHandler` and `txRegisterMessageHandler` respectively. The callback functions will be invoked from worker threads.

The connection state

When all is set up, the application enables the context's connection to the EyeX Engine by calling the `txEnableConnection` function. Enabling a connection means allowing a connection to be established and kept alive until it is disabled, or until the context is destroyed.

When the context's connection has been enabled, its state will change to one of the following:

- `TX_CONNECTIONSTATE_CONNECTED`
  A connection was successfully established to the EyeX Engine.
- `TX_CONNECTIONSTATE_TRYINGTOCONNECT`
  The first connection attempt timed out and now repeated tries will be made until a connection is successfully established.
- `TX_CONNECTIONSTATE_SERVERVERSIONTOOLOW/`
  `TX_CONNECTIONSTATE_SERVERVERSIONTOOHIGH`
  The EyeX client API and the EyeX Engine versions are incompatible. No more attempts to connect to the EyeX Engine will be made, and there is no way of establishing a working connection without upgrading or downgrading to compatible versions.

If an established connection goes down, the connection state is changed to `TX_CONNECTIONSTATE_TRYINGTOCONNECT`. When a connection is re-established, the state is changed back to `TX_CONNECTIONSTATE_CONNECTED` again. A successfully enabled connection will be in either of these two states until it is disabled by the application.

Whenever the connection state changes, the registered connection state changed handler is called with information on what the state was changed to.

Responding to queries

The registered query handler will receive callbacks with queries from the EyeX Engine, asking the application to supply information about the area of the screen where the user is currently looking. The application can then handle the query step by step:

1. Extract the query bounds from the query
2. Create a snapshot
3. Add interactors that intersect with the query bounds to the snapshot
4. Commit the snapshot to the EyeX Engine

Extracting the query bounds

Extraction of the query bounds from the query is done in two steps. First `txGetQueryBounds` is used to get a handle to the query bounds property. Then the values for x, y, width and height can be retrieved using the `txGetRectangularBoundsData` function.

Creating a snapshot

In order to create the snapshot, the application has to decide which interactors to create and add to the snapshot.

First of all, a snapshot object needs to be created. When creating a snapshot as a response to a query a convenient function to use is `txCreateSnapshotForQuery`. This way a snapshot is created with the same bounds and windows id as the query. Another useful alternative is `txCreateSnapshotWithQueryBounds` that uses the same bounds as the query. If, for some reason the application needs to create a snapshot with different bounds, the `txCreateSnapshot` function can be used in combination with `txCreateSnapshotBounds`.

When adding an interactor to a snapshot, the application has to make sure the window ID of the interactor has also been added to the snapshot. If `txCreateSnapshotForQuery` has not been used, window IDs are added to snapshots using the function `txAddSnapshotWindowId`.

The window ID must be the window handle ("HWND") of the top-level window containing the interactor, formatted in a character string as a decimal number. The C function `itoa` is convenient for this—just remember to use 10 for the base.

Adding the interactors

The word interactor has so far been used interchangeably to mean either the conceptual interactor—a specific region on the screen possible to do eye interaction with, or, the actual interactor object that is created and added to a snapshot. In this section the distinction is important, so, in this section, *interactor* refers to the conceptual interactor, *interactor bounds* refers to the region on the screen bounding the interactor, and *interactor object* refers to the actual object to be created.

In order to decide which interactor objects to create, the application has to find all interactors which have interactor bounds that intersect with the query bounds. (That is, are at least partially within the query bounds). Note that the query bounds are always in screen coordinates, so, if the application keeps its interactor bounds information in client coordinates, these will have to be re-mapped before comparison.

To create a valid interactor object, the following is required: an interactor ID, a parent ID, a window ID, and bounds. In addition to this, one or more behaviors can be created and added to the interactor, and a z value can be set. See the section called *Interactor bounds and nested interactors* for more information about z values.

For every intersecting interactor, a corresponding interactor object is created and added to the snapshot. This is done in a number of steps:

The interactor object is created using the function `txCreateInteractor`, passing in an interactor ID that is unique within the context, the parent interactor ID, and the window ID. If the interactor is the child of another interactor, then its parent ID should be set to the interactor ID of its parent. Otherwise, the parent ID should be set to `TX_LITERAL_ROOTID`.

Last but not least, all the interactor's behaviors have to be created and added to the interactor object. For example, for an interactor that has the activatable behavior, first an activatable behavior options struct is created and populated with the necessary information, and then the function `txCreateActivatableBehavior` is called to add the behavior to the interactor object.

When all interactor objects have been added to the snapshot, it is time to commit.

Committing the snapshot

The snapshot is committed to the EyeX Engine using the function `txCommitSnapshotAsync`.

One of the arguments to this function is a callback function that is called with the result of the commit when the snapshot has been received and validated by the EyeX Engine. From the result handle, a `TX_SNAPSHOTRESULTCODE` can be extracted using the `txGetAsyncResultCode` function, indicating if the snapshot validation went well or if the snapshot was malformed.

This callback is mostly useful during development and of limited value in production builds. It is possible to pass in a null pointer if you do not wish to receive the callback.

Handling events

Now is the time to use those interactor IDs that you chose so carefully, because the interactor IDs provide the link between the interactor objects that you provided in the snapshots, and the events that the engine sends to you. Use the `txGetEventInteractorId` function to extract the interactor ID from an event.

The typical way of handling an event is to get all its behaviors using the `txGetEventBehaviors` function, iterating through the behaviors one by one, and checking the behavior type and parameters as you go using the `txGetBehaviorType` function. This is your only choice if you have an interactor with multiple behaviors of the same type but with different parameters, for example, an interactor with multiple "Gaze point data" behaviors that differ only in the choice of filtering.

If your application uses only one or a few behaviors, then it can be more convenient to retrieve them one by one from the events, using the `txGetEventBehavior` function; if the call succeeds, you have a handle to the behavior of the specified type, and if it doesn't you can move on to the next behavior type.

The content of a behavior can be extracted using behavior specific functions. For example, for the activatable behavior the useful ones are: `txGetActivatableEventType` and `txGetActivationFocusChangedEventParams`.

Getting state information and handling state changes

If the context is connected, it is possible to retrieve state information synchronously or asynchronously by using the `txGetState` or `txGetStateAsync` functions respectively. To get the correct state, a path needs to be provided. See a list of available paths in the StatePaths constant list. If a correct path is provided, a handle to a StateBag object is returned. To extract state values from the state bag, use the `txGetStateValueAsInteger`, `GetStateValueAsReal`, `GetStateValueAsString`, `txGetStateValueAsRectangle`, `txGetStateValueAsSize2` or `txGetStateValueAsVector2` depending on the type of the state value. The EyeX Engine API also has functions to set state values in a similar manner, but this is not recommended.

To be notified of state changes, client applications can set up callback functions using `txRegisterStateChangedHandler`. In the callback function, the corresponding state bag and state values can be extracted as above. It is also possible to register a state observer using `txRegisterStateObserver` and get message about state changes as messages that can be subscribed to using `txRegisterMessageHandler`, but this is a bit more complicated. The client application can (and should) stop listening for state changes when not needed by using `txUnregisterStateChangedHandler` or `txUnregisterStateObserver` respectively.

Note that the state paths are hierarchical, so if a client application for example has set up a state changed handler for the `TX_STATEPATH_EYETRACKING` state path, it will be notified also when sub-states change. In this case sub-states are states related to eye tracking, such as the eye tracker connection state and the screen size of the screen where the eye tracker is mounted. See the MinimalStatusNotifications sample in the SDK package to learn more about how to use states.

Tearing down a context

When you have finished using a context, use the `txShutdownContext` function to free up all the resources that the client library has allocated on behalf of the context. This function can also perform a check for leaked resources, that is, interaction objects that have not been released using `txReleaseObject`—highly recommended for debug builds. Resource leaks are reported to the log.

The function disconnects from the engine and waits for any callback functions to return before freeing up the resources. The calling thread is blocked until the context has been properly cleaned up. If `txShutdownContext` has completed without errors, you can run `txReleaseContext` to release the handle to the context.

Redistributing an EyeX client application

The installer for your application must include the client library dll file and install it along with your executable file. Note that you may *not* install the dll in a system directory, because that could potentially cause version incompatibilities with other EyeX client applications.

The Tobii EyeX SDK license agreement gives you the permission to redistribute the client library dll with your application, free of charge, in most cases. The exceptions include high risk use applications, applications that might inflict on a person's privacy, and certain other niche applications. Please see the license agreement for more details; it is available in the SDK package and it can also be downloaded from the Tobii Developer Zone.

The client library depends on the Microsoft Visual C run-time libraries, version 110, and will not work unless these libraries are installed on the computer. The run-time libraries can be downloaded free of charge from Microsoft. You can also include them as a merge module in your installer.

Using the C++ API

The C++ API is a header-only wrapper around the C API that offers the same functionality in a different format: there are classes for the interaction objects, as well as for some other key concepts.

The C++ header files can be found in the include/eyex-cpp directory in the SDK package.

What you must know about pixels

Points on the screen are always given in **physical pixels** in the EyeX Engine API. Unfortunately, this isn't as simple as it sounds. But in order to explain why, we need some background on how Windows handles multiple display monitors, and why a pixel in your application isn't always the same as a pixel on the screen.

Client and Screen coordinates

The coordinate system most commonly used by developers is the *client coordinate system*, which is relative to the top left corner of a window. The window can be your application's top level window, or it can be a child window such as a button. Each window has its own client coordinate system, which is used when laying out its contents.

There is also a coordinate system that relates to the pixels on the screen. That is, a system that is fixed relative to the screen and doesn't move around with a window. This coordinate system is called *screen coordinates* in Windows.

Mapping between screen coordinates and client coordinates is performed using the `ClientToScreen` and `ScreenToClient` functions (or the `MapWindowsPoints` function if your application supports right-to-left mirroring). This is standard procedure.

This means that—

- If you received, for example, the user's gaze point from the EyeX Engine, and you want to get its position relative to your application's top level window, then you should use the `ScreenToClient` function to map the coordinates to the client coordinate system of your window.
- When preparing a snapshot for the EyeX Engine that includes region-bound interactors corresponding to user interface elements, and you know the bounds of those user interface elements in client coordinates, then you should use the `ClientToScreen` function to map them to screen coordinates.

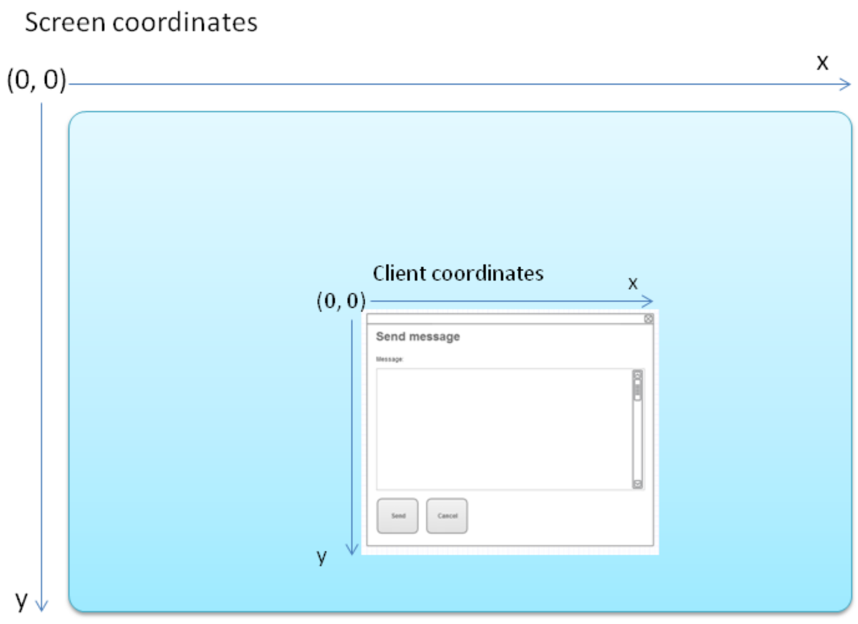


Figure 2 Screen vs client coordinates

Multiple display monitors

In a system with multiple display monitors there is still a single screen coordinate system that spans all of the monitors. This is referred to as the *virtual screen*. Pixel coordinates on the virtual screen can be negative because the origin (0, 0) is always on the primary monitor. So if you place another monitor to the left of the primary, then it will have negative x coordinates.

If you are using multiple screens, you can place the eye tracker on any screen you want. Once you have set up the EyeX Engine to use a particular screen, it will make sure that the eye tracker coordinates are mapped properly.

DPI scaling and DPI awareness

The **DPI scaling**[2] feature, also known as DPI virtualization, was first introduced with Windows Vista. It enables the user to set the screen DPI to a value larger than 100%, making everything appear larger on screen. This can be useful with very large monitors, with high-dpi monitors, or for users who have impaired eyesight.

Applications may indicate to Windows whether they take care of the scaling themselves or not, that is, if they are **DPI aware** or not. If they don't declare anything, then Windows will assume that the application does not handle scaling, and will scale the window automatically if the DPI setting is above 100%. Windows performs the scaling by rendering the application's window to an intermediate buffer instead of to the screen, and then drawing a magnified screen image from the buffer.

Windows also has some clever tricks to protect the application from having to know anything about the scaling being performed. Unfortunately, these clever tricks break the transformation between

---

[2] DPI is an abbreviation for Dots Per Inch, a unit often used when stating the resolution of a screen or a printer.

client coordinates and screen coordinates, so that the coordinates reported by the application to the EyeX Engine will be wrong. **Therefore all EyeX client applications should be DPI aware.**

An application which is DPI aware will always be able to respond correctly to the EyeX Engine's queries and events, and as a side effect you also get a better looking user interface without scaling artifacts.

Note that some GUI frameworks, such as the Windows Presentation Foundation, always declare DPI awareness, so you might not have to do it yourself. If you are uncertain if this is the case, it is easily tested: set your monitor DPI setting to "Larger" and run your application full-screen. Any offsets should be easily noticed.

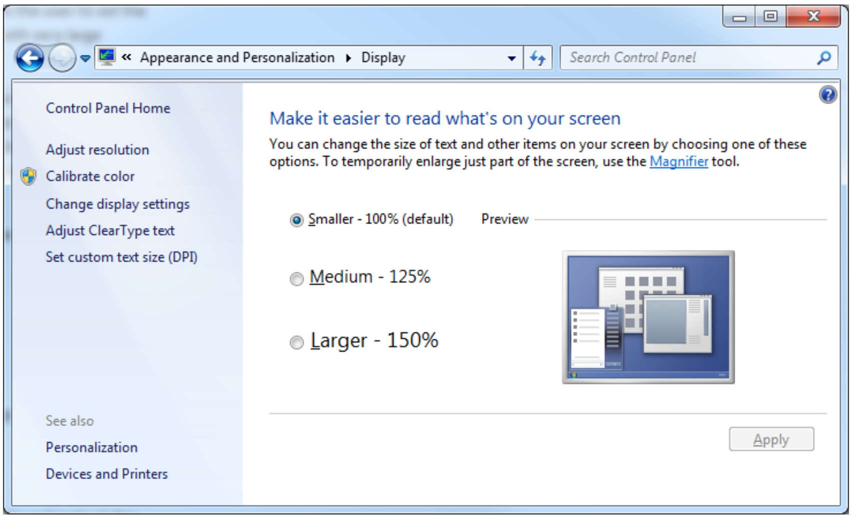


Figure 3 The Display settings page in the Control Panel, where the user can change the DPI setting.

Windows 8.1 takes the concept of DPI awareness and scaling even further, by introducing a per-monitor DPI setting. This is not yet fully supported.

For more information about screen coordinates, multi-monitor setups, and DPI awareness, see

- Window Layout and Mirroring on MSDN
- Multiple Display Monitors on MSDN
- Writing DPI-Aware Desktop and Win32 Applications on MSDN

EyeX Engine API reference

The EyeX Engine API reference explains all the concepts in the API and how the EyeX Engine and a client application will work together to create an Eye Experience. The concepts in the API are language, platform and GUI framework agnostic.

The Data Streams, States, and Behaviors for region-bound interactors are described at the end of the API reference. The reference has been written so that it can be read through top-down with later sections referring to previous sections.

The Client application

Whenever you find the term **client application**, or just application, in this document, it refers to *your* application. Or to one of the sample applications, or any other application that makes use of the services provided by the EyeX Engine—which plays the part of the server in this instantiation of the classic Client/Server architecture.

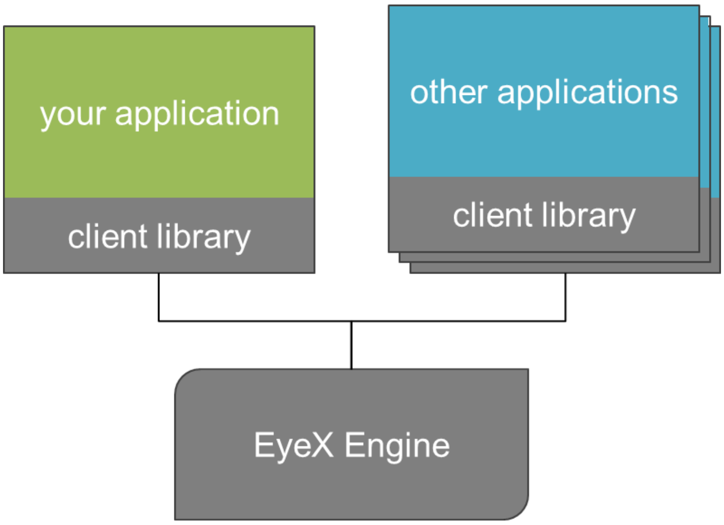


Figure 4 Your application and its relationship to the EyeX Engine.

Something that all client applications have in common is that they use a client library provided with the SDK to connect to the engine.

Interactors

Anything that the user can interact with using eye-gaze is called an **interactor** in the language of the EyeX Engine. For example, an interactor can be an activatable (clickable) button, or a widget that is expanded when the user's gaze falls within its bounds. An interactor can also be a stream of filtered data where the user interacts by moving the head or by appearing in front of the screen after being away from it.

Some modes of eye-gaze interaction take place within a particular region on the screen, as in the case of the button and widget examples above. Other eye-gaze interaction modes are not tied to any particular region of the screen, as in the case of a stream of eye-gaze data. For example, an information kiosk application or ATM could use an eye-gaze data stream to sense that a user has appeared in front of it, and switch its user interface into a different mode at that point.

The EyeX Engine treats the interactors which are used for eye-gaze interaction within a particular screen region quite differently from those interactors that are not. To clearly distinguish the one kind from the other, we will refer to the former kind as **region-bound interactors**, and the latter kind as **global interactors**. Both will be described in more detail below.

Region-bound interactors

Region-bound interactors usually map one-to-one with the visual elements/components in the GUI framework used to create the application. This is by convenience and not a requirement: it makes sense, because it is it easier to maintain the relations between the interactors, and because end users expect objects to respect visual hierarchies and window bounds.

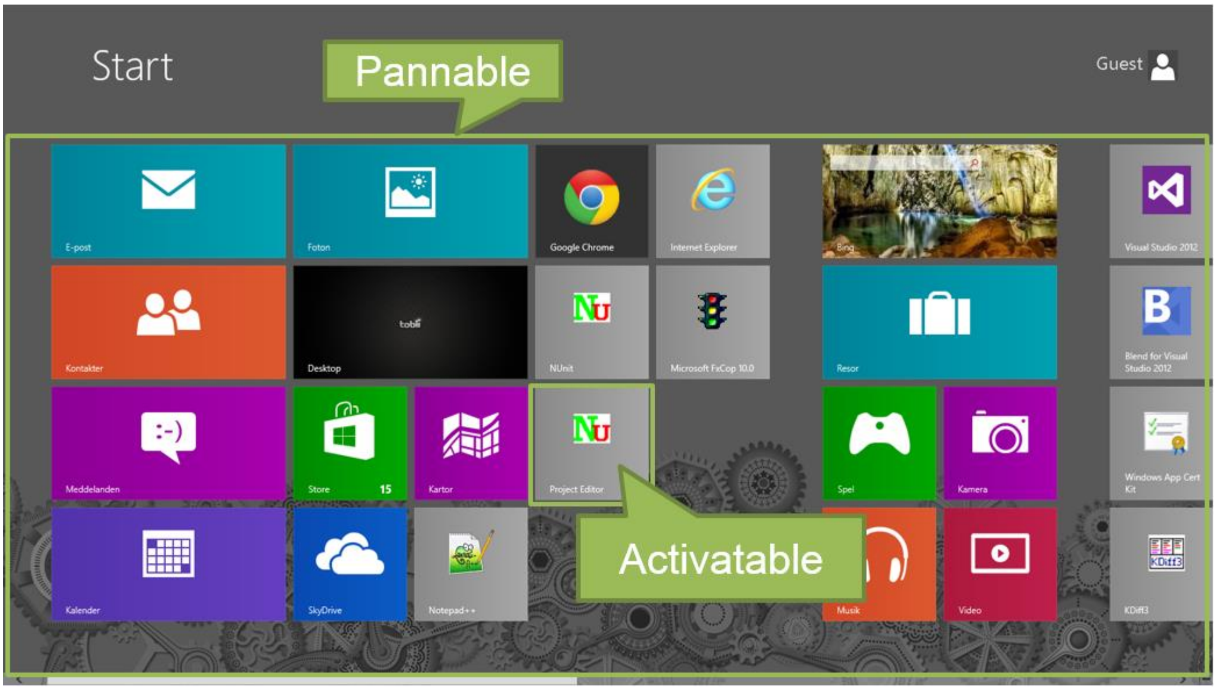


Figure 5 Examples of region-bound interactors

The EyeX Engine considers all region-bound interactors to be transient. The engine will continuously query the application for region-bound interactors based on the user's gaze point. It will remember the interactors long enough to decide what interaction is currently going on, but then it will discard the information. As the user's gaze point moves to a new region of interest, new queries are sent to the application and a new batch of interactors are sent back to the engine. And so on.

Non-rectangular interactors

By default, a region-bound interactor has a rectangular shape. To define non-rectangular shapes, you need to define a **weighted stencil mask** (or just **mask**) on the interactor. A weighted stencil mask is a bitmap that spans the area of the interactor. The interactable parts of the area are represented by non-zero values in the bitmap. The rest of the area is considered transparent, and cannot be interacted with. The resolution of the bitmap does not need to be as high as the screen resolution. Usually, a low-resolution bitmap works just as well and is better from a performance point of view.

Global interactors

Global interactors are used for subscribing to **data streams** that aren't associated with any specific part of the screen.

Once you have told the EyeX Engine about a global interactor, the engine will remember it as long as the connection with your application remains, or until you explicitly tell the engine to remove the interactor. So, while region-bound interactors are committed continuously in response to queries from the engine, a global interactor is usually only committed once per established connection.

A common usage pattern is to set up a global interactor when the application starts, and to send it to the engine as soon as the connection is established, or re-established—for example, due to a switch of users. (The EyeX Engine restarts automatically every time Windows switches users.)

Interactor ID's

The one thing that makes the EyeX Engine recognize an interactor, regardless of how it moves around and how its behaviors change, is the interactor's ID. It is your responsibility, as a developer, to ensure that all interactor IDs are indeed unique—at least within their respective contexts, as described below.

The interactor IDs can be any string values, and since almost anything can be converted to a string, that leaves you with plenty of options. So, what does a good interactor ID look like?

In the rather common case when an interactor maps directly to a user interface component, and that component already carries a sufficiently unique ID, it's good practice to let the interactor ID match the component ID. Not only will that give you reliable, unique, constant ID's; it will also simplify the mapping between interactors and components.

In other cases there are no clear-cut guidelines. Just try to choose ID's that make sense in your domain.

Interaction Behaviors

An interaction behavior, or **behavior** for short, is a particular mode of eye-gaze interaction that an interactor may provide. The catalog of behaviors is by far the most important part of the EyeX Engine API, because each behavior represents a piece of functionality that your application can use. The available behaviors are described in more detail later in this document.

Some behaviors are intrinsically region-bound, and some are not. It is really the behaviors that determine whether an interactor should be region-bound or global.

So, how much use would an interactor be if it didn't have any behaviors? Actually, there is a case where behavior-less interactors are indeed quite useful. A region-bound interactor without behaviors is effectively just preventing eye-gaze interaction on the part of the screen that it covers, and is commonly called an *occluder*.

The EyeX Engine adds occluders representing all top-level windows[3] automatically, in order to prevent any parts of a window which are covered by other windows to take part in eye-gaze interaction. The interactors defined by your application will be considered as children to the top-level window interactors.

[3] "Top-level window" is a Windows term for a window that doesn't belong to another window. Top-level windows are typically displayed on the taskbar. Applications typically display one top-level window for the application itself, or one top-level window for each document opened in the application.

The Query-Snapshot cycle

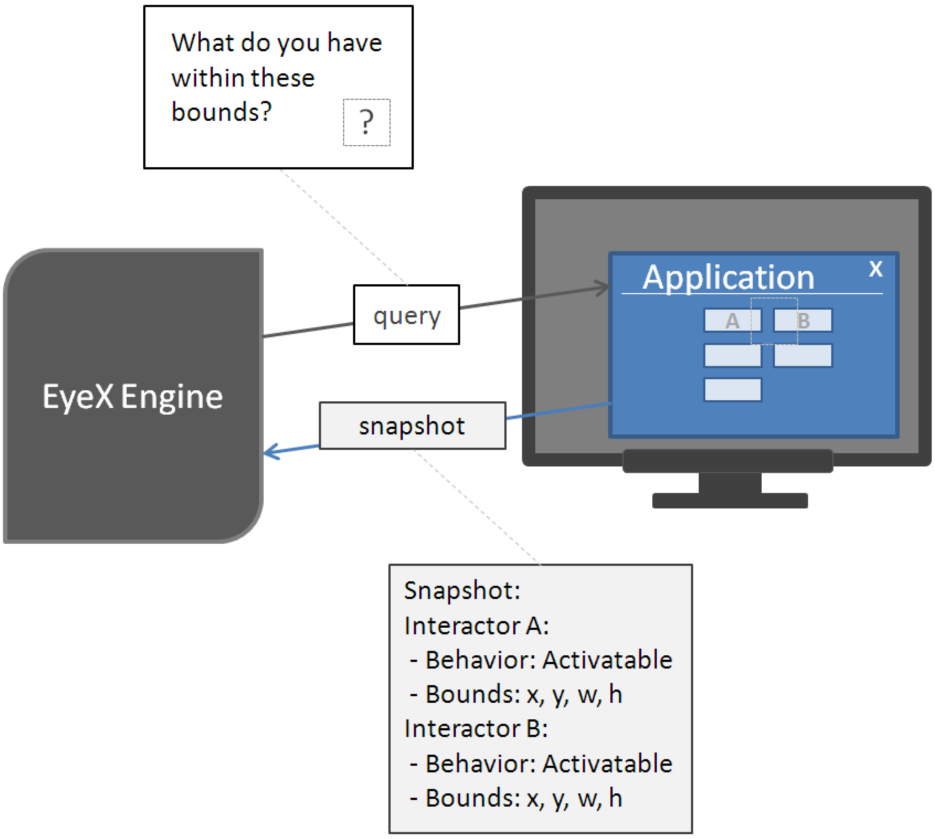


Figure 6 The Query-Snapshot cycle.

A key design principle of the EyeX Engine is that it senses what is on the screen one piece at a time, by making **queries** to the client applications. That is, it does *not* expect the applications to declare their entire gaze enabled user interface up front, but rather to feed the engine with information continuously, on request.

Note that this design principle doesn't prevent an application from keeping a repository, or cache, of its region-bound interactors, and respond to the engine's queries with cached information. Whether or not to use a repository is a design decision left to the application developer.

The queries roughly follow the user's gaze point. Queries are always specified with bounds, that is, a rectangular region on the screen, and with one or more window IDs. In areas on the screen where windows from different client applications are close, the query is sent to all applications, and each is responsible for keeping the engine updated on the region-bound interactors within its window(s). The window IDs in the queries identify the top-level window(s) that the engine wishes to receive interactor information for.

The packet of data that the client sends in response to a query is called an interaction snapshot, or **snapshot** for short. It contains the set of region-bound interactors that are at least partially within the query bounds, a timestamp, and the ID of the window that the snapshot concerns.

There is no one-to-one correspondence between queries and snapshots. If an application doesn't respond in a timely fashion, then the engine will simply assume that it didn't have any region-bound interactors to report—which may or may not be what the application intended.

An application may also act proactively and send the engine snapshots that it didn't ask for. This is how applications usually inform the engine of its global interactors. Animated interactors whose screen bounds change over time is another case where application-initiated updates can be useful.

The information in a snapshot should be considered as the *complete* description of all region-bound interactors within the snapshot bounds. If two snapshots with the same bounds are committed after another, and the first committed snapshot contains an interactor that is not included in the second snapshot, the engine will interpret this as if that interactor has been removed. As a consequence: don't stop responding to queries when your last interactor has gone off-stage—instead, keep sending empty snapshots so that the engine will know that they are gone.

The exception to this guideline is an application that doesn't use any region-bound interactors at all. Such an application doesn't even have to handle queries, because the global interactors are handled separately, as described in the section on global interactors above.

Events

As soon as the EyeX Engine has found that a particular kind of gaze interaction is taking place between the user and an interactor, it notifies the application that owns the interactor by sending it an **event**.

Events are used for both region-bound and global interactors, so an application should always be set up to receive and handle events from the engine.

Events are tagged with the ID of the interactor and the behavior(s) that triggered the event. Some events also include additional, behavior-specific parameters related to the interaction.

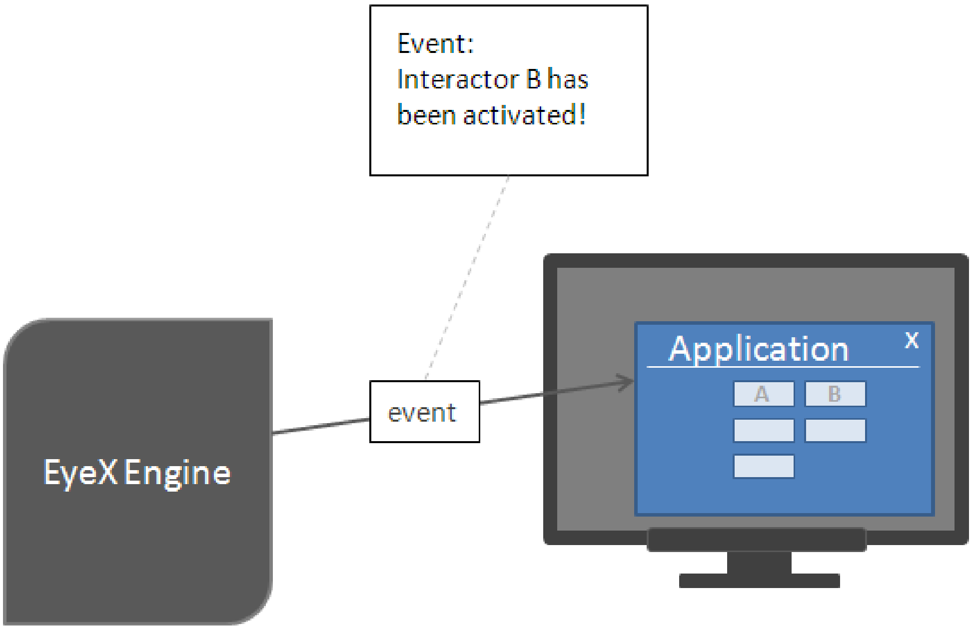


Figure 7 Event notification on a region-bound interactor

Interactor bounds and nested interactors

A region-bound interactor is always associated with a region, as the name implies. This region is called the interactor's **bounds**, and is currently defined as an axis-aligned rectangle on the screen. (If a weighted stencil mask is applied on the interactor, non-rectangular shapes can also be defined).

A region-bound interactor is also associated with a top-level window, and its bounds cannot extend outside the window—or, rather, its bounds will be clipped to the bounds of the window. This might seem like a severe restriction at first, but do remember that it applies to region-bound interactors only—the global interactors by definition do not have this restriction.

User interfaces are typically built as tree structures: starting from the window, there are layout containers, scroll containers, etc, until we arrive at the actual content that is visible on the screen. The contents are often only visible in part, such as in the case of a long, scrollable list where only a few items can be seen at any time, or when another window is covering part of the view. Users typically expect the invisible parts to be excluded from interaction.

Using only the bounds information, all region-bound interactors would appear to lay flat next to each other. Suppose two of them were overlapping, which one should the engine pick as the candidate for eye-gaze interaction? Instead of forcing the application developer to avoid overlaps by adjusting the interactor bounds, the API provides **nested interactors** to help out. The engine will consider child interactors to be in front of all its ancestor interactors.

Region-bound interactors can be organized in a tree structure just like the user interface components. Each interactor provides the ID of its parent if it has one, or otherwise a special ID representing the top-level window. Interactors that are children of the same parent interactor should specify a Z order (highest on top) if they overlap. Specifying parent-child relationships like this can be thought of as nesting interactors inside each other.

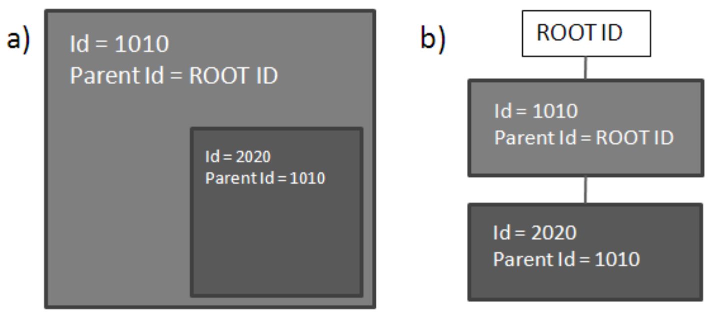


Figure 8 a) Nestled interactors, where the child interactor is overlapping its parent interactor. b) The corresponding interactor tree-structure.

The bounds of a child interactor may extend outside the bounds of its parent. Windows makes a distinction between *child windows* and *owned windows*, and a child interactor is more like an owned window than a child window in this sense.

When the EyeX Engine scans the area around the user's gaze point for interactors, it starts by determining which top-level windows there are in the region. Then it proceeds to search through the interactors attached to those windows, looking for interactors whose bounds contain or are close to the gaze point. During this process the engine makes use of both the parent-child relationships and the Z order information to decide what is on top of what.

The Z order is only compared between sibling interactors, and a sibling with a higher Z order will be considered to be in front of not only its sibling with lower Z order, but also to all children interactors of these siblings. Because of this, one has to be careful when constructing an interactor tree-structure so that the interactors overlap as intended. This is illustrated in Figure 9 and Figure 10.

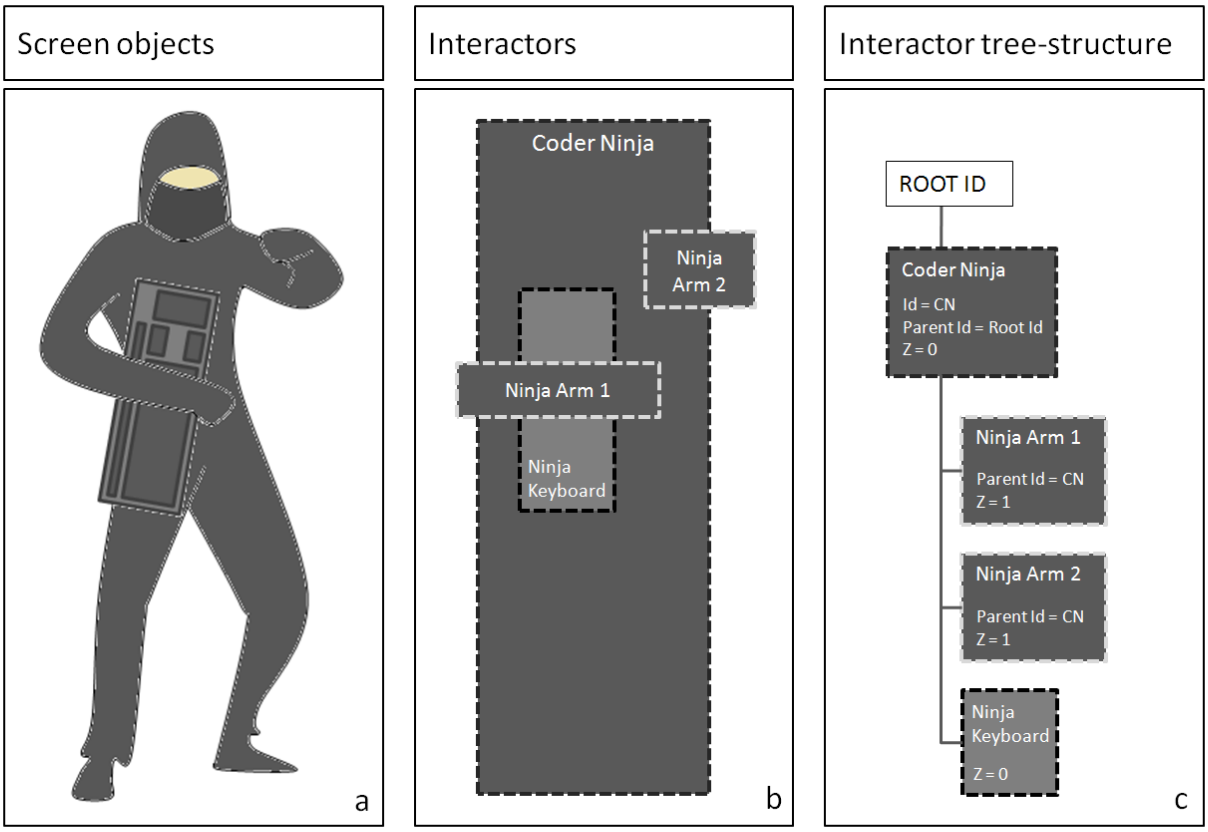


Figure 9 a) Coder Ninja with Ninja Keyboard b) Corresponding overlapping interactors. c) By making the Ninja Arm 1 and the Ninja Keyboard children of the Coder Ninja, but with different Z order, the EyeX Engine is told that they are both in front of the Coder Ninja, but that the arm is in front of the keyboard. The Z order of the other arm does not matter, since it does not overlap any of its siblings.

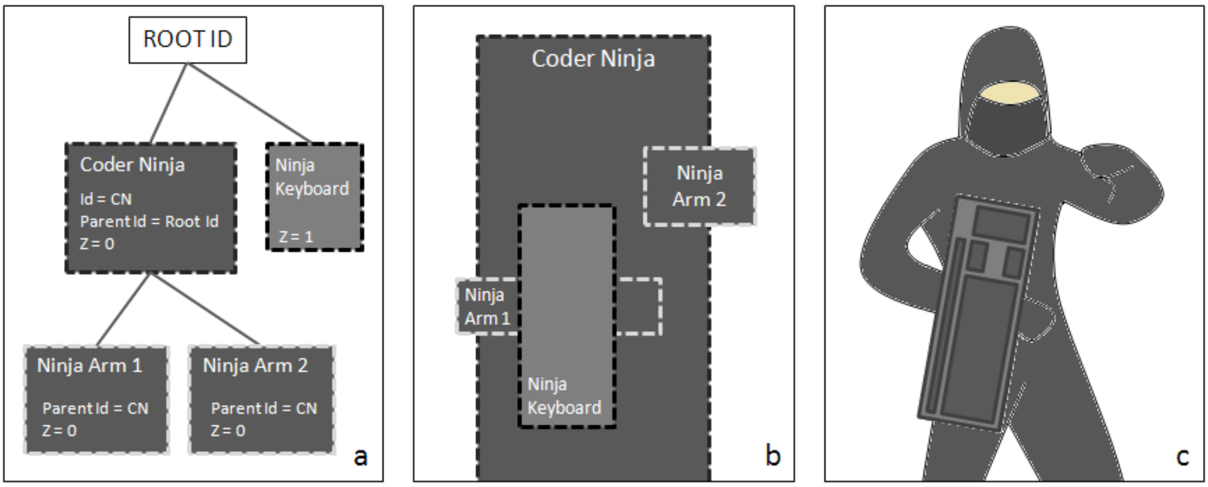


Figure 10 Making the Ninja Keyboard a higher Z order sibling of the Coder Ninja (a), would not only put it in front of the Coder Ninja but also all of Coder Ninja's child interactors, including the Ninja Arm 1 (b and c).

When the engine has identified the topmost interactor that is closest to the gaze point (if any) it looks at the interactor's behaviors to see what kind of user input it should expect and how to react on it.

The existence of nested interactors has some consequences for the application developer when preparing a snapshot:

- If an interactor in a snapshot references another interactor as its parent, then the parent interactor must also be included in the snapshot, even if it isn't within the snapshot (or query) bounds.
- If two interactors with overlapping bounds have the same parent interactor and the same Z order, then the EyeX Engine cannot decide which one is actually on top. The outcome will be random and the user experience inconsistent. So, ensure you are careful when defining the bounds and relationships of your interactors.

Contexts

A **context** represents a connection between an EyeX client application and the EyeX Engine. Applications typically create a context during startup and delete it on shutdown.

The application uses the context to register query and event handlers, and also to create **interaction objects** such as snapshots. Queries and events are also interaction objects, but they are normally not created by the application. An interaction object always belongs to a certain context, and interaction objects cannot be shared between contexts.

3D Coordinate systems

The coordinate system used for 3D points in the EyeX Engine, for example for the Eye position data stream, is relative to the screen where the eye tracker is mounted. The origin is at the center of the screen. The x axis extends to the right (as seen by the user) and the y axis upwards, both in the same plane as the display screen. The z axis extends towards the user, orthogonal to the screen plane. The units are measured in millimeters.

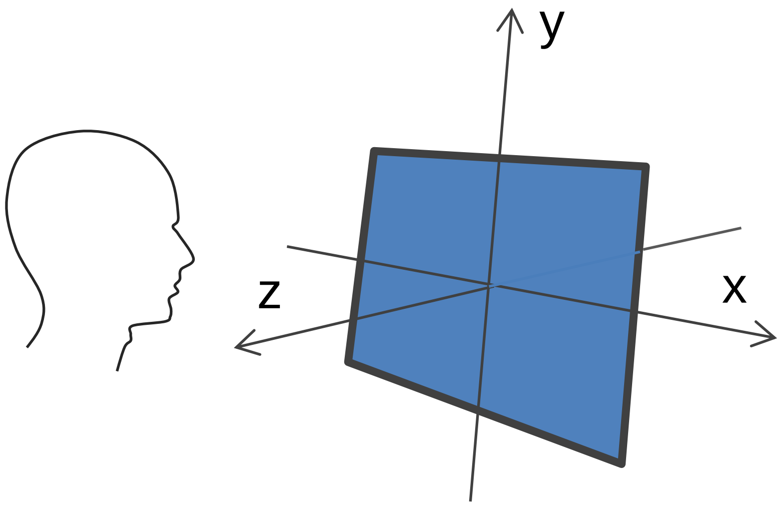


Figure 11 The 3D coordinate system used by the EyeX Engine.

Note that the y axes in the 3D coordinate system and the 2D coordinate system are different. The y axis in the 2D coordinate system crosses zero at the top of the screen and extends downwards; the y axis in the 3D coordinate system crosses zero at the middle of the screen and extends upwards.

The 3D coordinates can also be expressed relative to the *track box*, i.e. the volume in which the eye tracker is theoretically able to track the user's eyes. The track box coordinate system (TBCS) has its origin in the top, right corner located closest to the eye tracker. The TBCS is a normalized coordinate system: the location of the (1, 1, 1) point is the lower, left corner furthest away from the eye tracker.

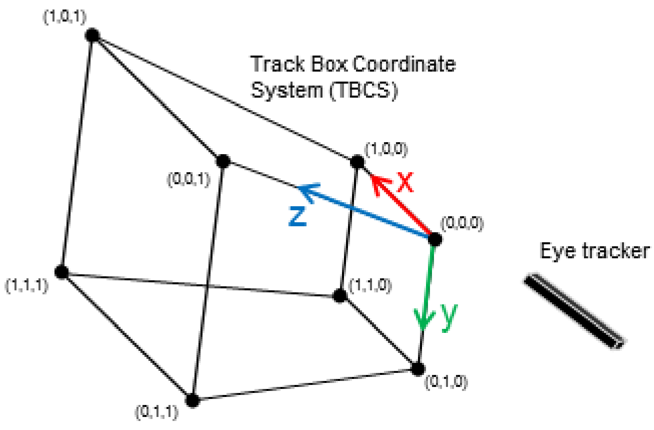


Figure 12 The track box coordinate system.

Data streams

From an EyeX Engine API point of view, data streams are just kinds of behaviors assigned to interactors. The typical way to setup a data stream is to create a global interactor, assign to it one of the data stream behaviors and then subscribe to the events raised for the interactor. Since the interactor is global, the engine will send data events no matter where the user is looking, as long as there is valid data to be sent.

The EyeX Engine also allows for assigning data stream behaviors to a region-bound interactor. In this case, data events will only be sent when the user is looking at the interactor. This could be used on a window level so that a client application only receives data if the user is looking at the client application's window.

Region-bound interactors with a data stream behavior need to be continuously sent to the EyeX Engine using the Query-Snapshot cycle (see *The Query-Snapshot cycle*). This means that these types of data streams tend to be more performance heavy than the straight forward global interactor data streams which only have to be sent once per established connection between the client application and the EyeX Engine. (Read more about the life cycle of global interactors in the section *Global interactors* above.)

Each data stream delivers one kind of data, for example the user's gaze point, and often comes in variants that differ for example in the choice of filtering.

Gaze point data

The Gaze point data behavior provides the user's gaze point on the screen as a data stream. The unfiltered data stream produces a new data point whenever the engine receives a valid eye-gaze data point from the eye tracker. No statements are made regarding the frame rate; it is what it is.

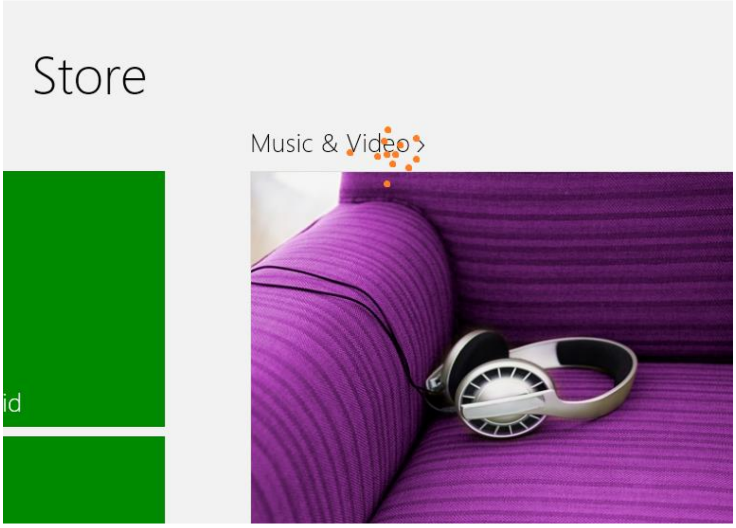


Figure 13 The gaze point is an inherently noisy signal. The orange dots on this screenshot represent the user's gaze point during a fraction of a second. Filtering makes the point cloud shrink towards its center, but also respond slower to rapid eye movements.

The gaze point is given as a single point. If the user has chosen to track a specific eye, then it's the gaze point from that eye. Otherwise the point is taken to be the average from both eyes.

Because the gaze point is an intrinsically noisy signal, the Gaze point data behavior provides a selection of filters that can be used to stabilize the signal. As usual when it comes to filtering, there is a trade-off between stability and responsiveness, so there cannot be a single filter that is the best choice for all applications. The choice of filters are:

- Unfiltered: no filtering performed by the engine. (Except for the removal of invalid data points and the averaging of the gaze points from both eyes.)
- Lightly filtered: an adaptive filter which is weighted based on both the age of the data points and the velocity of the eye movements. This filter is designed to remove noise while at the same time being responsive to quick eye movements.

Note: We expect that more filters will be added in later releases of the engine

The gaze point is given in physical pixels, see section *What you must know about pixels*.

Eye position data

The Eye position data behavior provides the user's eye positions in three-dimensional space as a data stream. This data stream can be used, for example, to control the camera perspective in a 3D game.

This data stream produces a new value whenever the engine receives a valid sample from the eye tracker, and no statements are made about the frame rate, just as for the Gaze point data behavior. The eye positions are given for the left and right eyes individually. See the section *3D Coordinate system* for a description of the coordinate system used.

The Eye position data behavior does not offer any filtering options at this time.

Fixation data

The Fixation data behavior provides information about when the user is fixating her eyes at a single location. This data stream can be used to get an understanding of where the user's attention is. In most cases, when a person is fixating at something for a long time, this means that the person's brain is processing the information at the point of fixation. If you want to know more about fixations, the Internet is your friend.

To get information about the length of each fixation, the Fixation data behavior provides **start time** and **end time** for each fixation, in addition to the **x and y point** of each individual gaze point within the fixations. Each fixation correspond to a series of fixation events: Begin, Data, Data, … , End.

When setting up the fixation data behavior, these fixation data modes are available:

- **Sensitive**, will result in many fixations, sometimes very close and in quick succession.
- **Slow**, will result in fairly stable fixations but may leave somewhat late.

States

The EyeX Engine keeps track on a number of **states** that are related to the current status of the eye tracking system, like configuration and user profiles, or to the user in relation to the eye tracking system, like user presence. Below is a list of the common states with a short description what status it tracks. Each state has a unique path that a client application can use to retrieve the information if needed. It is also possible to setup state changed handlers to get notified when a state changes.

User Presence
If a user is present before the eye tracker or not.

Gaze Tracking
If the user's eye-gaze is currently tracked or not. If none of the user's eyes can be tracked, there will be no events sent in the gaze point data stream.

User Profile Name
The name of the currently selected user calibration profile.

User Profiles
A list of names of all the available user calibration profiles.

Screen Bounds
The screen currently set up for eye tracking defined as an area on the virtual screen. Given as x,y coordinates of the upper left corner and the width and height, in pixels on the virtual screen.

Display Size
The physical size of the screen currently set up for eye tracking. Given as the width and height in millimeters.

Eye Tracking Device Status
Current status of the eye tracking device. This state will indicate if the EyeX system is ready to track your eyes (the "Tracking" state), or if there is something stopping it from doing that, for example: a missing user profile or screen setup, that the user has disabled eye tracking in the settings ("TrackingPaused"), or that no eye tracker device seem to be connected to the computer.

Behaviors for region-bound interactors

Behaviors for region-bound interactors are naturally associated with a region or an object on the screen. Or, to be more precise, with a region or object *in a window* on a screen, because region-bound interactors must always be associated with a window.

The behaviors for region-bound interactors either let the user perform an action on the object/region, such as activation (direct click or action), or provide some sort of monitoring of the user's eye-gaze on the object or region.

Gaze-aware behavior

An interactor with the Gaze-aware behavior represents a region or object on the screen that is sensitive to the user's eye-gaze. Possible uses of the behavior include widgets that expand on gaze, game characters that act differently when being watched, and other usages where the user interface adapts to what the user is looking at, or implicitly, is paying attention to.

The EyeX Engine raises one event when the user's gaze point enters the bounds of the interactor, and another event when it leaves. The event parameter "HasGaze" will be set to true or false according to if the engine considers the user looking or not looking at the region.

Add inertia by setting delay time

Note that the fact that the user is looking at something doesn't necessarily mean that she is paying attention to it, and also that this behavior is quite sensitive and can easily be triggered by noise and/or rapid eye movements. A common way of dealing with this uncertainty is to add some inertia to the interaction: make sure that the gaze point stays on the interactor for a while until the response is triggered, and don't release the trigger until the gaze point has been off for a while.

For a simple way to add inertia, there is a built-in delay parameter that can be set on a gaze-aware interactor to specify a delay between when the user's gaze point enters the interactor bounds and when the event is raised.

Nested Gaze-aware interactors

If the gaze-aware interactor has child interactors (see *Interactor bounds and nested interactors*) that also have the gaze-aware behavior, the gaze point will be considered to be within the parent interactor as long as it is within an unbroken hierarchy of gaze-aware child interactors. This applies even if the gaze point isn't within the bounds of the parent.

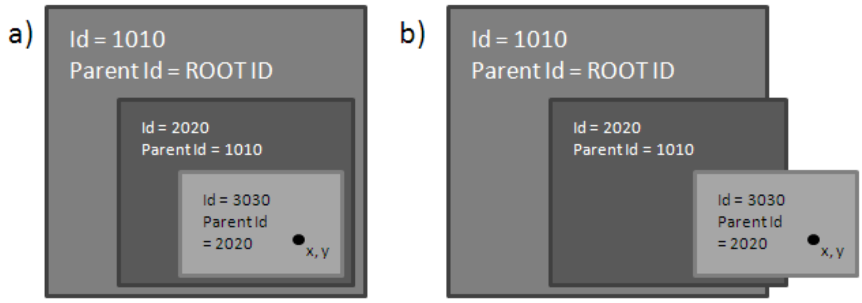


Figure 14 Nested Gaze-aware interactors: The gaze point at (x, y) is considered to be within the bounds of the parent interactors, no matter if it is a) geometrically within or b) geometrically outside the parent interactors' bounds. Moving the gaze point between child interactors does not trigger any additional events for the parent interactor.

Activatable behavior

An interactor which has the activatable behavior represents an action that the user can select and trigger using her eye-gaze. The actual triggering is usually performed using another input modality—because triggering actions entirely using eye-gaze isn't comfortable to most people.

The activation can be thought of as a mouse click, a touch tap, or the pressing of a button. It is up to the application developer to decide what happens on activation. Common usages include selecting an item from a menu, executing a command (for example, launching an application), navigating to a web page, flipping pages, firing the lasers, and so on.

To implement the activatable behavior for an interactor, the client application sends a number of *action commands* to the EyeX Engine and reacts to the *events* the EyeX Engine sends in return. In short: the client application informs the engine of the steps of the interaction the user is doing, the engine informs the client application which interactor the user is trying to interact with.

Activation event 'Activated'

There are two kinds of events associated with the activatable behavior. The most central event for the behavior is the *activated* event. This event is sent for the interactor that the user has looked at while triggering an activate command, for example by pressing a specific key on the keyboard. The *activated* event is the event the application should respond to by performing the action associated with the activatable interactor.

Activation event 'Activation Focus Changed'

To be able to correctly and quickly decide which activatable interactor the user is trying to activate, the EyeX Engine continuously keeps track of which interactor is in focus. Only one interactor can be in focus at a given time. There are two levels of focus: *tentative activation focus* and *activation focus*. The current state of these focuses are set as two separate parameters in the *activation focus changed event*.

While the user is just looking around, not pressing any activation keys or anything, the activatable interactor the user is looking at will have the so called *tentative activation focus*. If the user is not looking at any activatable interactor, then no interactor will have the tentative activation focus at that time. By default, the EyeX Engine will not send any events related to this kind of focus, but it is possible to set a parameter on the activatable behavior so that the interactor will receive events whenever the tentative activation focus changes. Please note that it might be costly for performance to have events fired continuously by the EyeX Engine when the tentative activation focus changes. In a future version of the engine we might introduce power states, where tentative activation focus changed events are not available in low power modes. That said, our current recommendation is to not have an interaction depend solely on tentative activation focus events.

When the user is just about to activate an activatable interactor using an activation button or command, the EyeX Engine enters a mode called activation mode. In this mode, the engine uses even more refined heuristics to decide which activatable interactor is truly in focus and about to be activated. In this mode, the interactor that the user is looking at will have *activation focus*. It will no longer have tentative activation focus. There will always be at most one interactor in focus, either with tentative activation focus or activation focus. The EyeX Engine enforces this rule across all applications, so that several applications that uses EyeX Interaction can run in windows next to each other without confusing the user with multiple focuses/highlights. By default, activatable interactors will always receive *activation focus changed* events when they get or lose the activation focus. This means that these focus changes are suitable to use for visualization of for example pressing or releasing an activatable button.

Action command 'Activate'
The action command corresponding to the *activated* event is the *activate* command. It can be used on its own or together with the *activation mode on/off* commands to implement an interaction involving activatable regions.

When the EyeX Engine receives an *activate* command it will decide which interactor is looked at and send an *activated* event for that interactor. As soon as the command is received and handled in the engine, the engine will switch the activation mode off.

Action command 'Activation Mode On'
In order for interactors to get the *activation focus* described above (see *Activation event 'Activation Focus Changed'*), the EyeX Engine first has to be in the *activation mode*. This mode is switched on using the *activation mode on* command. There is a caveat, though: the engine can only be in one mode simultaneously and some modes override others. The panning mode (described in section *Pannable behavior*) overrides the activation mode, but the activation mode cannot override the panning mode. If the EyeX Engine is in panning mode and receives an *activation mode on* command, it will ignore it. This means that it is not possible to get activation focus on an interactor while panning is ongoing. If this was not the case, the result could be distracting highlights on items one is just scrolling past. If the EyeX Engine on the other hand receives a *panning begin* command when already in activation mode, it will switch to panning mode, and then switch back to activation mode after *panning end* has been received and handled in the engine.

The recommended way to use the activation mode with activatable buttons[4] is to switch it on when the user is just about to activate the button, and then use the *activation focus true* event to trigger a visual feedback which button is about to be clicked (for example by making it look pressed in the GUI). If the activation is triggered by pressing a physical key on the keyboard, a good time to send the *activation mode on* command would be when the physical button is pressed down. Then, the *activate* command could be sent when the physical button is released.

Action command 'Activation Mode Off'
When the *activation mode on* command has been received by the EyeX Engine, the activation mode will stay on until the engine receives either an *activation mode off* or an *activate* command. This means that the client application typically never has to send an *activation mode off* command.

All action commands are global
All action commands are global, meaning that they apply to the EyeX Engine as a whole. This means that an activation command sent by your client application can trigger an activation of an activatable interactor in another client application. Since you most likely only will send an activation command when your client application window has input focus, it is most probable that your users will only focus on items in your application when invoking activation commands through it. But bear this possibility in mind when designing your application and its action command handling algorithms.

Design and visual feedback
The way you design your application can have a huge effect on the usefulness of the activatable behavior. Here are some guidelines to help you make the best use of this interaction concept:

[4] These recommendations are specific for clickable buttons and might not be suitable for other types of activatable regions and interactions involving the activatable behavior.

- Give the user something to focus on: a visual hotspot. This can be as simple as the caption on a button. Sometimes there are several visual hotspots on an interactor, for example, an icon and some text. That's fine too.
- Make sure that the visual hotspots of different interactors are sufficiently separated. For example, add more spacing around the visual elements, and/or make them larger. Note that spacing can be more effective than size.
- Be careful with any visual feedback given; it can be helpful but it can also be distracting. For example, instead of highlighting a whole button, you can highlight only the text or the visual hotspot of the button.

Pannable behavior

An interactor with the Pannable behavior represents a region on the screen that can be panned or scrolled using eye-gaze and a secondary input. For example: a reading pane could be scrolled vertically upwards or downwards as long the user holds down a specific keyboard key and looks at the top or bottom of the reading pane.

To implement the pannable behavior for a region, the client application sends *action commands* to the EyeX Engine and reacts to the *events* the EyeX Engine sends in return. In short: the client application informs the engine that a panning/scrolling interaction is ongoing, the engine informs the client application how to adapt the panning/scrolling speed and direction depending on where the user is looking.

Panning profiles and available panning directions

There are two parameters on the pannable behavior that specify the kind of panning it will generate: the panning profile and the available panning directions.

The panning profile decides what velocities to trigger when the user is looking at different parts of the pannable area. These velocities and trigger areas are optimized for the particular kind of panning or scrolling that the profile corresponds to. For example, the vertical panning profile uses velocities and trigger areas optimized for vertical scrolling, like scrolling a web page, where looking at the upper part of the pannable area will trigger upward velocities, while looking at the lower part will trigger downward velocities. The radial panning profile is optimized for panning in all directions, like panning around a map, and triggers velocities in any direction the user is looking.

In addition to the panning profile, there is also a parameter to set the available panning directions. This might at first seem like redundant information: if I have set a vertical panning profile, then the available panning directions should be up and down, right? Yes and no. The panning profile decides how the panning should work in general and limits the available panning directions according to the velocity profile. But even though a pannable area in general should be possible to scroll both up and down, it might not always be in a state where both directions are available. This is where the parameter for available panning directions comes in handy: it can be used to dynamically keep the set of available directions up-to-date with the current state of the pannable area. For example, it can be used to temporarily remove the up direction while a vertically scrollable area displays its topmost contents and cannot be scrolled up.

Setting the available panning directions only affects the panning behavior if it restricts the number of directions as compared to the directions available for the specific panning profile. If it is set to *none*, there will be no panning events raised by the EyeX Engine for this particular pannable interactor. If it is set to *all*, you will still only get vertical velocities if the panning profile is set to vertical.

Dynamically updating the available panning directions might be critical to make the panning work correctly at end-states. For example, if the GUI component that implements the pannable area has some automatic bouncing or rubber-band effect when reaching its end-states, you might get a very peculiar behavior if the contents is moved simultaneously by panning velocities and the bouncing mechanism. For other cases of GUI components and implementations it might be fine to leave the available panning directions unchanged throughout the interaction, and just ignore the panning events that are not applicable in a specific end-state.

Panning events

While the user is panning and looking at a pannable area, the EyeX Engine continuously raises panning events to the pannable interactor. The panning events contain velocity information: in what velocity the pannable area currently should be panned in the horizontal and vertical directions, expressed in pixels per second. It is these events the client application uses to continuously update the ongoing motion of the contents in the pannable area.

Note that the velocities in the panning events describe the panning behavior of the pannable area itself and not its content. For example, to scroll down a web page (a pannable area), the texts and images on the page (the content) should move up. So, depending on how the scrolling is implemented in the client application, the velocities in the panning events can be used as is, or they have to be inverted.

The EyeX Engine will keep firing panning events as long as there are velocity changes and as long as it believes the area can be panned in the direction indicated by the user's eye-gaze (and the available panning directions).

Action command: 'Panning Begin'

There are two action commands associated with a continuous panning interaction. To start panning, and put the the EyeX Engine in the panning mode, the client application sends the *panning begin* command to the engine.

While in the panning mode, the EyeX Engine continuously sends panning events with velocity information as described above in the section about panning events.

The panning mode overrides any ongoing activation mode.

Action command: 'Panning End'

The *panning end* command will end an ongoing panning and switch off the panning mode in the EyeX Engine. If the panning mode was entered from an ongoing activation mode, the engine will go back to the activation mode.

All action commands are global

All action commands are global, meaning that they apply to the EyeX Engine as a whole. This means that a panning begin command sent by your client application can trigger panning in another client application depending on where the user is looking. Bear this possibility in mind when designing your application and its action command handling algorithms.