Here are the classes, structs, unions and interfaces with brief descriptions:

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>nite::Array&lt; T &gt;</td>
</tr>
<tr>
<td>nite::BoundingBox</td>
</tr>
<tr>
<td>nite::GestureData</td>
</tr>
<tr>
<td>nite::HandData</td>
</tr>
<tr>
<td>nite::HandTracker</td>
</tr>
<tr>
<td>nite::HandTrackerFrameRef</td>
</tr>
<tr>
<td>nite::HandTracker::NewFrameListener</td>
</tr>
<tr>
<td>nite::UserTracker::NewFrameListener</td>
</tr>
<tr>
<td>nite::NiTE</td>
</tr>
<tr>
<td>NiteBoundingBox</td>
</tr>
<tr>
<td>NitePlane</td>
</tr>
<tr>
<td>NitePoint3f</td>
</tr>
<tr>
<td>NiteQuaternion</td>
</tr>
<tr>
<td>nite::Plane</td>
</tr>
<tr>
<td>nite::Point3f</td>
</tr>
<tr>
<td>nite::PoseData</td>
</tr>
<tr>
<td>nite::Quaternion</td>
</tr>
<tr>
<td>nite::Skeleton</td>
</tr>
<tr>
<td>nite::SkeletonJoint</td>
</tr>
<tr>
<td>nite::UserData</td>
</tr>
<tr>
<td>nite::UserMap</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>nite::UserTracker</td>
</tr>
<tr>
<td>nite::UserTrackerFrameRef</td>
</tr>
</tbody>
</table>

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by [doxygen](https://www.doxygen.org) 1.7.5.1
#nite::Array< T > Class

Template Reference

```cpp
#include <NiTE.h>
```

List of all members.
## Public Member Functions

<table>
<thead>
<tr>
<th>Array ()</th>
<th>int getSize () const</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool isEmpty () const</td>
<td></td>
</tr>
<tr>
<td>const T &amp; operator[] (int index) const</td>
<td></td>
</tr>
<tr>
<td>void setData (int size, T *data)</td>
<td></td>
</tr>
</tbody>
</table>
**Detailed Description**

```cpp
template<class T>
class nite::Array< T >
```

Provides a simple array class used throughout the API. Wraps a primitive array of objects, holding the elements and their count.
Default constructor. Creates an empty `Array` and sets the element count to zero.
Member Function Documentation

```cpp
template<class T>
int nite::Array< T >::getSize ( ) const [inline]
```

**Returns:**
Current number of elements in the `Array`.

```cpp
template<class T>
bool nite::Array< T >::isEmpty ( ) const [inline]
```

**Returns:**
true if there are elements in the `Array`, false otherwise.

```cpp
template<class T>
const T& nite::Array< T >::operator[] ( int index ) const [inline]
```

Implements the array indexing operator for the `Array` class.

```cpp
template<class T>
void nite::Array< T >::setData ( int size, T* data ) [inline]
```

Setter function for data. Causes this `Array` to wrap an existing primitive array of specified type.

**Parameters:**
- `[in] T` Type of objects the `Array` will contain
<table>
<thead>
<tr>
<th></th>
<th><strong>data</strong></th>
<th>Pointer to first object in list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>size</strong></td>
<td>Number of objects in list</td>
</tr>
</tbody>
</table>

The documentation for this class was generated from the following file:

- **NiTE.h**
#include `<NiTE.h>`

Inheritance diagram for nite::BoundingBox:

```
NiteBoundingBox

nite::BoundingBox
```

List of all members.
## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundingBox ()</td>
<td></td>
</tr>
<tr>
<td>BoundingBox (const Point3f &amp;min, const Point3f &amp;max)</td>
<td></td>
</tr>
</tbody>
</table>
**Detailed Description**

Represents a box in three dimensional space. The box is represented as two points, containing respectively its minimum and maximum x, y and z coordinate values.
Constructor & Destructor Documentation

nite::BoundingBox::BoundingBox ( ) [inline]

Default constructor. Does nothing.

nite::BoundingBox::BoundingBox ( const Point3f & min, const Point3f & max ) [inline]

Constructor. Creates a bounding box from two points in space, which represent that minimum and maximum value of all of its coordinates.

Parameters:

- [in] **min** A point containing the minimum x, y and z values of the bounding box.
- [in] **max** A point containing the maximum x, y and z values of the bounding box.

The documentation for this class was generated from the following file:

- **NiTE.h**
#include <NiTE.h>

Inheritance diagram for nite::GestureData:

List of all members.
Public Member Functions

<table>
<thead>
<tr>
<th>const Point3f &amp; getCurrentPosition() const</th>
</tr>
</thead>
<tbody>
<tr>
<td>GestureType getType() const</td>
</tr>
<tr>
<td>bool isComplete() const</td>
</tr>
<tr>
<td>bool isInProgress() const</td>
</tr>
</tbody>
</table>
**Detailed Description**

This class stores data on a gesture that is being detected. "Gestures" in this context indicate gestures detected directly from the depthmap, so there are no handpoint ID's at this stage. Objects of this class store the data for a specific instance of a specific gesture.

The type, position in space, and completion status of the gesture can all be tracked.

All positions are given in "real world" coordinates. See the OpenNI 2.0 documentation for more information on the coordinate systems used by OpenNI/NiTE.

**See also:**

- **GestureType** enumeration for a list of available hand gestures.
- **HandTracker** for more information on using gestures.
Member Function Documentation

```cpp
core Point3f nite::GestureData::getCurrentPosition ( ) const [inline]
```
This function provides the position in space of the projected gesture, using "real world" coordinates.

**Returns:**  
The position of the gesture being detected.

```cpp
GestureType nite::GestureData::getType ( ) const [inline]
```
This function is used to find out what type of gesture that this object is storing detection data for.

**Returns:**  
the type of gesture.

**See also:**  
`GestureType` for a list of valid gestures.

```cpp
bool nite::GestureData::isComplete ( ) const [inline]
```
This function indicates whether the complete gesture has been detected yet.

**Returns:**  
True if the gesture is complete, false otherwise.

**See also:**  
`isInProgress()` to detect gestures that are only partially complete.
bool nite::GestureData::isInProgress ( ) const [inline]

This function indicates whether a non-completed gesture has been detected. This can be useful for providing user feedback during the gesture.

Returns:
    True if a gesture is currently being detected. False if detection is complete or has not yet started.

See also:
    isComplete() to detect only the completed gesture rather than a gesture in progress.

The documentation for this class was generated from the following file:

- NiTE.h
#include <NiTE.h>

Inheritance diagram for nite::HandData:

![Inheritance Diagram for nite::HandData](image)

List of all members.
# Public Member Functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HandId</code></td>
<td><code>getId()</code> const</td>
</tr>
<tr>
<td><code>const Point3f &amp;</code></td>
<td><code>getPosition()</code> const</td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>isLost()</code> const</td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>isNew()</code> const</td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>isTouchingFov()</code> const</td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>isTracking()</code> const</td>
</tr>
</tbody>
</table>
Detailed Description

This class stores data on a single hand during a single frame of hand detection by the .

It can be used to find out where the hand is in space, the hand's ID, and the tracking status.

All positions are given in "real world" coordinates. See the OpenNI 2.0 documentation for more information on the coordinate systems used by OpenNI/NiTE.

See also:
   HandTracker for more information on hand tracking.
**Member Function Documentation**

**HandId nite::HandData::getId ( ) const [inline]**

Get the ID of the hand referred to by this data.

**Returns:**

`HandId` of this hand.

**const Point3f& nite::HandData::getPosition ( ) const [inline]**

Get the position in space of the hand during this frame.

**Returns:**

Position of the hand.

**bool nite::HandData::isLost ( ) const [inline]**

Indicates whether the hand tracking algorithm has lost track of the hand. This can happen, for example, if the hand leaves the field of view of the sensor or becomes occluded.

**Returns:**

True if this hand is no longer being tracked.

**bool nite::HandData::isNew ( ) const [inline]**

Indicates if the hand referred to by this object is newly tracked.

**Returns:**

True if this is the first frame that this hand has been detected in, false otherwise.
bool nite::HandData::isTouchingFov() const [inline]

Indicates whether the hand is at the edge of the sensor field of view.

**Returns:**
True if the hand is at the edge of the field of view, false otherwise.

bool nite::HandData::isTracking() const [inline]

Indicates whether this hand is being actively tracked.

**Returns:**
True if hand is being successfully tracked, false otherwise.

The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h>`

List of all members.
Classes

class NewFrameListener
## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandTracker ()</td>
<td></td>
</tr>
<tr>
<td>~HandTracker ()</td>
<td></td>
</tr>
<tr>
<td>void addNewFrameListener (NewFrameListener *pListener)</td>
<td></td>
</tr>
<tr>
<td>Status convertDepthCoordinatesToHand (int x, int y, int z, float *pOutX, float *pOutY) const</td>
<td></td>
</tr>
<tr>
<td>Status convertHandCoordinatesToDepth (float x, float y, float z, float *pOutX, float *pOutY) const</td>
<td></td>
</tr>
<tr>
<td>Status create (openni::Device *pDevice=NULL)</td>
<td></td>
</tr>
<tr>
<td>void destroy ()</td>
<td></td>
</tr>
<tr>
<td>float getSmoothingFactor () const</td>
<td></td>
</tr>
<tr>
<td>bool isValid () const</td>
<td></td>
</tr>
<tr>
<td>Status readFrame (HandTrackerFrameRef *pFrame)</td>
<td></td>
</tr>
<tr>
<td>void removeNewFrameListener (NewFrameListener *pListener)</td>
<td></td>
</tr>
<tr>
<td>Status setSmoothingFactor (float factor)</td>
<td></td>
</tr>
<tr>
<td>Status startGestureDetection (GestureType type)</td>
<td></td>
</tr>
<tr>
<td>Status startHandTracking (const Point3f &amp;position, HandId *pNewHandId)</td>
<td></td>
</tr>
<tr>
<td>void stopGestureDetection (GestureType type)</td>
<td></td>
</tr>
<tr>
<td>void stopHandTracking (HandId id)</td>
<td></td>
</tr>
</tbody>
</table>
Detailed Description

This is the main object of the Hand Tracker algorithm. It (along with UserTracker) is one of two main classes in NiTE. All NiTE algorithms are accessible through one of these two classes.

HandTracker provides access to all algorithms relates to tracking individual hands, as well as detecting gestures in the depthmap.

The core of the hand tracking is an algorithm that finds human hands in each from of the depthmap, and reports the position of those hands in space. This can be used for simple detection of higher level gestures and implimentation of gesture based user interfaces. Unlike full body tracking algorithms, handpoint based tracking works on users that are sitting and does not require a full body be visible.

Gesture tracking is generally used to initiate hand tracking. It allows detection of gestures in the raw depth map, without requiring hand points (in contrast to higher-level gestures that might be used to impliment a UI using handpoints). These gestures can be located in space to provide a hint to the hand tracking algorithm on where to start tracking.

The output of the HandTracker occurs one frame at a time. For each input depth frame, a hand tracking frame is output with hand positions, gesture positions, etc. A listener class is provided that allows for event driven reaction to each new frame as it arrives.

Note that creating a HandTracker requires a valid OpenNI 2.0 Device to be initialized in order to provide depth information. See the OpenNI 2.0 documentation for information on connecting a device and starting the stream of depth maps that will drive this algorithm.

See also:
  UserTracker If you want to track full body motion, skeletons, find a floor plane, or detect poses.
  NiTE For a list of static functions that must be used before using
HandTracker
Constructor & Destructor Documentation

nite::HandTracker::HandTracker ( ) [inline]

Default constructor. Creates an empty HandTracker with a NULL handle. This object will not be useful until the create() function is called.

See also:
- HandTracker::create() for a function to create and activate the algorithm.
- HandTracker::isValid() to determine whether create() has already been called.

nite::HandTracker::~HandTracker ( ) [inline]

Destructor. Automatically calls the provided destroy() function.
Member Function Documentation

void nite::HandTracker::addNewFrameListener (NewFrameListener)

Adds a NewFrameListener object to this HandTracker so that it will respond when a new frame is generated.

Parameters:
    [in] pListener Pointer to a listener to add.

See also:
    HandTracker::NewFrameListener for more information on using event based interaction with HandTracker

Status nite::HandTracker::convertDepthCoordinatesToHand (int, int, int, float*, float*)

In general, two coordinate systems are used in OpenNI 2.0. These conventions are also followed in NiTE 2.0.

Hand pont and gesture positions are provided in "Real World" coordinates, while the native coordinate system of depth maps is the "projective" system. In short, "Real World" coordinates locate objects using a Cartesian coordinate system with the origin at the sensor. "Projective" coordinates measure straight line distance from the sensor, and indicate x/y coordinates using pixels in the image (which is mathematically equivalent to specifying angles). See the OpenNI 2.0 documentation online for more information.

This function allows you to convert the native depth map coordinates
to the system used by the hand points. This might be useful for performing certain types of measurements (eg distance between a hand and an object identified only in the depth map).

Note that no output is given for the Z coordinate. Z coordinates remain the same when performing the conversion. An input value is still required for Z, since this can affect the x/y output.

**Parameters:**

- **[in] x** The input X coordinate using the "projective" coordinate system.
- **[in] y** The input Y coordinate using the "projective" coordinate system.
- **[in] z** The input Z coordinate using the "projective" coordinate system.
- **[out] pOutX** Pointer to a location to store the output X coordinate in the "real world" system.
- **[out] pOutY** Pointer to a location to store the output Y coordinate in the "real world" system.

**Returns:**

**Status** indicating success or failure of this operation. This is needed because the ability to convert between coordinate systems requires a properly initialized Device from OpenNI 2.0.

```cpp
Status nite::HandTracker::convertHandCoordinatesToDepth ( float x, float y, float z, float *pOutX, float *pOutY )
```

In general, two coordinate systems are used in OpenNI 2.0. These conventions are also followed in **NiTE 2.0**.
Hand point and gesture positions are provided in "Real World" coordinates, while the native coordinate system of depth maps is the "projective" system. In short, "Real World" coordinates locate objects using a Cartesian coordinate system with the origin at the sensor. "Projective" coordinates measure straight line distance from the sensor (perpendicular to the sensor face), and indicate x/y coordinates using pixels in the image (which is mathematically equivalent to specifying angles). See the OpenNI 2.0 documentation online for more information.

Note that no output is given for the Z coordinate. Z coordinates remain the same when performing the conversion. An input value is still required for Z, since this can affect the x/y output.

This function allows you to convert the coordinates of a hand point or gesture to the native coordinates of a depth map. This is useful if you need to find the hand position on the raw depth map.

**Parameters:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>x</td>
</tr>
<tr>
<td>[in]</td>
<td>y</td>
</tr>
<tr>
<td>[in]</td>
<td>z</td>
</tr>
<tr>
<td>[out]</td>
<td>pOutX</td>
</tr>
<tr>
<td>[out]</td>
<td>pOutY</td>
</tr>
</tbody>
</table>

**Returns:**

`Status` indicating success or failure of this operation. This is needed because the ability to convert between coordinate systems requires a properly initialized Device from OpenNI 2.0.

```
Status nite::HandTracker::create ( openni::Device * pDevice = NULL )
```
Creates and initializes an empty HandTracker. This function should be the first one called when a new HandTracker object is constructed.

An OpenNI device with depth capabilities is required for this algorithm to work. See the OpenNI 2.0 documentation for more information about using an OpenNI 2.0 compliant hardware device and creating a Device object.

**Parameters:**

- [in] `pDevice` A pointer to an initialized OpenNI 2.0 Device object that provides depth streams.

**Returns:**

A status code to indicate success/failure. Since this relies on an external hardware device, it is important for applications to check this value.

**See also:**

- Status enumeration for a list of all possible status values generated by this call.

```cpp
void nite::HandTracker::destroy() [inline]
```

Shuts down the hand tracker and releases all resources used by it.

This is the opposite of `create()`. This function is called automatically by the destructor in the current implementation, but it is good practice to run it manually when the algorithm is no longer required. Running this function more than once is safe -- it simply exits if called on a non-valid HandTracker.

```cpp
float nite::HandTracker::getSmoothingFactor() const [inline]
```

Queries the current hand smoothing factor.
Returns:
Current hand smoothing factor.

See also:
setSmoothingFactor for more information on the smoothing factor, and the means to change it.

bool nite::HandTracker::isValid () const [inline]

Indicates whether the HandTracker is valid.

When a new HandTracker is first constructed, this function will indicate that it is invalid (ie return False). Once the create() function has been successfully called, then this function will return True. If the destroy() function is called, this function will again indicate invalid.

It is safe to run create() and destroy() without calling this function -- both of those functions already check this value and return without doing anything if no action is required.

Returns:
True if the HandTracker object is correctly initialized, False otherwise.

See also:
create() function -- causes the UserTracker to become initialized.
destroy() function -- causes the UserTracker to become uninitialized.

Status nite::HandTracker::readFrame ( HandTrackerFrameRef * pFrame)

Gets the next snapshot of the algorithm. This causes all data to be generated for the next frame of the algorithm -- algorithm frames correspond to the input depth frames used to generate them.
### Parameters:

**pFrame** [out] A pointer that will be set to point to the next frame of data.

### Returns:

Status code indicating whether this operation was successful.

---

**void nite::HandTracker::removeNewFrameListener (NewFrameListener)**

Removes a NewFrameListener object from this HandTracker's list of listeners. The listener will no longer respond when a new frame is generated.

### Parameters:

[in] **pListener** Pointer to a listener to remove.

### See also:

HandTracker::NewFrameListener for more information on using event based interaction with HandTracker.

---

**Status nite::HandTracker::setSmoothingFactor (float factor)** [inline]

Control the smoothing factor of the hand points. Factor should be between 0 (no smoothing at all) and 1 (no movement at all).

Experimenting with this factor should allow you to fine tune the hand tracking performance. Higher values will produce smoother movement of the handpoints, but may make the handpoints feel less responsive to the user.

### Parameters:

[in] **factor** The smoothing factor.

### Returns:

Status code indicating success or failure of this operation.
Status nite::HandTracker::startGestureDetection (GestureType typ)

Start detecting a specific gesture. This function will cause the algorithm to start scanning the entire field of view for any hand that appears to be performing the gesture specified. Intermediate progress is available to aid in providing feedback to the user.

Gestures are detected from the raw depth map. They don't depend on hand points. They are most useful for determining where a hand is in space to start hand tracking. Unlike handpoints, they do not follow a specific hand, so they will react to a hand anywhere in the room.

If you want to detect user gestures for input purposes, it is often better to use a single “focus” gesture to start hand tracking, and then detect other gestures from the handpoints. This enables an application to focus on a single user, even in a crowded room.

Hand points can also be more computationally efficient. The gesture tracking algorithm for any given gesture uses about as much CPU bandwidth as the hand tracker. Adding more gestures or also running the hand tracker increases CPU consumption linearly. Finding gestures from hand points, on the other hand, can be done for negligible CPU cost once the handpoint algorithm has run. This means that user interface complexity will scale better with CPU complexity.

**Parameters:**

[in] @ref GestureType you wish to detect

**Returns:**

Status indicating success or failure of this operation.

**See also:**

UserTracker if you want to do full body poses instead of hand-only gestures

GestureType enumeration for list of available gestures.
**stopGestureDetection** to stop detection once it has started.

**Status** nite::HandTracker::startHandTracking ( const Point3f & pos, HandId * pNewHandId )

Starts tracking a hand at a specific point in space. Use of this function assumes that there actually is a hand in the location given. In general, the hand algorithm is much better at tracking a specific hand as it moves around than it is at finding the hand in the first place.

This function is typically used in conjunction with gesture detection. The position in space of the gesture is used to initiate hand tracking. It is also possible to start hand tracking without a gesture if your application will constrain users to place their hands in a certain known point in space. A final possibility is for applications or third party middleware to implement their own hand ‘finding’ algorithm either in depth or from some other information source, and using that data to initialize the hand tracker.

The position in space of the hand point is specified in "real world" coordinates. See OpenNI 2.0 documentation for more information on coordinate systems.

**Parameters:**
- [in] **Point** where hand is known/suspected to exist.
- [in] **ID** to assign a hand once tracking starts. This will be used to refer to the hand later.

**Returns:**
- **Status** code indicating success or failure of this operation.

**void nite::HandTracker::stopGestureDetection ( GestureType type );**

Stop detecting a specific gesture. This disables detection of the
specified gesture. Doing this when that gesture is no longer required prevents false detection and saves CPU bandwidth.

**Parameters:**

- [in] @ref GestureType you would like to stop detecting.

```cpp
void nite::HandTracker::stopHandTracking (HandId id) [inline]
```

Commands the algorithm to stop tracking a specific hand. Note that the algorithm may be tracking more than one hand. This function only halts tracking on the single hand specified.

**Parameters:**

- [in] id The HandId of the hand to quit tracking.

The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h>`

List of all members.
### Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HandTrackerFrameRef()</code></td>
<td>Constructor</td>
</tr>
<tr>
<td><code>HandTrackerFrameRef(const HandTrackerFrameRef &amp;other)</code></td>
<td>Copy constructor</td>
</tr>
<tr>
<td><code>~HandTrackerFrameRef()</code></td>
<td>Destructor</td>
</tr>
<tr>
<td><code>openni::VideoFrameRef getDepthFrame()</code></td>
<td>Get depth frame</td>
</tr>
<tr>
<td><code>int getFrameIndex()</code></td>
<td>Get frame index</td>
</tr>
<tr>
<td><code>const Array&lt;GestureData&gt; &amp; getGestures()</code></td>
<td>Get gestures</td>
</tr>
<tr>
<td><code>const Array&lt;HandData&gt; &amp; getHands()</code></td>
<td>Get hands</td>
</tr>
<tr>
<td><code>uint64_t getTimestamp()</code></td>
<td>Get timestamp</td>
</tr>
<tr>
<td><code>bool isValid()</code></td>
<td>Check validity</td>
</tr>
<tr>
<td><code>HandTrackerFrameRef &amp; operator=(const HandTrackerFrameRef &amp;other)</code></td>
<td>Assignment operator</td>
</tr>
<tr>
<td><code>void release()</code></td>
<td>Release resource</td>
</tr>
</tbody>
</table>
class HandTracker
Detailed Description

Snapshot of the Hand Tracker algorithm. It contains all output from a single frame of the Hand Tracking algorithm, which corresponds to a single input depth frame.

It holds all the hands identified at this time, as well as the detected gestures.

**See also:**

HandTracker for how this data is created
Constructor & Destructor Documentation

nite::HandTrackerFrameRef::HandTrackerFrameRef ( ) [inline]

Default constructor. Creates a new empty HandTrackerFrameRef. This object will not be valid or useful until it has been initialized with data from a specific frame.

nite::HandTrackerFrameRef::~HandTrackerFrameRef ( ) [inline]

Destructor. Calls the provided release() function.

nite::HandTrackerFrameRef::HandTrackerFrameRef ( const HandTrackerFrameRef & ) [inline]

Copy constructor. Creates a HandTrackerFrameRef that is identical to and existing one.

Parameters:
[in] @ref HandTrackerFrameRef to be copied.
Member Function Documentation

openni::VideoFrameRef nite::HandTrackerFrameRef::getDepthFrame

Provides the raw depth frame used to generate this Hand Tracker frame. See the OpenNI 2.0 documentation for more information on manipulating this data.

Returns:
VideoFrameRef used to create this object.

int nite::HandTrackerFrameRef::getFrameIndex ( ) const [inline]

Gets the frame index of the raw depth frame that generated this frame. Frame indexes are consecutive integer numbers assigned to frames by OpenNI. See the OpenNI documentation for more information.

Returns:
FrameIndex of the depth frame used to generate this User Tracker frame.

const Array<GestureData>& nite::HandTrackerFrameRef::getGestures

This function provides a list of all gestures being actively detected in this frame.

Returns:
Array of GestureData objects, one for each gesture being detected.

const Array<HandData>& nite::HandTrackerFrameRef::getHands (
This function provides a list of all hands being tracked in this frame.

**Returns:**

Array of HandData objects, one for each hand being tracked.

```cpp
uint64_t nite::HandTrackerFrameRef::getTimestamp() const
```

Get the timestamp in which this frame was processed. See the OpenNI 2.0 documentation for more information on how OpenNI assigns time stamps to depth frames.

**Returns:**

Timestamp in microseconds

```cpp
bool nite::HandTrackerFrameRef::isValid() const
```

Indicates whether this object has been loaded with valid data.

**Returns:**

True if this object contains valid data, false otherwise.

```cpp
HandTrackerFrameRef& nite::HandTrackerFrameRef::operator=(const
```

Operator. Implements the assignment operator for this class.

**Parameters:**

[in] other HandTrackerFrameRef to be assigned.

```cpp
void nite::HandTrackerFrameRef::release() [inline]
```

Properly releases all resources used by this object. This function is automatically called by the destructor, but it is a good practice to
release the resources as soon as they are no longer required. It is safe to run this function multiple times on the same object.
friend class HandTracker [friend]

The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h>`

List of all members.
## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewFrameListener ()</td>
<td>virtual void onNewFrame (HandTracker &amp;)=0</td>
</tr>
</tbody>
</table>
Friends

class HandTracker
Detailed Description

This is a listener class that is used to react to events generated by the HandTracker class.

To use this class, you must derive a class from it that implements the onNewFrame() function. This is the function that will be called when an event is generated.

Create a new instance of your derived class. Then, use the HandTracker::addNewFrameListener() function to add the listener to the HandTracker. When that HandTracker generates an onNewFrame event, the specified callback function will be called.

The onNewFrame event is currently the only event type that this listener is designed to work with.

See also:
   HandTracker for the source of this listener's events.
Default Constructor. Creates a new object of this type and configures it to correctly receive events.
Member Function Documentation

virtual void nite::HandTracker::NewFrameListener::onNewFrame (HandTracker)

This is the callback function for the event. It should be implemented in a class derived from NewFrameListener. This function will automatically be called when the OnNewFrame event is triggered.

**Parameters:**

[| in | A reference to the HandTracker that triggered the event is provided. |]
Friends And Related Function Documentation

friend class HandTracker [friend]

The documentation for this class was generated from the following file:

- NiTE.h

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
#include <NiTE.h>

List of all members.
Public Member Functions

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewFrameListener ()</td>
<td>virtual void onNewFrame (UserTracker &amp;)=0</td>
</tr>
</tbody>
</table>
Friends

class UserTracker
Detailed Description

This is a listener class that is used to react to events generated by the UserTracker class.

To use this class, you must derive a class from it that implements the onNewFrame() function. This is the function that will be called when an event is generated.

Create a new instance of your derived class. Then, use the UserTracker::addNewFrameListener() function to add the listener to the UserTracker. When that UserTracker generates an onNewFrame event, the specified callback function will be called.

The onNewFrame event is currently the only event type that this listener is designed to work with.

See also:

UserTracker for the source of this listener's events.
Default Constructor. Creates a new object of this type and configures it to correctly receive events.
Member Function Documentation

virtual void nite::UserTracker::NewFrameListener::onNewFrame ( UserTracker *tracker )

This is the callback function for the event. It should be implemented in a class derived from NewFrameListener. This function will automatically be called when the OnNewFrame event is triggered.

Parameters:

[in] A reference to the UserTracker that triggered the event is provided.
friend class UserTracker [friend]

The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h>`

List of all members.
## Static Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getVersion</code></td>
<td>Static Version</td>
</tr>
<tr>
<td><code>initialize</code></td>
<td>Static Status</td>
</tr>
<tr>
<td><code>shutdown</code></td>
<td>Static void</td>
</tr>
</tbody>
</table>
Detailed Description

The **NiTE** class is a static entry point to the library.

All applications that expect to use the **NiTE** middleware should run the **NiTE.initialize()** function before using any other API calls. Before the application exits, the application should run the **NiTE.shutdown()** function to properly close the API.
Member Function Documentation

**static Version nite::NiTE::getVersion()** [inline, static]

This function allows you to obtain the current version of the NiTE installation. This is useful for verifying application compatibility with the currently used version of NiTE, and for reporting this information to the user for troubleshooting purposes. In general, there should be minimal version conflict problems, as NiTE and OpenNI applications each contain their own copy of the API runtime files.

**Returns:**
The version of the current NiTE installation.

**static Status nite::NiTE::initialize()** [inline, static]

Initializes the NiTE API. This function should be called before using any other API functionality.

**Returns:**
A status code of type Status that indicates success/failure of the initialization.

**static void nite::NiTE::shutdown()** [inline, static]

Shuts down the API, properly releasing all resources used by NiTE. This should be called by all NiTE applications before they close.

The documentation for this class was generated from the following file:

- NiTE.h
NiteBoundingBox
Struct Reference

#include <NiteCTypes.h>

Inheritance diagram for NiteBoundingBox:

[legend]

List of all members.
## Public Attributes

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NitePoint3f</td>
<td>max</td>
</tr>
<tr>
<td>NitePoint3f</td>
<td>min</td>
</tr>
</tbody>
</table>
Detailed Description

3D Box
Member Data Documentation

NitePoint3f NiteBoundingBox::max

NitePoint3f NiteBoundingBox::min

The documentation for this struct was generated from the following file:

- NiteCTypes.h
NitePlane Struct
Reference

#include <NiteCTypes.h>

Inheritance diagram for NitePlane:

List of all members.
**Public Attributes**

<table>
<thead>
<tr>
<th>NitePoint3f</th>
<th>normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>NitePoint3f</td>
<td>point</td>
</tr>
</tbody>
</table>
Detailed Description

3D Plane
Member Data Documentation

NitePoint3f NitePlane::normal

NitePoint3f NitePlane::point

The documentation for this struct was generated from the following file:

- NiteCTypes.h

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
NitePoint3f Struct
Reference

#include <NiteCTypes.h>

Inheritance diagram for NitePoint3f:

```
NiTEPoint3f

nite::Point3f
```

List of all members.
## Public Attributes

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>x</td>
</tr>
<tr>
<td>float</td>
<td>y</td>
</tr>
<tr>
<td>float</td>
<td>z</td>
</tr>
</tbody>
</table>
Detailed Description

3D Point
Member Data Documentation

float NitePoint3f::x

float NitePoint3f::y

float NitePoint3f::z

The documentation for this struct was generated from the following file:

- NiteCTypes.h

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
#include `<NiteCTypes.h>`

Inheritance diagram for NiteQuaternion:

![Inheritance Diagram](legend)

List of all members.
Public Attributes

<table>
<thead>
<tr>
<th>float</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>x</td>
</tr>
<tr>
<td>float</td>
<td>y</td>
</tr>
<tr>
<td>float</td>
<td>z</td>
</tr>
</tbody>
</table>
Detailed Description

Quaternion
Member Data Documentation

float NiteQuaternion::w

float NiteQuaternion::x

float NiteQuaternion::y

float NiteQuaternion::z

The documentation for this struct was generated from the following file:

- NiteCTypes.h
#include <NITE.h>

Inheritance diagram for nite::Plane:

List of all members.
## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plane ()</td>
<td></td>
</tr>
<tr>
<td>Plane (const Point3f &amp;point, const Point3f &amp;normal)</td>
<td></td>
</tr>
</tbody>
</table>
Detailed Description

Enapsulates a geometrical plane. The plane is defined by a single point on that plane, and a normal vector to the plane. The normal vector is represented as a **Point3f**.
Constructor & Destructor Documentation

nite::Plane::Plane ( ) [inline]

Default constructor. Creates a plane that passes through the origin, and has a normal vector set to (0,0,0). Note that the default normal vector will make the plane generated by this function invalid for many mathematical operations, as it is a degenerate vector and has no direction.

nite::Plane::Plane ( const Point3f & point,
                     const Point3f & normal
                     ) [inline]

Constructor. Creates a new point, given a single point on that plane and a normal vector.

Parameters:

  [in] point  Any point on the plane
  [in] normal The normal vector of the plane, represented as a Point3f

The documentation for this class was generated from the following file:

- NiTE.h
#include <NiTE.h>

Inheritance diagram for nite::Point3f:

```
NiTEPoint3f

nite::Point3f
```

[legend]

List of all members.
## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point3f ()</td>
<td></td>
</tr>
<tr>
<td>Point3f (float x, float y, float z)</td>
<td></td>
</tr>
<tr>
<td>Point3f (const Point3f &amp;other)</td>
<td></td>
</tr>
<tr>
<td>bool operator!= (const Point3f &amp;other) const</td>
<td></td>
</tr>
<tr>
<td>Point3f &amp; operator= (const Point3f &amp;other)</td>
<td></td>
</tr>
<tr>
<td>bool operator== (const Point3f &amp;other) const</td>
<td></td>
</tr>
<tr>
<td>void set (float x, float y, float z)</td>
<td></td>
</tr>
</tbody>
</table>
Detailed Description

Encapsulates a single point in 3D space, storing the x/y/z coordinates as floating point numbers.

Can also be used to represent a three dimensional vector.
Constructor & Destructor Documentation

nite::Point3f::Point3f () [inline]

Default Constructor, creates a point with all three coordinate values set to 0.0

nite::Point3f::Point3f ( float x, float y, float z ) [inline]

Constructor. Creates a point from three given coordinates.

Parameters:
[in] x X coordinate of point to be created
[in] y Y coordinate of point to be created
[in] z Z coordinate of point to be created

nite::Point3f::Point3f ( const Point3f & other ) [inline]

Copy Constructor. Creates a new point that has the same coordinates as an existing point.

Parameters:
[in] other An existing point to be copied.
Member Function Documentation

```cpp
bool nite::Point3f::operator!=(const Point3f & other) const [inline]
```

Inequality operator. Tests two points to see if they are different.

**Parameters:**
- `[in]` `other` One of two points to test

**Returns:**
- True if either x, y, or z coordinate of first point differs from the matching coordinate of the second point.

```cpp
Point3f& nite::Point3f::operator=(const Point3f & other) [inline]
```

Assignment operator. Sets all coordinates of the point on the left to be equal to the coordinate values of the point on the left.

**Parameters:**
- `[in]` `other` Point to copy coordinates from

**Returns:**
- Reference to a point containing a copy of the input points coordinates.

```cpp
bool nite::Point3f::operator==(const Point3f & other) const [inline]
```

Equality operator. Tests two points to determine whether they have matching coordinate values.

**Parameters:**
- `[in]` `other` One of the points to be tested against

**Returns:**
True if all three coordinate values of left point match all three coordinate values of right point.

```cpp
void nite::Point3f::set(float x,
                        float y,
                        float z
                        ) [inline]
```

Setter function for all coordinates of the point.

**Parameters:**
- `[in] x` Desired new X coordinate of the point.
- `[in] y` Desired new Y coordinate of the point.
- `[in] z` Desired new Z coordinate of the point.

The documentation for this class was generated from the following file:

- **NiTE.h**
#include `<NiTE.h>`

Inheritance diagram for nite::PoseData:

```
  NitePoseData
  ↑
nite::PoseData
```

List of all members.
## Public Member Functions

<table>
<thead>
<tr>
<th>PoseType</th>
<th>getType () const</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>isEntered () const</td>
</tr>
<tr>
<td>bool</td>
<td>isExited () const</td>
</tr>
<tr>
<td>bool</td>
<td>isHeld () const</td>
</tr>
</tbody>
</table>
Detailed Description

This class wraps the output from the NiTE Pose Detection algorithms.

NiTE provides the ability for applications to detect whether a user is in a specific pose. This is most commonly used for skeleton calibration. The algorithm passes through a number of different states during the process of Pose Detection.

For each user being analyzed, this class stores the pose being detected and the state of that detection.
Member Function Documentation

**PoseType nite::PoseData::getType() const** [inline]

Get the type of this pose.

**Returns:**
The type of pose being detected, formatted as a `PoseType`.

**bool nite::PoseData::isEntered() const** [inline]

Checks whether this is the first frame in which the user has assumed the pose indicated by `PoseData::getType()`.

**Returns:**
True if the user is in the pose indicated AND the user was not in that pose on the previous frame. Returns false otherwise.

**bool nite::PoseData::isExited() const** [inline]

Check if the user has exited the pose in this frame.

**Returns:**
True if the user is NOT in the pose indicated, and was in the previous frame. Returns false otherwise.

**bool nite::PoseData::isHeld() const** [inline]

Checks if the user is currently in the pose indicated by the `PoseData::getType()` function.

**Returns:**
True if the user is in the pose indicated, False otherwise.
The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h>`

Inheritance diagram for nite::Quaternion:

```
NiTEQuatenn

nite::Quaternion

[legend]
```

List of all members.
Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quaternion ()</td>
<td></td>
</tr>
<tr>
<td>Quaternion (float $w$, float $x$, float $y$, float $z$)</td>
<td></td>
</tr>
</tbody>
</table>
Detailed Description

Represents a Quaternion. The Quaternion is stored as four floating point numbers. (The quaternions are a number system that extends the complex number system from two dimensions to four.)
Constructor & Destructor Documentation

nite::Quaternion::Quaternion ( ) [inline]

Default Constructor. Creates a new Quaternion with the value of all four components set to zero.

nite::Quaternion::Quaternion ( float \textit{w}, float \textit{x}, float \textit{y}, float \textit{z}) [inline]

Constructor. Creates a new Quaternion with each component specified.

\textbf{Parameters:}\n
- \textit{w} Desired first quaternion component value.
- \textit{x} Desired second quaternion component value.
- \textit{y} Desired third quaternion component value.
- \textit{z} Desired fourth quaternion component value.

The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h`

Inheritance diagram for nite::Skeleton:

[legend]

List of all members.
Public Member Functions

const SkeletonJoint & getJoint (JointType type) const
SkeletonState getState () const
Detailed Description

This is the main class for the skeleton algorithm.

The purpose of the skeleton algorithm is to analyze a user outline supplied by the User Tracker algorithm, and to locate the position of that user's joints in space (eg knees, elbows, head, etc). The orientation of the user's joints is also calculated. Where joints are not visible, the algorithm will make a best guess about the joint. For all data calculated, confidence values are also created to help an application understand if the algorithm is sure about the data, or if it is "guessing".

Nite offers two types of skeletons algorithms. One requires the user to assume a specific "calibration pose" before calibration can be attempted. The other does not require a specific pose from the user, but will require a few seconds to settle down as it learns a user's body shape. The algorithm used sometimes depends on the platform being used -- for example, the ARM version of Nite currently only offers the version that requires a calibration pose.

All skeleton joint positions are given in "real world" data coordinates, using the same conventions as OpenNI 2.0. See the online documentation for OpenNI 2.0 for more information on the coordinate systems used with the skeleton.

See also:

SkeletonJoint for the output data format of the skeleton.
JointType enumeration for a list of all joints detected by the skeleton.
UserTracker to understand where the input data to the skeleton comes from.
SkeletonState enumeration for a list of possible skeleton states.
Member Function Documentation

const SkeletonJoint& nite::Skeleton::getJoint ( JointType type ) const

Get a specific joint of the skeleton. The object returned by this function can be queried to determine position, orientation, and confidence values for the joint.

Parameters:
   [in] type  The type of the joint to retrieve from the skeleton.

Returns:  
   Pointer to a SkeletonJoint containing the requested data.

See also:  
   JointType for a list of possible joints to request from the skeleton.

SkeletonState nite::Skeleton::getState ( ) const [inline]

Get the state of the skeleton. This is useful for determining whether an active skeleton exists, and whether it is calibrating. Some of the states are also error conditions that attempt to provide a hint about why the skeleton is not calibrating. This can be useful for providing feedback to a user about where they need to stand, assuming a calibration pose, etc.

Returns:  
   Current state of the skeleton.

See also:  
   SkeletonState enumeration for a list of possible states

The documentation for this class was generated from the following file:
#include <NiTE.h>

Inheritance diagram for nite::SkeletonJoint:

List of all members.
### Public Member Functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>const Quaternion &amp;</td>
<td>getOrientation() const</td>
<td>const</td>
</tr>
<tr>
<td>float</td>
<td>getOrientationConfidence() const</td>
<td>const</td>
</tr>
<tr>
<td>const Point3f &amp;</td>
<td>getPosition() const</td>
<td>const</td>
</tr>
<tr>
<td>float</td>
<td>getPositionConfidence() const</td>
<td>const</td>
</tr>
<tr>
<td>JointType</td>
<td>getType() const</td>
<td>const</td>
</tr>
</tbody>
</table>
Detailed Description

This class provides a data structure to store the output of the Skeleton algorithm.

The basic purpose of the skeleton algorithm is to locate each of a user's joints, and to estimate the position and orientation of each joint in space. The output from the Skeleton algorithm is stored as a series of SkeletonJoint objects.

See also: nite::Skeleton for more information about the Skeleton algorithm.
Member Function Documentation

**const Quaternion**& nite::SkeletonJoint::getOrientation ( ) const

Get the current orientation of the joint represented by this object. This orientation will be represented by a Quaternion.

A confidence value is also available via the **getOrientationConfidence()** function to indicate how sure the algorithm is of this value.

**Returns:**
Joint orientation

**float** nite::SkeletonJoint::getOrientationConfidence ( ) const

Indicates how sure the NiTE skeleton algorithm is about the orientation data stored about this joint. The value is between 0 and 1, with increasing value indicating increasing confidence.

**Returns:**
Confidence value between 0 and 1

**const Point3f**& nite::SkeletonJoint::getPosition ( ) const

Get the current position of the joint. This will be represented as a point in space. The coordinates used are "real world" coordinates. See the OpenNI2 documentation for more information on coordinate systems.

A confidence value is also available via the **getPositionConfidence()** function to indicate how sure the algorithm is of this value.

**Returns:**
Position in space of the joint.

```cpp
float nite::SkeletonJoint::getPositionConfidence ( ) const [inline]
```

Indicates how sure the NiTE skeleton algorithm is about the position data stored about this joint. The value is between 0 and 1, with increasing value indicating increasing confidence.

**Returns:**
- Confidence value between 0 and 1

```cpp
JointType nite::SkeletonJoint::getType ( ) const [inline]
```

Gets the type of the joint represented by this particular object.

**Returns:**
- Type of the joint.

**See also:**
- JointType enumeration for a list of all possible joint types.

The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h>`

Inheritance diagram for nite::UserData:

```
<table>
<thead>
<tr>
<th>NiteUserData</th>
</tr>
</thead>
<tbody>
<tr>
<td>nite::UserData</td>
</tr>
</tbody>
</table>
```

List of all members.
## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const BoundingBox &amp; getBoundingBox () const</code></td>
<td>get bounding box</td>
</tr>
<tr>
<td><code>const Point3f &amp; getCenterOfMass () const</code></td>
<td>get center of mass</td>
</tr>
<tr>
<td><code>UserId getId () const</code></td>
<td>get user ID</td>
</tr>
<tr>
<td><code>const PoseData &amp; getPose (PoseType type) const</code></td>
<td>get pose data</td>
</tr>
<tr>
<td><code>const Skeleton &amp; getSkeleton () const</code></td>
<td>get skeleton</td>
</tr>
<tr>
<td><code>bool isLost () const</code></td>
<td>check if lost</td>
</tr>
<tr>
<td><code>bool isNew () const</code></td>
<td>check if new</td>
</tr>
<tr>
<td><code>bool isVisible () const</code></td>
<td>check if visible</td>
</tr>
</tbody>
</table>
**Detailed Description**

Provides the current information available about a specific user detected by the **UserTracker**.

Note that some data is provided in depth ("projective") coordinates, and other data is provided in world ("real world") coordinates. See the OpenNI 2.0 documentation for information on the two coordinate systems used and how to convert between them.

**See also:**
- **UserTracker** for more information on how this data is generated.
- **Skeleton** for more information on a user's skeleton
- **UserMap** for information on locating a specific user in the scene.
Member Function Documentation

`const BoundingBox& nite::UserData::getBoundingBox() const [inline]`

Get a bounding box around the user. This provides a minimum volume cube that completely contains the user. Coordinates are "projective", so they are suitable for direct use with a depthmap.

**Returns:**

`BoundingBox` that surrounds the user in the `UserMap`

`const Point3f& nite::UserData::getCenterOfMass() const [inline]`

Find the geometric center of mass of the user. This value is given in "real world" coordinates, so it must be converted before being superimposed over a raw depthmap. It is suitable for direct use with skeleton and hand point output.

**Returns:**

`Point3f` indicating position in space of the user.

`UserId nite::UserData::getId() const [inline]`

Get the ID of the user. This ID is persistent -- it will not change as long as the user is being continuously tracked.

**Returns:**

`UserId` of the user being tracked.

`const PoseData& nite::UserData::getPose(PoseType type) const`

Get all information about a specific pose for this user. This will only be available if pose detection is enabled on this user for the `PoseType`
specified.

**Parameters:**

[in] ```@ref PoseType``` to retrieve data for.

**Returns:**
```
PoseData ``` for this pose type and this user.

**See also:**

`UserTracker` for more information on enabling Pose detected.

---

```const Skeleton& nite::UserData::getSkeleton( ) const [inline]```

Get the full skeleton of this user. This will only be available if ```Skeleton``` tracking has been enabled for this user, and the skeleton is calibrated and being actively tracked.

**Returns:**
```
Skeleton ``` of the user.

**See also:**

`UserTracker` for information on enabling skeleton tracking.

---

```bool nite::UserData::isLost( ) const [inline]```

Check if the user is lost. This will happen once, in the first frame in which the user was declared lost. This user will not be provided in future frames, and his UserId may be assigned to a new user that appears in view.

**Returns:**

True if the user is no longer being tracked, False otherwise.

---

```bool nite::UserData::isNew( ) const [inline]```
Checks whether this user has been newly detected.

**Returns:**
True if this is the first frame this user appears in, False otherwise.

```cpp
bool nite::UserData::isVisible () const [inline]
```

Check if the user is currently visible in the field of view.

**Returns:**
True if the user is visible and detected, false otherwise.

The documentation for this class was generated from the following file:

- **NiTE.h**
#include `<NiTE.h>`

Inheritance diagram for nite::UserMap:

![Inheritance Diagram](legend)

List of all members.
### Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getHeight () const</code></td>
<td></td>
</tr>
<tr>
<td><code>const UserId * getPixels () const</code></td>
<td></td>
</tr>
<tr>
<td><code>int getStride () const</code></td>
<td></td>
</tr>
<tr>
<td><code>int getWidth () const</code></td>
<td></td>
</tr>
</tbody>
</table>
Friends

class UserTrackerFrameRef
Detailed Description

This class stores the output from the nite::UserTracker algorithm.

The basic purpose of the User Tracker algorithm is to analyze a depth frame and to locate all users in the scene, to indicate which pixels belong to which user, and which pixels belong to the background.

The output is in the form of a two dimensional array with the same dimensions as the depth frame that was used as the input to UserTracker. Each element of the array corresponds to a single pixel from the depth map, and indicates the UserID of the user which occupies that pixel.

The UserId value 0 is used to indicate pixels that contain no users (ie background).

See also: UserTracker for more information about this algorithm.
Member Function Documentation

```
int nite::UserMap::getHeight ( ) const [inline]
```

Gets the height of the UserMap array. This value will match the Y resolution of the depth map that was used to create the UserMap. This value can be used, along with the value returned by `getStride()` to determine the size of the array in bytes.

**Returns:**
- Height of the UserMap, measured in pixels.

```
const UserId* nite::UserMap::getPixels ( ) const [inline]
```

Returns a direct pointer to the array containing the UserMap data. This array has type UserId, and a size equal to y*stride. If your application needs to calculate the size value directly, then see `UserMap::getHeight()` and `UserMap::getStride()`.

**Returns:**
- Pointer directly to the array data

```
int nite::UserMap::getStride ( ) const [inline]
```

This function returns the stride of the array returned by `getPixels`. The stride of the array is defined as the width of a single row of that array, measured in bytes. This value can be multiplied by the value returned by `getHeight()` to calculate the size of the array.

**Returns:**
- Stride of the array, measured in bytes.

```
int nite::UserMap::getWidth ( ) const [inline]
```
Gets the width of the UserMap array. This value will match the X resolution of the depth map that was used to create the UserMap.

**Returns:**

Width of the UserMap, measured in pixels.
Friends And Related Function Documentation

friend class UserTrackerFrameRef [friend]

The documentation for this class was generated from the following file:

- NiTE.h
#include `<NiTE.h>`

List of all members.
## Classes

<table>
<thead>
<tr>
<th>class</th>
<th>NewFrameListener</th>
</tr>
</thead>
</table>

## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserTracker ()</td>
<td></td>
</tr>
<tr>
<td>~UserTracker ()</td>
<td></td>
</tr>
<tr>
<td>void addNewFrameListener (NewFrameListener *pListener)</td>
<td></td>
</tr>
<tr>
<td>Status convertDepthCoordinatesToJoint (int x, int y, int z, float *pOutX, float *pOutY) const</td>
<td></td>
</tr>
<tr>
<td>Status convertJointCoordinatesToDepth (float x, float y, float z, float *pOutX, float *pOutY) const</td>
<td></td>
</tr>
<tr>
<td>Status create (openni::Device *pDevice=NULL)</td>
<td></td>
</tr>
<tr>
<td>void destroy ()</td>
<td></td>
</tr>
<tr>
<td>float getSkeletonSmoothingFactor () const</td>
<td></td>
</tr>
<tr>
<td>bool isValid ()</td>
<td></td>
</tr>
<tr>
<td>Status readFrame (UserTrackerFrameRef *pFrame)</td>
<td></td>
</tr>
<tr>
<td>void removeNewFrameListener (NewFrameListener *pListener)</td>
<td></td>
</tr>
<tr>
<td>Status setSkeletonSmoothingFactor (float factor)</td>
<td></td>
</tr>
<tr>
<td>Status startPoseDetection (UserId user, PoseType type)</td>
<td></td>
</tr>
<tr>
<td>Status startSkeletonTracking (UserId id)</td>
<td></td>
</tr>
<tr>
<td>void stopPoseDetection (UserId user, PoseType type)</td>
<td></td>
</tr>
<tr>
<td>void stopSkeletonTracking (UserId id)</td>
<td></td>
</tr>
</tbody>
</table>
Detailed Description

This is the main object of the User Tracker algorithm. It provides access to one half of the algorithms provided by NiTE. Scene segmentation, skeleton, floor plane detection, and pose detection are all provided by this class.

The first purpose of the User Tracker algorithm is to find all of the active users in a specific scene. It individually tracks each human it finds, and provides the means to separate their outline from each other and from the background. Once the scene has been segmented, the User Tracker is also used to initiate Skeleton Tracking and Pose Detection algorithms.

Each user is provided an ID as they are detected. The user ID remains constant as long as the user remains in the frame. If a user leaves the field of view of the camera, or tracking of that user is otherwise lost, the user may have a different ID when he is detected again. There is currently no mechanism that provides persistent recognition of individuals when they are not being actively tracking. If this functionality is desired, it will need to be implemented at the application level.

A listener class is provided to allow event-based interaction with this algorithm.

See also:

- UserMap for the output format of the User Tracker algorithm
- UserData for additional data output by this format
- Skeleton if you are also interested in tracking a user's skeleton
- NiTE for a couple of static functions that must be run before User Tracker can be used
- HandTracker for Gesture and Hand tracking algorithms.
Constructor & Destructor Documentation

nite::UserTracker::UserTracker ( ) [inline]

Default constructor. Creates an empty UserTracker with a NULL handle. This object will not be useful until the create() function is called.

See also:
    UserTracker::create() for a function to create and activate the algorithm.
    UserTracker::isValid() to determine whether create() has already been called.

nite::UserTracker::~UserTracker ( ) [inline]

Destructor. Automatically calls the provided destroy() function.
Member Function Documentation

**void nite::UserTracker::addNewFrameListener (NewFrameListener**

 Adds a NewFrameListener object to this **UserTracker** so that it will respond when a new frame is generated.

**Parameters:**

- **[in]** `pListener` Pointer to a listener to add.

**See also:**

- **UserTracker::NewFrameListener** for more information on using event based interaction with **UserTracker**

**Status nite::UserTracker::convertDepthCoordinatesToJoint (int**

 In general, two coordinate systems are used in OpenNI 2.0. These conventions are also followed in **NiTE** 2.0.

**Skeleton** joint positions are provided in "Real World" coordinates, while the native coordinate system of depth maps is the "projective" system. In short, "Real World" coordinates locate objects using a Cartesian coordinate system with the origin at the sensor. "Projective" coordinates measure straight line distance from the sensor, and indicate x/y coordinates using pixels in the image (which is mathematically equivalent to specifying angles). See the OpenNI 2.0 documentation online for more information.

This function allows you to convert the native depth map coordinates
to the system used by the joints. This might be useful for performing certain types of measurements (eg distance between a joint and an object identified only in the depth map).

Note that no output is given for the Z coordinate. Z coordinates remain the same when performing the conversion. An input value is still required for Z, since this can affect the x/y output.

**Parameters:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>x</td>
<td>The input X coordinate using the &quot;projective&quot; coordinate system.</td>
</tr>
<tr>
<td>[in]</td>
<td>y</td>
<td>The input Y coordinate using the &quot;projective&quot; coordinate system.</td>
</tr>
<tr>
<td>[in]</td>
<td>z</td>
<td>The input Z coordinate using the &quot;projective&quot; coordinate system.</td>
</tr>
<tr>
<td>[out]</td>
<td>pOutX</td>
<td>Pointer to a location to store the output X coordinate in the &quot;real world&quot; system.</td>
</tr>
<tr>
<td>[out]</td>
<td>pOutY</td>
<td>Pointer to a location to store the output Y coordinate in the &quot;real world&quot; system.</td>
</tr>
</tbody>
</table>

**Returns:**

`Status` indicating success or failure of this operation. This is needed because the ability to convert between coordinate systems requires a properly initialized Device from OpenNI 2.0.

```cpp
Status nite::UserTracker::convertJointCoordinatesToDepth ( float x, float y, float z, float * pOutX, float * pOutY )
```

In general, two coordinate systems are used in OpenNI 2.0. These conventions are also followed in NiTE 2.0.
**Skeleton** joint positions are provided in "Real World" coordinates, while the native coordinate system of depth maps is the "projective" system. In short, "Real World" coordinates locate objects using a Cartesian coordinate system with the origin at the sensor. "Projective" coordinates measure straight line distance from the sensor (perpendicular to the sensor face), and indicate x/y coordinates using pixels in the image (which is mathematically equivalent to specifying angles). See the OpenNI 2.0 documentation online for more information.

Note that no output is given for the Z coordinate. Z coordinates remain the same when performing the conversion. An input value is still required for Z, since this can affect the x/y output.

This function allows you to convert the coordinates of a **SkeletonJoint** to the native coordinates of a depth map. This is useful if you need to find the joint position on the raw depth map.

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in] x</td>
<td>The input X coordinate using the &quot;real world&quot; coordinate system.</td>
</tr>
<tr>
<td>[in] y</td>
<td>The input Y coordinate using the &quot;real world&quot; coordinate system.</td>
</tr>
<tr>
<td>[in] z</td>
<td>The input Z coordinate using the &quot;real world&quot; coordinate system.</td>
</tr>
<tr>
<td>[out] pOutX</td>
<td>Pointer to a location to store the output X coordinate in the &quot;projective&quot; system.</td>
</tr>
<tr>
<td>[out] pOutY</td>
<td>Pointer to a location to store the output Y coordinate in the &quot;projective&quot; system.</td>
</tr>
</tbody>
</table>

**Returns:**

**Status** indicating success or failure of this operation. This is needed because the ability to convert between coordinate systems requires a properly initialized Device from OpenNI 2.0.

```c++
nite::UserTracker::create ( openni::Device * pDevice = NULL )
```
Creates and initializes an empty User Tracker. This function should be the first one called when a new `UserTracker` object is constructed.

An OpenNI device with depth capabilities is required for this algorithm to work. See the OpenNI 2.0 documentation for more information about using an OpenNI 2.0 compliant hardware device and creating a Device object.

**Parameters:**

- `[in] pDevice` A pointer to an initialized OpenNI 2.0 Device object that provides depth streams.

**Returns:**

A status code to indicate success/failure. Since this relies on an external hardware device, it is important for applications to check this value.

**See also:**

`Status` enumeration for a list of all possible status values generated by this call.

```cpp
tvoid nite::UserTracker::destroy ( ) [inline]
```

Shuts down the user tracker and releases all resources used by it.

This is the opposite of `create()`. This function is called automatically by the destructor in the current implementation, but it is good practice to run it manually when the algorithm is no longer required. Running this function more than once is safe -- it simply exits if called on a non-valid `UserTracker`.

```cpp
tfloat nite::UserTracker::getSkeletonSmoothingFactor ( ) const [in]
```

Queries the current skeleton smoothing factor.
Returns:
Current skeleton smoothing factor.

See also:
setSkeletonSmoothingFactor for more information on the smoothing factor, and the means to change it.

bool nite::UserTracker::isValid ( ) const [inline]

Indicates whether the UserTracker is valid.

When a new UserTracker is first constructed, this function will indicate that it is invalid (ie return False). Once the create() function has been successfully called, then this function will return True. If the destroy() function is called, this function will again indicate invalid.

It is safe to run create() and destroy() without calling this function -- both of those functions already check this value and return without doing anything if no action is required.

Returns:
True if the UserTracker object is correctly initialized, False otherwise.

See also:
create() function -- causes the UserTracker to become initialized.
destroy() function -- causes the UserTracker to become uninitialized.

Status nite::UserTracker::readFrame (UserTrackerFrameRef * pFrame)

Gets the next snapshot of the algorithm. This causes all data to be generated for the next frame of the algorithm -- algorithm frames correspond to the input depth frames used to generate them.
Parameters:

*pFrame* [out] A pointer that will be set to point to the next frame of data.

Returns:

Status code indicating whether this operation was successful.

---

**void nite::UserTracker::removeNewFrameListener** *(NewFrameListener)*

Removes a *NewFrameListener* object from this UserTracker's list of listeners. The listener will no longer respond when a new frame is generated.

Parameters:

* [in] *pListener* Pointer to a listener to remove.

See also:

*UserTracker::NewFrameListener* for more information on using event based interaction with *UserTracker*.

---

**Status nite::UserTracker::setSkeletonSmoothingFactor** *(float factor)*

Control the smoothing factor of the skeleton joints. Factor should be between 0 (no smoothing at all) and 1 (no movement at all).

Experimenting with this factor should allow you to fine tune the skeleton performance. Higher values will produce smoother operation of the skeleton, but may make the skeleton feel less responsive to the user.

Parameters:

* [in] *factor* The smoothing factor.

Returns:

Status code indicating success or failure of this operation.
**Status nite::UserTracker::startPoseDetection** (UserId user, PoseType type)

This function commands the **UserTracker** to start detecting specific poses for a specific user.

**Parameters:**
- [in] **user** The UserID of the user that you would like to detect a pose for.
- [in] **type** The type of pose you would like to detect.

**Returns:**
**Status** code indicating success or failure of this operation.

**See also:**
- **PoseData** For more information on pose detection and the output it generates.
- **PoseType** enumeration for a list of the available poses that can be detected.

**Status nite::UserTracker::startSkeletonTracking** (UserId id)

Requests that the **Skeleton** algorithm starts tracking a specific user. Once started, the skeleton will provide information on the joint position and orientation for that user during each new frame of the **UserTracker**.

Note that the computational requirements of calculating a skeleton increase linearly with the number of users tracked. Tracking too many users may result in poor performance and high CPU utilization. If performance slows to the point where the skeleton is not calculated at the full frame rate of the depth data used to generate it, the algorithm tends to perform poorly.
Parameters:

[in] **UserID** The ID number of the user to calculate a skeleton for.

Returns:

Status code indicating success or failure of this operation.

See also:

`nite::Skeleton` for more information on the skeleton algorithm.

```cpp
void nite::UserTracker::stopPoseDetection (UserId user,
                                           PoseType type)
```

This function commands the pose detection algorithm to stop detecting a specific pose for a specific user. Since it is possible to detect multiple poses from multiple users, it is possible that detection of a different pose on the same user (or the same pose on a different user) may continue after this function is called.

Parameters:

[in] **user** The UserID of the user to stop detecting a specific pose for.

[in] **type** The `PoseType` of the pose to stop detecting.

```cpp
void nite::UserTracker::stopSkeletonTracking (UserId id)
```

Stops skeleton tracking for a specific user. If multiple users are being tracked, this will only stop tracking for the user specified -- skeleton calculation will continue for remaining users.

Parameters:

[in] **UserID** of the person to stop tracking.
See also: `nite::Skeleton` for more information on the skeleton algorithm.

The documentation for this class was generated from the following file:

- `NiTE.h`

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
#include <NiTE.h>

List of all members.
Public Member Functions

<table>
<thead>
<tr>
<th>Function/Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserTrackerFrameRef ()</td>
</tr>
<tr>
<td>UserTrackerFrameRef (const UserTrackerFrameRef &amp;other)</td>
</tr>
<tr>
<td>~UserTrackerFrameRef ()</td>
</tr>
<tr>
<td>openni::VideoFrameRef getDepthFrame ()</td>
</tr>
<tr>
<td>const Plane &amp; getFloor () const</td>
</tr>
<tr>
<td>float getFloorConfidence () const</td>
</tr>
<tr>
<td>int getFrameIndex () const</td>
</tr>
<tr>
<td>uint64_t getTimestamp () const</td>
</tr>
<tr>
<td>const UserData * getUserById (UserId id) const</td>
</tr>
<tr>
<td>const UserMap &amp; getUserMap () const</td>
</tr>
<tr>
<td>const Array&lt;UserData&gt; &amp; getUsers () const</td>
</tr>
<tr>
<td>bool isValid () const</td>
</tr>
<tr>
<td>UserTrackerFrameRef &amp; operator= (const UserTrackerFrameRef &amp;other)</td>
</tr>
<tr>
<td>void release ()</td>
</tr>
</tbody>
</table>
### Friends

<table>
<thead>
<tr>
<th>class</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>UserTracker</td>
</tr>
</tbody>
</table>
Detailed Description

This class stores a snapshot of a single frame of output from the UserTracker algorithm. This frame will correspond to a single input depth frame. It holds all info on users, as well as information on the floor plane.

See also:
- **UserData** For detailed info on specific users
- **Skeleton** For user skeleton data
- **UserMap** For user positions in the frame
- **UserTracker** For information on starting the user tracker algorithms to detect all of this data.
- **HandTrackerFrameRef** for the equivalent information when hand tracking.
Constructor & Destructor Documentation

nite::UserTrackerFrameRef::UserTrackerFrameRef ( ) [inline]

Default Constructor. Creates an empty UserTrackerFrameRef.

nite::UserTrackerFrameRef::~UserTrackerFrameRef ( ) [inline]

Destructor. Frees the resources used by a UserTrackerFrameRef.

nite::UserTrackerFrameRef::UserTrackerFrameRef ( const UserTrackerFrameRef & ) [inline]

Copy Constructor. Creates a new UserTrackerFrameRef identical to an existing one.
Member Function Documentation

openni::VideoFrameRef nite::UserTrackerFrameRef::getDepthFrame()

Get the raw depth frame that originated this output. Each UserTracker frame is generated to correspond with a single depth frame. This function gives you access to the raw frame data.

See the OpenNI 2.0 documentation for information on manipulating VideoFrameRef data.

**Returns:**

VideoFrameRef with the raw depth that generated this frame.

const Plane& nite::UserTrackerFrameRef::getFloor() const [inline]

Obtains an estimate of the floor plane of the scene currently in view.

**Returns:**

Plane that indicates a best guess of where the floor of the scene is.

**See also:**

getFloorConfidence() for a confidence value in this calculation.

float nite::UserTrackerFrameRef::getFloorConfidence() const [inline]

NiTE generates a confidence value between 0 and 1 for the floor plane calculation. Lower values indicate less confidence.

**Returns:**

Value between 0 and 1 indicating confidence in the accuracy of the floor plane.
See also:

`getFloor()` to obtain the actual plane.

```cpp
int nite::UserTrackerFrameRef::getFrameIndex ( ) const [inline]
```

Gets the frame index of the raw depth frame that generated this `UserTracker` frame. Frame indexes are consecutive integer numbers assigned to frames by OpenNI. See the OpenNI documentation for more information.

**Returns:**

FrameIndex of the depth frame used to generate this User Tracker frame.

```cpp
uint64_t nite::UserTrackerFrameRef::getTimestamp ( ) const [inline]
```

Get the timestamp in which this frame was processed. See the OpenNI 2.0 documentation for more information on how OpenNI assigns time stamps to depth frames.

**Returns:**

Timestamp in microseconds

```cpp
const UserData* nite::UserTrackerFrameRef::getUserById ( UserId
```

Provides access to the `UserData` object for a specific user, indexed by `UserId`.

**Parameters:**

- `@ref` `UserId` The ID of the user you would like data from.

**Returns:**

Pointer to the corresponding `UserData` object.
See also:

- **UserData** for the information contained in this object.
- **UserMap** if what you really want is to know where in the image the user is.

```cpp
const UserMap& nite::UserTrackerFrameRef::getUserMap() const
```

Get the segmentation of the scene. The **UserTracker** algorithm locates each user in the field of view, and figures out which pixels correspond to which user, as well as which pixels correspond to the background (ie, not to any user).

**Returns:**

- **UserMap** showing which pixels are occupied by what users.

```cpp
const Array<UserData>& nite::UserTrackerFrameRef::getUsers() const
```

Get an **Array** of all the users available in this frame.

**Returns:**

- **Array** of **UserData** with one entry for each user in this frame.

```cpp
bool nite::UserTrackerFrameRef::isValid() const [inline]
```

Indicates whether this **UserTrackerFrameRef** points to valid frame data. When first constructed, the **UserTrackerFrameRef** will be invalid until assigned a frame handle.

**Returns:**

- True if this object contains valid data, False otherwise.
Impliments the assignment operator for the `UserTrackerFrameRef`.

**Parameters:**

- `[in] other` *UserTrackerFrameRef* to be assigned to this one.

```cpp
void nite::UserTrackerFrameRef::release () [inline]
```

Properly releases all resources used by this object. This will be automatically called by the destructor, but it is good practice to call this as soon as the object is no longer required. Calling this function multiple times on the same object is safe.
Friends And Related Function Documentation

friend class User [friend]

friend class UserTracker [friend]

The documentation for this class was generated from the following file:

- NiTE.h
## Class Index

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>G</th>
<th>H</th>
<th>N</th>
<th>P</th>
<th>Q</th>
<th>S</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NiTE (nite)</strong></td>
<td><strong>NiteBoundingBox</strong></td>
<td><strong>NitePlane</strong></td>
<td><strong>NitePoint3f</strong></td>
<td><strong>HandTrackerFrameRef (nite)</strong></td>
<td><strong>UserTracker::NewFrameListener (nite)</strong></td>
<td><strong>Plane (nite)</strong></td>
<td><strong>Point3f (nite)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Array (nite)</strong></td>
<td><strong>HandData (nite)</strong></td>
<td><strong>HandTracker (nite)</strong></td>
<td><strong>BoundingBox (nite)</strong></td>
<td><strong>HandTracker::NewFrameListener (nite)</strong></td>
<td><strong>GestureData (nite)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This inheritance list is sorted roughly, but not completely, alphabetically:

- nite::Array\< T \>
- nite::HandTracker
- nite::HandTrackerFrameRef
- nite::HandTracker::NewFrameListener
- nite::UserTracker::NewFrameListener
- nite::NiTE
- NiteBoundingBox
  - nite::BoundingBox
- NiteGestureData
  - nite::GestureData
- NiteHandData
  - nite::HandData
- NitePlane
  - nite::Plane
- NitePoint3f
  - nite::Point3f
- NitePoseData
  - nite::PoseData
- NiteQuaternion
  - nite::Quaternion
- NiteSkeleton
  - nite::Skeleton
- NiteSkeletonJoint
  - nite::SkeletonJoint
- **NiteUserData**
  - `nite::UserData`
- **NiteUserMap**
  - `nite::UserMap`
- `nite::UserTracker`
- `nite::UserTrackerFrameRef`
Here is a list of all class members with links to the classes they belong to:

- **a** -
  - addNewFrameListener() : nite::UserTracker, nite::HandTracker
  - Array() : nite::Array< T >

- **b** -
  - BoundingBox() : nite::BoundingBox

- **c** -
  - convertDepthCoordinatesToHand() : nite::HandTracker
  - convertDepthCoordinatesToJoint() : nite::UserTracker
  - convertHandCoordinatesToDepth() : nite::HandTracker
  - convertJointCoordinatesToDepth() : nite::UserTracker
  - create() : nite::UserTracker, nite::HandTracker

- **d** -
  - destroy() : nite::UserTracker, nite::HandTracker

- **g** -
  - getBoundingBox() : nite::UserData
  - getCenterOfMass() : nite::UserData
  - getCurrentPosition() : nite::GestureData
  - getDepthFrame() : nite::UserTrackerFrameRef,
nite::HandTrackerFrameRef

- getFloor() : nite::UserTrackerFrameRef
- getFloorConfidence() : nite::UserTrackerFrameRef
- getFrameIndex() : nite::UserTrackerFrameRef, nite::HandTrackerFrameRef
- getGestures() : nite::HandTrackerFrameRef
- getHands() : nite::HandTrackerFrameRef
- getHeight() : nite::UserMap
- getId() : nite::UserData, nite::HandData
- getJoint() : nite::Skeleton
- getOrientation() : nite::SkeletonJoint
- getOrientationConfidence() : nite::SkeletonJoint
- getPixels() : nite::UserMap
- getPose() : nite::UserData
- getPosition() : nite::SkeletonJoint, nite::HandData
- getPositionConfidence() : nite::SkeletonJoint
- getSize() : nite::Array< T >
- getSkeleton() : nite::UserData
- getSkeletonSmoothingFactor() : nite::UserTracker
- getSmoothingFactor() : nite::HandTracker
- getState() : nite::Skeleton
- getStride() : nite::UserMap
- getTimestamp() : nite::UserTrackerFrameRef, nite::HandTrackerFrameRef
- getType() : nite::SkeletonJoint, nite::PoseData, nite::GestureData
- getUserById() : nite::UserTrackerFrameRef
- getUserMap() : nite::UserTrackerFrameRef
- getUsers() : nite::UserTrackerFrameRef
- getVersion() : nite::NiTE
- getWidth() : nite::UserMap

- h -

- HandTracker : nite::HandTrackerFrameRef, nite::HandTracker::NewFrameListener, nite::HandTracker
- HandTrackerFrameRef() : nite::HandTrackerFrameRef
- i -

- initialize() : nite::NiTE
- isComplete() : nite::GestureData
- isEmpty() : nite::Array< T >
- isEntered() : nite::PoseData
- isExited() : nite::PoseData
- isHeld() : nite::PoseData
- isInProgress() : nite::GestureData
- isLost() : nite::UserData, nite::HandData
- isNew() : nite::HandData, nite::UserData
- isTouchingFov() : nite::HandData
- isTracking() : nite::HandData
- isValid() : nite::HandTrackerFrameRef, nite::UserTracker, nite::HandTracker, nite::UserTrackerFrameRef
- isVisible() : nite::UserData

- m -

- max : NiteBoundingBox
- min : NiteBoundingBox

- n -

- NewFrameListener() : nite::UserTracker::NewFrameListener, nite::HandTracker::NewFrameListener
- normal : NitePlane

- o -

- onNewFrame() : nite::UserTracker::NewFrameListener, nite::HandTracker::NewFrameListener
- operator!=() : nite::Point3f
- operator=() : nite::Point3f, nite::UserTrackerFrameRef, nite::HandTrackerFrameRef
- operator==() : nite::Point3f
- operator[]() : nite::Array< T >
- p -
  - Plane() : nite::Plane
  - point : NitePlane
  - Point3f() : nite::Point3f

- q -
  - Quaternion() : nite::Quaternion

- r -
  - readFrame() : nite::UserTracker , nite::HandTracker
  - release() : nite::HandTrackerFrameRef , nite::UserTrackerFrameRef
  - removeNewFrameListener() : nite::HandTracker , nite::UserTracker

- s -
  - set() : nite::Point3f
  - setData() : nite::Array< T >
  - setSkeletonSmoothingFactor() : nite::UserTracker
  - setSmoothingFactor() : nite::HandTracker
  - shutdown() : nite::NiTE
  - startGestureDetection() : nite::HandTracker
  - startHandTracking() : nite::HandTracker
  - startPoseDetection() : nite::UserTracker
  - startSkeletonTracking() : nite::UserTracker
  - stopGestureDetection() : nite::HandTracker
  - stopHandTracking() : nite::HandTracker
  - stopPoseDetection() : nite::UserTracker
  - stopSkeletonTracking() : nite::UserTracker

- u -
  - User : nite::UserTrackerFrameRef
  - UserTracker : nite::UserTrackerFrameRef , nite::UserTracker , nite::UserTracker::NewFrameListener
- UserTrackerFrameRef() : nite::UserTrackerFrameRef, nite::UserMap, nite::UserTrackerFrameRef

- w -

  - w : NiteQuaternion

- x -

  - x : NitePoint3f, NiteQuaternion

- y -

  - y : NitePoint3f, NiteQuaternion

- z -

  - z : NitePoint3f, NiteQuaternion

- ~ -

  - ~HandTracker() : nite::HandTracker
  - ~HandTrackerFrameRef() : nite::HandTrackerFrameRef
  - ~UserTracker() : nite::UserTracker
  - ~UserTrackerFrameRef() : nite::UserTrackerFrameRef
Here is a list of all namespaces with brief descriptions:

| nite |  |

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
<table>
<thead>
<tr>
<th>Main Page</th>
<th>Namespaces</th>
<th>Classes</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace List</td>
<td>Namespace Members</td>
<td>Classes</td>
<td>Typedefs</td>
</tr>
</tbody>
</table>

**nite Namespace Reference**
## Classes

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Array</code></td>
</tr>
<tr>
<td><code>BoundingBox</code></td>
</tr>
<tr>
<td><code>GestureData</code></td>
</tr>
<tr>
<td><code>HandData</code></td>
</tr>
<tr>
<td><code>HandTracker</code></td>
</tr>
<tr>
<td><code>HandTrackerFrameRef</code></td>
</tr>
<tr>
<td><code>NiTE</code></td>
</tr>
<tr>
<td><code>Plane</code></td>
</tr>
<tr>
<td><code>Point3f</code></td>
</tr>
<tr>
<td><code>PoseData</code></td>
</tr>
<tr>
<td><code>Quaternion</code></td>
</tr>
<tr>
<td><code>Skeleton</code></td>
</tr>
<tr>
<td><code>SkeletonJoint</code></td>
</tr>
<tr>
<td><code>UserData</code></td>
</tr>
<tr>
<td><code>UserMap</code></td>
</tr>
<tr>
<td><code>UserTracker</code></td>
</tr>
<tr>
<td><code>UserTrackerFrameRef</code></td>
</tr>
</tbody>
</table>
### Typedefs

<table>
<thead>
<tr>
<th>Typedef</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef short int</td>
<td><strong>HandId</strong></td>
<td></td>
</tr>
<tr>
<td>typedef short int</td>
<td><strong>UserId</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Enumerations

<table>
<thead>
<tr>
<th>Enum</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GestureType</strong></td>
<td>GESTURE_WAVE, GESTURE_CLICK, GESTURE_HAND_RAISE</td>
</tr>
<tr>
<td><strong>JointType</strong></td>
<td>JOINT_HEAD, JOINT_NECK, JOINT_LEFT_SHOULDER, JOINT_RIGHT_SHOULDER,</td>
</tr>
<tr>
<td></td>
<td>JOINT_LEFT Elbow, JOINT_RIGHT Elbow, JOINT_LEFT Hand,</td>
</tr>
<tr>
<td></td>
<td>JOINT_RIGHT Hand, JOINT_TORSO, JOINT_LEFT HIP, JOINT_RIGHT HIP,</td>
</tr>
<tr>
<td></td>
<td>JOINT_LEFT KNEE, JOINT_RIGHT KNEE, JOINT_LEFT FOOT,</td>
</tr>
<tr>
<td></td>
<td>JOINT_RIGHT FOOT</td>
</tr>
<tr>
<td><strong>PoseType</strong></td>
<td>POSE_PSI, POSE_CROSSED_HANDS</td>
</tr>
<tr>
<td><strong>SkeletonState</strong></td>
<td>SKELETON_NONE, SKELETON_CALIBRATING, SKELETON_TRACKED,</td>
</tr>
<tr>
<td></td>
<td>SKELETON_CALIBRATION_ERROR_NOT_IN_POSE, SKELETON_CALIBRATION_ERROR_HIPS,</td>
</tr>
<tr>
<td></td>
<td>SKELETON_CALIBRATION_ERROR_HEAD, SKELETON_CALIBRATION_ERROR_LEGS,</td>
</tr>
<tr>
<td></td>
<td>SKELETON_CALIBRATION_ERROR_TORSO</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>STATUS_OK, STATUS_ERROR, STATUS_BAD_USER_ID, STATUS_OUT_OF_FLOW</td>
</tr>
</tbody>
</table>
Typedef Documentation

typedef short int nite::HandId

typedef short int nite::UserId

UserId is a persistent ID for a specific user detected by the UserTracker algorithm. While the user is known, it will have the same ID.
# Enumeration Type Documentation

**enum nite::GestureType**

Available gestures types

**Enumerator:**

- GESTURE_WAVE
- GESTURE_CLICK
- GESTURE_HAND_RAISE

**enum nite::JointType**

Available joints in skeleton

**Enumerator:**

- JOINT_HEAD
- JOINT_NECK
- JOINT_LEFT_SHOULDER
- JOINT_RIGHT_SHOULDER
- JOINT_LEFT_ELBOW
- JOINT_RIGHT_ELBOW
- JOINT_LEFT_HAND
- JOINT_RIGHT_HAND
- JOINT_TORSO
- JOINT_LEFT_HIP
- JOINT_RIGHT_HIP
- JOINT_LEFT_KNEE
- JOINT_RIGHT_KNEE
- JOINT_LEFT FOOT
- JOINT_RIGHT FOOT
### enum nite::PoseType

**Enumerator:**
- `POSE_PSI`
- `POSE_CROSSED_HANDS`

### enum nite::SkeletonState

Possible states of the skeleton

**Enumerator:**
- `SKELETON_NONE`
  - No skeleton - skeleton was not requested
- `SKELETON_CALIBRATING`
  - Skeleton requested but still unavailable
- `SKELETON_TRACKED`
  - Skeleton available
- `SKELETON_CALIBRATION_ERROR_NOT_IN_POSE`
  - Possible reasons as to why skeleton is unavailable
- `SKELETON_CALIBRATION_ERROR_HANDS`
<table>
<thead>
<tr>
<th>SKELETON_CALIBRATION_ERROR_HEAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKELETON_CALIBRATION_ERROR_LEGS</td>
</tr>
<tr>
<td>SKELETON_CALIBRATION_ERROR_TORSO</td>
</tr>
</tbody>
</table>

```cpp
class nite::Status {
  enum Possible failure values
  {
    STATUS_OK,
    STATUS_ERROR,
    STATUS_BAD_USER_ID,
    STATUS_OUT_OF_FLOW,
  }
}
```

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by `doxygen` 1.7.5.1
Here is a list of all namespace members with links to the namespace documentation for each member:

- **g** -
  
  - GESTURE_CLICK : [nite](#)
  - GESTURE_HAND_RAISE : [nite](#)
  - GESTURE_WAVE : [nite](#)
  - GestureType : [nite](#)

- **h** -
  
  - HandId : [nite](#)

- **j** -
  
  - JOINT_HEAD : [nite](#)
  - JOINT_LEFT_ELBOw : [nite](#)
  - JOINT_LEFT_FOOT : [nite](#)
  - JOINT_LEFT_HAND : [nite](#)
  - JOINT_LEFT_HIP : [nite](#)
  - JOINT_LEFT_KNEE : [nite](#)
  - JOINT_LEFT_SHOULDER : [nite](#)
  - JOINT_NECK : [nite](#)
  - JOINT_RIGHT_ELBOw : [nite](#)
  - JOINT_RIGHT_FOOT : [nite](#)
  - JOINT_RIGHT_HAND : [nite](#)
  - JOINT_RIGHT_HIP : [nite](#)
  - JOINT_RIGHT_KNEE : [nite](#)
- p -

- s -

- u -
Here is a list of all files with brief descriptions:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NiTE.h</td>
<td>[code]</td>
</tr>
<tr>
<td>NiteCTypes.h</td>
<td>[code]</td>
</tr>
<tr>
<td>Nite Enums.h</td>
<td>[code]</td>
</tr>
</tbody>
</table>
NiTE.h File Reference

#include "NiteCAPI.h"  #include <OpenNI.h>
#include "NiteEnums.h"

Go to the source code of this file.
## Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nite::Array&lt;T&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::BoundingBox</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::GestureData</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::HandData</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::HandTracker</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::HandTrackerFrameRef</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::HandTracker::NewFrameListener</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::UserTracker::NewFrameListener</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::NiTE</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::Plane</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::Point3f</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::PoseData</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::Quaternion</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::Skeleton</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::SkeletonJoint</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::UserData</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::UserMap</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::UserTracker</code></td>
<td></td>
</tr>
<tr>
<td><code>nite::UserTrackerFrameRef</code></td>
<td></td>
</tr>
</tbody>
</table>
Namespaces

namespace nite
## Typedefs

<table>
<thead>
<tr>
<th>Typedef</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nite::HandId</code></td>
<td>typedef short int</td>
</tr>
<tr>
<td><code>nite::UserId</code></td>
<td>typedef short int</td>
</tr>
</tbody>
</table>
## Enumerations

```cpp
define enum nite::GestureType { nite::GESTURE_WAVE, nite::GESTURE_CLICK, nite::GESTURE_HAND_RAISE }

define enum nite::JointType {
  nite::JOINT_HEAD, nite::JOINT_NECK,
  nite::JOINT_LEFT_SHOULDER,
  nite::JOINT_RIGHT_SHOULDER,
  nite::JOINT_LEFT_ELBOW, nite::JOINT_RIGHT_ELBOW,
  nite::JOINT_LEFT_HAND, nite::JOINT_RIGHT_HAND,
  nite::JOINT_TORSO, nite::JOINT_LEFT_HIP,
  nite::JOINT_RIGHT_HIP, nite::JOINT_LEFT_KNEE,
  nite::JOINT_RIGHT_KNEE, nite::JOINT_LEFT_FOOT,
  nite::JOINT_RIGHT_FOOT
}

define enum nite::PoseType { nite::POSE_PSI, nite::POSE_CROSSED_HANDS }

define enum nite::SkeletonState {
  nite::SKELETON_NONE,
  nite::SKELETON_CALIBRATING,
  nite::SKELETON_TRACKED,
  nite::SKELETON_CALIBRATION_ERROR_NOT_IN_POSE,
  nite::SKELETON_CALIBRATION_ERROR_HANDS,
  nite::SKELETON_CALIBRATION_ERROR_HEAD,
  nite::SKELETON_CALIBRATION_ERROR_LEGS,
  nite::SKELETON_CALIBRATION_ERROR_TORSO
}

define enum nite::Status { nite::STATUS_OK, nite::STATUS_ERROR, nite::STATUS_BAD_USER_ID, nite::STATUS_OUT_OF_FLOW }
```
#include <OniCTypes.h> #include "NiteCEnums.h"

Go to the source code of this file.
Classes

<table>
<thead>
<tr>
<th>struct</th>
<th>NiteBoundingBox</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct</td>
<td>NitePlane</td>
</tr>
<tr>
<td>struct</td>
<td>NitePoint3f</td>
</tr>
<tr>
<td>struct</td>
<td>NiteQuaternion</td>
</tr>
</tbody>
</table>
## Defines

```
#define NITE_JOINT_COUNT 15
#define NITE_POSE_COUNT 2
```
## Typedefs

<table>
<thead>
<tr>
<th>Type Definition</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef short int</td>
<td>NiteHandId</td>
</tr>
<tr>
<td>typedef struct NiteHandTracker *</td>
<td>NiteHandTrackerHandle</td>
</tr>
<tr>
<td>typedef short int</td>
<td>NiteUserId</td>
</tr>
<tr>
<td>typedef struct NiteUserTracker *</td>
<td>NiteUserTrackerHandle</td>
</tr>
</tbody>
</table>
Define Documentation

#define NITE_JOINT_COUNT 15

#define NITE_POSE_COUNT 2
Typedef Documentation

typedef short int NiteHandId

typedef struct NiteHandTracker* NiteHandTrackerHandle

typedef short int NiteUserId

typedef struct NiteUserTracker* NiteUserTrackerHandle
NiteEnums.h File
Reference

Go to the source code of this file.
## Enumerations

<table>
<thead>
<tr>
<th><code>enum</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GestureType</code></td>
<td>GESTURE_WAVE, GESTURE_CLICK, GESTURE_HAND_RAISE</td>
</tr>
<tr>
<td><code>JointType</code></td>
<td>JOINT_HEAD, JOINT_NECK, JOINT_LEFT_SHoulder, JOINT_RIGHT_SHoulder, JOINT_LEFT_Elbow, JOINT_RIGHT_Elbow, JOINT_LEFT_HAND, JOINT_RIGHT_HAND, JOINT_TORSO, JOINT_LEFT_Hip, JOINT_RIGHT_Hip, JOINT_LEFT_Knee, JOINT_RIGHT_Knee, JOINT_LEFT_Foot, JOINT_RIGHT_Foot</td>
</tr>
<tr>
<td><code>PoseType</code></td>
<td>POSE_PSI, POSE_CROSSED_HANDS</td>
</tr>
<tr>
<td><code>SkeletonState</code></td>
<td>SKELETON_NONE, SKELETON_CALIBRATING, SKELETON_TRACKED, SKELETON_CALIBRATION_ERROR_NOT_IN_POSE, SKELETON_CALIBRATION_ERROR_HANDS, SKELETON_CALIBRATION_ERROR_HEAD, SKELETON_CALIBRATION_ERROR_LEGS, SKELETON_CALIBRATION_ERROR_TORSO</td>
</tr>
<tr>
<td><code>Status</code></td>
<td>STATUS_OK, STATUS_ERROR, STATUS_BAD_USER_ID, STATUS_OUT_OF_FLOW</td>
</tr>
</tbody>
</table>
Enumeration Type Documentation

enum GestureType

Available gestures types

**Enumerator:**

- GESTURE_WAVE
- GESTURE_CLICK
- GESTURE_HAND_RAISE

enum JointType

Available joints in skeleton

**Enumerator:**

- JOINT_HEAD
- JOINT_NECK
- JOINT_LEFT_SHOULDER
- JOINT_RIGHT_SHOULDER
- JOINT_LEFT_ELBOW
- JOINT_RIGHT_ELBOW
- JOINT_LEFT_HAND
- JOINT_RIGHT_HAND
- JOINT_TORSO
- JOINT_LEFT_HIP
- JOINT_RIGHT_HIP
- JOINT_LEFT_KNEE
- JOINT_RIGHT_KNEE
- JOINT_LEFT_FOOT
- JOINT_RIGHT_FOOT
### enum PoseType

**Enumerator:**
- `POSE_PSI`
- `POSE_CROSSED_HANDS`

### enum SkeletonState

Possible states of the skeleton

**Enumerator:**
- `SKELETON_NONE`  
  No skeleton - skeleton was not requested
- `SKELETON_CALIBRATING`  
  Skeleton requested but still unavailable
- `SKELETON_TRACKED`  
  Skeleton available
- `SKELETON_CALIBRATION_ERROR_NOT_IN_POSE`  
  Possible reasons as to why skeleton is unavailable
- `SKELETON_CALIBRATION_ERROR_HANDS`
enum Status

Possible failure values

**Enumerator:**

- STATUS_OK
- STATUS_ERROR
- STATUS_BAD_USER_ID
- STATUS_OUT_OF_FLOW
Here is a list of all file members with links to the files they belong to:

- **g** -
  
  - GESTURE_CLICK : Nite Enums.h
  - GESTURE_HAND_RAISE : Nite Enums.h
  - GESTURE_WAVE : Nite Enums.h
  - GestureType : Nite Enums.h

- **j** -
  
  - JOINT_HEAD : Nite Enums.h
  - JOINT_LEFT_ELBOW : Nite Enums.h
  - JOINT_LEFT_FOOT : Nite Enums.h
  - JOINT_LEFT_HAND : Nite Enums.h
  - JOINT_LEFT_HIP : Nite Enums.h
  - JOINT_LEFT_KNEE : Nite Enums.h
  - JOINT_LEFT_SHOULDER : Nite Enums.h
  - JOINT_NECK : Nite Enums.h
  - JOINT_RIGHT_ELBOW : Nite Enums.h
  - JOINT_RIGHT_FOOT : Nite Enums.h
  - JOINT_RIGHT_HAND : Nite Enums.h
  - JOINT_RIGHT_HIP : Nite Enums.h
  - JOINT_RIGHT_KNEE : Nite Enums.h
  - JOINT_RIGHT_SHOULDER : Nite Enums.h
  - JOINT_TORSO : Nite Enums.h
  - JointType : Nite Enums.h
- NITE_JOINT_COUNT : NiteCTypes.h
- NITE_POSE_COUNT : NiteCTypes.h
- NiteHandId : NiteCTypes.h
- NiteHandTrackerHandle : NiteCTypes.h
- NiteUserId : NiteCTypes.h
- NiteUserTrackerHandle : NiteCTypes.h

- p -

- POSE_CROSSED_HANDS : NiteEnums.h
- POSE_PSI : NiteEnums.h
- PoseType : NiteEnums.h

- s -

- SKELETON_CALIBRATING : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_HANDS : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_HEAD : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_LEGS : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_NOT_IN_POSE : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_TORSO : NiteEnums.h
- SKELETON_NONE : NiteEnums.h
- SKELETON_TRACKED : NiteEnums.h
- SkeletonState : NiteEnums.h
- Status : NiteEnums.h
- STATUS_BAD_USER_ID : NiteEnums.h
- STATUS_ERROR : NiteEnums.h
- STATUS_OK : NiteEnums.h
- STATUS_OUT_OF_FLOW : NiteEnums.h
Go to the textual class hierarchy

nite::Array<T>

NiteGestureData

NiteHandData

nite::HandTracker

nite::HandTracker::NewFrameListener

nite::HandTrackerFrameRef

nite::NITE

NitePoseData

NiteSkeleton

NiteSkeletonJoint

NiteUserData

NiteUserMap

nite::UserTracker

nite::UserTracker::NewFrameListener

nite::UserTrackerFrameRef
#ifndef _NITE_H_
#define _NITE_H_

#include "NiteCAPI.h"
#include <OpenNI.h>

// Summary of use cases, modules, facades
namespace nite {
#ifndef _NITE_DECLARE_VERSION(Version)

_NITE_DECLARE_VERSION(Version);

#include "NiteEnums.h"

// General
_NITE_DECLARE_VERSION(Version);
class Point3f : public NitePoint3f
{
public:
    Point3f()
    {
        x = y = z = 0.0f;
    }
    Point3f(float x, float y, float z)
    {
        this->set(x, y, z);
    }
    Point3f(const Point3f& other)
    {
        *this = other;
    }
    void set(float x, float y, float z)
    {
        this->x = x;
        this->y = y;
        this->z = z;
    }
    Point3f& operator=(const Point3f& other)
    {
        set(other.x, other.y, other.z);
        return *this;
    }
    bool operator==(const Point3f& other) const
    {
        return x == other.x && y == other.y && z == other.z;
    }
bool operator!=(const Point3f& other) const
{
    return !operator==(other);
};

class Plane : public NitePlane
{
public:
    Plane()
    {
        this->point = Point3f();
        this->normal = Point3f();
    }

    Plane(const Point3f& point, const Point3f & normal)
    {
        this->point = point;
        this->normal = normal;
    }

    Quaternion Quaternion()
    {
        x = y = z = w = 0;
    }

    Quaternion(float w, float x, float y, float z)
    {

```cpp
this->x = x;
this->y = y;
this->z = z;
this->w = w;
}
}

class BoundingBox : public NiteBoundingBox {
public:
BoundingBox()
{}
BoundingBox(const Point3f& min, const Point3f& max)
{
    this->min = min;
    this->max = max;
}
};

template<class T>
class Array {
public:
Array() : m_size(0), m_data(NULL) {}
void setData(int size, T* data) {m_data = data; m_size = size;}
const T& operator[](int index) const {return m_data[index];}
int getSize() const {return m_size;}
bool isEmpty() const {return m_size == 0 ;}
private:
Array(const Array&);
Array& operator=(const Array&);
int m_size;

T* m_data;

};

// UserTracker
typedef short int UserId;

class PoseData : protected NitePoseData
{

public:

PoseType getType() const {return (PoseType)type;}

bool isHeld() const {return (state & NITE_POSE_STATE_IN_POSE) != 0;}

bool isEntered() const {return (state & NITE_POSE_STATE_ENTER) != 0;}

bool isExited() const {return (state & NITE_POSE_STATE_EXIT) != 0;}

};

class UserMap : private NiteUserMap
{

public:

const UserId* getPixels() const {return pixels;}

int getWidth() const {return width;}

int getHeight() const {return height;}

int getStride() const {return stride;}

friend class UserTrackerFrameRef;

};
class SkeletonJoint : private NiteSkeletonJoint
{
    public:
    JointType getType() const {return (JointType)jointType;}
    
    const Point3f& getPosition() const {return (Point3f&)position;}
    
    float getPositionConfidence() const {return positionConfidence;}

    const Quaternion& getOrientation() const {return (Quaternion&)orientation;}

    float getOrientationConfidence() const {return orientationConfidence;}
};

class Skeleton : private NiteSkeleton
{
    public:
    const SkeletonJoint& getJoint(JointType type) const {return (SkeletonJoint&)joints[type];}
    SkeletonState getState() const {return (SkeletonState)state;}
};

class UserData : private NiteUserData
{
    public:
    UserId getId() const {return id;}
    
    const BoundingBox& getBoundingBox() const {return (const BoundingBox&)boundingBox;}
}
    const Point3f& getCenterOfMass() const { return (const Point3f&)centerOfMass; }

    bool isNew() const { return (state & NITE_USER_STATE_NEW) != 0; }

    bool isVisible() const { return (state & NITE_USER_STATE_VISIBLE) != 0; }

    bool isLost() const { return (state & NITE_USER_STATE_LOST) != 0; }

    const Skeleton& getSkeleton() const { return (const Skeleton&)skeleton; }

    const PoseData& getPose(PoseType type) const { return (const PoseData&)poses[type]; }

    class UserTrackerFrameRef
    {
    public:

        UserTrackerFrameRef() : m_pFrame(NULL), m_userTrackerHandle(NULL) {}

        ~UserTrackerFrameRef() {
            release();
        }

        UserTrackerFrameRef(const UserTrackerFrameRef& other) : m_pFrame(NULL) {
            *this = other;
        }
    };
UserTrackerFrameRef& operator=(const UserTrackerFrameRef& other)
{
    setReference(other.m_userTrackerHandle, other.m_pFrame);
    niteUserTrackerFrameAddRef(m_userTrackerHandle, m_pFrame);
    return *this;
}

bool isValid() const
{
    return m_pFrame != NULL;
}

void release()
{
    if (m_pFrame != NULL)
    {
        niteUserTrackerFrameRelease(m_userTrackerHandle, m_pFrame);
    }
    m_pFrame = NULL;
    m_userTrackerHandle = NULL;
}

const UserData* getUserById(UserId id) const
{
    for (int i = 0; i < m_users.getSize(); ++i)
    {
        if (m_users[i].getId() == id)
        {
            return &m_users[i];
        }
    }
    return NULL;
}
const Array<UserData>& getUsers() const { return m_users; }

float getFloorConfidence() const { return m_pFrame->floorConfidence; }

const Plane& getFloor() const { return (const Plane&)m_pFrame->floor; }

openni::VideoFrameRef getDepthFrame() { return m_depthFrame; }

const UserMap& getUserMap() const {
    return static_cast<const UserMap&>(m_pFrame->userMap);
}

uint64_t getTimestamp() const { return m_pFrame->timestamp; }

int getFrameIndex() const { return m_pFrame->frameIndex; }

private:
    friend class User;
    friend class UserTracker;

    Array<UserData> m_users;

    void setReference(NiteUserTrackerHandle userTrackerHandle, NiteUserTrackerFrame* pFrame)
        {
            release();
            m_userTrackerHandle = userTrackerHandle;
        }
m_pFrame = pFrame;
m_depthFrame._setFrame(pFrame->pDept
hFrame);
m_users.setData(m_pFrame->userCount,
(UserData*)m_pFrame->pUser);
}
NiteUserTrackerFrame* m_pFrame;
NiteUserTrackerHandle m_userTrackerHandle;
openni::VideoFrameRef m_depthFrame;
};

class UserTracker
{
public:

    class NewFrameListener
    {
        public:

            NewFrameListener() : m_pUserTracker(NULL)
            {
                m_userTrackerCallbacks.readyForNextFrame = newFrameCallback;
            }

            virtual void onNewFrame(UserTracker& ) = 0;

private:

    NiteUserTrackerCallbacks m_userTrackerCallbacks;

    NiteUserTrackerCallbacks& getCallbacks() {return m_userTrackerCallbacks;}

static void ONI_CALLBACK_TYPE newFrameCallback(void* pCookie)
{
    NewFrameListener* pListener = (NewFrameListener*)pCookie;
    pListener->onNewFrame(*pListener->m_pUserTracker);
}

friend class UserTracker;

void setUserTracker(UserTracker* pUserTracker)
{
    m_pUserTracker = pUserTracker;
}

UserTracker* m_pUserTracker;

UserTracker() : m_userTrackerHandle(NULL)
{

}

~UserTracker()
{
    destroy();
}

Status create(openni::Device* pDevice = NULL)
{
    if (isValid())
    {
        // tracker already active
        return STATUS_OUT_OFFLOW;
    }
}
if (pDevice == NULL)
{
    return (Status)niteInitializeUserTracker(&m_userTrackerHandle);
}
return (Status)niteInitializeUserTrackerByDevice(pDevice, &m_userTrackerHandle);

void destroy()
{
    if (isValid())
    {
        niteShutdownUserTracker(m_userTrackerHandle);
        m_userTrackerHandle = NULL;
    }
}

Status readFrame(UserTrackerFrameRef* pFrame)
{
    NiteUserTrackerFrame *pNiteFrame = NULL;
    Status rc = (Status)niteReadUserTrackerFrame(m_userTrackerHandle, &pNiteFrame);
    pFrame->setReference(m_userTrackerHandle, pNiteFrame);
    return rc;
}

bool isValid() const
{
    return m_userTrackerHandle != NULL;
}
Status setSkeletonSmoothingFactor(float factor) {
    return (Status)niteSetSkeletonSmoothing(m_userTrackerHandle, factor);
}

float getSkeletonSmoothingFactor() const {
    float factor;
    Status rc = (Status)niteGetSkeletonSmoothing(m_userTrackerHandle, &factor);
    if (rc != STATUS_OK) {
        factor = 0;
    }
    return factor;
}

Status startSkeletonTracking(UserId id) {
    return (Status)niteStartSkeletonTracking(m_userTrackerHandle, id);
}

void stopSkeletonTracking(UserId id) {
    niteStopSkeletonTracking(m_userTrackerHandle, id);
}

Status startPoseDetection(UserId user, PoseType type) {
    return (Status)niteStartPoseDetection(m_userTrackerHandle, (NiteUserId)user, (NitePose
void stopPoseDetection(UserId user, PoseType type) {
    niteStopPoseDetection(m_userTrackerHandle, (NiteUserId)user, (NitePoseType)type);
}

void addNewFrameListener(NewFrameListener *pListener) {
    niteRegisterUserTrackerCallbacks(m_userTrackerHandle, &pListener->getCallbacks(), pListener);
    pListener->setUserTracker(this);
}

void removeNewFrameListener(NewFrameListener* pListener) {
    niteUnregisterUserTrackerCallbacks(m_userTrackerHandle, &pListener->getCallbacks());
    pListener->setUserTracker(NULL);
}

Status convertJointCoordinatesToDepth(float x, float y, float z, float* pOutX, float* pOutY) const {
    return (Status)niteConvertJointCoordinatesToDepth(m_userTrackerHandle, x, y, z, pOutX, pOutY);
}

Status convertDepthCoordinatesToJoint(int
x, int y, int z, float* pOutX, float* pOutY) const

    { return (Status)niteConvertDepthCoordinatesToJoint(m_userTrackerHandle, x, y, z, pOutX, pOutY); }

private:
    NiteUserTrackerHandle m_userTrackerHandle;
};

// HandTracker
typedef short int HandId;

class GestureData : protected NiteGestureData
{
    public:
        GestureType getType() const { return (GestureType)type; }

        const Point3f& getCurrentPosition() const { return (Point3f&)currentPosition; }

        bool isComplete() const { return (state & NITE_GESTURE_STATE_COMPLETED) != 0; }

        bool isInProgress() const { return (state & NITE_GESTURE_STATE_IN_PROGRESS) != 0; }
};

class HandData : protected NiteHandData
{
    public:

HandId getId() const { return id; }

const Point3f& getPosition() const { return (Point3f&)position; }

bool isNew() const { return (state & NITE_HAND_STATE_NEW) != 0; }

bool isLost() const { return state == NITE_HAND_STATE_LOST; }

bool isTracking() const { return (state & NITE_HAND_STATE_TRACKED) != 0; }

bool isTouchingFov() const { return (state & NITE_HAND_STATE_TOUCHING_FOV) != 0; }

class HandTrackerFrameRef
{
public:
    HandTrackerFrameRef() : m_pFrame(NULL), m_handTracker(NULL)
    {}

    ~HandTrackerFrameRef()
    {
        release();
    }

    HandTrackerFrameRef(const HandTrackerFrameRef& other) : m_pFrame(NULL)
    {
        *this = other;
    }

    HandTrackerFrameRef& operator=(const HandTrackerFrameRef& other) { return *this = other; }
};
dTrackerFrameRef& other) {
    setReference(other.m_handTracker, other.m_pFrame);
    niteHandTrackerFrameAddRef(m_handTracker, m_pFrame);
    return *this;
}

bool isValid() const {
    return m_pFrame != NULL;
}

void release() {
    if (m_pFrame != NULL) {
        niteHandTrackerFrameRelease(m_handTracker, m_pFrame);
    }
    m_pFrame = NULL;
    m_handTracker = NULL;
}

const Array<HandData>& getHands() const { return m_hands; }

const Array<GestureData>& getGestures() const { return m_gestures; }

openni::VideoFrameRef getDepthFrame() const {
    return m_depthFrame;
}
uint64_t getTimestamp() const {return m_pFrame->timestamp;}

int getFrameIndex() const {return m_pFrame->frameIndex;}

private:
friend class HandTracker;

void setReference(NiteHandTrackerHandle handTracker, NiteHandTrackerFrame* pFrame)
{
    release();
m_handTracker = handTracker;
m_pFrame = pFrame;
m_depthFrame._setFrame(pFrame->pDepthFrame);
m_hands.setData(m_pFrame->handCount, (HandData*)m_pFrame->pHands);
m_gestures.setData(m_pFrame->gestureCount, (GestureData*)m_pFrame->pGestures);
}

NiteHandTrackerFrame* m_pFrame;
NiteHandTrackerHandle m_handTracker;
openni::VideoFrameRef m_depthFrame;
Array<HandData> m_hands;
Array<GestureData> m_gestures;
};

class HandTracker
{
public:
    class NewFrameListener
    {
    
    
    
    
};

    class NewFrameListener
    {
    
    
    
    
};

    class HandTracker
    {
    
    
    
    
};
public:
    NewFrameListener() : m_pHandTracker(NULL)
    {
        m_handTrackerCallbacks.readyForNextFrame = newFrameCallback;
    }

    virtual void onNewFrame(HandTracker&) = 0;

private:
    friend class HandTracker;

    NiteHandTrackerCallbacks m_handTrackerCallbacks;

    NiteHandTrackerCallbacks& getCallbacks() {return m_handTrackerCallbacks;}

    static void ONI_CALLBACK_TYPE newFrameCallback(void* pCookie)
    {
        NewFrameListener* pListener = (NewFrameListener*)pCookie;
        pListener->onNewFrame(*pListener->m_pHandTracker);
    }

    void setHandTracker(HandTracker* pHandTracker)
    {
        m_pHandTracker = pHandTracker;
    }

    HandTracker* m_pHandTracker;

    HandTracker() : m_handTrackerHandle(NULL)
    {
    }
~HandTracker()
{
    destroy();
}

Status create(openni::Device* pDevice = NULL)
{
    if (isValid())
    {
        // tracker already active
        return STATUS_OUT_OF_FLOW;
    }
    if (pDevice == NULL)
    {
        return (Status)niteInitializeHandTracker(&m_handTrackerHandle);
        // Pick a device
    }
    return (Status)niteInitializeHandTrackerByDevice(pDevice, &m_handTrackerHandle);
}

void destroy()
{
    if (isValid())
    {
        niteShutdownHandTracker(m_handTrackerHandle);
        m_handTrackerHandle = NULL;
    }
}

Status readFrame(HandTrackerFrameRef* pFrame)
NiteHandTrackerFrame *pNiteFrame = NULL;

Status rc = (Status)niteReadHandTrackerFrame(m_handTrackerHandle, &pNiteFrame);
pFrame->setReference(m_handTrackerHandle, pNiteFrame);

return rc;

bool isValid() const
{
    return m_handTrackerHandle != NULL;
}

Status setSmoothingFactor(float factor)
{
    return (Status)niteSetHandSmoothingFactor(m_handTrackerHandle, factor);
}

float getSmoothingFactor() const
{
    float factor;
    Status rc = (Status)niteGetHandSmoothingFactor(m_handTrackerHandle, &factor);
    if (rc != STATUS_OK)
    {
        factor = 0;
    }
    return factor;
}

Status startHandTracking(const Point3f& position, HandId* pNewHandId)
{
return (Status)niteStartHandTracking(m_handTrackerHandle, (const NitePoint3f*)&position, pNewHandId);
}

void stopHandTracking(HandId id)
{
    niteStopHandTracking(m_handTrackerHandle, id);
}

void addNewFrameListener(NewFrameListener * pListener)
{
    niteRegisterHandTrackerCallbacks(m_handTrackerHandle, &pListener->getCallbacks(), pListener);
    pListener->setHandTracker(this);
}

void removeNewFrameListener(NewFrameListener* pListener)
{
    niteUnregisterHandTrackerCallbacks(m_handTrackerHandle, &pListener->getCallbacks());
    pListener->setHandTracker(NULL);
}

Status startGestureDetection(GestureType type)
{
    return (Status)niteStartGestureDetection(m_handTrackerHandle, (NiteGestureType)type);
}

void stopGestureDetection(GestureType type)
niteStopGestureDetection(m_handTrackerHandle, (NiteGestureType)type);

Status convertHandCoordinatesToDepth(float x, float y, float z, float* pOutX, float* pOutY) const
{
    return (Status)niteConvertHandCoordinatesToDepth(m_handTrackerHandle, x, y, z, pOutX, pOutY);
}

Status convertDepthCoordinatesToHand(int x, int y, int z, float* pOutX, float* pOutY) const
{
    return (Status)niteConvertDepthCoordinatesToHand(m_handTrackerHandle, x, y, z, pOutX, pOutY);
}

private:
NiteHandTrackerHandle m_handTrackerHandle;

class NiTE
{
public:
    static Status initialize()
    {
        return (Status)niteInitialize();
    }

    static void shutdown()
niteShutdown();

static Version getVersion()
{
    NiteVersion version = niteGetVersion();
    union
    {
        NiteVersion* pC;
        Version* pCpp;
    } a;
    a.pC = &version;
    return *a.pCpp;
}

private:
NiTE();

} // namespace nite

#endif  // _NITE_H_
This is the complete list of members for `nite::Array< T >`, including all inherited members.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Array()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>getSize() const</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>isEmpty() const</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>operator[](int index) const</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>setData(int size, T *data)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by **doxygen** 1.7.5.1
Graph Legend

This page explains how to interpret the graphs that are generated by doxygen.

Consider the following example:

```c++
/*!
Invisible class because of truncation */
class Invisible { }

/*!
Truncated class, inheritance relation is hidden */
class Truncated : public Invisible { }

/*
Class not documented with doxygen comments */
class Undocumented { }

/*!
Class that is inherited using public inheritance */
class PublicBase : public Truncated { }

/*!
A template class */
template<class T> class Templ { }

/*!
Class that is inherited using protected inheritance */
class ProtectedBase { }

/*!
Class that is inherited using private inheritance */
class PrivateBase { }
```
/*!
Class that is used by the Inherited class */
class Used {
};

/*!
Super class that inherits a number of other classes */
class Inherited : public PublicBase,
protected ProtectedBase,
private PrivateBase,
public Undocumented,
public Templ<int>
{
  private:
  Used *m_usedClass;
};

This will result in the following graph:

The boxes in the above graph have the following meaning:

- A filled gray box represents the struct or class for which the graph is generated.
- A box with a black border denotes a documented struct or class.
- A box with a grey border denotes an undocumented struct or class.
- A box with a red border denotes a documented struct or class for which not all inheritance/containment relations are shown. A graph is truncated if it does not fit within the specified boundaries.

The arrows have the following meaning:
• A dark blue arrow is used to visualize a public inheritance relation between two classes.
• A dark green arrow is used for protected inheritance.
• A dark red arrow is used for private inheritance.
• A purple dashed arrow is used if a class is contained or used by another class. The arrow is labeled with the variable(s) through which the pointed class or struct is accessible.
• A yellow dashed arrow denotes a relation between a template instance and the template class it was instantiated from. The arrow is labeled with the template parameters of the instance.
nité::BoundingBox Member List

This is the complete list of members for nité::BoundingBox, including all inherited members.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nité::BoundingBox()</td>
<td></td>
</tr>
<tr>
<td>nité::BoundingBox(const Point3f &amp;min, const Point3f &amp;max)</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>NiteBoundingBox</td>
</tr>
<tr>
<td>min</td>
<td>NiteBoundingBox</td>
</tr>
</tbody>
</table>
nite::GestureData Member List

This is the complete list of members for nite::GestureData, including all inherited members.

- `get_current_position() const nite::GestureData [inline]`
- `get_type() const nite::GestureData [inline]`
- `is_complete() const nite::GestureData [inline]`
- `is_in_progress() const nite::GestureData [inline]`
nite::HandData Member List

This is the complete list of members for nite::HandData, including all inherited members.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getId() const</td>
<td>nite::HandData</td>
</tr>
<tr>
<td>getPosition() const</td>
<td>nite::HandData</td>
</tr>
<tr>
<td>isLost() const</td>
<td>nite::HandData</td>
</tr>
<tr>
<td>isNew() const</td>
<td>nite::HandData</td>
</tr>
<tr>
<td>isTouchingFov() const</td>
<td>nite::HandData</td>
</tr>
<tr>
<td>isTracking() const</td>
<td>nite::HandData</td>
</tr>
</tbody>
</table>
# nite::HandTracker Member List

This is the complete list of members for `nite::HandTracker`, including all inherited members.

<table>
<thead>
<tr>
<th>Function</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addNewFrameListener</code> (NewFrameListener *pListener)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>convertDepthCoordinatesToHand</code> (int x, int y, int z, float *pOutX, float *pOutY) const</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>convertHandCoordinatesToDepth</code> (float x, float y, float z, float *pOutX, float *pOutY) const</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>create</code> (openni::Device *pDevice=NULL)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>destroy</code></td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>getSmoothingFactor</code> () const</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>HandTracker()</code></td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>isValid</code> () const</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>readFrame</code> (HandTrackerFrameRef *pFrame)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>removeNewFrameListener</code> (NewFrameListener *pListener)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>setSmoothingFactor</code> (float factor)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>startGestureDetection</code> (GestureType type)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>startHandTracking</code> (const Point3f &amp;position, HandId *pNewHandId)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>stopGestureDetection</code> (GestureType type)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>stopHandTracking</code> (HandId id)</td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
<tr>
<td><code>~HandTracker()</code></td>
<td><code>nite::HandTracker</code> [inl]</td>
</tr>
</tbody>
</table>
nite::HandTrackerFrameRef Member List

This is the complete list of members for nite::HandTrackerFrameRef, including all inherited members.

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDepthFrame() const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>getFrameIndex() const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>getGestures() const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>getHands() const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>getTimestamp() const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>HandTracker class</td>
<td>nite::HandTrackerFrameRef [friend]</td>
</tr>
<tr>
<td>HandTrackerFrameRef()</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>HandTrackerFrameRef(const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>HandTrackerFrameRef &amp;other)</td>
<td></td>
</tr>
<tr>
<td>isValid() const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>operator=(const</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>HandTrackerFrameRef &amp;other)</td>
<td></td>
</tr>
<tr>
<td>release()</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
<tr>
<td>~HandTrackerFrameRef()</td>
<td>nite::HandTrackerFrameRef [inline]</td>
</tr>
</tbody>
</table>
**nite::HandTracker::NewFrameListener Member List**

This is the complete list of members for `nite::HandTracker::NewFrameListener`, including all inherited members.

<table>
<thead>
<tr>
<th>HandTracker class</th>
<th>nite::HandTracker::NewFrameListener [friend]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewFrameListener()</td>
<td>nite::HandTracker::NewFrameListener [inline]</td>
</tr>
<tr>
<td>onNewFrame(HandTracker &amp;)=0</td>
<td>nite::HandTracker::NewFrameListener [pure virtual]</td>
</tr>
</tbody>
</table>

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by [doxygen](http://www.stackoverflow.com) 1.7.5.1
# nite::UserTracker::NewFrameListener Member List

This is the complete list of members for `nite::UserTracker::NewFrameListener`, including all inherited members.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NewFrameListener()</code></td>
<td><code>nite::UserTracker::NewFrameListener [inline]</code></td>
</tr>
<tr>
<td><code>onNewFrame(UserTracker &amp;)=0</code></td>
<td><code>nite::UserTracker::NewFrameListener [pure virtual]</code></td>
</tr>
<tr>
<td><code>UserTracker class</code></td>
<td><code>nite::UserTracker::NewFrameListener [friend]</code></td>
</tr>
</tbody>
</table>

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
nite::NiTE Member List

This is the complete list of members for nite::NiTE, including all inherited members.

- `getVersion()` nite::NiTE [inline, static]
- `initialize()` nite::NiTE [inline, static]
- `shutdown()` nite::NiTE [inline, static]
#ifdef _NITE_C_TYPES_H_
#define _NITE_C_TYPES_H_

#include <OniCTypes.h>
#include "NiteCEnums.h"

typedef short int NiteUserId;

typedef struct NiteUserTracker* NiteUserTrackerHandle;

#define NITE_JOINT_COUNT 15
#define NITE_POSE_COUNT 2
typedef struct{
    float x, y, z;
} NitePoint3f;

typedef struct{
    float x, y, z, w;
} NiteQuaternion;

typedef struct{
    NiteJointType jointType;
    NitePoint3f position;
    float positionConfidence;
    NiteQuaternion orientation;
    float orientationConfidence;
} NiteSkeletonJoint;

typedef struct{
    NitePoint3f min;
    NitePoint3f max;
} NiteBoundingBox;

typedef struct{
    NitePoseType type;
    int state;
} NitePoseData;

typedef struct{
    NiteSkeletonJoint joints[NITE_JOINT_COUNT];
}
typedef struct {
  NiteUserId id;
  NiteBoundingBox boundingBox;
  NitePoint3f centerOfMass;
  int state;
  NiteSkeleton skeleton;
  NitePoseData poses[NITE_POSE_COUNT];
} NiteUserData;

typedef struct {
  NiteUserId* pixels;
  int width;
  int height;
  int stride;
} NiteUserMap;

typedef struct {
  NitePoint3f point;
  NitePoint3f normal;
} NitePlane;

typedef struct {
  int userCount;
  NiteUserData* pUser;
}
NiteUserMap userMap;
OniFrame* pDepthFrame;
unsigned long long timestamp;
int frameIndex;
float floorConfidence;
NitePlane floor;
}
NiteUserTrackerFrame;

typedef struct
{
OniGeneralCallback readyForNextFrame;
} NiteUserTrackerCallbacks;

typedef short int NiteHandId;

typedef struct
{
NiteHandId id;
NitePoint3f position;
int state;
} NiteHandData;

typedef struct
{
NiteGestureType type;
NitePoint3f currentPosition;
int state;
} NiteGestureData;

typedef struct
{
int handCount;
NiteHandData* pHands;
}
int gestureCount;
NiteGestureData* pGestures;
OniFrame* pDepthFrame;
unsigned long long timestamp;
int frameIndex;
}	NiteHandTrackerFrame;

typedef struct
{
OniGeneralCallback readyForNextFrame;
} NiteHandTrackerCallbacks;

#define _NITE_DECLARE_VERSION(name)

typedef struct
{
int major;

int minor;

}
00183    int maintenance;

00184

00186    int build;

00187 } name;
00188
00189  _NITE_DECLARE_VERSION(NiteVersion);
00190
00191
00192 typedef struct NiteHandTracker* NiteHandTrackerHandle;
00193
00194 #endif // _NITE_C_TYPES_H_

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
NiteBoundingBox Member List

This is the complete list of members for NiteBoundingBox, including all inherited members.

- `max NiteBoundingBox`
- `min NiteBoundingBox`
NitePlane Member List

This is the complete list of members for NitePlane, including all inherited members.

- normal NitePlane
- point NitePlane
NitePoint3f Member List

This is the complete list of members for NitePoint3f, including all inherited members.

x NitePoint3f
y NitePoint3f
z NitePoint3f
NiteQuaternion Member List

This is the complete list of members for NiteQuaternion, including all inherited members.

- w NiteQuaternion
- x NiteQuaternion
- y NiteQuaternion
- z NiteQuaternion
nite::Plane Member List

This is the complete list of members for nite::Plane, including all inherited members.

| normal     | NitePlane  |
| Plane()    | nite::Plane [inline] |
| Plane(const Point3f &point, const Point3f &normal) | nite::Plane [inline] |
| point      | NitePlane  |
## nite::Point3f Member List

This is the complete list of members for nite::Point3f, including all inherited members.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator!=(const Point3f &amp;other) const</td>
<td>nite::Point3f [inline]</td>
</tr>
<tr>
<td>operator=(const Point3f &amp;other)</td>
<td>nite::Point3f [inline]</td>
</tr>
<tr>
<td>operator==(const Point3f &amp;other) const</td>
<td>nite::Point3f [inline]</td>
</tr>
<tr>
<td>Point3f()</td>
<td>nite::Point3f [inline]</td>
</tr>
<tr>
<td>Point3f(float x, float y, float z)</td>
<td>nite::Point3f [inline]</td>
</tr>
<tr>
<td>Point3f(const Point3f &amp;other)</td>
<td>nite::Point3f [inline]</td>
</tr>
<tr>
<td>set(float x, float y, float z)</td>
<td>nite::Point3f [inline]</td>
</tr>
<tr>
<td>x</td>
<td>NitePoint3f</td>
</tr>
<tr>
<td>y</td>
<td>NitePoint3f</td>
</tr>
<tr>
<td>z</td>
<td>NitePoint3f</td>
</tr>
</tbody>
</table>
nite::PoseData Member List

This is the complete list of members for nite::PoseData, including all inherited members.

- **getType() const**
- **isEntered() const**
- **isExited() const**
- **isHeld() const**

*Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1*
nite::Quaternion Member List

This is the complete list of members for nite::Quaternion, including all inherited members.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quaternion()</td>
<td>nite::Quaternion [inline]</td>
</tr>
<tr>
<td>Quaternion(float w, float x, float y, float z)</td>
<td>nite::Quaternion [inline]</td>
</tr>
<tr>
<td>w</td>
<td>NiteQuaternion</td>
</tr>
<tr>
<td>x</td>
<td>NiteQuaternion</td>
</tr>
<tr>
<td>y</td>
<td>NiteQuaternion</td>
</tr>
<tr>
<td>z</td>
<td>NiteQuaternion</td>
</tr>
</tbody>
</table>

Generated on Thu Jun 6 2013 17:48:15 for NiTE 2.0 by doxygen 1.7.5.1
nite::Skeleton Member List

This is the complete list of members for nite::Skeleton, including all inherited members.

- `getJoint(JointType type) const nite::Skeleton` [inline]
- `getState() const nite::Skeleton` [inline]
## nite::SkeletonJoint Member List

This is the complete list of members for `nite::SkeletonJoint`, including all inherited members.

<table>
<thead>
<tr>
<th>Member Method</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getOrientation()</code> const</td>
<td><code>nite::SkeletonJoint</code> [inline]</td>
</tr>
<tr>
<td><code>getOrientationConfidence()</code> const</td>
<td><code>nite::SkeletonJoint</code> [inline]</td>
</tr>
<tr>
<td><code>getPosition()</code> const</td>
<td><code>nite::SkeletonJoint</code> [inline]</td>
</tr>
<tr>
<td><code>getPositionConfidence()</code> const</td>
<td><code>nite::SkeletonJoint</code> [inline]</td>
</tr>
<tr>
<td><code>getType()</code> const</td>
<td><code>nite::SkeletonJoint</code> [inline]</td>
</tr>
</tbody>
</table>
# nite::UserData Member List

This is the complete list of members for `nite::UserData`, including all inherited members.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBoundingBox() const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
<tr>
<td>getCenterOfMass() const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
<tr>
<td>getId() const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
<tr>
<td>getPose(PoseType type) const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
<tr>
<td>getSkeleton() const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
<tr>
<td>isLost() const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
<tr>
<td>isNew() const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
<tr>
<td>isVisible() const</td>
<td><code>nite::UserData</code> [inline]</td>
</tr>
</tbody>
</table>
nite::UserMap Member List

This is the complete list of members for nite::UserMap, including all inherited members.

- getHeight() const
- getPixels() const
- getStride() const
- getWidth() const
- UserTrackerFrameRef class nite::UserMap [friend]
# nite::UserTracker Member List

This is the complete list of members for **nite::UserTracker**, including all inherited members.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addNewFrameListener</code></td>
<td>(NewFrameListener *pListener)</td>
</tr>
<tr>
<td><code>convertDepthCoordinatesToJoint</code></td>
<td>(int x, int y, int z, float *pOutX, float *pOutY) const</td>
</tr>
<tr>
<td><code>convertJointCoordinatesToDepth</code></td>
<td>(float x, float y, float z, float *pOutX, float *pOutY) const</td>
</tr>
<tr>
<td><code>create</code></td>
<td>(openni::Device *pDevice=NULL)</td>
</tr>
<tr>
<td><code>destroy</code></td>
<td>()</td>
</tr>
<tr>
<td><code>getSkeletonSmoothingFactor</code></td>
<td>() const</td>
</tr>
<tr>
<td><code>isValid</code></td>
<td>() const</td>
</tr>
<tr>
<td><code>readFrame</code></td>
<td>(UserTrackerFrameRef *pFrame)</td>
</tr>
<tr>
<td><code>removeNewFrameListener</code></td>
<td>(NewFrameListener *pListener)</td>
</tr>
<tr>
<td><code>setSkeletonSmoothingFactor</code></td>
<td>(float factor)</td>
</tr>
<tr>
<td><code>startPoseDetection</code></td>
<td>(UserId user, PoseType type)</td>
</tr>
<tr>
<td><code>startSkeletonTracking</code></td>
<td>(UserId id)</td>
</tr>
<tr>
<td><code>stopPoseDetection</code></td>
<td>(UserId user, PoseType type)</td>
</tr>
<tr>
<td><code>stopSkeletonTracking</code></td>
<td>(UserId id)</td>
</tr>
<tr>
<td><code>UserTracker</code></td>
<td>()</td>
</tr>
<tr>
<td><code>~UserTracker</code></td>
<td>()</td>
</tr>
</tbody>
</table>
nite::UserTrackerFrameRef Member List

This is the complete list of members for nite::UserTrackerFrameRef, including all inherited members.

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDepthFrame()</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>getFloor() const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>getFloorConfidence() const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>getFrameIndex() const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>getTimestamp() const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>getUserById(UserId id) const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>getUserMap() const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>getUsers() const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>isValid() const</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>operator=(const UserTrackerFrameRef &amp;other)</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>release()</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>User class</td>
<td>nite::UserTrackerFrameRef</td>
<td>[friend]</td>
</tr>
<tr>
<td>UserTracker class</td>
<td>nite::UserTrackerFrameRef</td>
<td>[friend]</td>
</tr>
<tr>
<td>UserTrackerFrameRef()</td>
<td>nite::UserTrackerFrameRef</td>
<td>[friend]</td>
</tr>
<tr>
<td>UserTrackerFrameRef(const UserTrackerFrameRef &amp;other)</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
<tr>
<td>~UserTrackerFrameRef()</td>
<td>nite::UserTrackerFrameRef</td>
<td>[inline]</td>
</tr>
</tbody>
</table>
- **a** -
  - addNewFrameListener() : `nite::UserTracker`, `nite::HandTracker`
  - Array() : `nite::Array< T >`

- **b** -
  - BoundingBox() : `nite::BoundingBox`

- **c** -
  - `convertDepthCoordinatesToHand()` : `nite::HandTracker`
  - `convertDepthCoordinatesToJoint()` : `nite::UserTracker`
  - `convertHandCoordinatesToDepth()` : `nite::HandTracker`
  - `convertJointCoordinatesToDepth()` : `nite::UserTracker`
  - `create()` : `nite::UserTracker`, `nite::HandTracker`

- **d** -
  - `destroy()` : `nite::UserTracker`, `nite::HandTracker`

- **g** -
  - `getBoundingBox()` : `nite::UserData`
  - `getCenterOfMass()` : `nite::UserData`
  - `getCurrentPosition()` : `nite::GestureData`
  - `getDepthFrame()` : `nite::UserTrackerFrameRef`, `nite::HandTrackerFrameRef`
- h -

- HandTracker() : nite::HandTracker
- HandTrackerFrameRef() : nite::HandTrackerFrameRef

- i -
- n -
  - NewFrameListener() : nite::UserTracker::NewFrameListener, nite::HandTracker::NewFrameListener

- o -
  - onNewFrame() : nite::UserTracker::NewFrameListener, nite::HandTracker::NewFrameListener
  - operator!=( ) : nite::Point3f
  - operator==( ) : nite::Point3f, nite::UserTrackerFrameRef, nite::HandTrackerFrameRef
  - operator==( ) : nite::Point3f
  - operator[]( ) : nite::Array< T >

- p -
  - Plane() : nite::Plane
  - Point3f() : nite::Point3f

- q -
  - Quaternion() : nite::Quaternion
- r -
  - readFrame() : nite::UserTracker, nite::HandTracker
  - release() : nite::HandTrackerFrameRef, nite::UserTrackerFrameRef
  - removeNewFrameListener() : nite::HandTracker, nite::UserTracker

- s -
  - set() : nite::Point3f
  - setData() : nite::Array<T>
  - setSkeletonSmoothingFactor() : nite::UserTracker
  - setSmoothingFactor() : nite::HandTracker
  - shutdown() : nite::NiTE
  - startGestureDetection() : nite::HandTracker
  - startHandTracking() : nite::HandTracker
  - startPoseDetection() : nite::UserTracker
  - startSkeletonTracking() : nite::UserTracker
  - stopGestureDetection() : nite::HandTracker
  - stopHandTracking() : nite::HandTracker
  - stopPoseDetection() : nite::UserTracker
  - stopSkeletonTracking() : nite::UserTracker

- u -
  - UserTracker() : nite::UserTracker
  - UserTrackerFrameRef() : nite::UserTrackerFrameRef

- ~ ~ -
  - ~HandTracker() : nite::HandTracker
  - ~HandTrackerFrameRef() : nite::HandTrackerFrameRef
  - ~UserTracker() : nite::UserTracker
  - ~UserTrackerFrameRef() : nite::UserTrackerFrameRef
- max : `NiteBoundingBox`
- min : `NiteBoundingBox`
- normal : `NitePlane`
- point : `NitePlane`
- w : `NiteQuaternion`
- x : `NitePoint3f`, `NiteQuaternion`
- y : `NitePoint3f`, `NiteQuaternion`
- z : `NitePoint3f`, `NiteQuaternion`
- HandTracker: `nite::HandTrackerFrameRef`, `nite::HandTracker::NewFrameListener`
- User: `nite::UserTrackerFrameRef`
- UserTracker: `nite::UserTrackerFrameRef`, `nite::UserTracker::NewFrameListener`
- UserTrackerFrameRef: `nite::UserMap`
- HandId: `nite`
- UserId: `nite`
- GestureType : nite
- JointType : nite
- PoseType : nite
- SkeletonState : nite
- Status : nite
- g -

- GESTURE_CLICK : nite
- GESTURE_HAND_RAISE : nite
- GESTURE_WAVE : nite

- j -

- JOINT_HEAD : nite
- JOINT_LEFT_ELBOW : nite
- JOINT_LEFT_FOOT : nite
- JOINT_LEFT_HAND : nite
- JOINT_LEFT_HIP : nite
- JOINT_LEFT_KNEE : nite
- JOINT_LEFT_SHOULDER : nite
- JOINT_NECK : nite
- JOINT_RIGHT_ELBOW : nite
- JOINT_RIGHT_FOOT : nite
- JOINT_RIGHT_HAND : nite
- JOINT_RIGHT_HIP : nite
- JOINT_RIGHT_KNEE : nite
- JOINT_RIGHT_SHOULDER : nite
- JOINT_TORSO : nite

- p -

- POSE_CROSSED_HANDS : nite
- s -

- POSE_PSI : nite

- SKELETON_CALIBRATING : nite
- SKELETON_CALIBRATION_ERROR_HANDS : nite
- SKELETON_CALIBRATION_ERROR_HEAD : nite
- SKELETON_CALIBRATION_ERROR_LEGS : nite
- SKELETON_CALIBRATION_ERROR_NOT_IN_POSE : nite
- SKELETON_CALIBRATION_ERROR_TORSO : nite
- SKELETON_NONE : nite
- SKELETON_TRACKED : nite
- STATUS_BAD_USER_ID : nite
- STATUS_ERROR : nite
- STATUS_OK : nite
- STATUS_OUT_OF_FLOW : nite
/* NiTE 2.0 */

#ifdef _NITE_ENUMS_H_
#define _NITE_ENUMS_H_
#endif

typedef enum
{
    JOINT_HEAD,
    JOINT_NECK,
    JOINT_LEFT_SHOULDER,
    JOINT_RIGHT_SHOULDER,
    JOINT_LEFT_ELBOW,
    JOINT_RIGHT_ELBOW,
    JOINT_LEFT_HAND,
    JOINT_RIGHT_HAND,
}
typedef enum 
{
  SKELETON_NONE,
  SKELETON_CALIBRATING,
  SKELETON_TRACKED,
  SKELETON_CALIBRATION_ERROR_NOT_IN_POSE,
  SKELETON_CALIBRATION_ERROR_HANDS,
  SKELETON_CALIBRATION_ERROR_HEAD,
  SKELETON_CALIBRATION_ERROR_LEGS,
  SKELETON_CALIBRATION_ERROR_TORSO
} SkeletonState;

typedef enum
{
  STATUS_OK,
  STATUS_ERROR,
  STATUS_BAD_USER_ID,
  STATUS_OUT_OF_FLOW
} Status;

typedef enum
{
  POSE_PSI,
  POSE_CROSSED_HANDS
}
typedef enum {
    GESTURE_WAVE,
    GESTURE_CLICK,
    GESTURE_HAND_RAISE
} GestureType;

# endif // _NITE_ENUMS_H_
<table>
<thead>
<tr>
<th>Typedefs</th>
<th>Enumerations</th>
<th>Enumerator</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td>NiteHandId : NiteCTypes.h</td>
<td>NiteHandTrackerHandle : NiteCTypes.h</td>
<td>NiteUserId : NiteCTypes.h</td>
<td>NiteUserTrackerHandle : NiteCTypes.h</td>
</tr>
</tbody>
</table>
- GestureType: `Nite Enums.h`
- JointType: `Nite Enums.h`
- PoseType: `Nite Enums.h`
- SkeletonState: `Nite Enums.h`
- Status: `Nite Enums.h`
- g -

- GESTURE_CLICK : NiteEnums.h
- GESTURE_HAND_RAISE : NiteEnums.h
- GESTURE_WAVE : NiteEnums.h

- j -

- JOINT_HEAD : NiteEnums.h
- JOINT_LEFT_ELBO : NiteEnums.h
- JOINT_LEFT_FOOT : NiteEnums.h
- JOINT_LEFT_HAND : NiteEnums.h
- JOINT_LEFT_HIP : NiteEnums.h
- JOINT_LEFT_KNEE : NiteEnums.h
- JOINT_LEFT_SHOULDER : NiteEnums.h
- JOINT_NECK : NiteEnums.h
- JOINT_RIGHT_ELBO : NiteEnums.h
- JOINT_RIGHT_FOOT : Nite Enums.h
- JOINT_RIGHT_HAND : NiteEnums.h
- JOINT_RIGHT_HIP : NiteEnums.h
- JOINT_RIGHT_KNEE : NiteEnums.h
- JOINT_RIGHT_SHOULDER : NiteEnums.h
- JOINT_TORSO : NiteEnums.h

- p -

- POSE_CROSSED_HANDS : NiteEnums.h
- s -

- POSE_PSI : NiteEnums.h

- SKELETON_CALIBRATING : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_HANDS : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_HEAD : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_LEGS : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_NOT_IN_POSE : NiteEnums.h
- SKELETON_CALIBRATION_ERROR_TORSO : NiteEnums.h
- SKELETON_NONE : NiteEnums.h
- SKELETON_TRACKED : NiteEnums.h
- STATUS_BAD_USER_ID : NiteEnums.h
- STATUS_ERROR : NiteEnums.h
- STATUS_OK : Nite Enums.h
- STATUS_OUT_OF_FLOW : Nite Enums.h
- NITE_JOINT_COUNT : NiteCTypes.h
- NITE_POSE_COUNT : NiteCTypes.h