# Omek Beckon SDK
# TI DM3730™ Edition

## Developer's Guide

BECKON SDK 2.4, TI DM3730 EDITION

# Content

# Welcome to Omek Beckon SDK, TI DM3730 Edition

The aim of this guide is to provide you with all the details you need to get started using the Beckon 2.4 SDK on the BeagleBoard-xM. The Beckon SDK provides a robust solution for real-time, 3D motion tracking using a single depth sensor. With the supplied tools you can quickly and efficiently deploy full body tracking and gesture recognition in your applications.

This release of Omek's Beckon 2.4 SDK, TI DM3730 Edition is intended for use on the BeagleBoard-xM. The BeagleBoard-xM is a low-cost, fan-less single-board computer based on low-power Texas Instruments processors (here, the DM3730) featuring the ARM Cortex-A series core with all of the expandability of today's desktop machines, but without the bulk, expense, or noise.

The Beckon SDK, TI DM3730 Edition employs the core technology underpinning Omek's PC-version SDK but is optimized for use on an embedded platform. This was achieved by off-loading the algorithms to the DSP in order to free up the ARM, allowing users to develop applications on a Linux PC and run them on the ARM on the BeagleBoard-xM.

Due to the embedded processing environment associated with the BeagleBoard, there are some limitations regarding the functionality of this version, as compared to the PC version of the SDK. The features included in this release are listed below.

Omek is constantly upgrading and improving its solution and will be providing additional developer tools in the future. The Support Portal provides details on new releases and improved functionalities.

## Beckon SDK Development Suite Components

The Beckon SDK Development Suite, TI DM3730 Edition includes the following components:

- Ubuntu-based PC cross-development platform, including Beckon engine, APIs, sample code, and camera support for Kinect and ASUS cameras
- Complete image containing Angstrom OS, DSP-based Beckon engine, and pre-compiled OpenNI camera drivers
- Three sample demos – QT Tracker and OpenGL ES stick figure skeleton, and GUI gesture recognition
- Support for a single skeleton, at 20fps
- Support for front-facing camera placement
- Five gestures provided per hand, including (for both right and left hands) left and right scroll, up and down scroll, and push

## Downloads

The Beckon SDK components are included in two tarballs available in the Downloads section on the support site:

- **Tarball 1 (*Beckon-Image-2.4.16175.tar.gz*):** An image with everything needed to run the Beckon Engine and Demos on the BeagleBoard-xM:
  - The Beckon Engine, which runs on the BeagleBoard-xM
  - Angstrom Linux OS
  - OpenNI Camera Drivers
  - Qt for Application Development
  - 3 Compiled Sample Demos
- **Tarball 2 (*Beckon-SDK-2.4.16236.tar.gz*):** The Beckon SDK. This tool, which is installed on the Linux PC for cross-compiling enables you to develop your own gesture-based applications. Using the Beckon SDK you have access to the scene intelligence – at the blob, skeleton, or gesture level – for use by applications written in C++.

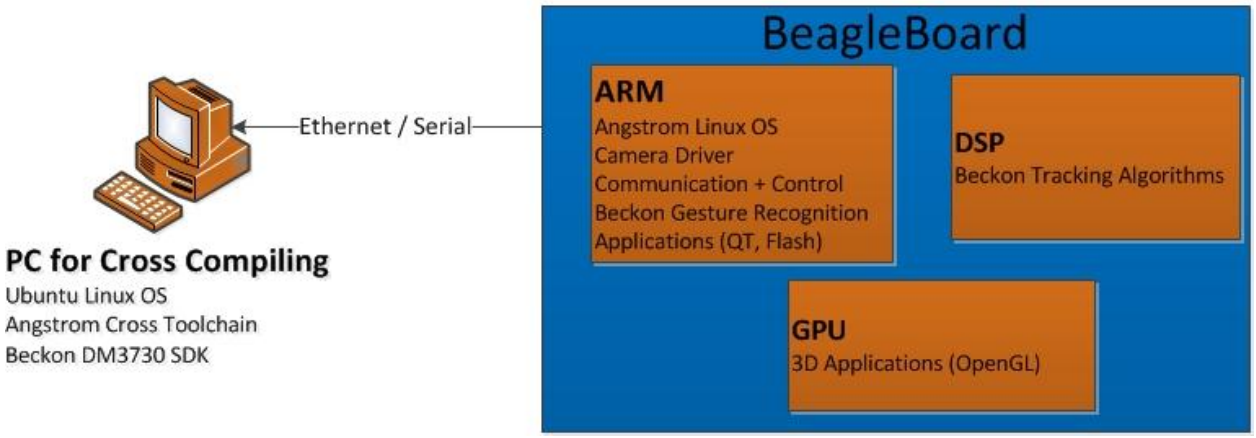


*Figure 1: Diagram showing the cross-compilation development environment. The DSP runs the Beckon Tracking Algorithms opening up valuable processing space on the ARM run applications. Communication between the ARM and the DSP is facilitated through DSPLink.*

## Hardware Requirements

- **BeagleBoard-xM**. Details on where to buy can be found in: http://beagleboard.org/buy.
  ***Note: You need a BeagleBoard-xM, as this does not run on older versions of BeagleBoard. Any reference to BeagleBoard in this document implies use of the BeagleBoard-xM***
- Ubuntu 10.04, or higher, Linux PC
- Accessories:
  - **Supported Camera**. Currently, the Beckon 2.4, TI DM3730 Edition supports the Asus Xtion Pro and the Kinect cameras, however, in the coming months there will be extended support for additional cameras.
    - **Asus Xtion Pro.** Details on where to buy can be found here: http://us.estore.asus.com/index.php?l=product_detail&p=3397
    - **Microsoft Kinect.** Note that you only need to buy the Microsoft Kinect and **not** an Xbox Kinect. Details on where to buy can be found here: http://www.xbox.com/en-US/Xbox360/WhereToBuy?xr=shellnav

    ***Note: only one camera is needed, either the Asus or the Microsoft Kinect***
  - **USB Keyboard**
  - **USB Mouse**
  - **Display monitor with HDMI or DVI inputs**
  - **HDMI to HDMI or HDMI to DVI Cable**
  - **Micro SD Card Reader / Writer**
  - **Micro SD Card,** minimum size = 1 GB

    ***Note: the SD Card will be formatted prior to use so make sure everything on it is backed up or can be erased!***

## Software Requirements

- Linux Development PC with Ubuntu version 10.04 or higher installed

# Recommended Workflow for Getting Started

To familiarize yourself with the Beckon 2.4 SDK running on the BeagleBoard-xM, start off by burning the Image and testing out the demos that are included.

The recommended steps are as follows:

## Step 1: Download and Extract the Image Package

**To download and extract the Image packages:**

1. Download the tar image onto your Linux PC.
2. Extract the tar image onto your Linux PC by entering **tar -xvzf imagefilename** in the command line.

## Step 2: Burn the Image Package and Boot the System

**To create an SD card:**

1. Ensure you have bash installed in your Linux PC system. To install Bash on your Ubuntu system, run **sudo apt-get install bash** in the command
2. Insert the SD card into the Linux machine using a card reader/writer.
3. Run dmesg in the command line. The following description should appear:

```
[ 5230.797273] sd 10:0:0:0: Attached scsi generic sg1 type 0
[ 5231.411571] sd 10:0:0:0: [sdb] 7744512 512-byte logical blocks: (3.96
GB/3.69 GiB)
[ 5231.412452] sd 10:0:0:0: [sdb] Write Protect is off
[ 5231.412458] sd 10:0:0:0: [sdb] Mode Sense: 03 00 00 00
[ 5231.413724] sd 10:0:0:0: [sdb] No Caching mode page present
[ 5231.413730] sd 10:0:0:0: [sdb] Assuming drive cache: write through
[ 5231.417729] sd 10:0:0:0: [sdb] No Caching mode page present
[ 5231.417734] sd 10:0:0:0: [sdb] Assuming drive cache: write through
```

This means that in the example above the card was identified as /dev/sdb. **Make sure the correct size of the card is shown**.
***Note: Your card might be identified as /dev/sdc or /dev/sdd. In this manual the identified device is referred to as /dev/sdX***

4. Issue the following command:

```
sudo bash mkcard.sh /dev/sdX beagle-omek
```

***WARNING:*** **If you mistakenly use the main OS device name this script can erase your entire OS.**

5. When the script finishes you can pull the card out and put it in the BeagleBoard-xM.

**To activate the BeagleBoard:**

6. Connect the BeagleBoard power supply.

7. Plug the camera into the BeagleBoard USB port.
8. Ensure the mouse and keyboard are connected to the BeagleBoard's USB port.
9. Connect the BeagleBoard to the monitor using an HDMI cable.
10. Turn on the power to the BeagleBoard-xM and wait for the boot process to reach the login screen, login as user "root" with an empty password

*Once the prior steps have been completed you are ready to run the sample demos on the BeagleBoard-xM.*

## Run the Sample Demos

Once you have the image running on the BeagleBoard-xM you are ready to test out the demos that are included in the package. The Beagleboard-xM image comes pre-loaded with three demos that you can activate to familiarize yourself with the Beckon middleware.

***Note: The sources of the demos exist as part of the SDK and can also be used in application development. There is more detail on this in the Sample Code section of this document.***

The demos included are:

1. **Qt Tracking Sample:** This demo shows the raw depth data with annotations of a skeleton's tracked joints in 2D (i.e., dots on the joints), implemented using Qt.
2. **TrackingViewer3D Sample:** The sample displays a 3D stick-figure showing the tracked skeleton joints, written using OpenGL ES.
3. **GestureDemo Sample:** The demo allows the user to select from a predefined list of gestures (i.e., _rightScrollRight, _rightClick, _leftScrollLeft, etc…) and have an application show a message each time the requested gesture is recognized. For additional detail on how to incorporate gestures into an application see the Gestures section.

### QtTracking Sample

The sample displays simple GUI with two windows:


*Figure 2*

- 
- **Left Window** - the depth image as received from camera
- **Right Window** - the player mask indicating the tracked person with marked skeleton joints. The player mask is a mask where all player pixels are marked in red and all background pixels are black.

An ESC key terminates the application.

**To run QtTracking Sample:**

1. To run live tracking using camera input:

```
# sh ./tracking.sh
```

2. You can record a camera sequence using the tools provided with the PC version of OpenNI. To run prerecorded camera sequence (OpenNI sequence file (.oni)):

```
# sh ./tracking.sh -seq
path/to/directory/containing/the/sequence/file/
```

***Note: the path points to a directory containing the sequence, not to the sequence file itself.***

## TrackingViewer3D Sample

The sample displays a stick-figure (see Figure below) showing the tracked skeleton joints.

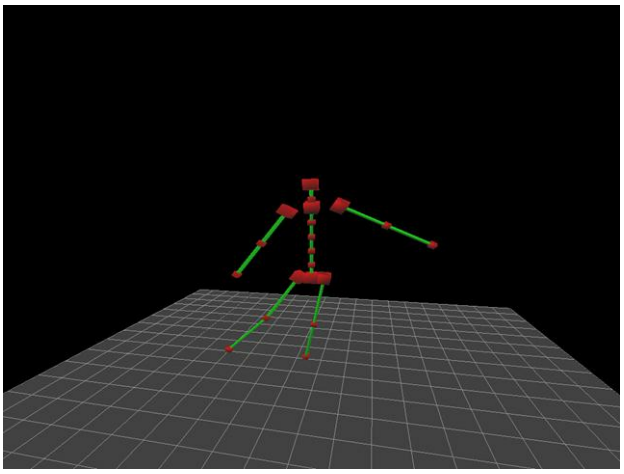

*Figure 3*

1. To run live tracking using camera input:

```
# sh ./tracking3d.sh -gest <gesture name> -gest ...
```

2. To run pre-recorded camera sequence (OpenNI sequence file (.oni)):

```
# sh ./tracking3d.sh -seq
path/to/directory/containing/the/sequence/file/
```

***Note: the path points to a directory containing the sequence, not to the sequence file itself.***

## GestureDemo Sample

The sample tracks a human and prints each time a requested gesture is activated

1. To run live tracking using camera input:

```
# sh ./gestures.sh -gest <gesture name> -gest ...
```

2. To run a pre-recorded camera sequence (OpenNI sequence file (.oni)):

```
# sh ./gestures.sh -seq
path/to/directory/containing/the/sequence/file/ -gest <gesture name> -
gest ...
```

The supported list of gestures are:

- _rightClick _leftClick
- _rightScrollRight _rightScrollLeft
- _rightScrollUp _rightScrollDown
- _leftScrollRight _leftScrollLeft
- _leftScrollUp _leftScrollDown

***Note: The path points to a directory containing the sequence, not to the sequence file itself.***

# Setting Up the Linux PC

## Setting up your Development PC

- Download and open the SDK tarball package from the Downloads section on the Support Site – extract it onto your Linux development PC (this is the actual SDK which contains headers, libraries, documentation, make files, etc…)
- The SDK provides the tools to develop a motion-tracking and/or gesture-based application that can run on the BeagleBoard-xM.
- After developing an application, cross-compile it onto the Linux PC and then copy the image onto the BeagleBoard-xM via external USB storage or Ethernet, moving the files using ftp, udp, etc…from the development PC to the board.

***Note: if you develop an OpenGL ES-based application, the processing will go on the PowerVR GPU and not on the ARM. This provides you with more freedom for processing on the ARM.***

## Notes on Running the BeagleBoard-xM

- Every time you stop an application – you need to reset the DSP. It is advisable to run the **/home/root/bin/reset-dsp.sh** script before each execution of a sample or any other application using DSP (found on the provided Angstrom image).

# Beckon Tracking Engine Initialization

## Setting Up Beckon SDK Capabilities

Adding the Beckon 2.4 SDK capabilities to your application can be done easily using only a few lines of code. The access point of your application to the Beckon SDK is through an instance of the **IMotionSensor** class.

First, you should include the class header file in your code:

```
#include "Shadow/IMotionSensor.h"
```

Beckon SDK supports two modes of operation, Live camera mode and Sequence mode.

Live camera mode is used for running the application with a connected camera. Sequence mode is used for running the application without a camera, based on an offline pre-recorded sequence stored in a file.

### To setup the Beckon SDK to run in Live camera mode:

- Create an instance of the `IMotionSensor` using the following code:

```
IMotionSensor *pSensor = IMotionSensor::createCameraSensor();
```

The `createCameraSensor` method sets up the sensor. This automatically detects the sensor type and connects to it with the appropriate depth resolution and frame rate. Note that the camera must be physically connected to the host at this point.

### To setup the Beckon SDK to run in Sequence mode:

- Create an instance of the `IMotionSensor` using the following code:

```
IMotionSensor *pSensor =
IMotionSensor::createSequenceSensor("C:/data/seq1/");
```

In this example, the `createSequenceSensor` method simulates a live sensor using the data stored in the sequence located at the folder specified by the second parameter.

From this point onwards, the code is identical for both modes of operation, so that an application can easily switch between the two modes. This ability is very useful during the development phase, as the application can be created and then debugged and tested without being connected to a camera.

***Note: Both the `createSequenceSensor` and the `createCameraSensor` methods allocate memory for the `IMotionSensor` object, and thus you must physically release its memory at the end of its use.***

## Running Motion Tracking

To invoke motion tracking, do the following:

```
bool bHasNewImage = false;
while (pSensor->getSensor()->isAlive()){
   if (pSensor->processNextImage(true, bHasNewImage)
== OMK_SUCCESS && bHasNewImage) {
      int curFrame = pSensor->getSensor()->getFramenum();
       }
}
```

The `isAlive` method returns **true** in live camera mode as long as a camera is connected. In Sequence mode it returns **true** if it has not yet reached the end of the sequence. The `processNextImage` method has an option for running in blocking or immediate mode.

The next two sections describe in depth how to use the `IMotionSensor` object for connecting visual elements to tracking results, and for registering gestures.

# Connecting Visual Elements to Tracking Results

Once the `IMotionSensor` object has been instantiated and initialized, you can access the full body skeleton of the player. The `ISkeleton` object is kept in memory and is updated per frame based on the player's movements in the scene. The `ISkeleton` object holds the rotation and translation data for each joint, as well as some additional data.

## Skeleton Setup

The following joint 3D positions (in world space) and orientations (both in world and local space) are supported. Local joint rotations are relative to the skeleton's T-Pose as shown in Figure 3 below:

| Joint Description | Joint Name |
|---|---|
| **Head** | head |
| **Torso** | torso |
| **Hips** | hips |
| **Tips of both hands** | leftFingerTip, rightFingerTip |
| **Shoulders** | leftShoulder, rightShoulder |
| **Knees** | leftKnee, rightKnee |
| **Neck** | neck |
| **Collar bones** | leftCollar, rightCollar |
| **Elbows** | leftElbow, rightElbow |
| **Waist** | waist |
| **Left and right hips** | leftHip, rightHip |
| **Spine** | spine1, spine2, spine3, spine4 |
| **Feet** | leftFoot, rightFoot |

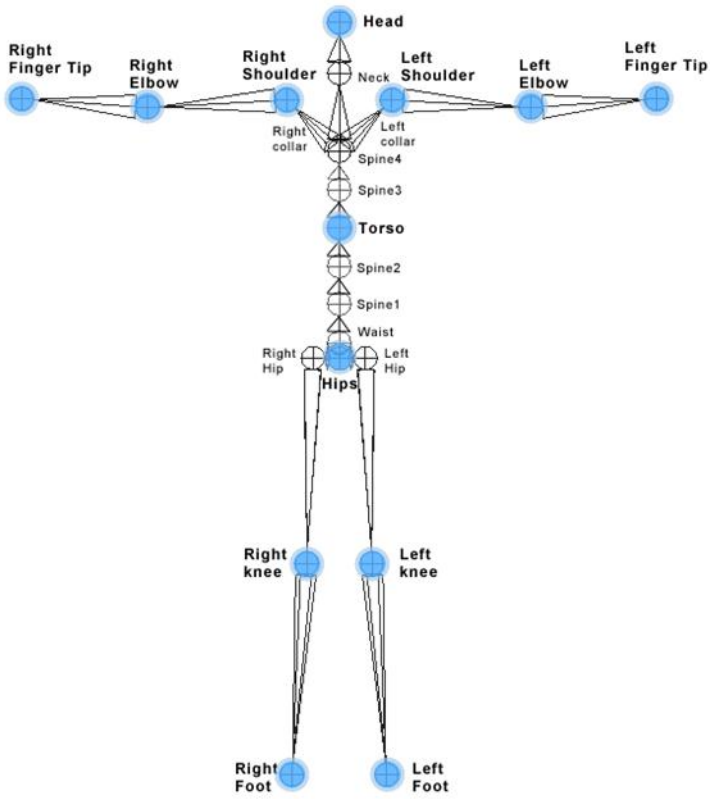


*Figure 4: Diagram of skeleton joints tracked by Beckon SDK*

## Coordinate System

In the Beckon SDK the coordinate systems are given in centimeters, with the X pointing to the right, the Y axis pointing down and the Z axis pointing into the screen. The origin of the coordinate system is the camera position.

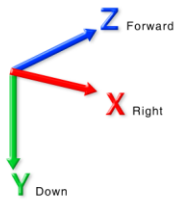


*Figure 5: Coordinate system*

## Acquiring Skeleton Data

You need to allocate a new skeleton object and pass it to the `getSkeleton` method of the `IMotionSensor`:

```
ISkeleton *pSkeleton = IMotionSensor::createSkeleton();
pSensor->getSkeleton(pSkeleton);
```

The acquired skeleton is based on the last frame, and each joint can be queried separately for its internal data in the following manner:

| Joint Description | Data Type | Description |
|---|---|---|
| **pSkeleton->getJointPosition(joint, true)** | 3D point | 3D point in world space |
| **pSkeleton->getJointPosition(joint, false)** | 2D point | 2D point in image space, range: [0…width, 0…height] |
| **pSkeleton->getJointRotation(joint, false)** | Quaternion | local 3D rotation |
| **pSkeleton->getJointRotation(joint, true)** | Quaternion | global 3D rotation |
| **pSkeleton->getJointConfidence(joint)** | Unsigned int | Accuracy measure in the range [0,100] indicating the % of confidence in the tracked position. |
| **pSkeleton->getCenterOfMass()** | 3D point | 3D point in world space indicating the torso center of mass |
| **pSkeleton->getTorsoWidth()** | Float | The width (in world space) of the torso [cm] |
| **pSkeleton->getCalibratedHeight()** | Float | The height of the player [cm] |
| **pSkeleton->getCalibratedArmLength()** | Float | The length of the arm [cm] |
| **pSkeleton->getCalibratedLegLength()** | Float | The length of the leg [cm] |
| **pSkeleton->getCalibratedShoulderWidth()** | Float | The distance between the 2 shoulders [cm] |

By default, the skeleton acquired by `getSkeleton` is a smooth skeleton, fitted to a human body model based on the player's dimensions. The smoothed skeleton minimizes motion noise and provides higher-quality tracking data. This naturally adds latency, though, to the tracking of the skeleton. If you wish to modify the strength of the smoothing on any particular joint, you can do so by calling the following method:

```
pSensor->setSmoothStrength(0, JointID_torso, 3);
```

The specific call above would cause the torso of the first tracked skeleton to use 3 frames for smoothing. 0 would turn off smoothing completely, and higher values lead to heavier smoothing (with more latency.)

If you wish to get the raw skeleton before smoothing and human body fit, you can do so by calling the following method instead:

```
pSensor->getRawSkeleton(pSkeleton);
```

## Multiple Players

While the PC version of our SDK supports multiple players, currently the TI DM3730 solution only supports a single tracked player. Future versions will support multiple player-tracking; you can learn about updates regarding future releases and new features by checking our support forum: http://support.omekinteractive.com/

## Player Blobs

In addition to tracking skeletons, you can also get data related to the blob of the player, by calling the following method:

```
pSensor->copyPlayerMask(pMask, maskSize, label, width, height,
center3D, center2D);
```

This gives you access to a silhouette mask of the tracked player which can be used as part of the application, or just for the display of the user's position relative to the scene. In this binary mask, white pixels correspond to pixels occupied by the player. The returned `width` and `height` parameters are filled with the dimensions of the blob's 2D bounding rectangle. The `center3D` parameter is filled with the blob's 3D center of mass in world space [cm]. These blobs correspond to the green blobs seen in the Tracking Viewer. The `center2D` parameter is filled with the blob's projected 2D center of mass.

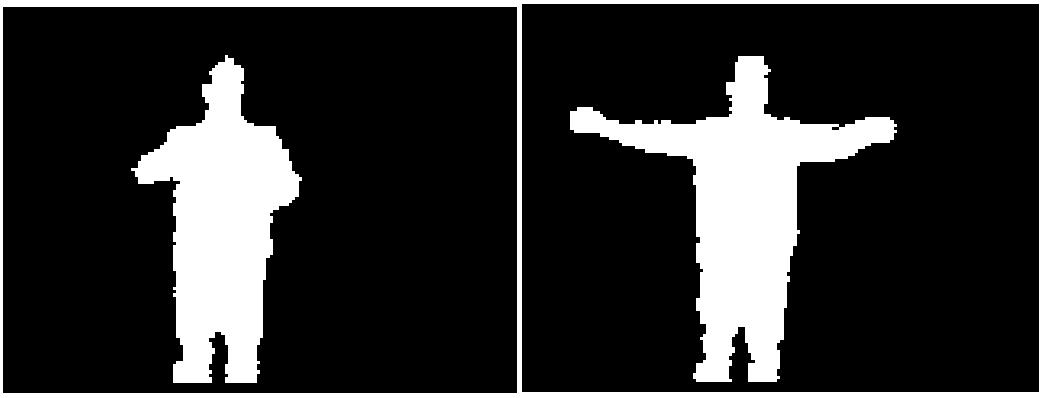

*Figure 6: Example of player blobs*

# Accessing Other Data

## Accessing Sensor Parameters

You can access sensor parameters by querying the `ISensor` class. For example, in order to query the dimensions of the depth image, call the following method:

```
ISensor *pCamera = pMotionSensor->getSensor();
int width = pCamera->getImageWidth(IMAGE_TYPE_DEPTH);
int height = pCamera->getImageHeight(IMAGE_TYPE_DEPTH);
```

You can also query the frames per second and bit resolution using this class.

To check if the sensor is connected, call the following method:

```
pCamera->isAlive();
```

In addition, you can get some useful information per frame when running with a sequence or a sensor. You can query the current frame number from the start of the session by calling:

```
pCamera->getFrameNum()
```

You can get a time stamp for each frame by calling:

```
pCamera->getFrameTimeStamp()
```

## Accessing Raw Sensor Data

In addition to the skeleton data, the `IMotionSensor` interface also provides access to the underlying depth data. You can get the depth data by calling the following method:

```
pCamera->copyRawImage(pImage, bufferSize, widthStep, false);
```

This is useful for applications that wish to display the depth data. It is also useful for debugging applications and ensuring the tracking results match correctly with the respective image. `pImage` must be allocated to the size of (*width* x *height*), where *width* and *height* are the dimensions of the image output by the specific sensor in use.

*Figure 7: depth image acquired using the above method*

## Checking Method Return Codes

In addition to logging, you can also query the result of each method in runtime by checking its return value, `OMKStatus`. This value gives you specific information on the success of each function, or the reason for its failure. For a full list of status options, see the header file `ShadowDefines.h`.

# Gestures

In addition to tracking, the Beckon SDK also provides powerful gesture recognition capabilities, through which you can access gestures, activate them and be notified when they happen.

The SDK includes a pre-defined, ready-to-use gesture pack.

See the GestureDemo code sample, for a full demonstration of using and coding gestures.

## GUI Gesture Pack

The GUI Gesture Pack (GP) is specifically designed for GUI control in which the user needs to navigate through an application. All GUI gestures should be done while the user is facing the camera.

The GUI GP includes the following gestures for both the Right and Left hands:

- Side Scroll Left
- Side Scroll Right
- Scroll Up
- Scroll Down
- Click

## Side Scroll Left, Side Scroll Right

The side scroll gestures are done by holding the hand in front of the body, at the shoulder level and moving the hand to a certain side and back to the initial position (i.e. Scroll Left is done by moving the hand from the shoulder level to the left and back). The movement should be fluent and fast and completed in a horizontal line (take less than 1 second). See Figure below.

The gestures you can register are:

- Right Hand: {_rightScrollRight, _rightScrollLeft}
- Left Hand: {_leftScrollLeft, _leftScrollRight}

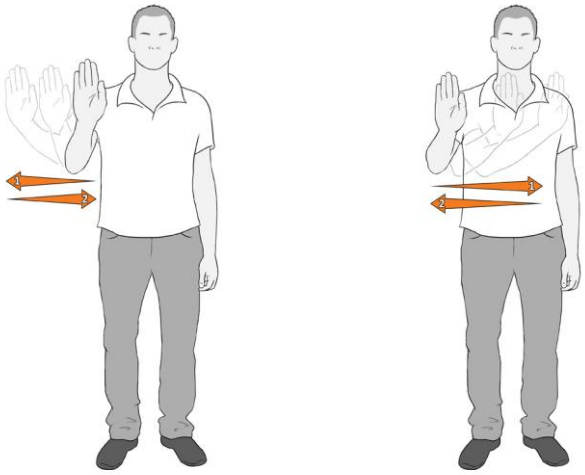

*Figure 8: Side scrolls*

## Scroll Up

The scroll up gesture is done by holding the hand in front of the body, at the shoulder level, and moving the hand up to the head level and back to the initial position. The movement should be fluent and fast and in a straight vertical line (take less than 1 second). See Figure below.

The gestures you can register are:

- Right Hand: {_rightScrollUp}
- Left Hand: {_leftScrollUp}

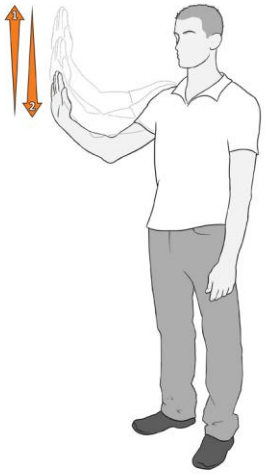

*Figure 9: Scroll up*

## Scroll Down

The scroll down gesture is done by moving the hand from the shoulder level down below to the hip area and then back to the initial position. The movement should be fluent and fast (take less than 1 second) and the hand should be far from the body.

The gestures you can register are:

- Right Hand: {_rightScrollDown}
- Left Hand: {_leftScrollDown}

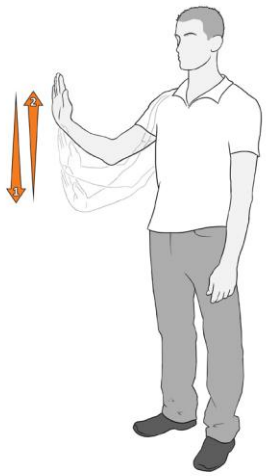

*Figure 10: Scroll down*

## Click

The click gesture is done by moving the hand from the body forward. The movement should be fluent and fast (take less than 1 second) and the hand height should be around the chest-shoulder level.

The gestures you can register are:

- Right Hand: {_rightClick}
- Left Hand: {_leftClick}

*Figure 11: Click*

## Getting Gesture Events

In order to listen to gesture events, you must first register for the gestures you are interested in. Gestures can be seamlessly registered and unregistered throughout the lifetime of the application. In order to register a gesture, call the following:

```
pSensor->enableGesture("_rightScrollRight");
```

Once registered, the gesture is checked for in each frame, and can be queried by inserting the following in the main program loop:

```
while (pSensor->hasMoreGestures()) {
   const IFiredEvent* pGesture = pSensor->popNextGesture();
   string gestureName = pGesture->getName();
   if (gestureName == "_rightScrollRight")
      // … handle event code here …
   pSensor->releaseGesture(pGesture);
}
```

For each frame, you can get a stack of gestures that were fired for that frame. You can then query the Gesture object for the type of gesture.

Alternatively, if you wish to listen to gesture events by using an event based callback mechanism instead, you can subclass the `IGestureListener` interface and register your new class to `IMotionSensor`:

```
class MyGestureListener : public IGestureListener
{
public:
   virtual void onGesture(Gesture& sender)
   {
           cout << sender.getName();
   }
};

MyGestureListener *pGListener = new MyGestureListener();
bRes = pSensor->addGestureListener("ManualGesture", pGListener);
```

***NOTE: It is good practice to minimize the complexity of the code in the body of the gesture listener, since it affects the processing time of the tracking algorithm.***

If you wish to unregister a gesture, simply call the following:

```
pSensor->disableGesture("_rightScrollRight");
```

***NOTE: Each gesture takes up additional processing on the ARM; therefore, heavy applications should avoid using all the gestures at once. This can be handled by enabling / disabling any gesture "on the fly" at any time. For example, you can write an application that starts when the user performs the gesture "_rightClick"; then after the application starts you should disable this gesture if it is no longer used.***

# Alerts

Alerts are useful for understanding when a critical problem is encountered by the tracking algorithms. Like gestures, alerts need to be enabled by the developer in order to get access to their events. In order to enable an alert, call the following:

```
pSensor->enableAlert(Alert_PlayerEnters);
```

Once enabled, the alert is checked for in each frame, and can be queried by inserting the following in the main program loop:

```
while (pSensor->hasMoreAlerts()) {
    const IFiredEvent * pAlert = pSensor->popNextAlert();
    string alertEvent = pAlert->getName();
if (alertEvent == "playerEnters")
// do something here…
    pSensor->releaseAlert(pAlert);
}
```

For each frame, you can get a queue of alerts that were fired for that frame. If you wish to disable an alert, call the following:

```
pSensor->disableAlert(Alert_PlayerEnters);
```

The following alerts are available:

| Alert | Description | Event Names* |
|---|---|---|
| **Calibration** | Indicates different stages of the automatic player calibration process. | • CalibrationBegun<br>• CalibrationDone |
| **Location** | Sends an event when the player is in an extreme location and cannot be fully tracked. Note that these alerts are different based on each sensor's range and LED intensity. | • TooClose<br>• TooFar<br>• CloseToSide<br>• OutOfFrame<br>• GoodLocation |
| **Background** | Sends an event when the player is too close to a wall in the back | • CloseToWall<br>• GoodBackground |
| **Sensor** | Sends events with various indications regarding the sensor operation, such as connection, disconnection, hardware failure, when sensor has been physically moved during operation. | • SensorCamDisconnected<br>• SensorCamReconnected<br>• SensorProblem<br>• SensorMoved |
| **Players** | Sends an event when a player enters or leaves the camera's view. | • PlayerEnters<br>• PlayerLeaves |

* When used in the code, the names of the alerts should be preceded by the prefix Alert_ (i.e. `Alert_PlayerEnters`). See the GestureDemo code sample below, for a full demonstration of using alerts.

# Sample Code

The Beckon 2.4 SDK comes with the following sample applications (in the **Beckon-SDK-{version}/samples** directory), complete with full source code and makefiles:

## Prerequisites

The following tools should be installed on your Linux computer:

**The Linux (Angstrom distro) ARM cross-toolchain**:

**To install the Linux ARM cross toolchain:**

1. Download the toolchain from the following URL:
   ***http://www.angstrom-distribution.org/toolchains/angstrom-2011.03-i686-linux-armv7a-linux-gnueabi-toolchain-qte-4.6.3.tar.bz2***
2. Extract the downloaded archive into your system root directory:

```
# sudo tar -xvjf angstrom-2011.03-i686-linux-armv7a-linux-gnueabi-toolchain-qte-4.6.3.tar.bz2 -C /
```

***Note: If you extract the archive into a different directory, you should adjust the following instructions as well as the samples' makefiles accordingly.***

**Qt embedded SDK for Angstrom/BeagleBoard.**

**To install the Qt embedded SDK for Angstrom/BeagleBoard:**

1. Install zlib library:

```
# sudo sudo apt-get install zlib1g-dev
```

2. Go to the following URL and follow the instructions (ignore the two last steps - *Setting Up Qt-Creator* and *Setting up the BeagleBoard-xM*):
   ***http://treyweaver.blogspot.com/2010/10/setting-up-qt-development-environment.html***

   **Note: in the Building Qt section execute install step using root privileges. e.g.:**

```
# sudo make install
```

**To complete the Qt Creator setup:**

1. Install Qt Creator

```
# sudo apt-get install qtcreator
```

2. Start Qt Creator.
3. Go to: **Tools > Options > Tool Chains**, select **Add** -> GCC and insert the full compiler path (/usr/local/angstrom/arm/arm-angstrom-linux-gnueabi/bin/g++).
4. Go to: **Tools > Options > Qt4**, select **Add** and provide full path to qmake (/opt/qt-arm/bin/qmake).
5. Press **Apply** and **OK**.

## QtTracking Sample

This sample displays simple GUI with two windows:

- **Left window**: The depth image as received from camera (due to quantization, the quality of the displayed image is reduced relative to the actual depth data received from camera).
- **Right window**: The binary mask indicating the tracked person with marked skeleton joints.

An ESC key terminates the application.

### To build QtTracking Sample:

1. Start Qt Creator.
2. Go to: **File > open file or project**, select QtTracking-SDK.pro project (from Beckon-SDK-{version}/samples/QtTracking) and click Finish.
3. Go to Projects tab (on the left).
    a. In Qt version select **Qt 4.6.2 (qt-arm)/**
    b. In Build steps:
        i. Remove the existing qmake step.
        ii. Go to: Add Build Step > Custom Process Step.
        iii. Check the Enable custom process step box
        iv. Command arguments: QtTracking-SDK.pro -r -spec qws/linux-DM3730-g++
        v. Command: provide full path to qmake (/opt/qt-arm/bin/qmake)
        vi. Move the newly created step upwards (so it will appear before the Make step)
4. Run Build -> Build Project command

The result executable will be found in the Beckon-SDK-{version}/bin directory.

### To run QtTracking Sample:

1. Mount or copy the bin dir to the Beagleboard.
2. Do any of the following:
    - To run live tracking using camera input:

```
# ./QtTracking-SDK –qws
```

    - To run prerecorded camera sequence (OpenNI sequence file (.oni)):

```
# ./QtTracking-SDK -qws -seq
path/to/directory/containing/the/sequence/file/
```

***Note: The path points to a directory containing the sequence not to the sequence file itself.***

    - To record a sequence enable the recording functionality by adding the *-rec* parameter:

```
# ./QtTracking-SDK -qws –rec
```

The REC button appears on the right. Clicking the REC button starts the recording of a depth data sequence (every new recorded sequence is stored in a newly created directory with date/time in its name). Clicking the button again stops recording. The maximum number of frames recorded sequence is the number of frames specified in **MAX_FRAMES_TO_RECORD** global variable defined in the Tracking.cpp source file.

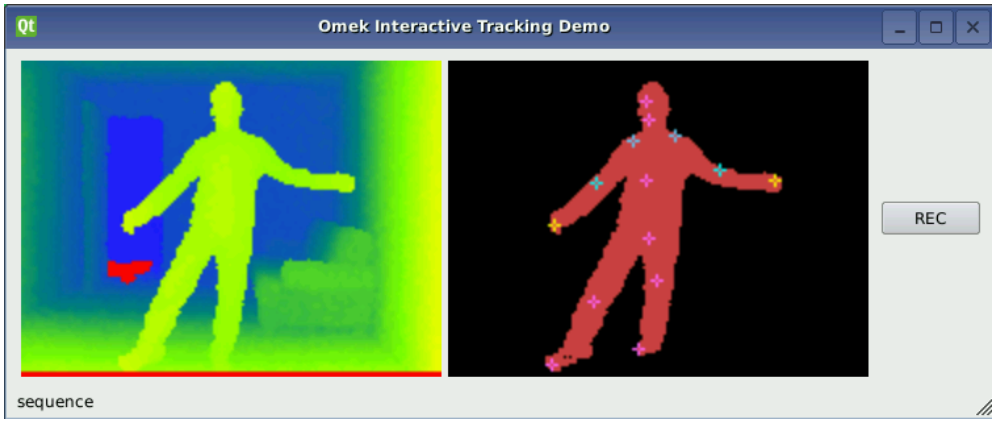


*Figure 12: Recording sequence*

## TrackingViewer3D Sample

The sample displays a stick-figure showing the tracked skeleton joints. Pressing the **space** key toggles between the front and side views. Pressing the **R** key starts/stops recording of a sequence.

The sequence is stored in the same way as in the QtTracking sample.

### To build TrackingViewer3D sample:

1. Enter Beckon-SDK-{version}/samples/TrackingViewer3D directory and run make.

The result executable can be found in the Beckon-SDK-{version}/bin directory.

### To run TrackingViewer3D sample:

1. Mount or copy the bin dir to the Beagleboard.
2. Do any of the following:
   - To run live tracking using camera input:

```
# ./TrackingViewer3D
```

   - To run prerecorded camera sequence:

```
# ./TrackingViewer3D -seq
path/to/directory/containing/the/sequence/file/
```

***Note: The path points to a directory containing the sequence not to the sequence file itself.***

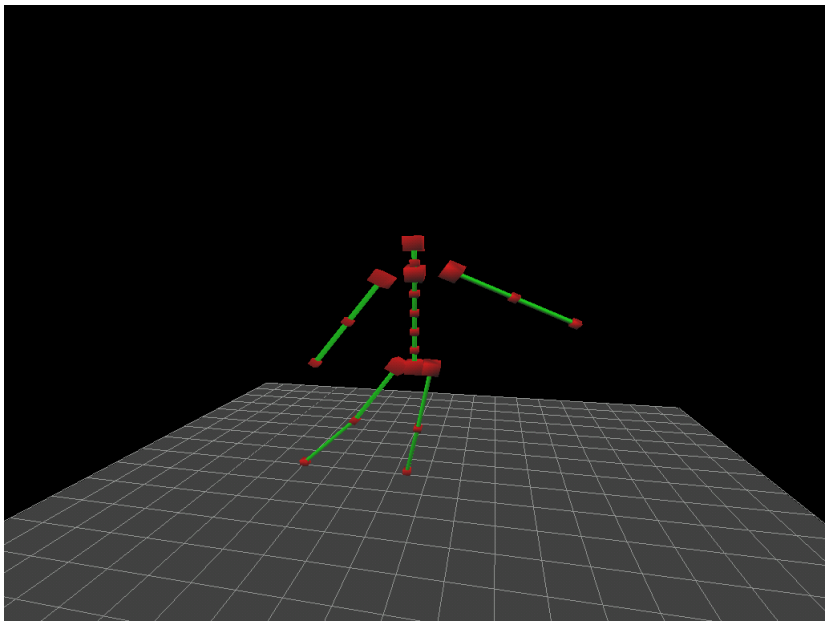

*Figure 13: Tracking Viewer 3D sample*

## GestureDemo Sample

The sample track a human and prints each time a requested gesture is activated

### To build GestureDemo sample:

1. Enter Beckon-SDK-{version}/samples/GestureDemo directory and run make.

The result executable will be found in the Beckon-SDK-{version}/bin directory.

### To run GestureDemo sample:

1. Mount or copy the bin dir to the Beagleboard.
2. Do any of the following:
   - To run live tracking using camera input:

```
# ./GestureDemo -gest <gesture name> -gest ...
```

     Supported gestures are:

     _rightClick _leftClick

     _rightScrollRight _rightScrollLeft _rightScrollUp _rightScrollDown

     _leftScrollRight _leftScrollLeft _leftScrollUp _leftScrollDown

   - To run prerecorded camera sequence:

```
# ./GestureDemo -seq path/to/directory/containing/the/sequence/file/
-gest <gesture name> -gest ...
```

***Note: The path points to a directory containing the sequence not to the sequence file itself.***

***Additional Notes:***

***1) It is advised to run /home/root/bin/reset-dsp.sh script before every execution of a sample or any other application using DSP (can be found on the provided Angstrom image).***

***2) Both pre-built samples can be found in /home/root/demo directory on the provided Angstrom image.***

# Support

Omek is dedicated to providing you support quickly and efficiently. All support issues are handled through the Support Forum where there is free access to user forums, software downloads and the knowledgebase. You can access your account here: http://support.omekinteractive.com/. We welcome your feedback and input on how to continue to improve our offering.