Contents

About Azure Kinect DK

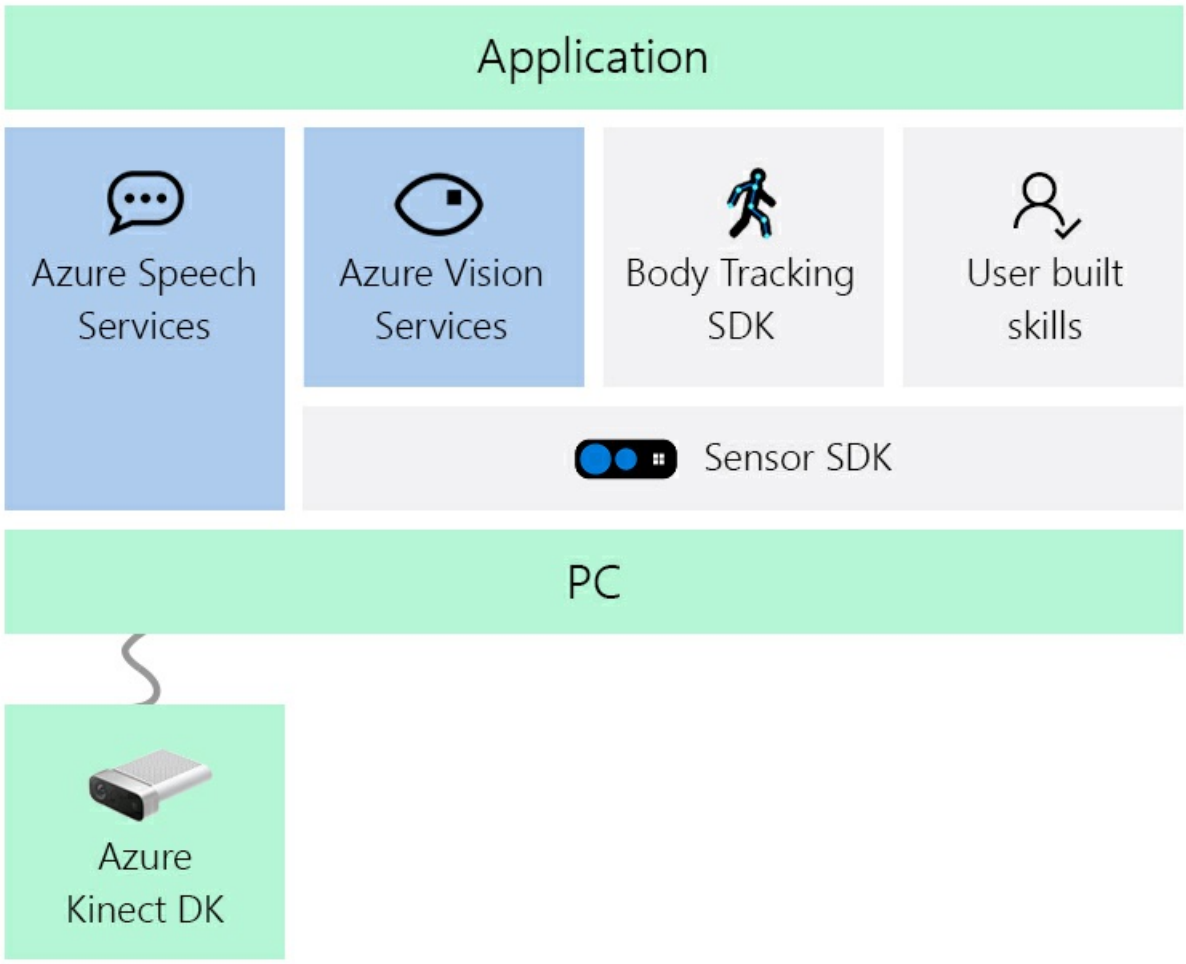
11/12/2019 • 2 minutes to read • Edit Online

Azure Kinect DK is a developer kit with advanced AI sensors that provide sophisticated computer vision and speech models. Kinect contains a depth sensor, spatial microphone array with a video camera, and orientation sensor as an all in-one small device with multiple modes, options, and software development kits (SDKs). It is available for purchase in Microsoft online store.

The Azure Kinect DK development environment consists of the following multiple SDKs:

- Sensor SDK for low-level sensor and device access.
- Body Tracking SDK for tracking bodies in 3D.
- Speech Cognitive Services SDK for enabling microphone access and Azure cloud-based speech services.

In addition, Cognitive Vision services can be used with the device RGB camera.



Azure Kinect Sensor SDK

The Azure Kinect Sensor SDK provides low-level sensor access for Azure Kinect DK hardware sensors and device configuration.

To learn more about Azure Kinect Sensor SDK, see Using Sensor SDK.

Azure Kinect Sensor SDK features

The Sensor SDK has the following features that work once installed and run on the Azure Kinect DK:

- Depth camera access and mode control (a passive IR mode, plus wide and narrow field-of-view depth modes)
- RGB camera access and control (for example, exposure and white balance)
- Motion sensor (gyroscope and accelerometer) access
- Synchronized Depth-RGB camera streaming with configurable delay between cameras
- External device synchronization control with configurable delay offset between devices
- Camera frame meta-data access for image resolution, timestamp, etc.
- Device calibration data access

Azure Kinect Sensor SDK tools

The following tools are available in the Sensor SDK:

- A viewer tool to monitor device data streams and configure different modes.
- A sensor recording tool and playback reader API that uses the Matroska container format.
- An Azure Kinect DK firmware update tool.

Azure Kinect Body Tracking SDK

The Body Tracking SDK includes a Windows library and runtime to track bodies in 3D when used with the Azure Kinect DK hardware.

Azure Kinect Body Tracking features

The following body-tracking features are available on the accompanying SDK:

- Provides body segmentation.
- Contains an anatomically correct skeleton for each partial or full body in FOV.
- Offers a unique identity for each body.
- Can track bodies over time.

Azure Kinect Body Tracking tools

- Body Tracker has a viewer tool to track bodies in 3D.

Speech Cognitive services SDK

The Speech SDK enables Azure-connected speech services.

Speech services

- Speech-to-text
- Speech translation
- Text-to-Speech

> **NOTE**
> The Azure Kinect DK does not have speakers.

For additional details and information, visit Speech Service documentation.

Vision services

The following Azure Cognitive Vision Services provide Azure services that can identify and analyze content within images and videos.

- Computer vision
- Face

- Video indexer
- Content moderator
- Custom vision

Services evolve and improve constantly, so remember to check regularly for new or additional Cognitive services to improve your application. For an early look on emerging new services, check out the Cognitive services labs.

Azure Kinect hardware requirements

The Azure Kinect DK integrates Microsoft's latest sensor technology into single USB connected accessory. For more information, see Azure Kinect DK Hardware Specification.

Next steps

You now have an overview of Azure Kinect DK. The next step is to dive in and set it up!

Quickstart: Set up Azure Kinect DK

Quickstart: Set up your Azure Kinect DK

2/14/2020 • 2 minutes to read • Edit Online

This quickstart provides guidance about how to set up your Azure Kinect DK. We'll show you how to test sensor stream visualization and use the Azure Kinect Viewer.

If you don't have an Azure subscription, create a free account before you begin.

System requirements

Check System requirements to verify that your host PC configuration meets all Azure Kinect DK minimum requirements.

Set up hardware

> **NOTE**
> Make sure to remove the camera protective film before using the device.

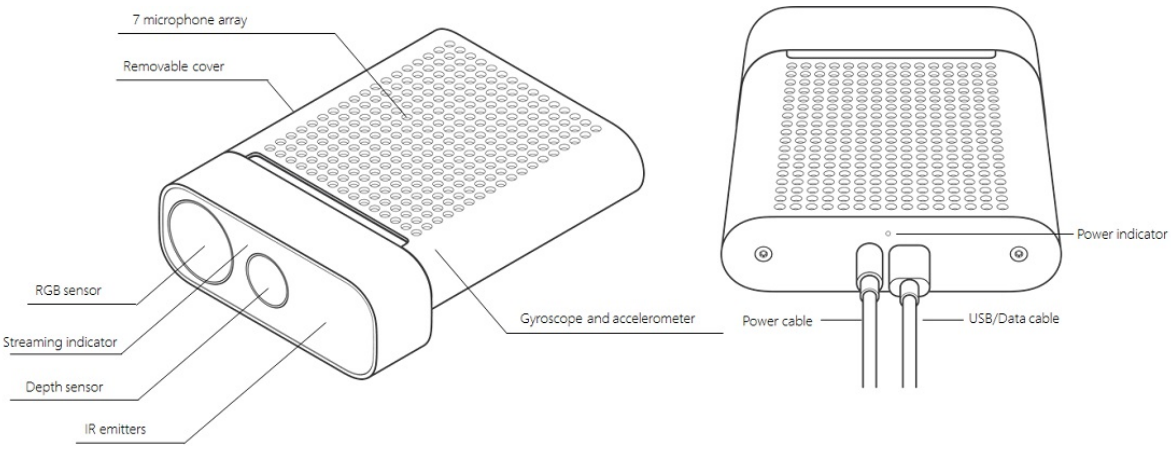
1. Plug the power connector into the power jack on the back of your device. Connect the USB power adapter to the other end of the cable, and then plug the adapter into a power outlet.
2. Connect the USB data cable into your device, and then to a USB 3.0 port on your PC.

   > **NOTE**
   > For best results, connect the cable directly to the device and to the PC. Avoid using extensions or extra adapters in the connection.

3. Verify that the power indicator LED (next to the USB cable) is solid white.

   Device power-on takes a few seconds. The device is ready to use when the front-facing LED streaming indicator turns off.

   For more information about the power indicator LED, see What does the light mean?



Download the SDK

1. Select the link to Download the SDK.
2. Install the SDK on your PC.

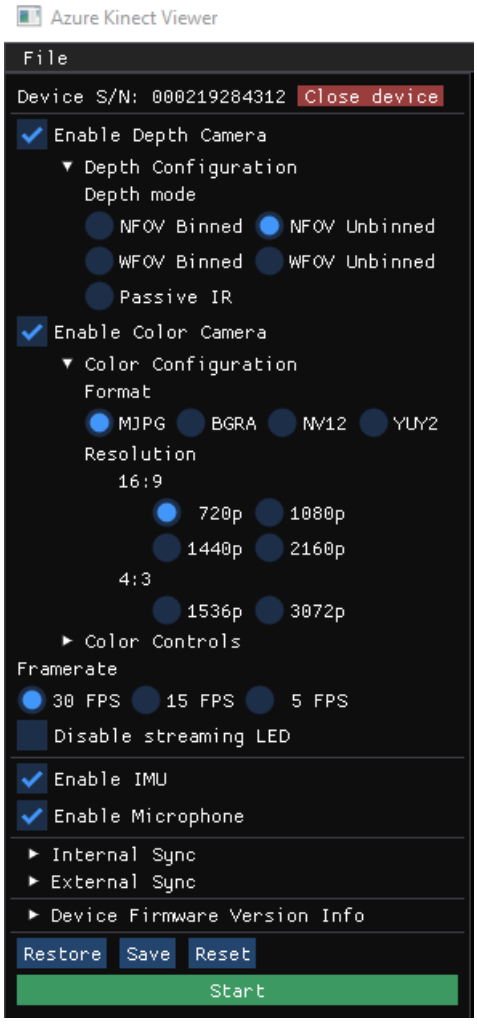
Update Firmware

To work properly, the SDK requires the latest version of the device firmware. To check and update your firmware version,follow the steps in Update Azure Kinect DK firmware.

Verify that the device streams data

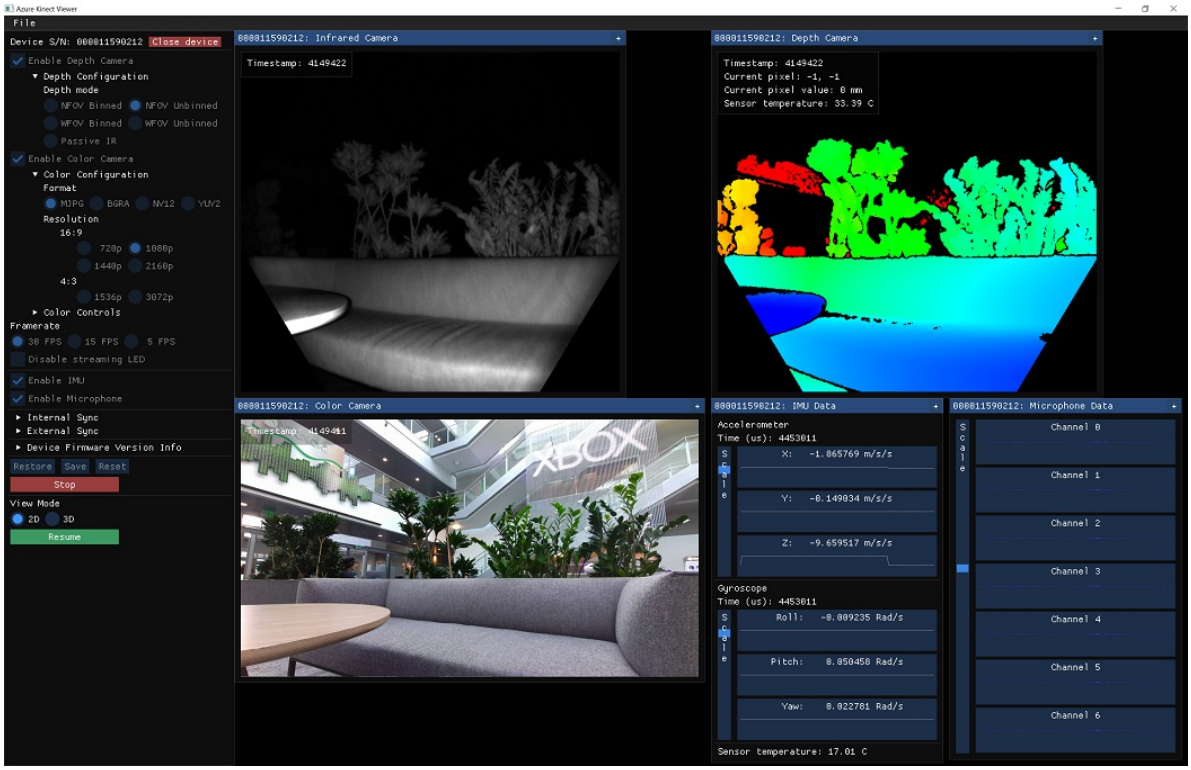
1. Launch the Azure Kinect Viewer. You can start this tool by using one of these methods:

   - You can launch the viewer from the command line or by double-clicking the executable file. The file, `k4aviewer.exe`, resides in the SDK tools directory (for example, `C:\Program Files\Azure Kinect SDK vX.Y.Z\tools\k4aviewer.exe`, where `X.Y.Z` is the installed version of the SDK).
   - You can launch Azure Kinect Viewer from the device **Start** menu.
2. In Azure Kinect Viewer, select **Open Device** > **Start**.



3. Verify that the tool visualizes each sensor stream:

   - Depth camera
   - Color camera
   - Infrared camera
   - IMU
   - Microphones



You're done with your Azure Kinect DK setup. Now you can start developing your application or integrating services.

If you have any issues, check Troubleshooting.

See also

- Azure Kinect DK hardware information
- Update device firmware
- Learn more about Azure Kinect Viewer

Next steps

After the Azure Kinect DK is ready and working, you can also learn how to

Record sensor streams to a file

Quickstart: Record Azure Kinect sensor streams to a file

11/12/2019 • 2 minutes to read • Edit Online

This quickstart provides information about how you can use the Azure Kinect recorder tool to record data streams from the Sensor SDK to a file.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

This quickstart assumes:

- You have the Azure Kinect DK connected to your host PC and powered properly.
- You have finished setting up the hardware.

Create recording

1. Open a command prompt, and provide the path to the Azure Kinect recorder, located in the installed tools directory as `k4arecorder.exe`. For example: `C:\Program Files\Azure Kinect SDK\tools\k4arecorder.exe`.
2. Record 5 seconds.

   ```
   k4arecorder.exe -l 5 %TEMP%\output.mkv
   ```

3. By default, the recorder uses the NFOV Unbinned depth mode and outputs 1080p RGB at 30 fps including IMU data. For a complete overview of recording options and tips, refer to Azure Kinect recorder.

Play back recording

You can use the Azure Kinect Viewer to play back a recording.

1. Launch `k4aviewer.exe`
2. Unfold the **Open Recording** tab and open your recording.

Next steps

Now that you've learned how to record sensor streams to a file, it's time to

Build your first Azure Kinect application

Quickstart: Build your first Azure Kinect application

8/28/2019 • 3 minutes to read • Edit Online

Getting started with the Azure Kinect DK? This quickstart will get you up and running with the device!

If you don't have an Azure subscription, create a free account before you begin.

The following functions are covered:

- `k4a_device_get_installed_count()`
- `k4a_device_open()`
- `k4a_device_get_serialnum()`
- `k4a_device_start_cameras()`
- `k4a_device_stop_cameras()`
- `k4a_device_close()`

Prerequisites

1. Set up the Azure Kinect DK device.
2. Download and install the Azure Kinect Sensor SDK.

Headers

There's only one header that you'll need, and that's `k4a.h`. Make sure your compiler of choice is set up with the SDK's lib and include folders. You'll also need the `k4a.lib` and `k4a.dll` files linked up. You may want to refer to adding the Azure Kinect library to your project.

```
#include <k4a/k4a.h>
```

Finding an Azure Kinect DK device

Multiple Azure Kinect DK devices can be connected to your computer. We'll first start by finding out how many, or if any are connected at all using the `k4a_device_get_installed_count()` function. This function should work right away, without any additional setup.

```
uint32_t count = k4a_device_get_installed_count();
```

Once you've determined there's a device connected to the computer, you can open it using `k4a_device_open()`. You can provide the index of the device you want to open, or you can just use `K4A_DEVICE_DEFAULT` for the first one.

```
// Open the first plugged in Kinect device
k4a_device_t device = NULL;
k4a_device_open(K4A_DEVICE_DEFAULT, &device);
```

As with most things in the Azure Kinect library, when you open something, you should also close it when you're finished with it! When you're shutting down, remember to make a call to `k4a_device_close()`.

```
k4a_device_close(device);
```

Once the device is open, we can make a test to ensure it's working. So let's read the device's serial number!

```
// Get the size of the serial number
size_t serial_size = 0;
k4a_device_get_serialnum(device, NULL, &serial_size);

// Allocate memory for the serial, then acquire it
char *serial = (char*)(malloc(serial_size));
k4a_device_get_serialnum(device, serial, &serial_size);
printf("Opened device: %s\n", serial);
free(serial);
```

Starting the cameras

Once you've opened the device, you'll need to configure the camera with a `k4a_device_configuration_t` object. Camera configuration has a number of different options. Choose the settings that best fit your own scenario.

```
// Configure a stream of 4096x3072 BRGA color data at 15 frames per second
k4a_device_configuration_t config = K4A_DEVICE_CONFIG_INIT_DISABLE_ALL;
config.camera_fps       = K4A_FRAMES_PER_SECOND_15;
config.color_format     = K4A_IMAGE_FORMAT_COLOR_BGRA32;
config.color_resolution = K4A_COLOR_RESOLUTION_3072P;

// Start the camera with the given configuration
k4a_device_start_cameras(device, &config);

// ...Camera capture and application specific code would go here...

// Shut down the camera when finished with application logic
k4a_device_stop_cameras(device);
```

Error handling

For the sake of brevity and clarity, we don't show error handling in some inline examples. However, error handling is always important! Many functions will return a general success/failure type `k4a_result_t`, or a more specific variant with detailed information such as `k4a_wait_result_t`. Check the docs or IntelliSense for each function to see what error messages you should expect to see from it!

You can use the `K4A_SUCCEEDED` and `K4A_FAILED` macros to check the result of a function. So instead of just opening an Azure Kinect DK device, we might guard the function call like this:

```
// Open the first plugged in Kinect device
k4a_device_t device = NULL;
if ( K4A_FAILED( k4a_device_open(K4A_DEVICE_DEFAULT, &device) ) )
{
    printf("Failed to open k4a device!\n");
    return;
}
```

Full source

```
#pragma comment(lib, "k4a.lib")
#include <k4a/k4a.h>

#include <stdio.h>
#include <stdlib.h>

int main()
{
    uint32_t count = k4a_device_get_installed_count();
    if (count == 0)
    {
        printf("No k4a devices attached!\n");
        return 1;
    }

    // Open the first plugged in Kinect device
    k4a_device_t device = NULL;
    if (K4A_FAILED(k4a_device_open(K4A_DEVICE_DEFAULT, &device)))
    {
        printf("Failed to open k4a device!\n");
        return 1;
    }

    // Get the size of the serial number
    size_t serial_size = 0;
    k4a_device_get_serialnum(device, NULL, &serial_size);

    // Allocate memory for the serial, then acquire it
    char *serial = (char*)(malloc(serial_size));
    k4a_device_get_serialnum(device, serial, &serial_size);
    printf("Opened device: %s\n", serial);
    free(serial);

    // Configure a stream of 4096x3072 BRGA color data at 15 frames per second
    k4a_device_configuration_t config = K4A_DEVICE_CONFIG_INIT_DISABLE_ALL;
    config.camera_fps       = K4A_FRAMES_PER_SECOND_15;
    config.color_format     = K4A_IMAGE_FORMAT_COLOR_BGRA32;
    config.color_resolution = K4A_COLOR_RESOLUTION_3072P;

    // Start the camera with the given configuration
    if (K4A_FAILED(k4a_device_start_cameras(device, &config)))
    {
        printf("Failed to start cameras!\n");
        k4a_device_close(device);
        return 1;
    }

    // Camera capture and application specific code would go here

    // Shut down the camera when finished with application logic
    k4a_device_stop_cameras(device);
    k4a_device_close(device);

    return 0;
}
```

Next steps

Learn how to find and open a Azure Kinect DK device using Sensor SDK

Find and open a device

Quickstart: Set up Azure Kinect body tracking

1/24/2020 • 2 minutes to read • Edit Online

This quickstart will guide you through the process of getting body tracking running on your Azure Kinect DK.

System requirements

The Body Tracking SDK requires a NVIDIA GPU installed in the host PC. The recommended body tracking host PC requirement is described in system requirements page.

Install software

Install the latest NVIDIA Driver

Download and install the latest NVIDIA driver for your graphics card. Older drivers may not be compatible with the CUDA binaries redistributed with the body tracking SDK.

Visual C++ Redistributable for Visual Studio 2015

Download and install Visual C++ Redistributable for Visual Studio 2015.

Set up hardware

Set up Azure Kinect DK

Launch the Azure Kinect Viewer to check that your Azure Kinect DK is set up correctly.

Download the Body Tracking SDK

1. Select the link to Download the Body Tracking SDK
2. Install the Body Tracking SDK on your PC.

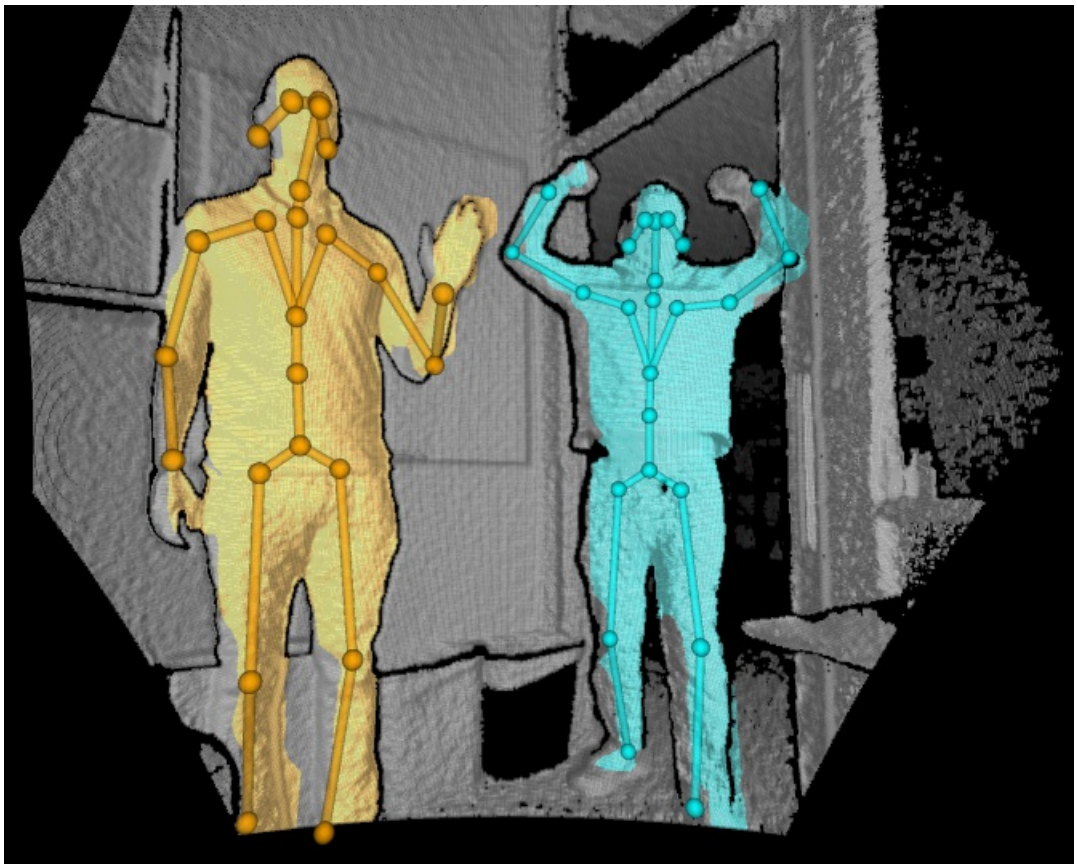
Verify body tracking

Launch the **Azure Kinect Body Tracking Viewer** to check that the Body Tracking SDK is set up correctly. The viewer is installed with the SDK msi installer. You can find it at your start menu or at `<SDK Installation Path>\tools\k4abt_simple_3d_viewer.exe`.

If you don't have a powerful enough GPU and still want to test the result, you can launch the the **Azure Kinect Body Tracking Viewer** in the command line by the following command:

```
<SDK Installation Path>\tools\k4abt_simple_3d_viewer.exe CPU
```

If everything is set up correctly, a window with a 3D point cloud and tracked bodies should appear.



Examples

You can find the examples about how to use the body tracking SDK here.

Next steps

Build your first body tracking application

Quickstart: Build an Azure Kinect body tracking application

1/24/2020 • 4 minutes to read • Edit Online

Getting started with the Body Tracking SDK? This quickstart will get you up and running with body tracking! You can find more examples in this Azure-Kinect-Sample repo.

Prerequisites

- Set up Azure Kinect DK
- Set up Body Tracking SDK
- Walk through how to build your first Azure Kinect application quickstart.
- Familiarize yourself with the following Sensor SDK functions:
  - k4a_device_open()
  - k4a_device_start_cameras()
  - k4a_device_stop_cameras()
  - k4a_device_close()
- Review the documentation on the following Body Tracking SDK functions:
  - k4abt_tracker_create()
  - k4abt_tracker_enqueue_capture()
  - k4abt_tracker_pop_result()
  - k4abt_tracker_shutdown()
  - k4abt_tracker_destroy()

Headers

Body tracking uses a single header, `k4abt.h`. Include this header in addition to `k4a.h`. Make sure your compiler of choice is set up for both the Sensor SDK and the Body Tracking SDK `lib` and `include` folders. You also need to link to `k4a.lib` and `k4abt.lib` files. Running the application requires `k4a.dll`, `k4abt.dll`, `onnxruntime.dll`, and `dnn_model.onnx` to be in the applications execution path.

```
#include <k4a/k4a.h>
#include <k4abt.h>
```

Open device and start the camera

Your first body tracking application assumes a single Azure Kinect device connected to the PC.

Body tracking builds on the Sensor SDK. To use body tracking, you first need to open and configure the device. Use the k4a_device_open() function to open the device and then configure it with a k4a_device_configuration_t object. For best results set the depth mode to `K4A_DEPTH_MODE_NFOV_UNBINNED` or `K4A_DEPTH_MODE_WFOV_2X2BINNED`. The body tracker will not run if the depth mode is set to `K4A_DEPTH_MODE_OFF` or `K4A_DEPTH_MODE_PASSIVE_IR`.

You can find more information on finding and opening the device on this page.

You can find more information on Azure Kinect depth modes on these pages: hardware specification and k4a_depth_mode_t enumerations.

```
k4a_device_t device = NULL;
k4a_device_open(0, &device);

// Start camera. Make sure depth camera is enabled.
k4a_device_configuration_t deviceConfig = K4A_DEVICE_CONFIG_INIT_DISABLE_ALL;
deviceConfig.depth_mode = K4A_DEPTH_MODE_NFOV_UNBINNED;
deviceConfig.color_resolution = K4A_COLOR_RESOLUTION_OFF;
k4a_device_start_cameras(device, &deviceConfig);
```

Create the tracker

The first step in getting body tracking results is to create a body tracker. It needs the sensor calibration k4a_calibration_t structure. The sensor calibration can be queried using the k4a_device_get_calibration() function.

```
k4a_calibration_t sensor_calibration;
k4a_device_get_calibration(device, deviceConfig.depth_mode, deviceConfig.color_resolution,
&sensor_calibration);

k4abt_tracker_t tracker = NULL;
k4abt_tracker_configuration_t tracker_config = K4ABT_TRACKER_CONFIG_DEFAULT;
k4abt_tracker_create(&sensor_calibration, tracker_config, &tracker);
```

Get captures from the Azure Kinect device

You can find more information on retrieving image data on this page.

```
// Capture a depth frame
k4a_device_get_capture(device, &capture, TIMEOUT_IN_MS);
```

Enqueue the capture and pop the results

The tracker internally maintains an input queue and an output queue to asynchronously process the Azure Kinect DK captures more efficiently. The next step is to use the `k4abt_tracker_enqueue_capture()` function to add a new capture to the input queue. Use the `k4abt_tracker_pop_result()` function to pop a result from the output queue. The timeout value is dependent on the application and controls the queueing wait time.

Your first body tracking application uses the real-time processing pattern. Refer to get body tracking results for a detailed explanation of the other patterns.

```
k4a_wait_result_t queue_capture_result = k4abt_tracker_enqueue_capture(tracker, sensor_capture,
K4A_WAIT_INFINITE);
k4a_capture_release(sensor_capture); // Remember to release the sensor capture once you finish using it
if (queue_capture_result == K4A_WAIT_RESULT_FAILED)
{
    printf("Error! Adding capture to tracker process queue failed!\n");
    break;
}

k4abt_frame_t body_frame = NULL;
k4a_wait_result_t pop_frame_result = k4abt_tracker_pop_result(tracker, &body_frame, K4A_WAIT_INFINITE);
if (pop_frame_result == K4A_WAIT_RESULT_SUCCEEDED)
{
    // Successfully popped the body tracking result. Start your processing
    ...

    k4abt_frame_release(body_frame); // Remember to release the body frame once you finish using it
}
```

Access the body tracking result data

The body tracking results for each sensor capture are stored in a body frame k4abt_frame_t structure. Each body frame contains three key components: a collection of body structs, a 2D body index map, and the input capture.

Your first body tracking application only accesses the number of detected bodies. Refer to access data in body frame for detailed explanation of data in a body frame.

```
size_t num_bodies = k4abt_frame_get_num_bodies(body_frame);
printf("%zu bodies are detected!\n", num_bodies);
```

Clean up

The final step is to shut down the body tracker and release the body tracking object. You also need to stop and close the device.

```
k4abt_tracker_shutdown(tracker);
k4abt_tracker_destroy(tracker);
k4a_device_stop_cameras(device);
k4a_device_close(device);
```

Full source

```
#include <stdio.h>
#include <stdlib.h>

#include <k4a/k4a.h>
#include <k4abt.h>

#define VERIFY(result, error)                                                                            \
    if(result != K4A_RESULT_SUCCEEDED)                                                                   \
    {                                                                                                    \
        printf("%s \n - (File: %s, Function: %s, Line: %d)\n", error, __FILE__, __FUNCTION__, __LINE__); \
        exit(1);                                                                                         \
    }                                                                                                    \

int main()
{
    k4a_device_t device = NULL;
```

```
    VERIFY(k4a_device_open(0, &device), "Open K4A Device failed!");

    // Start camera. Make sure depth camera is enabled.
    k4a_device_configuration_t deviceConfig = K4A_DEVICE_CONFIG_INIT_DISABLE_ALL;
    deviceConfig.depth_mode = K4A_DEPTH_MODE_NFOV_UNBINNED;
    deviceConfig.color_resolution = K4A_COLOR_RESOLUTION_OFF;
    VERIFY(k4a_device_start_cameras(device, &deviceConfig), "Start K4A cameras failed!");

    k4a_calibration_t sensor_calibration;
    VERIFY(k4a_device_get_calibration(device, deviceConfig.depth_mode, deviceConfig.color_resolution,
&sensor_calibration),
        "Get depth camera calibration failed!");

    k4abt_tracker_t tracker = NULL;
    k4abt_tracker_configuration_t tracker_config = K4ABT_TRACKER_CONFIG_DEFAULT;
    VERIFY(k4abt_tracker_create(&sensor_calibration, tracker_config, &tracker), "Body tracker initialization
failed!");

    int frame_count = 0;
    do
    {
        k4a_capture_t sensor_capture;
        k4a_wait_result_t get_capture_result = k4a_device_get_capture(device, &sensor_capture,
K4A_WAIT_INFINITE);
        if (get_capture_result == K4A_WAIT_RESULT_SUCCEEDED)
        {
            frame_count++;
            k4a_wait_result_t queue_capture_result = k4abt_tracker_enqueue_capture(tracker, sensor_capture,
K4A_WAIT_INFINITE);
            k4a_capture_release(sensor_capture); // Remember to release the sensor capture once you finish
using it
            if (queue_capture_result == K4A_WAIT_RESULT_TIMEOUT)
            {
                // It should never hit timeout when K4A_WAIT_INFINITE is set.
                printf("Error! Add capture to tracker process queue timeout!\n");
                break;
            }
            else if (queue_capture_result == K4A_WAIT_RESULT_FAILED)
            {
                printf("Error! Add capture to tracker process queue failed!\n");
                break;
            }

            k4abt_frame_t body_frame = NULL;
            k4a_wait_result_t pop_frame_result = k4abt_tracker_pop_result(tracker, &body_frame,
K4A_WAIT_INFINITE);
            if (pop_frame_result == K4A_WAIT_RESULT_SUCCEEDED)
            {
                // Successfully popped the body tracking result. Start your processing

                size_t num_bodies = k4abt_frame_get_num_bodies(body_frame);
                printf("%zu bodies are detected!\n", num_bodies);

                k4abt_frame_release(body_frame); // Remember to release the body frame once you finish using
it
            }
            else if (pop_frame_result == K4A_WAIT_RESULT_TIMEOUT)
            {
                //  It should never hit timeout when K4A_WAIT_INFINITE is set.
                printf("Error! Pop body frame result timeout!\n");
                break;
            }
            else
            {
                printf("Pop body frame result failed!\n");
                break;
            }
        }
        else if (get_capture_result == K4A_WAIT_RESULT_TIMEOUT)
```

```
        {
            // It should never hit time out when K4A_WAIT_INFINITE is set.
            printf("Error! Get depth frame time out!\n");
            break;
        }
        else
        {
            printf("Get depth capture returned error: %d\n", get_capture_result);
            break;
        }

    } while (frame_count < 100);

    printf("Finished body tracking processing!\n");

    k4abt_tracker_shutdown(tracker);
    k4abt_tracker_destroy(tracker);
    k4a_device_stop_cameras(device);
    k4a_device_close(device);

    return 0;
}
```

Next steps

Get body tracking results

Azure Kinect DK depth camera

11/12/2019 • 5 minutes to read • Edit Online

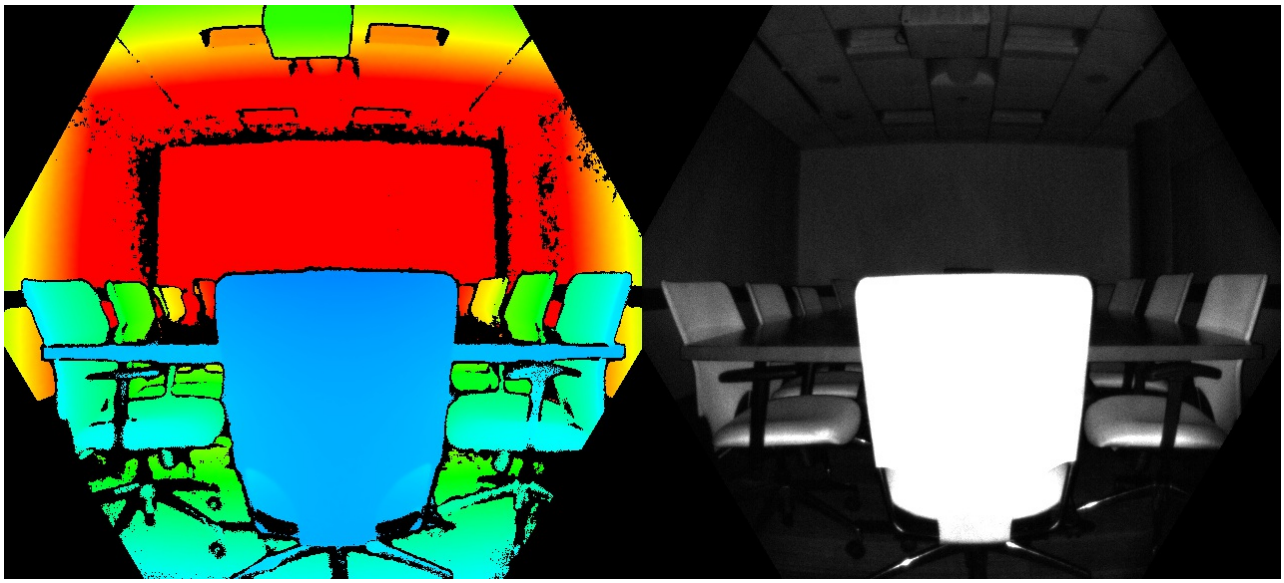
This page covers how to use the depth camera in your Azure Kinect DK. The depth camera is the second of the two cameras. As covered in previous sections, the other camera is the RGB camera.

Operating principles

The Azure Kinect DK depth camera implements the Amplitude Modulated Continuous Wave (AMCW) Time-of-Flight (ToF) principle. The camera casts modulated illumination in the near-IR (NIR) spectrum onto the scene. It then records an indirect measurement of the time it takes the light to travel from the camera to the scene and back.

These measurements are processed to generate a depth map. A depth map is a set of Z-coordinate values for every pixel of the image, measured in units of millimeters.

Along with a depth map, we also obtain a so-called clean IR reading. The value of pixels in the clean IR reading is proportional to the amount of light returned from the scene. The image looks similar to a regular IR image. The figure below shows an example depth map (left) and a corresponding clean IR image (right).



Key features

Technical characteristics of the depth camera include:

- 1-Megapixel ToF imaging chipwith advanced pixel technology enabling higher modulation frequencies and depth precision.
- Two NIR Laser diodes enabling near and wide field-of-view (FoV) depth modes.
- The world's smallest ToF pixel, at 3.5µm by 3.5µm.
- Automatic per pixel gain selection enabling large dynamic range allowing near and far objects to be captured cleanly.
- Global shutter that allows for improved performance in sunlight.
- Multi-phase depth calculation method that enables robust accuracy even in the presence of chip, laser, and power supply variation.
- Low systematic and random errors.

The depth camera transmits raw modulated IR images to the host PC. On the PC, the GPU accelerated depth engine software converts the raw signal into depth maps.The depth camera supports several modes. The **narrow field of view (FoV)** modes are ideal for scenes with smaller extents in X- and Y-dimensions, but larger extents in the Z-dimension. If the scene has large X- and Y-extents, but smaller Z-ranges, the **wide FoV modes** are better suited.

The depth camera supports **2x2 binning modes** to extend the Z-range in comparison to the corresponding **unbinned modes**. Binning is done at the cost of lowering image resolution. All modes can be run at up to 30 frames-per-second (fps) with exception of the 1 megapixel (MP) mode that runs at a maximum frame rate of 15 fps. The depth camera also provides a **passive IR mode**. In this mode, the illuminators on the camera aren't active and only ambient illumination is observed.

Camera performance

The camera's performance is measured as systematic and random errors.

Systematic Error

Systematic error is defined as the difference between the measured depth after noise removal and the correct (ground truth) depth. We compute the temporal average over many frames of a static scene to eliminate depth noise as much as possible. More precisely, the systematic error is defined as:

$$E_{systematic} = \frac{\sum_{t=1}^{N} d_t}{N} - d_{gt}$$

Where d_t denotes the measure depth at time t, N is the number of frames used in the averaging procedure and d_{gt} is the ground truth depth.

The depth camera's systematic error specification is excluding multi-path interference (MPI). MPI is when one sensor pixel integrates light that's reflected by more than one object. MPI is partly mitigated in our depth camera using higher modulation frequencies, along with the depth invalidation, which we'll introduce later.

Random error

Let's assume we take 100 images of the same object without moving the camera. The depth of the object will be

slightly different in each of the 100 images. This difference is caused by shot noise. Shot noise is the number of photons hitting the sensor varies by a random factor over time. We define this random error on a static scene as the standard deviation of depth over time computed as:

$$E_{random} = \sqrt{\frac{\sum_{t=1}^{N}(d_t - \bar{d})^2}{N}}$$

Where N denotes the number of depth measurements, d_t represents the depth measurement at time t and d denotes the mean value computed over all depth measurements d_t.

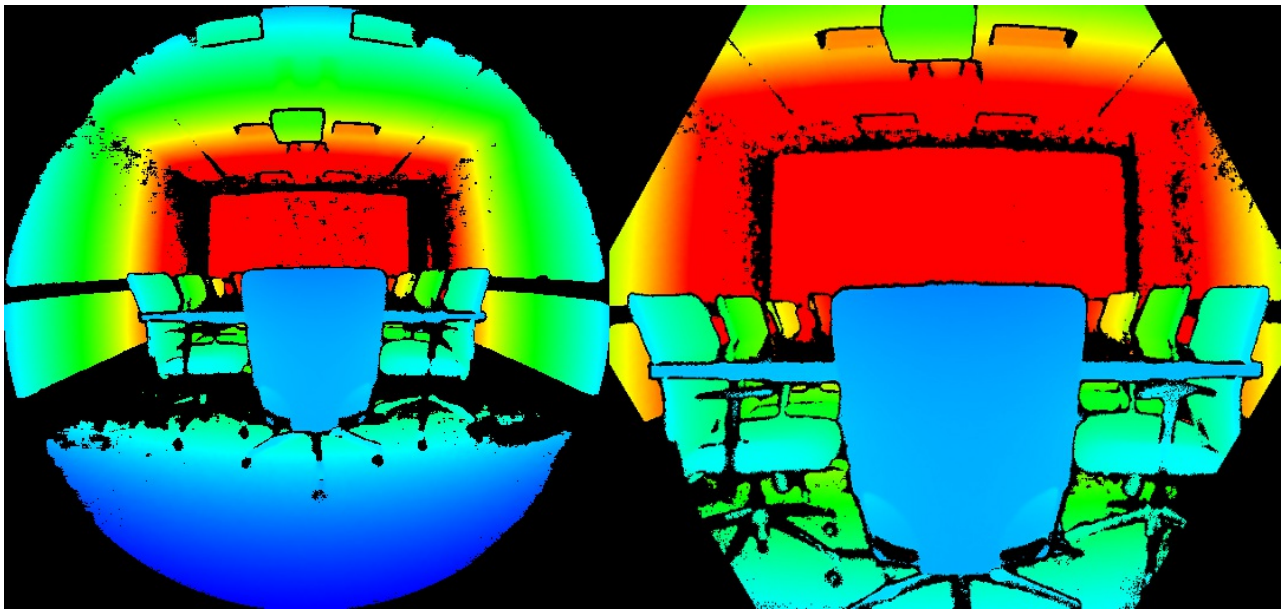
Invalidation

In certain situations, the depth camera may not provide correct values for some pixels. In these situations depth pixels are invalidated. Invalid pixels are indicated by the depth value equals to 0. Reasons for the depth engine being unable to produce correct values include:

- Outside of active IR illumination mask
- Saturated IR signal
- Low IR signal
- Filter outlier
- Multi-path interference

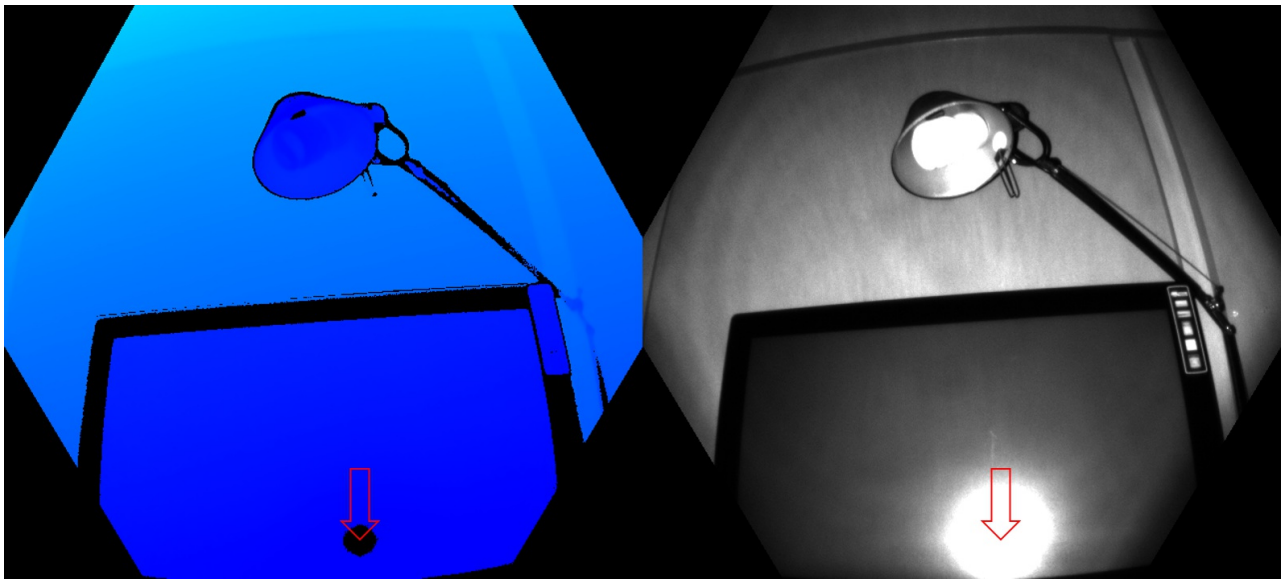
Illumination Mask

Pixels are invalidated when they're outside of the active IR illumination mask. We don't recommend using the signal of such pixels to compute depth. The figure below, shows the example of invalidation by illumination mask. The invalidated pixels are the black-color pixels outside the circle in the wide FoV modes (left), and the hexagon in the narrow FoV modes (right).

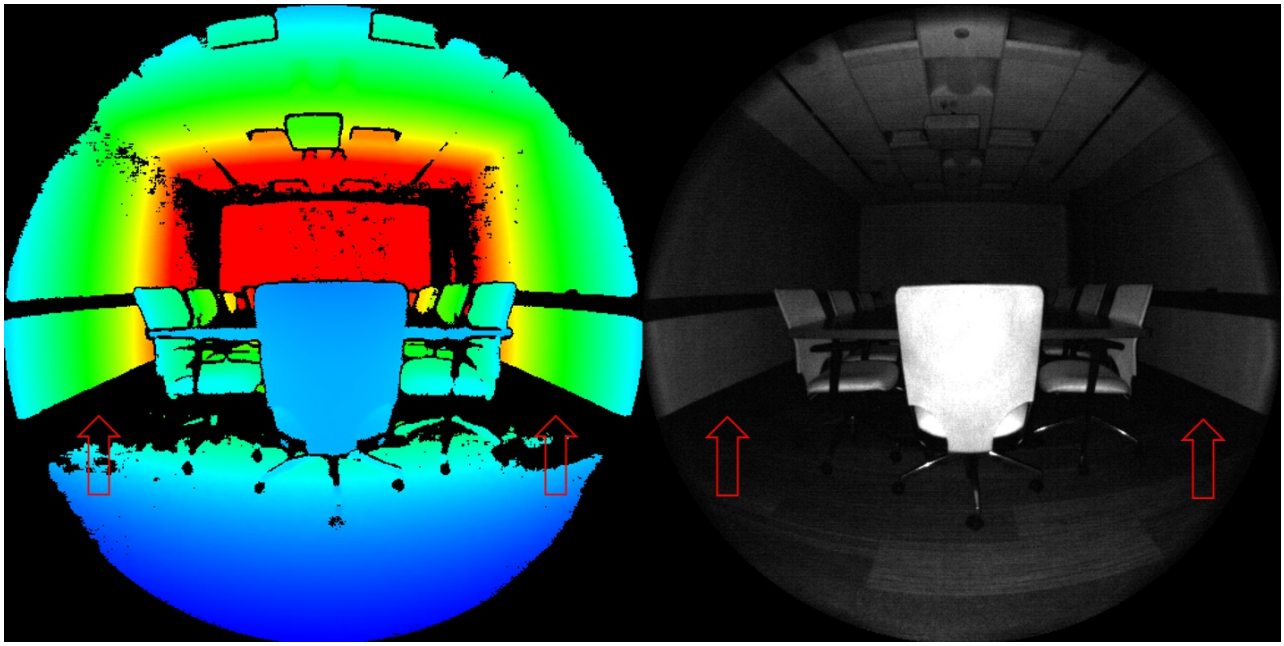


Signal strength

Pixels are invalidated when they contain a saturated IR signal. When pixels are saturated, phase information is lost. The image below, shows the example of invalidation by a saturated IR signal. See arrows pointed to the example pixels in both the depth and IR images.



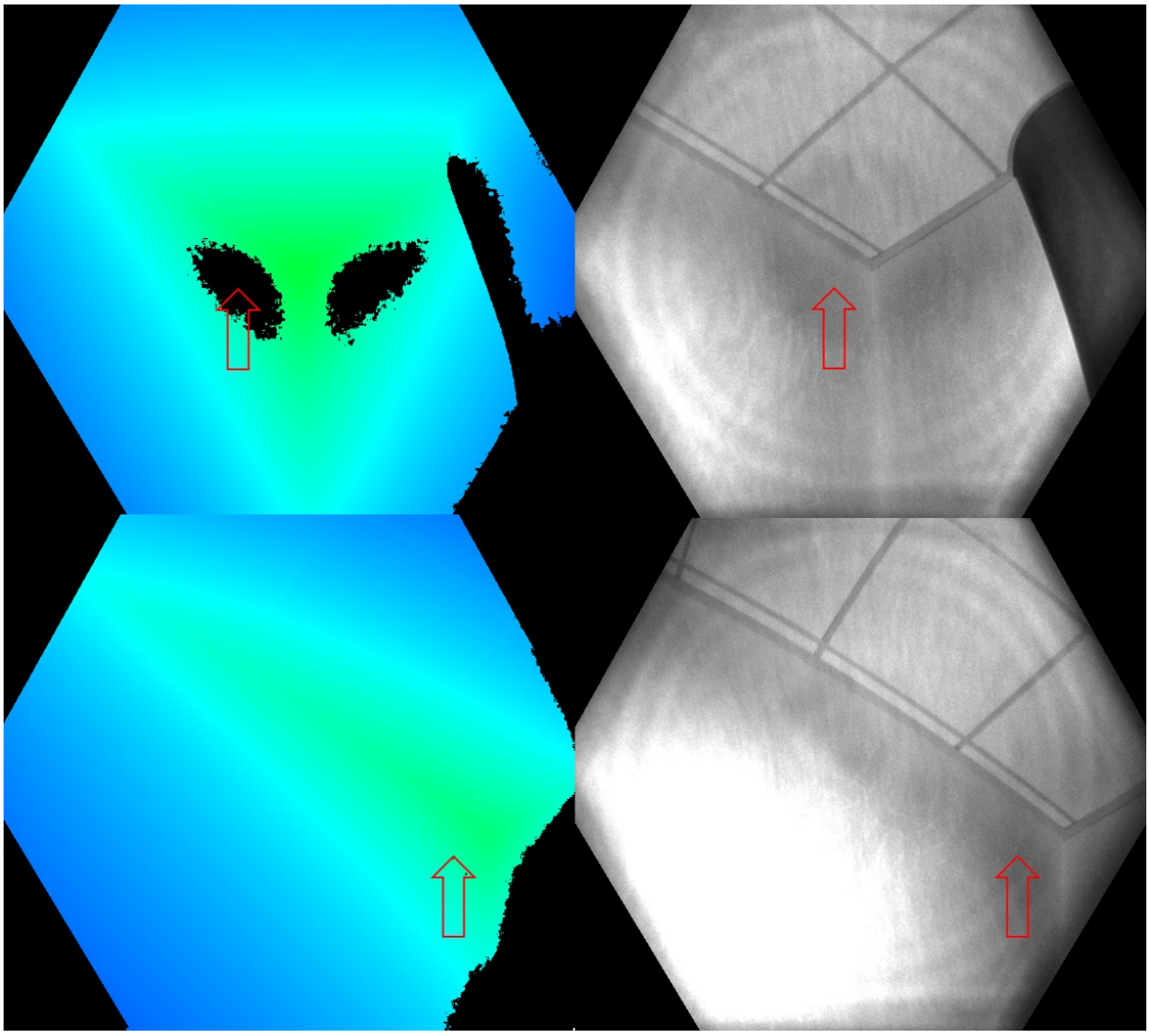
Invalidation can also occur when the IR signal isn't strong enough to generate depth. The below figure, shows the example of invalidation by a low IR signal. See the arrows pointed to example pixels in both the depth and IR images.



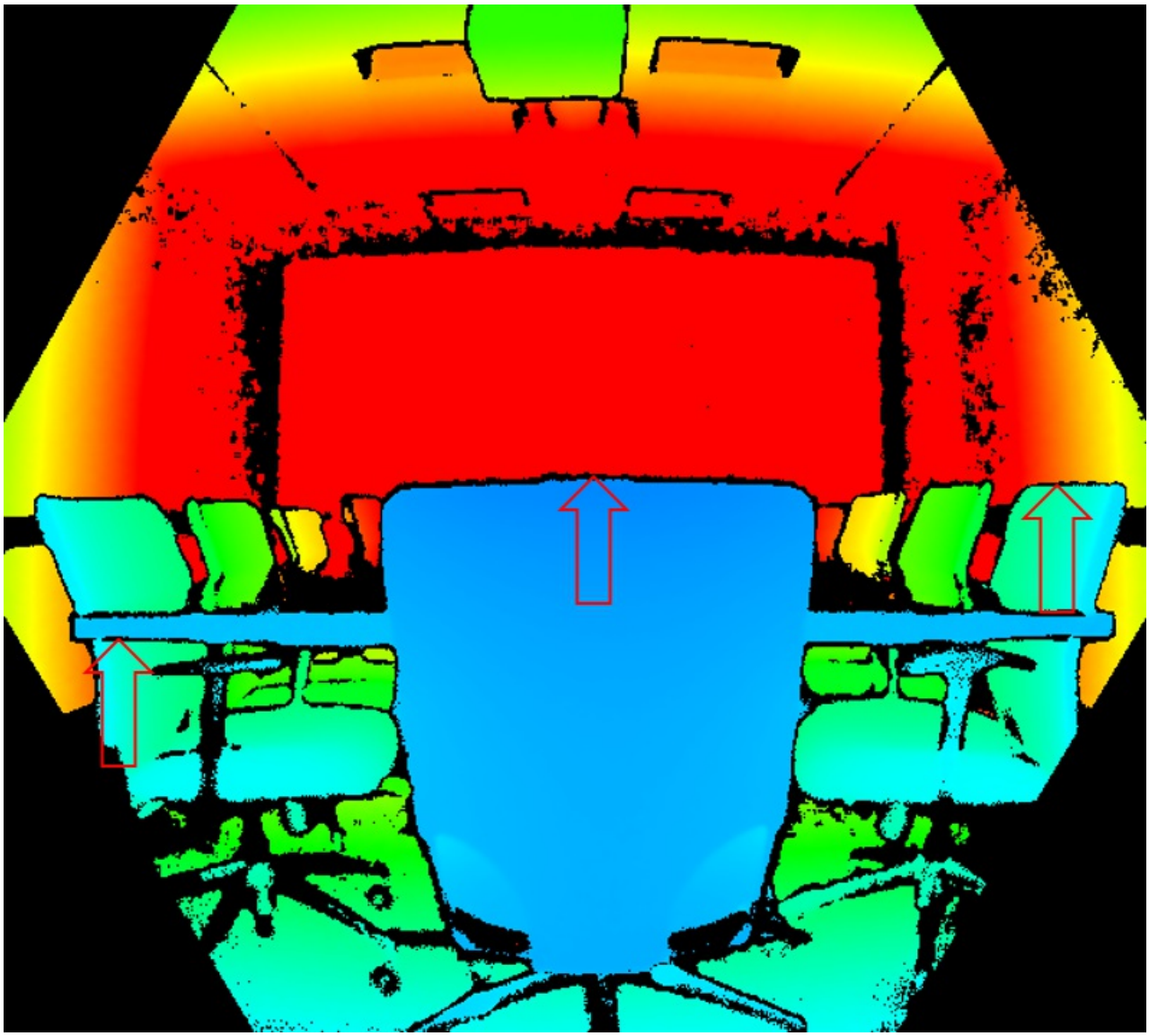
Ambiguous depth

Pixels can also be invalidated if they received signals from more than one object in the scene. A common case where this sort of invalidation can be seen is in corners. Because of the scene geometry, the IR light from the camera reflected off one wall and onto the other. This reflected light causes ambiguity in the measured depth of the pixel. Filters in the depth algorithm detect these ambiguous signals and invalidate the pixels.

The figures below show examples of invalidation by multi-path detection. You also can see how the same surface area that was invalidated from one camera view (top row) may appear again from a different camera view (bottom row). This image demonstrates that surfaces invalidated from one perspective may be visible from another.



Another common case of multipath is pixels that contain the mixed signal from foreground and background (such as around object edges). During fast motion, you may see more invalidated pixels around the edges. The additional invalidated pixels are because of the exposure interval of the raw depth capture,



Next steps

Coordinate systems

Azure Kinect DK coordinate systems

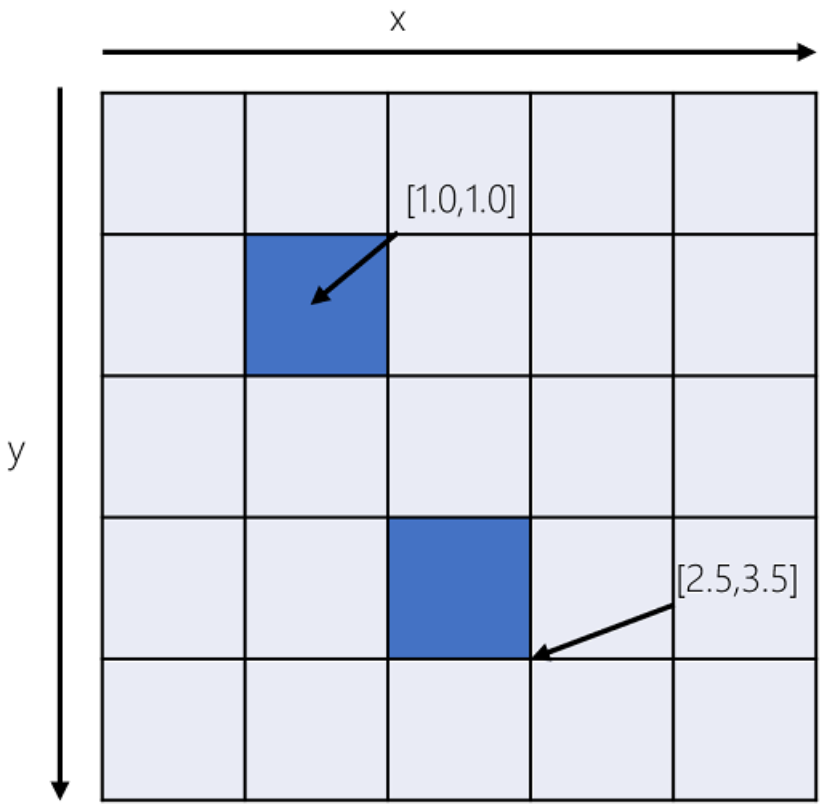
7/2/2019 • 2 minutes to read • Edit Online

In this article, we describe conventions used for 2D and 3D coordinate systems. There are separate coordinate systems associated with each sensor's device and the calibration functions allowed to transform points between them. The transformation functions transform entire images between coordinate systems.

2D coordinate systems

Both depth and color cameras are associated with an independent 2D coordinate system. An [x,y]-coordinate is represented in units of pixels where *x* ranges from 0 to width-1 and *y* ranges from 0 to height-1. Width and height depend on the chosen mode in which depth and color cameras are operated. The pixel coordinate `[0,0]` corresponds to the top-left pixel of the image. Pixel coordinates can be fractional representing subpixel coordinates.

The 2D coordinate system is 0-centered, that is, the subpixel coordinate `[0.0, 0.0]` represents the center and `[0.5,0.5]` the bottom-right corner of the pixel, as shown below.



3D coordinate systems

Each camera, the accelerometer, and the gyroscope, are associated with an independent 3D coordinate space system.

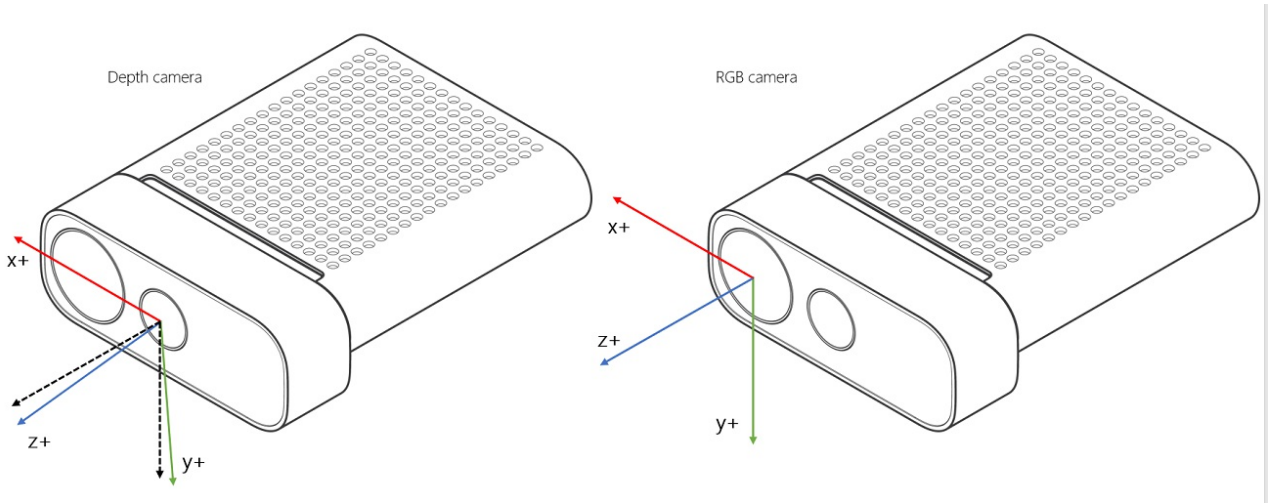
Points in the 3D-coordinate systems are represented as metric [X,Y,Z]-coordinate triplets with units in millimeters.

Depth and color camera

The origin `[0,0,0]` is located at the focal point of the camera. The coordinate system is oriented such that the positive X-axis points right, the positive Y-axis points down, and the positive Z-axis points forward.

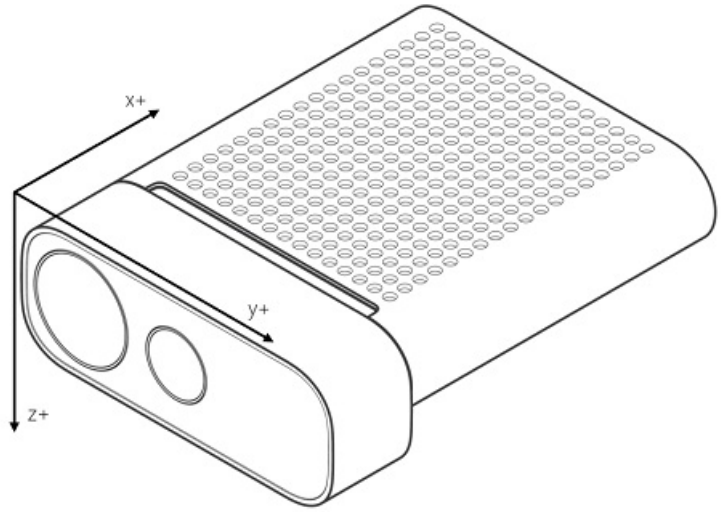
The depth camera is tilted 6 degrees downwards of the color camera, as shown below.

There are two illuminators used by the depth camera. The illuminator used in narrow field-of-view (NFOV) modes is aligned with the depth camera case, so, the illuminator is not tilted. The illuminator used in wide field-of-view (WFOV) modes is tilted an additional 1.3 degrees downward relative to the depth camera.



Gyroscope and accelerometer

The gyroscope's origin `[0,0,0]` is identical to the origin of the depth camera. The origin of the accelerometer coincides with its physical location. Both the accelerometer and gyroscope coordinate systems are right-handed. The coordinate system's positive X-axis points backward, the positive Y-axis points left, and the positive Z-axis points down, as shown below.



Next Steps

Learn about Azure Kinect Sensor SDK

Azure Kinect body tracking joints

12/10/2019 • 2 minutes to read • Edit Online

Azure Kinect body tracking can track multiple human bodies at the same time. Each body includes an ID for temporal correlation between frames and the kinematic skeleton. The number of bodies detected in each frame can be acquired using `k4abt_frame_get_num_bodies()`.

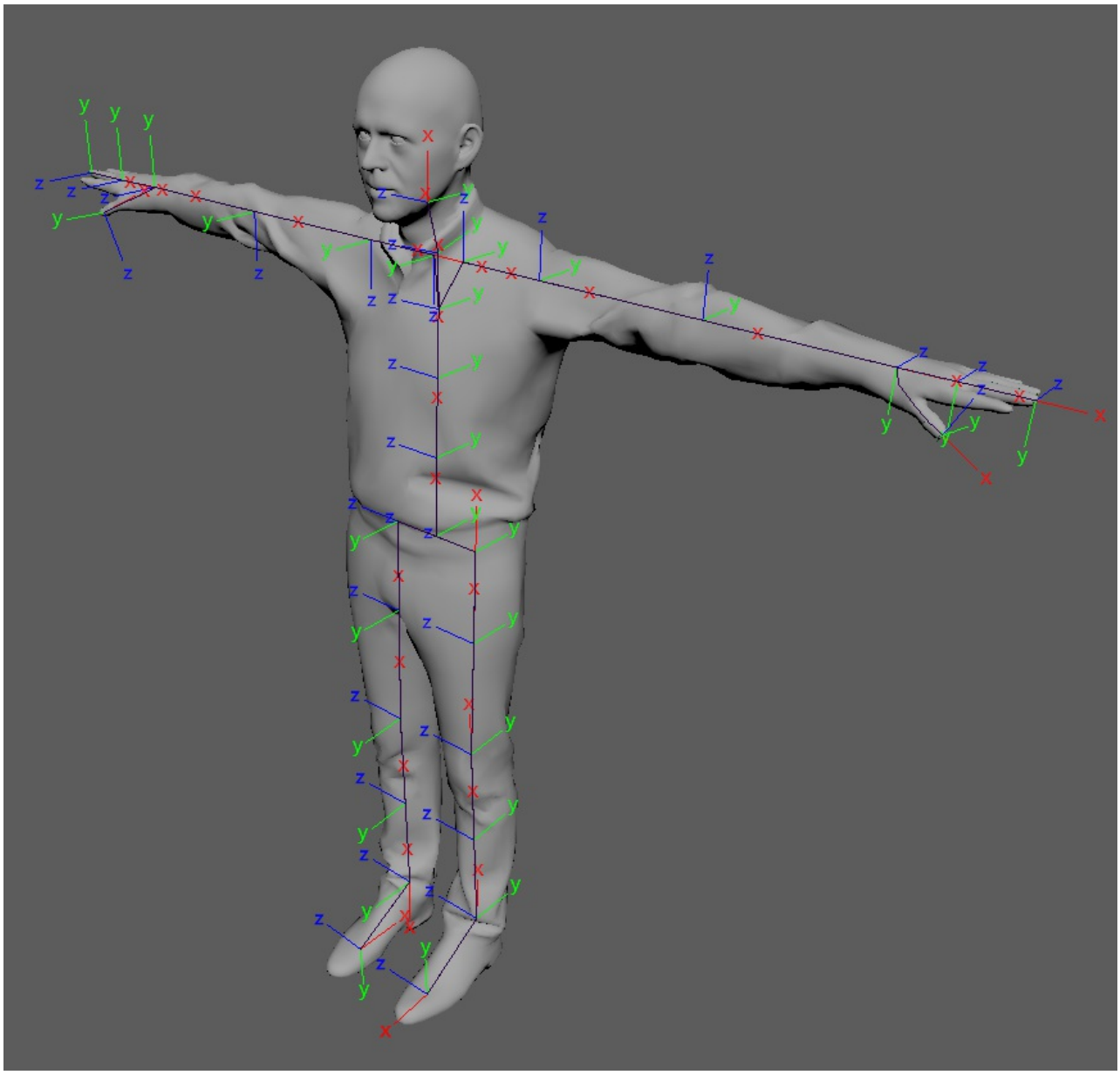
Joints

Joint position and orientation are estimates relative to the global depth sensor frame of reference. The position is specified in millimeters. The orientation is expressed as a normalized quaternion.

Joint coordinates

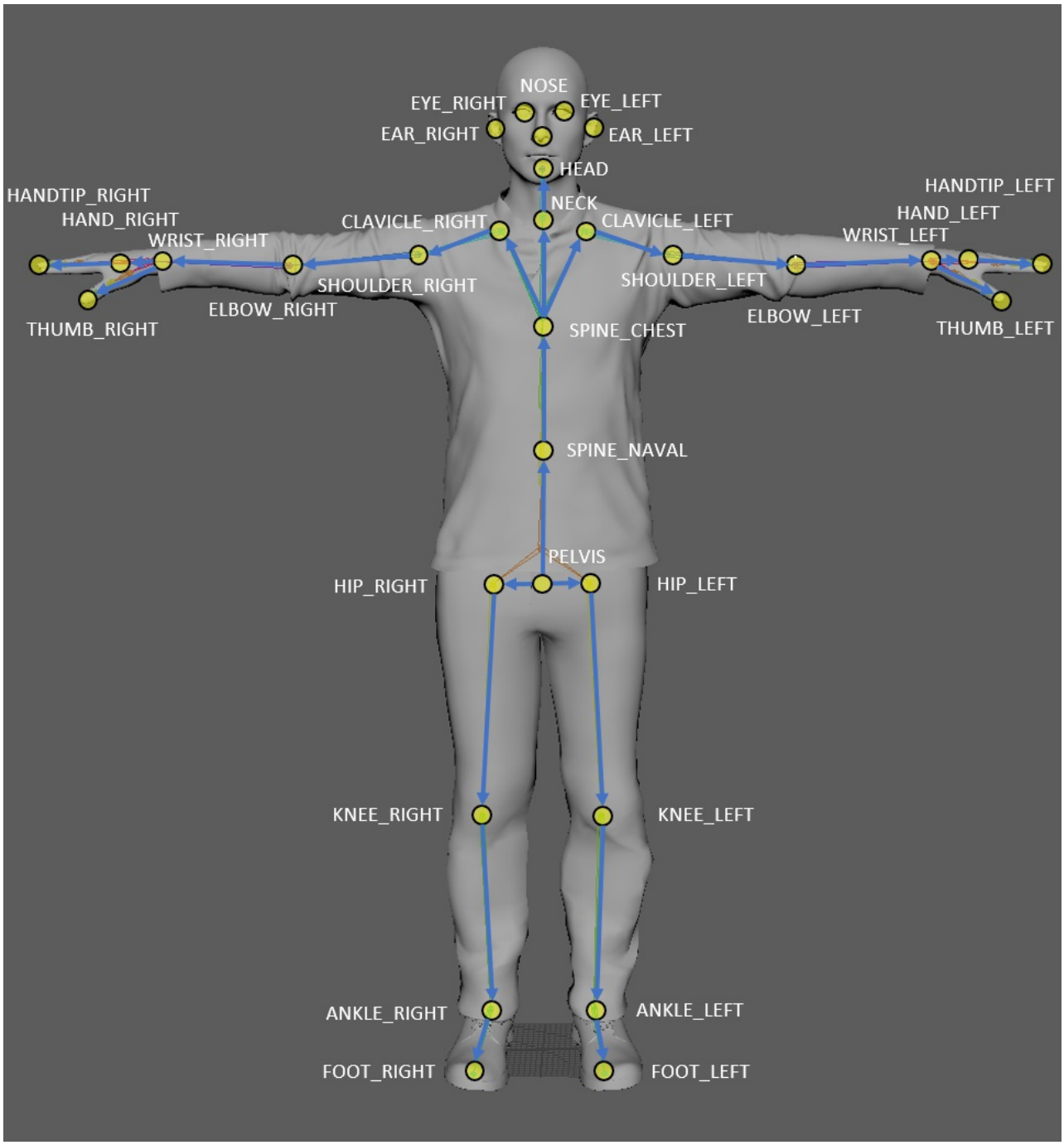
The position and orientation of each joint form its own joint coordinate system. All joint coordinate systems are absolute coordinate systems relative to the depth camera 3D coordinate system.

> **NOTE**
> Joint coordinates are in axis orientation. Axis orientation is widely used with commercial avatars, game engines, and rendering software. Using axis orientation simplifies mirrored movements e.g. raise both arms by 20 degrees.



Joint hierarchy

A skeleton includes 32 joints with the joint hierarchy flowing from the center of the body to the extremities. Each connection (bone) links the parent joint with a child joint. The figure illustrates the joint locations and connection relative to the human body.



The following table enumerates the standard joint connections.

INDEX	JOINT NAME	PARENT JOINT
0	PELVIS	-
1	SPINE_NAVAL	PELVIS
2	SPINE_CHEST	SPINE_NAVAL
3	NECK	SPINE_CHEST
4	CLAVICLE_LEFT	SPINE_CHEST
5	SHOULDER_LEFT	CLAVICLE_LEFT
6	ELBOW_LEFT	SHOULDER_LEFT

INDEX	JOINT NAME	PARENT JOINT
7	WRIST_LEFT	ELBOW_LEFT
8	HAND_LEFT	WRIST_LEFT
9	HANDTIP_LEFT	HAND_LEFT
10	THUMB_LEFT	WRIST_LEFT
11	CLAVICLE_RIGHT	SPINE_CHEST
12	SHOULDER_RIGHT	CLAVICLE_RIGHT
13	ELBOW_RIGHT	SHOULDER_RIGHT
14	WRIST_RIGHT	ELBOW_RIGHT
15	HAND_RIGHT	WRIST_RIGHT
16	HANDTIP_RIGHT	HAND_RIGHT
17	THUMB_RIGHT	WRIST_RIGHT
18	HIP_LEFT	PELVIS
19	KNEE_LEFT	HIP_LEFT
20	ANKLE_LEFT	KNEE_LEFT
21	FOOT_LEFT	ANKLE_LEFT
22	HIP_RIGHT	PELVIS
23	KNEE_RIGHT	HIP_RIGHT
24	ANKLE_RIGHT	KNEE_RIGHT
25	FOOT_RIGHT	ANKLE_RIGHT
26	HEAD	NECK
27	NOSE	HEAD
28	EYE_LEFT	HEAD
29	EAR_LEFT	HEAD
30	EYE_RIGHT	HEAD
31	EAR_RIGHT	HEAD

Next steps

Body tracking index map

Azure Kinect body tracking index map

1/24/2020 • 2 minutes to read • Edit Online

The body index map includes the instance segmentation map for each body in the depth camera capture. Each pixel maps to the corresponding pixel in the depth or IR image. The value for each pixel represents which body the pixel belongs to. It can be either background (value `K4ABT_BODY_INDEX_MAP_BACKGROUND`) or the index of a detected `k4abt_body_t`.

NOTE

The body index is different than the body id. You can query the body id from a given body index by calling API: k4abt_frame_get_body_id().

Using body index map

The body index map is stored as a `k4a_image_t` and has the same resolution as the depth or IR image. Each pixel is an 8-bit value. It can be queried from a `k4abt_frame_t` by calling `k4abt_frame_get_body_index_map`. The developer is responsible for releasing the memory for the body index map by calling `k4a_image_release()`.

Next steps

Build your first body tracking app

About Azure Kinect Sensor SDK

11/12/2019 • 2 minutes to read • Edit Online

This article provides an overview of the Azure Kinect Sensor software development kit (SDK), its features, and tools.

Features

The Azure Kinect Sensor SDK provides cross-platform low-level access for Azure Kinect device configuration and hardware sensors streams, including:

- Depth camera access and mode control (a passive IR mode, plus wide and narrow field-of-view depth modes)
- RGB camera access and control (for example, exposure and white balance)
- Motion sensor (gyroscope and accelerometer) access
- Synchronized Depth-RGB camera streaming with configurable delay between cameras
- External device synchronization control with configurable delay offset between devices
- Camera frame meta-data access for image resolution, timestamp, etc.
- Device calibration data access

Tools

- An Azure Kinect viewer to monitor device data streams and configure different modes.
- An Azure Kinect recorder and playback reader API that uses the Matroska container format.
- An Azure Kinect DK firmware update tool.

Sensor SDK

- Download Sensor SDK.
- The Sensor SDK is available in open source on GitHub.
- For more information about usage, see Sensor SDK API documentation.

Next steps

Now you learned about Azure Kinect sensor SDK, you can also:

Download sensor SDK code

Find and open device

Find then open the Azure Kinect device

11/12/2019 • 2 minutes to read • Edit Online

This article describes how you can find, then open your Azure Kinect DK. The article explains how to handle the case where there are multiple devices connected to your machine.

You can also refer to the SDK Enumerate Example that demonstrates how to use the functions in this article.

The following functions are covered:

- `k4a_device_get_installed_count()`
- `k4a_device_open()`
- `k4a_device_get_serialnum()`
- `k4a_device_close()`

Discover the number of connected devices

First get the count of currently connected Azure Kinect devices using `k4a_device_get_installed_count()`.

```
uint32_t device_count = k4a_device_get_installed_count();

printf("Found %d connected devices:\n", device_count);
```

Open a device

To get information about a device, or to read data from it, you need to first open a handle to the device using `k4a_device_open()`.

```
k4a_device_t device = NULL;

for (uint8_t deviceIndex = 0; deviceIndex < device_count; deviceIndex++)
{
    if (K4A_RESULT_SUCCEEDED != k4a_device_open(deviceIndex, &device))
    {
        printf("%d: Failed to open device\n", deviceIndex);
        continue;
    }

    ...

    k4a_device_close(device);
}
```

The `index` parameter of `k4a_device_open()` indicates which device to open if there are more than one connected. If you only expect a single device to be connected, you can pass an argument of `K4A_DEVICE_DEFAULT` or 0 to indicate the first device.

Anytime you open a device you need to call `k4a_device_close()` when you're done using the handle. No other handles can be opened to the same device until you've closed the handle.

Identify a specific device

The order devices enumerate by index won't change until devices are attached or detached. To identify a physical device, you should use the device's serial number.

To read the serial number from the device, use the `k4a_device_get_serialnum()` function after you've opened a handle.

This example demonstrates how to allocate the right amount of memory to store the serial number.

```
char *serial_number = NULL;
size_t serial_number_length = 0;

if (K4A_BUFFER_RESULT_TOO_SMALL != k4a_device_get_serialnum(device, NULL, &serial_number_length))
{
    printf("%d: Failed to get serial number length\n", deviceIndex);
    k4a_device_close(device);
    device = NULL;
    continue;
}

serial_number = malloc(serial_number_length);
if (serial_number == NULL)
{
    printf("%d: Failed to allocate memory for serial number (%zu bytes)\n", deviceIndex,
serial_number_length);
    k4a_device_close(device);
    device = NULL;
    continue;
}

if (K4A_BUFFER_RESULT_SUCCEEDED != k4a_device_get_serialnum(device, serial_number, &serial_number_length))
{
    printf("%d: Failed to get serial number\n", deviceIndex);
    free(serial_number);
    serial_number = NULL;
    k4a_device_close(device);
    device = NULL;
    continue;
}

printf("%d: Device \"%s\"\n", deviceIndex, serial_number);
```

Open the default device

In most applications, there will only be a single Azure Kinect DK attached to the same computer. If you only need to connect to the single expected device, you can call `k4a_device_open()` with `index` of `K4A_DEVICE_DEFAULT` to open the first device.

```
k4a_device_t device = NULL;
uint32_t device_count = k4a_device_get_installed_count();

if (device_count != 1)
{
    printf("Unexpected number of devices found (%d)\n", device_count);
    goto Exit;
}

if (K4A_RESULT_SUCCEEDED != k4a_device_open(K4A_DEVICE_DEFAULT, &device))
{
    printf("Failed to open device\n");
    goto Exit;
}
```

Next steps

Retrieve Images

Retrieve IMU Samples

Retrieve Azure Kinect image data

11/12/2019 • 2 minutes to read • Edit Online

This page provides details about how to retrieve images from the Azure Kinect. The article demonstrates how to capture and access images coordinated between the device's color and depth cameras.device. To access images, you must first open and configure the device, then you can capture images. Before you configure and capture an image, you must Find and open device.

You can also refer to the SDK Streaming Example that demonstrates how to use the functions in this article.

The following functions are covered:

- `k4a_device_start_cameras()`
- `k4a_device_get_capture()`
- `k4a_capture_get_depth_image()`
- `k4a_image_get_buffer()`
- `k4a_image_release()`
- `k4a_capture_release()`
- `k4a_device_stop_cameras()`

Configure and start the device

The two cameras available on your Kinect device support multiple modes, resolutions, and output formats. For a complete list, refer to the Azure Kinect Development Kit hardware specifications.

The streaming configuration is set using values in the `k4a_device_configuration_t` structure.

```
k4a_device_configuration_t config = K4A_DEVICE_CONFIG_INIT_DISABLE_ALL;
config.camera_fps = K4A_FRAMES_PER_SECOND_30;
config.color_format = K4A_IMAGE_FORMAT_COLOR_MJPG;
config.color_resolution = K4A_COLOR_RESOLUTION_2160P;
config.depth_mode = K4A_DEPTH_MODE_NFOV_UNBINNED;

if (K4A_RESULT_SUCCEEDED != k4a_device_start_cameras(device, &config))
{
    printf("Failed to start device\n");
    goto Exit;
}
```

Once cameras are started, they'll continue to capture data until `k4a_device_stop_cameras()` is called or the device is closed.

Stabilization

When starting up devices using the multi device synchronization feature, it is highly recommended to do so using a fixed exposure setting. With a manual exposure set, it can take up to eight captures from the device before images and framerate stabilize. With auto exposure, it can take up to 20 captures before images and framerate stabilize.

Get a capture from the device

Images are captured from the device in a correlated manner. Each captured image contains a depth image, an IR

image, a color image, or a combination of images.

By default, the API will only return a capture once it has received all of the requested images for the streaming mode. You can configure the API to return partial captures with only depth or color images as soon as they're available by clearing the `synchronized_images_only` parameter of the `k4a_device_configuration_t`.

```
// Capture a depth frame
switch (k4a_device_get_capture(device, &capture, TIMEOUT_IN_MS))
{
case K4A_WAIT_RESULT_SUCCEEDED:
    break;
case K4A_WAIT_RESULT_TIMEOUT:
    printf("Timed out waiting for a capture\n");
    continue;
    break;
case K4A_WAIT_RESULT_FAILED:
    printf("Failed to read a capture\n");
    goto Exit;
}
```

Once the API has successfully returned a capture, you must call `k4a_capture_release()` when you have completed using the capture object.

Get an image from the capture

To retrieve a captured image, call the appropriate function for each image type. One of:

- `k4a_capture_get_color_image()`
- `k4a_capture_get_depth_image()`
- `k4a_capture_get_ir_image()`

You must call `k4a_image_release()` on any `k4a_image_t` handle returned by these functions once you're done using the image.

Access image buffers

`k4a_image_t` has many accessor functions to get properties of the image.

To access the image's memory buffer, use k4a_image_get_buffer.

The following example demonstrates how to access a captured depth image. This same principle applies to other image types. However, make sure you replace the image-type variable with the correct image type, such as IR, or color.

```
// Access the depth16 image
k4a_image_t image = k4a_capture_get_depth_image(capture);
if (image != NULL)
{
    printf(" | Depth16 res:%4dx%4d stride:%5d\n",
            k4a_image_get_height_pixels(image),
            k4a_image_get_width_pixels(image),
            k4a_image_get_stride_bytes(image));

    // Release the image
    k4a_image_release(image);
}

// Release the capture
k4a_capture_release(capture);
```

Next steps

Now you know how to capture, and coordinate the cameras' images between the color and depth, using your Azure Kinect device. You also can:

Retrieve IMU samples

Access microphones

Retrieve Azure Kinect IMU samples

11/12/2019 • 2 minutes to read • Edit Online

The Azure Kinect device provides access to Inertial Motion Units (IMUs), including both the accelerometer and gyroscope types. To access IMUs samples, you must first open and configure your device, then capture IMU data. For more information, see find and open device.

IMU samples are generated at a much higher frequency than images. Samples are reported to the host at a lower rate than they're sampled. When waiting for an IMU sample, multiple samples will frequently become available at the same time.

See the Azure Kinect DK hardware specification for details on the IMU reporting rate.

Configure and start cameras

> **NOTE**
> IMU sensors can only work when the color and/or the depth cameras are running. IMU sensors cannot work alone.

To start the cameras, use k4a_device_start_cameras().

```
k4a_device_configuration_t config = K4A_DEVICE_CONFIG_INIT_DISABLE_ALL;
config.camera_fps = K4A_FRAMES_PER_SECOND_30;
config.color_format = K4A_IMAGE_FORMAT_COLOR_MJPG;
config.color_resolution = K4A_COLOR_RESOLUTION_2160P;

if (K4A_RESULT_SUCCEEDED != k4a_device_start_cameras(device, &config))
{
    printf("Failed to start cameras\n");
    goto Exit;
}

if (K4A_RESULT_SUCCEEDED != k4a_device_start_imu(device))
{
    printf("Failed to start imu\n");
    goto Exit;
}
```

Access IMU samples

Each k4a_imu_sample_t contains an accelerometer and gyroscope reading captured at nearly the same time.

You can get the IMU samples either on the same thread you get image captures, or on separate threads.

To retrieve IMU samples as soon as they're available, you may want to call `k4a_device_get_imu_sample()` on its own thread. The API also has sufficient internal queuing to allow you to only check for samples after each image capture is returned.

Because there's some internal queueing of IMU samples, you can use the following pattern without dropping any data:

1. Wait on a capture, at any frames rate.
2. Process the capture.

3. Retrieve all the queued IMU samples.
4. Repeat waiting on the next capture.

To retrieve all the currently queued IMU samples, you can call `k4a_device_get_imu_sample()` with a `timeout_in_ms` of 0 in a loop until the function returns `K4A_WAIT_RESULT_TIMEOUT`. `K4A_WAIT_RESULT_TIMEOUT` indicates that there are no queued samples and none have arrived in the timeout specified.

Usage example

```
k4a_imu_sample_t imu_sample;

// Capture a imu sample
switch (k4a_device_get_imu_sample(device, &imu_sample, TIMEOUT_IN_MS))
{
case K4A_WAIT_RESULT_SUCCEEDED:
    break;
case K4A_WAIT_RESULT_TIMEOUT:
    printf("Timed out waiting for a imu sample\n");
    continue;
    break;
case K4A_WAIT_RESULT_FAILED:
    printf("Failed to read a imu sample\n");
    goto Exit;
}

// Access the accelerometer readings
if (imu_sample != NULL)
{
    printf(" | Accelerometer temperature:%.2f x:%.4f y:%.4f z: %.4f\n",
            imu_sample.temperature,
            imu_sample.acc_sample.xyz.x,
            imu_sample.acc_sample.xyz.y,
            imu_sample.acc_sample.xyz.z);
}
```

Next steps

Now you know how to work with IMU samples, you also can

Access microphone input data

Access Azure Kinect DK microphone input data

11/15/2019 • 2 minutes to read • Edit Online

The Speech SDK quickstarts provide examples of how to use the Azure Kinect DK microphone array in various programming languages. For example, see the **Recognize speech in C++ on Windows by using the Speech SDK** quickstart. The code is available from GitHub.

Access the microphone array also through Windows API. See the following docs for details on Windows documentation:

- Windows Audio architecture
- Windows.Media.Capture documentation
- Tutorial for webcam capture
- USB audio information

You can also review the microphone array hardware specification.

Next steps

Speech Services SDK

Use Azure Kinect Sensor SDK image transformations

11/12/2019 • 6 minutes to read • Edit Online

Follow the specific functions to use and transform images between coordinated camera systems in your Azure Kinect DK.

k4a_transformation functions

All functions prefixed with *k4a_transformation* operate on whole images. They require the transformation handle k4a_transformation_t obtained via k4a_transformation_create() and are unallocated via k4a_transformation_destroy(). You can also refer to the SDK Transformation Example that demonstrates how to use the three functions in this topic.

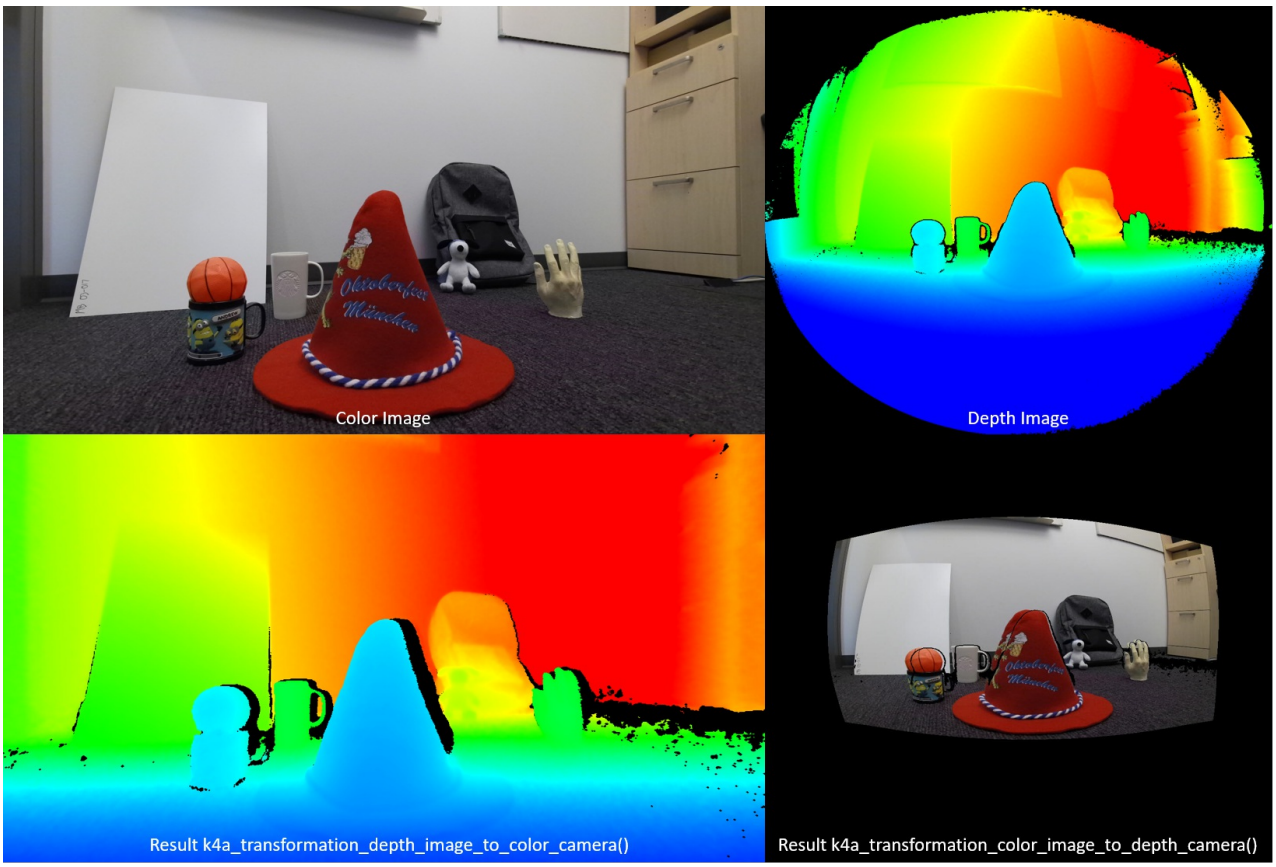
The following functions are covered:

- `k4a_transformation_depth_image_to_color_camera()`
- `k4a_transformation_depth_image_to_color_camera_custom()`
- `k4a_transformation_color_image_to_depth_camera()`
- `k4a_transformation_depth_image_to_point_cloud()`

k4a_transformation_depth_image_to_color_camera

Overview

The function k4a_transformation_depth_image_to_color_camera() transforms the depth map from the viewpoint of the depth camera into the viewpoint of the color camera. This function is designed to produce so-called RGB-D images, where D represents an additional image channel recording the depth value. As seen in the figure below, the color image and the output of k4a_transformation_depth_image_to_color_camera() look as if they were taken from the same viewpoint, that is, the viewpoint of the color camera.



Implementation

This transformation function is more complex than simply calling k4a_calibration_2d_to_2d() for every pixel. It warps a triangle mesh from the geometry of the depth camera into the geometry of the color camera. The triangle mesh is used to avoid generating holes in the transformed depth image. A Z-buffer ensures that occlusions are handled correctly. GPU acceleration is enabled for this function by default.

Parameters

Input parameters are the transformation handle and a depth image. The depth image resolution must match the `depth_mode` specified at creation of the transformation handle. For example, if the transformation handle was created using the 1024x1024 `K4A_DEPTH_MODE_WFOV_UNBINNED` mode, the resolution of the depth image must be 1024x1024 pixels. The output is a transformed depth image that needs to be allocated by the user via calling k4a_image_create(). The resolution of the transformed depth image must match the `color_resolution` specified at creation of the transformation handle. For example, if the color resolution was set to `K4A_COLOR_RESOLUTION_1080P`, the output image resolution must be 1920x1080 pixels. The output image stride is set to `width * sizeof(uint16_t)`, as the image stores 16-bit depth values.

k4a_transformation_depth_image_to_color_camera_custom

Overview

The function k4a_transformation_depth_image_to_color_camera_custom() transforms the depth map and a custom image from the viewpoint of the depth camera into the viewpoint of the color camera. As an extension of k4a_transformation_depth_image_to_color_camera(), this function is designed to produce a corresponding custom image for which each pixel matches the corresponding pixel coordinates of the color camera additional to the transformed depth image.

Implementation

This transformation function produces the transformed depth image the same way as k4a_transformation_depth_image_to_color_camera(). To transform the custom image, this function provides options of using linear interpolation or nearest neighbor interpolation. Using linear interpolation could create new values in the transformed custom image. Using nearest neighbor interpolation will prevent values not present in the original image from appearing in the output image but will result in less smooth image. The custom image should be single channel 8-bit or 16-bit. GPU acceleration is enabled for this function by default.

Parameters

Input parameters are the transformation handle, a depth image, a custom image and the interpolation type. The depth image and custom image resolution must match the `depth_mode` specified at creation of the transformation handle. For example, if the transformation handle was created using the 1024x1024 `K4A_DEPTH_MODE_WFOV_UNBINNED` mode, the resolution of the depth image and custom image must be 1024x1024 pixels. The `interpolation_type` should be either `K4A_TRANSFORMATION_INTERPOLATION_TYPE_LINEAR` or `K4A_TRANSFORMATION_INTERPOLATION_TYPE_NEAREST`. The output is a transformed depth image and a transformed custom image that need to be allocated by the user via calling k4a_image_create(). The resolution of the transformed depth image and transformed custom image must match the `color_resolution` specified at creation of the transformation handle. For example, if the color resolution was set to `K4A_COLOR_RESOLUTION_1080P`, the output image resolution must be 1920x1080 pixels. The output depth image stride is set to `width * sizeof(uint16_t)`, as the image stores 16-bit depth values. The input custom image and transformed custom image must be of format `K4A_IMAGE_FORMAT_CUSTOM8` or `K4A_IMAGE_FORMAT_CUSTOM16`, corresponding image stride should be set accordingly.

k4a_transformation_color_image_to_depth_camera

Overview

The function k4a_transformation_color_image_to_depth_camera() transforms the color image from the viewpoint of the color camera into the viewpoint of the depth camera (see figure above). It can be used to generate RGB-D images.

Implementation

For every pixel of the depth map, the function uses the pixel's depth value to compute the corresponding subpixel

coordinate in the color image. We then look up the color value at this coordinate in the color image. Bilinear interpolation is performed in the color image to obtain the color value at subpixel precision. A pixel that does not have an associated depth reading is assigned to a BGRA value of `[0,0,0,0]` in the output image. GPU acceleration is enabled for this function by default. As this method produces holes in the transformed color image and does not handle occlusions, we recommend using the function k4a_transformation_depth_image_to_color_camera() instead.

Parameters

The input parameters are the transformation handle, a depth image, and a color image. The resolutions of depth and color images must match the depth_mode and color_resolution specified at creation of the transformation handle. The output is a transformed color image that needs to be allocated by the user via calling k4a_image_create(). The resolution of the transformed color image must match the depth_resolution specified at creation of the transformation handle. The output image stores four 8-bit values representing BGRA for every pixel. Therefore, the stride of the image is `width * 4 * sizeof(uint8_t)`. The data order is pixel interleaved, that is, blue value - pixel 0, green value - pixel 0, red value - pixel 0, alpha value - pixel 0, blue value - pixel 1 and so on.

k4a_transformation_depth_image_to_point_cloud

Overview

The function k4a_transformation_depth_image_to_point_cloud() converts a 2D depth map taken by a camera into a 3D point cloud in the coordinate system of the same camera. The camera can thereby be the depth or color camera.

Implementation

The function gives equivalent results to running k4a_calibration_2d_to_2d() for every pixel, but is computationally more efficient. When calling k4a_transformation_create(), we precompute a so-called xy-lookup table that stores x- and y-scale factors for every image pixel. When calling k4a_transformation_depth_image_to_point_cloud(), we obtain a pixel's 3D X-coordinate by multiplying the pixel's x-scale factor with the pixel's Z-coordinate. Analogously, the 3D Y-coordinate is computed by multiplication with the y-scale factor. The fast point cloud example of the SDK demonstrates how the xy-table is computed. Users can follow the example code to implement their own version of this function, for example, to speed up their GPU pipeline.

Parameters

The input parameters are the transformation handle, a camera specifier, and a depth image. If the camera specifier is set to depth, the resolution of the depth image must match the depth_mode specified at creation of the transformation handle. Otherwise, if the specifier is set to the color camera, the resolution must match the resolution of the chosen color_resolution. The output parameter is an XYZ image that needs to be allocated by the user via calling k4a_image_create(). The XYZ image resolution must match the resolution of the input depth map. We store three signed 16-bit coordinate values in millimeter for every pixel. The XYZ image stride is therefore set to `width * 3 * sizeof(int16_t)`. The data order is pixel interleaved, that is, X-coordinate – pixel 0, Y-coordinate – pixel 0, Z-coordinate – pixel 0, X-coordinate – pixel 1 and so on. If a pixel cannot be converted to 3D, the function assigns the values `[0,0,0]` to the pixel.

Samples

Transformation example

Next steps

Now you know how to use Azure Kinect sensor SDK image transformation functions, you also can learn about

Azure Kinect sensor SDK calibration functions

Also you can review

Coordinate systems

Use Azure Kinect calibration functions

11/12/2019 • 3 minutes to read • Edit Online

The calibration functions allow for transforming points between the coordinate systems of each sensor on the Azure Kinect device. Applications requiring conversion of whole images may take advantage of the accelerated operations available in transformation functions.

Retrieve calibration data

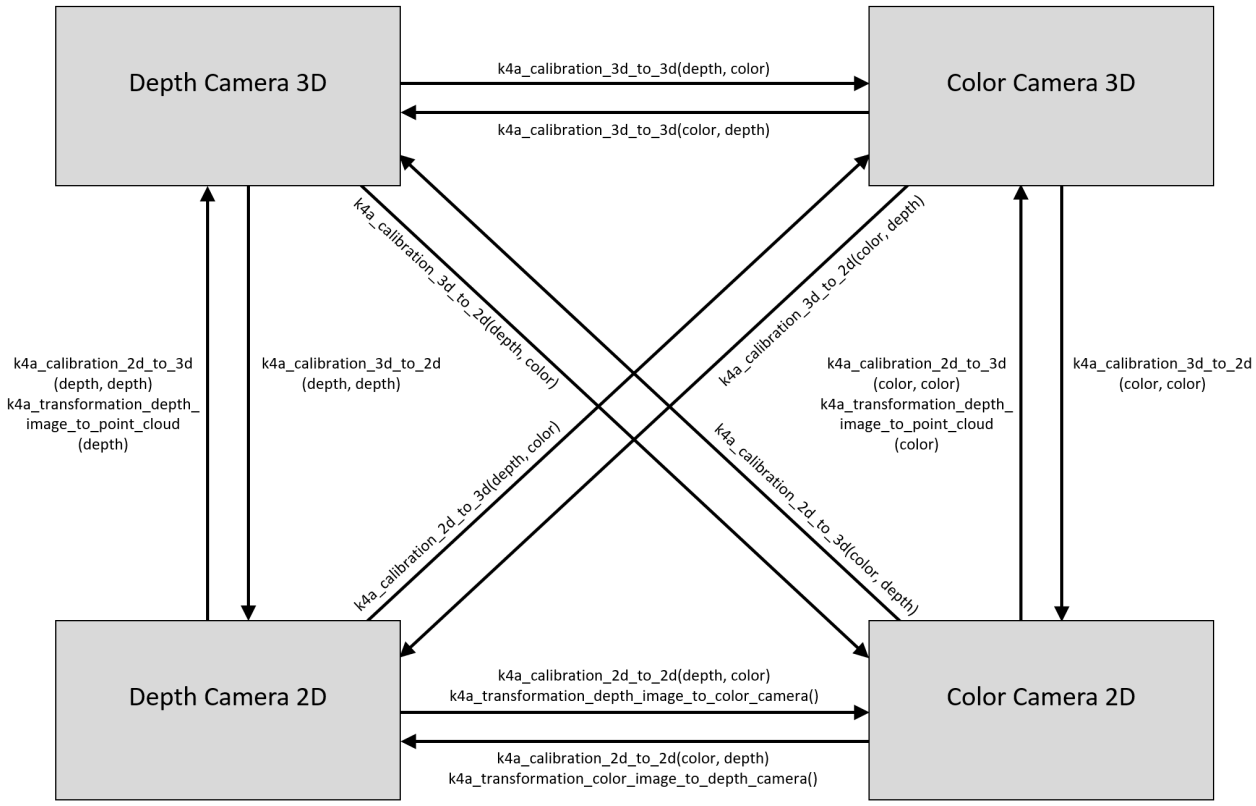
It is necessary to retrieve the device calibration to perform coordinate system transformations. The calibration data is stored in the k4a_calibration_t data type. It is obtained from the device via the function k4a_device_get_calibration(). The calibration data is not only specific to each device, but also to the operating mode of the cameras. Hence k4a_device_get_calibration() requires the `depth_mode` and `color_resolution` parameters as input.

OpenCV Compatibility

The calibration parameters are compatible with OpenCV. For more information about the individual camera calibration parameters, see also OpenCV documentation. Also see OpenCV compatibility example of the SDK that demonstrates conversion between the k4a_calibration_t type and the corresponding OpenCV data structures.

Coordinate Transformation Functions

The figure below shows the different coordinate systems of Azure Kinect as well as the functions to convert between them. We omit the 3D coordinate systems of gyroscope and accelerometer to keep the figure simple.



Remark on lens distortion: 2D coordinates always refer to the distorted image in the SDK. The undistortion example of the SDK demonstrates image undistortion. In general, 3D points will never be affected by lens distortion.

Convert between 3D coordinate systems

The function k4a_calibration_3d_to_3d() converts a 3D point of the source coordinate system to a 3D point of the target coordinate system using the camera's extrinsic calibration. Source and target can be set to any of the four 3D coordinate systems, that is, color camera, depth camera, gyroscope, or accelerometer. If source and target are identical, the unmodified input 3D point is returned as output.

Convert between 2D and 3D coordinate systems

The function k4a_calibration_3d_to_2d() converts a 3D point of the source coordinate system to a 2D pixel coordinate of the target camera. This function is often referred to as project function. While the source can be set to any of the four 3D coordinate systems, the target must be the depth or color camera. If source and target are different, the input 3D point is converted to the 3D coordinate system of the target camera using k4a_calibration_3d_to_3d(). Once the 3D point is represented in the target camera coordinate system, the corresponding 2D pixel coordinates are computed using the target camera's intrinsic calibration. If a 3D point falls out of the visible area of the target camera, the valid value is set to 0.

The function k4a_calibration_2d_to_3d() converts a 2D pixel coordinate of the source camera to a 3D point of the target camera coordinate system. The source must be color or depth camera. The target can be set to any of the four 3D coordinate systems. In addition to the 2D pixel coordinate, the pixel's depth value (in millimeters) in the source camera's image is required as an input to the function, one way to derive the depth value in the color camera geometry is to use the function k4a_transformation_depth_image_to_color_camera(). The function computes the 3D ray leading from the source camera's focal point through the specified pixel coordinate using the source camera's intrinsic calibration. The depth value is then used to find the exact location of the 3D point on this ray. This operation is often referred to as unproject function. If source and target cameras are different, the function transforms the 3D point to the coordinate system of the target via k4a_calibration_3d_to_3d(). If a 2D pixel coordinate falls out of the visible area of the source camera, the valid value is set to 0.

Converting between 2D coordinate systems

The function k4a_calibration_2d_to_2d() converts a 2D pixel coordinate of the source camera to a 2D pixel coordinate of the target camera. Source and target must be set to color or depth camera. The function requires the pixel's depth value (in millimeters) in the source camera image as an input, one way to derive the depth value in the color camera geometry is to use the function k4a_transformation_depth_image_to_color_camera(). It calls k4a_calibration_2d_to_3d() to convert to a 3D point of the source camera system. It then calls k4a_calibration_3d_to_2d() to convert to a 2D pixel coordinate of the target camera image. The valid value is set to 0, if k4a_calibration_2d_to_3d() or k4a_calibration_3d_to_2d() returns an invalid result.

Related samples

- OpenCV compatibility example
- Undistortion example
- Fast point cloud example

Next steps

Now you know about camera calibrations, you may also learn how to

Capture device synchronization

Also you can review

Coordinate systems

Capture Azure Kinect device synchronization

11/12/2019 • 2 minutes to read • Edit Online

The Azure Kinect hardware can align the capture time of color and depth images. Alignment between the cameras on the same device is **internal synchronization**. Capture time alignment across multiple connected devices is **external synchronization**.

Device internal synchronization

Image capture between the individual cameras is synchronized in hardware. In every k4a_capture_t that contains images from both the color and depth sensor, the images' timestamps are aligned based on the operating mode of the hardware. By default the images of a capture are center of exposure aligned. The relative timing of depth and color captures can be adjusted using the `depth_delay_off_color_usec` field of k4a_device_configuration_t.

Device external synchronization

See setup external synchronization for hardware setup.

The software for each connected device must be configured to operate in a **master** or **subordinate** mode. This setting is configured on the k4a_device_configuration_t.

When using external synchronization, subordinate cameras should always be started before the master for the timestamps to align correctly.

Subordinate mode

```
k4a_device_configuration_t deviceConfig;
deviceConfig.wired_sync_mode = K4A_WIRED_SYNC_MODE_SUBORDINATE
```

Master mode

```
k4a_device_configuration_t deviceConfig;
deviceConfig.wired_sync_mode = K4A_WIRED_SYNC_MODE_MASTER;
```

Retrieving synchronization jack state

To programmatically retrieve the current state of the synchronization input and synchronization output jacks, use the k4a_device_get_sync_jack function.

Next steps

Now you know how to enable and capture device synchronization. You also can review how to use

Azure Kinect sensor SDK record and playback API

The Azure Kinect playback API

11/12/2019 • 3 minutes to read • Edit Online

The sensor SDK provides an API for recording device data to a Matroska (.mkv) file. The Matroska container format stores video tracks, IMU samples, and device calibration. Recordings can be generated using the provided k4arecorder command-line utility. Recordings can also be customized and recorded directly using the record API.

For more information about the recording API, see `k4a_record_create()`.

For more information on the Matroska file format specifications, see the Recording File Format page.

Use the playback API

Recording files can be opened using the playback API. The playback API provides access to sensor data in the same format as the rest of the sensor SDK.

Open a record file

In the following example, we open a recording using `k4a_playback_open()`, print the recording length, and then close the file with `k4a_playback_close()`.

```
k4a_playback_t playback_handle = NULL;
if (k4a_playback_open("recording.mkv", &playback_handle) != K4A_RESULT_SUCCEEDED)
{
    printf("Failed to open recording\n");
    return 1;
}

uint64_t recording_length = k4a_playback_get_last_timestamp_usec(playback_handle);
printf("Recording is %lld seconds long\n", recording_length / 1000000);

k4a_playback_close(playback_handle);
```

Read captures

Once the file is open, we can start reading captures from the recording. This next example will read each of the captures in the file.

```
k4a_capture_t capture = NULL;
k4a_stream_result_t result = K4A_STREAM_RESULT_SUCCEEDED;
while (result == K4A_STREAM_RESULT_SUCCEEDED)
{
    result = k4a_playback_get_next_capture(playback_handle, &capture);
    if (result == K4A_STREAM_RESULT_SUCCEEDED)
    {
        // Process capture here
        k4a_capture_release(capture);
    }
    else if (result == K4A_STREAM_RESULT_EOF)
    {
        // End of file reached
        break;
    }
}
if (result == K4A_STREAM_RESULT_FAILED)
{
    printf("Failed to read entire recording\n");
    return 1;
}
```

Seek within a recording

Once we've reached the end of the file, we may want to go back and read it again. This process could be done by reading backwards with `k4a_playback_get_previous_capture()`, but it could be very slow depending on the length of the recording. Instead we can use the `k4a_playback_seek_timestamp()` function to go to a specific point in the file.

In this example, we specify timestamps in microseconds to seek to various points in the file.

```
// Seek to the beginning of the file
if (k4a_playback_seek_timestamp(playback_handle, 0, K4A_PLAYBACK_SEEK_BEGIN) != K4A_RESULT_SUCCEEDED)
{
    return 1;
}

// Seek to the end of the file
if (k4a_playback_seek_timestamp(playback_handle, 0, K4A_PLAYBACK_SEEK_END) != K4A_RESULT_SUCCEEDED)
{
    return 1;
}

// Seek to 10 seconds from the start
if (k4a_playback_seek_timestamp(playback_handle, 10 * 1000000, K4A_PLAYBACK_SEEK_BEGIN) !=
K4A_RESULT_SUCCEEDED)
{
    return 1;
}

// Seek to 10 seconds from the end
if (k4a_playback_seek_timestamp(playback_handle, -10 * 1000000, K4A_PLAYBACK_SEEK_END) != K4A_RESULT_SUCCEEDED)
{
    return 1;
}
```

Read tag information

Recordings can also contain various metadata such as the device serial number and firmware versions. This metadata is stored in recording tags, which can be accessed using the `k4a_playback_get_tag()` function.

```
// Print the serial number of the device used to record
char serial_number[256];
size_t serial_number_size = 256;
k4a_buffer_result_t buffer_result = k4a_playback_get_tag(playback_handle, "K4A_DEVICE_SERIAL_NUMBER",
&serial_number, &serial_number_size);
if (buffer_result == K4A_BUFFER_RESULT_SUCCEEDED)
{
    printf("Device serial number: %s\n", serial_number);
}
else if (buffer_result == K4A_BUFFER_RESULT_TOO_SMALL)
{
    printf("Device serial number too long.\n");
}
else
{
    printf("Tag does not exist. Device serial number was not recorded.\n");
}
```

Record tag list

Below is a list of all the default tags that may be included in a recording file. Many of these values are available as part of the `k4a_record_configuration_t` struct, and can be read with the `k4a_playback_get_record_configuration()` function.

If a tag doesn't exist, it's assumed to have the default value.

TAG NAME	DEFAULT VALUE	`K4A_RECORD_CONFIGURATION_T` FIELD	NOTES
`K4A_COLOR_MODE`	"OFF"	`color_format` / `color_resolution`	Possible values: "OFF", "MJPG_1080P", "NV12_720P", "YUY2_720P", and so on
`K4A_DEPTH_MODE`	"OFF"	`depth_mode` / `depth_track_enabled`	Possible values: "OFF, "NFOV_UNBINNED", "PASSIVE_IR", and so on
`K4A_IR_MODE`	"OFF"	`depth_mode` / `ir_track_enabled`	Possible values: "OFF", "ACTIVE", "PASSIVE"
`K4A_IMU_MODE`	"OFF"	`imu_track_enabled`	Possible values: "ON", "OFF"
`K4A_CALIBRATION_FILE`	"calibration.json"	N/A	See `k4a_device_get_raw_calibration()`
`K4A_DEPTH_DELAY_NS`	"0"	`depth_delay_off_color_usec`	Value stored in nanoseconds, API provides microseconds.
`K4A_WIRED_SYNC_MODE`	"STANDALONE"	`wired_sync_mode`	Possible values: "STANDALONE", "MASTER", "SUBORDINATE"
`K4A_SUBORDINATE_DELAY_NS`	"0"	`subordinate_delay_off_master_usec`	Value stored in nanoseconds, API provides microseconds.
`K4A_COLOR_FIRMWARE_VERSION`	""	N/A	Device color firmware version, for example "1.x.xx"

TAG NAME	DEFAULT VALUE	K4A_RECORD_CONFIGURATION_T FIELD	NOTES
K4A_DEPTH_FIRMWARE_VERSION	""	N/A	Device depth firmware version, for example "1.x.xx"
K4A_DEVICE_SERIAL_NUMBER	""	N/A	Recording device serial number
K4A_START_OFFSET_NS	"0"	start_timestamp_offset_usec	See Timestamp Synchronization below.
K4A_COLOR_TRACK	None	N/A	See Recording File Format - Identifying tracks.
K4A_DEPTH_TRACK	None	N/A	See Recording File Format - Identifying tracks.
K4A_IR_TRACK	None	N/A	See Recording File Format - Identifying tracks.
K4A_IMU_TRACK	None	N/A	See Recording File Format - Identifying tracks.

Timestamp synchronization

The Matroska format requires that recordings must start with a timestamp of zero. When externally syncing cameras, the first timestamp from of each device can be non-zero.

To preserve the original timestamps from the devices between recording and playback, the file stores an offset to apply to the timestamps.

The `K4A_START_OFFSET_NS` tag is used to specify a timestamp offset so that files can be resynchronized after recording. This timestamp offset can be added to each timestamp in the file to reconstruct the original device timestamps.

The start offset is also available in the `k4a_record_configuration_t` struct.

Get body tracking results

1/24/2020 • 2 minutes to read • Edit Online

Body Tracking SDK uses a body tracker object to process Azure Kinect DK captures and generates body tracking results. It also maintains global status of the tracker, processing queues and the output queue. There are three steps in using the body tracker:

- Create a tracker
- Capture depth and IR images using Azure Kinect DK
- Enqueue the capture and pop the results.

Create a tracker

The first step in using body tracking is to create a tracker and requires passing in the sensor calibration k4a_calibration_t structure. The sensor calibration can be queried using the Azure Kinect Sensor SDK k4a_device_get_calibration() function.

```
k4a_calibration_t sensor_calibration;
if (K4A_RESULT_SUCCEEDED != k4a_device_get_calibration(device, device_config.depth_mode,
K4A_COLOR_RESOLUTION_OFF, &sensor_calibration))
{
    printf("Get depth camera calibration failed!\n");
    return 0;
}

k4abt_tracker_t tracker = NULL;
k4abt_tracker_configuration_t tracker_config = K4ABT_TRACKER_CONFIG_DEFAULT;
if (K4A_RESULT_SUCCEEDED != k4abt_tracker_create(&sensor_calibration, tracker_config, &tracker))
{
    printf("Body tracker initialization failed!\n");
    return 0;
}
```

Capture depth and IR images

Image capture using Azure Kinect DK is covered in the retrieve images page.

> **NOTE**
>
> `K4A_DEPTH_MODE_NFOV_UNBINNED` or `K4A_DEPTH_MODE_WFOV_2X2BINNED` modes are recommended for best performance and accuracy. Do not use the `K4A_DEPTH_MODE_OFF` or `K4A_DEPTH_MODE_PASSIVE_IR` modes.

The supported Azure Kinect DK modes are described in the Azure Kinect DK hardware specification and k4a_depth_mode_t enumerations.

```
// Capture a depth frame
switch (k4a_device_get_capture(device, &capture, TIMEOUT_IN_MS))
{
case K4A_WAIT_RESULT_SUCCEEDED:
    break;
case K4A_WAIT_RESULT_TIMEOUT:
    printf("Timed out waiting for a capture\n");
    continue;
    break;
case K4A_WAIT_RESULT_FAILED:
    printf("Failed to read a capture\n");
    goto Exit;
}
```

Enqueue the capture and pop the results

The tracker internally maintains an input queue and an output queue to asynchronously process the Azure Kinect DK captures more efficiently. Use the k4abt_tracker_enqueue_capture() function to add a new capture to the input queue. Use the k4abt_tracker_pop_result() function o pop a result from the output queue. Use of the timeout value is dependent on the application and controls the queuing wait time.

Real-time processing

Use this pattern for single-threaded applications that need real-time results and can accommodate dropped frames. The `simple_3d_viewer` sample located in GitHub Azure-Kinect-Samples is an example of real-time processing.

```
k4a_wait_result_t queue_capture_result = k4abt_tracker_enqueue_capture(tracker, sensor_capture, 0);
k4a_capture_release(sensor_capture); // Remember to release the sensor capture once you finish using it
if (queue_capture_result == K4A_WAIT_RESULT_FAILED)
{
    printf("Error! Adding capture to tracker process queue failed!\n");
    break;
}

k4abt_frame_t body_frame = NULL;
k4a_wait_result_t pop_frame_result = k4abt_tracker_pop_result(tracker, &body_frame, 0);
if (pop_frame_result == K4A_WAIT_RESULT_SUCCEEDED)
{
    // Successfully popped the body tracking result. Start your processing
    ...

    k4abt_frame_release(body_frame); // Remember to release the body frame once you finish using it
}
```

Synchronous processing

Use this pattern for applications that do not need real-time results or cannot accommodate dropped frames.

Processing throughput may be limited.

The `simple_sample.exe` sample located in GitHub Azure-Kinect-Samples is an example of synchronous processing.

```
k4a_wait_result_t queue_capture_result = k4abt_tracker_enqueue_capture(tracker, sensor_capture,
K4A_WAIT_INFINITE);
k4a_capture_release(sensor_capture); // Remember to release the sensor capture once you finish using it
if (queue_capture_result != K4A_WAIT_RESULT_SUCCEEDED)
{
    // It should never hit timeout or error when K4A_WAIT_INFINITE is set.
    printf("Error! Adding capture to tracker process queue failed!\n");
    break;
}

k4abt_frame_t body_frame = NULL;
k4a_wait_result_t pop_frame_result = k4abt_tracker_pop_result(tracker, &body_frame, K4A_WAIT_INFINITE);
if (pop_frame_result != K4A_WAIT_RESULT_SUCCEEDED)
{
    // It should never hit timeout or error when K4A_WAIT_INFINITE is set.
    printf("Error! Popping body tracking result failed!\n");
    break;
}
// Successfully popped the body tracking result. Start your processing
...

k4abt_frame_release(body_frame); // Remember to release the body frame once you finish using it
```

Next steps

Access data in body frame

Access data in body frame

1/24/2020 • 2 minutes to read • Edit Online

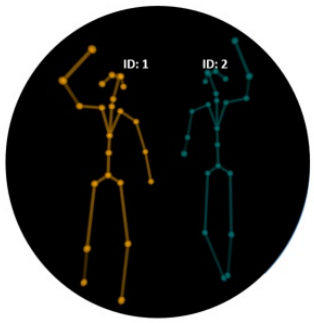
This article describes the data contained in a body frame and the functions to access that data.

The following functions are covered:

- k4abt_frame_get_body_id()
- k4abt_frame_get_body_index_map()
- k4abt_frame_get_body_skeleton()
- k4abt_frame_get_capture()
- k4abt_frame_get_num_bodies()
- k4abt_frame_get_device_timestamp_usec()

Key components of a body frame

Each body frame contains a collection of body structs, a 2D body index map, and the input capture that generated this result.



A collection of body structs

Each body struct contains:
- Body ID – the tracked ID of the body.
- Joint Positions – the 3D position of each joint.
- Joint Orientations – the orientation of each joint coordinate frame represented in quaternion.

2D body index map

The 2D instance segmentation map segments each body instance from the background.

Input capture

The input capture that used to generate the body tracking results.

Access the collection of body structs

Multiple bodies might be detected in a single capture. You can query the number of bodies by calling the k4abt_frame_get_num_bodies() function.

```
size_t num_bodies = k4abt_frame_get_num_bodies(body_frame);
```

You use the k4abt_frame_get_body_id() and k4abt_frame_get_body_skeleton() functions to iterate through each body index to find the body ID and joint position/orientation information.

```
for (size_t i = 0; i < num_bodies; i++)
{
    k4abt_skeleton_t skeleton;
    k4abt_frame_get_body_skeleton(body_frame, i, &skeleton);
    uint32_t id = k4abt_frame_get_body_id(body_frame, i);
}
```

Access the body index map

You use the k4abt_frame_get_body_index_map() function to access the body index map. Refer to body index map for detailed explanation of the body index map. Make sure to release the body index map when it is no longer needed.

```
k4a_image_t body_index_map = k4abt_frame_get_body_index_map(body_frame);
...  // Do your work with the body index map
k4a_image_release(body_index_map);
```

Access the input capture

The body tracker is an asynchronous API. The original capture may already have been released by the time the result is popped. Use the k4abt_frame_get_capture() function to query the input capture used to generate this body tracking result. The reference count for the k4a_capture_t is increased each time this function is called. Use k4a_capture_release() function when the capture is no longer needed.

```
k4a_capture_t input_capture = k4abt_frame_get_capture(body_frame);
... // Do your work with the input capture
k4a_capture_release(input_capture);
```

Next steps

Azure Kinect Body Tracking SDK

Add Azure Kinect library to your Visual Studio project

11/12/2019 • 2 minutes to read • Edit Online

This article walks you through the process of adding Azure Kinect NuGet package to your Visual Studio Project.

Install Azure Kinect NuGet package

To install the Azure Kinect NuGet package:

1. You can find detailed instructions for installing a NuGet package in Visual Studio in the Quickstart: Install and use a package in Visual Studio.
2. To add the package, you can use Package Manager UI by right-clicking References and choosing Manage NuGet Packages from Solution Explorer.
3. Choose nuget.org as the Package source, select Browse tab, and search for `Microsoft.Azure.Kinect.Sensor`.
4. Select that package from the list and install.

Use Azure Kinect NuGet package

Once the package is added, add header file includes to the source code, such as:

- `#include <k4a/k4a.h>`
- `#include <k4arecord/record.h>`
- `#include <k4arecord/playback.h>`

Next steps

Now you are ready to build your first application

Update Azure Kinect DK firmware

1/8/2020 • 2 minutes to read • Edit Online

This document provides guidance on how to update device firmware on your Azure Kinect DK.

Azure Kinect DK doesn't update firmware automatically. You can use Azure Kinect Firmware Tool to update firmware manually to the latest available version.

Prepare for firmware update

1. Download SDK.
2. Install the SDK.
3. In the SDK install location under (SDK install location)\tools\ you should find:
   - AzureKinectFirmwareTool.exe
   - A Firmware .bin file in the firmware folder, such as *AzureKinectDK_Fw_1.5.926614.bin*.
4. Connect your device to host PC and power it up also.

> **IMPORTANT**
> Keep the USB and power supply connected during the firmware update. Removing either connection during update may put the firmware into a corrupted state.

Update device firmware

1. Open a command prompt in the (SDK install location)\tools\ folder.
2. Update Firmware using the Azure Kinect Firmware Tool

   ```
   AzureKinectFirmwareTool.exe -u <device_firmware_file.bin>
   ```

   Example:

   ```
   AzureKinectFirmwareTool.exe -u firmware\AzureKinectDK_Fw_1.5.926614.bin
   ```

3. Wait until the firmware update finishes. It can take a few minutes depending on the image size.

Verify device firmware version

1. Verify the firmware is updated.

   ```
   AzureKinectFirmwareTool.exe -q
   ```

2. View the following example.

   ```
   >AzureKinectFirmwareTool.exe -q
   ```

== Azure Kinect DK Firmware Tool == Device Serial Number: 000805192412 Current Firmware Versions: RGB camera firmware: 1.6.102 Depth camera firmware: 1.6.75 Depth config file: 6109.7 Audio firmware: 1.6.14 Build Config: Production Certificate Type: Microsoft ```

3. If you see the above output, your firmware is updated.

4. After firmware update, you can run Azure Kinect viewer to verify all sensors are working as expected.

Troubleshooting

Firmware updates can fail for several reasons. When a firmware update fails, try the following mitigation steps:

1. Try to run the firmware update command a second time.
2. Confirm the device is still connected by querying for the firmware version. AzureKinectFirmareTool.exe
3. If all else fails, follow the recovery steps to revert to the factory firmware and try again.

For any additional issues, see Microsoft support pages

Next steps

Azure Kinect Firmware Tool

Use Azure Kinect recorder with external synchronized devices

11/12/2019 • 2 minutes to read • Edit Online

This article provides guidance on how the Azure Kinect Recorder can record data external synchronization configured devices.

Prerequisites

- Set up multiple Azure Kinect DK units for external synchronization.

External synchronization constraints

- Master device can't have SYNC IN cable connected.
- Master device must stream RGB camera to enable synchronization.
- All units must use the same camera configuration (framerate and resolution).
- All units must run the same device firmware (update firmware instructions).
- All subordinate devices must be started before the master device.
- The same exposure value should be set on all devices.
- Each subordinate's *Delay off master* setting is relative to the master device.

Record when each unit has a host PC

In the example below, each device has its own dedicated host PC. It's recommended you connect devices to dedicated PCs to prevent issues with USB bandwidth and CPU/GPU usage.

Subordinate-1

1. Set up recorder for the first unit

```
k4arecorder.exe --external-sync sub -e -8 -r 5 -l 10 sub1.mkv
```

2. Device starts waiting

```
Device serial number: 000011590212
Device version: Rel; C: 1.5.78; D: 1.5.60[6109.6109]; A: 1.5.13
Device started
[subordinate mode] Waiting for signal from master
```

Subordinate-2

1. Set up recorder for the second unit

```
k4arecorder.exe --external-sync sub -e -8 -r 5 -l 10 sub2.mkv
```

2. Device starts waiting

```
Device serial number: 000011590212
Device version: Rel; C: 1.5.78; D: 1.5.60[6109.6109]; A: 1.5.13
Device started
[subordinate mode] Waiting for signal from master
```

Master

1. Start recording on master

```
>k4arecorder.exe --external-sync master -e -8 -r 5 -l 10 master.mkv
```

2. Wait until recording finished

Recording when multiple units connected to single host PC

You can have multiple Azure Kinect DKs connected to a single host PC. However, that can be very demanding to USB bandwidth and host compute. To reduce the demand:

- Connect each device into own USB host controller.
- Have a powerful GPU that can handle depth engine for each device.
- Record only needed sensors and use lower framerate.

Always start subordinate devices first and the master last.

Subordinate-1

1. Start recorder on subordinate

```
>k4arecorder.exe --device 1 --external-sync subordinate --imu OFF -e -8 -r 5 -l 5 output-2.mkv
```

2. The device goes into waiting state

Master

1. Start master device

```
>k4arecorder.exe --device 0 --external-sync master --imu OFF -e -8 -r 5 -l 5 output-1.mkv
```

2. Wait recording to finish

Playing recording

You can use the Azure Kinect viewer to play back recording.

Tips

- Use manual exposure for recording synchronized cameras. RGB camera auto-exposure may impact time-synchronization.
- Restarting subordinate device will cause synchronization to be lost.
- Some camera modes support 15 fps max. We recommended that you don't mix modes/frame rates between devices
- Connecting multiple units to single PC can easily saturate USB bandwidth, consider using separate host PC per device. Pay attention to CPU/GPU compute as well.
- Disable the microphone and IMU if they aren't needed to improve reliability.

For any issues see Troubleshooting

See also

- Set up external sync
- Azure Kinect Recorder for recorder settings and additional information.
- Azure Kinect Viewer for playing recordings or setting RGB camera properties not available through recorder.

- Azure Kinect Firmware Tool for updating device firmware.

Azure Kinect Viewer

11/12/2019 • 2 minutes to read • Edit Online

The Azure Kinect Viewer, found under the installed tools directory as `k4aviewer.exe` (for example, `C:\Program Files\Azure Kinect SDK vX.Y.Z\tools\k4aviewer.exe`, where `X.Y.Z` is the installed version of the SDK), can be used to visualize all device data streams to:

- Verify sensors are working correctly.
- Help positioning the device.
- Experiment with camera settings.
- Read device configuration.
- Playback recordings made with Azure Kinect Recorder.

For more information about Azure Kinect viewer, watch How to use Azure Kinect video.

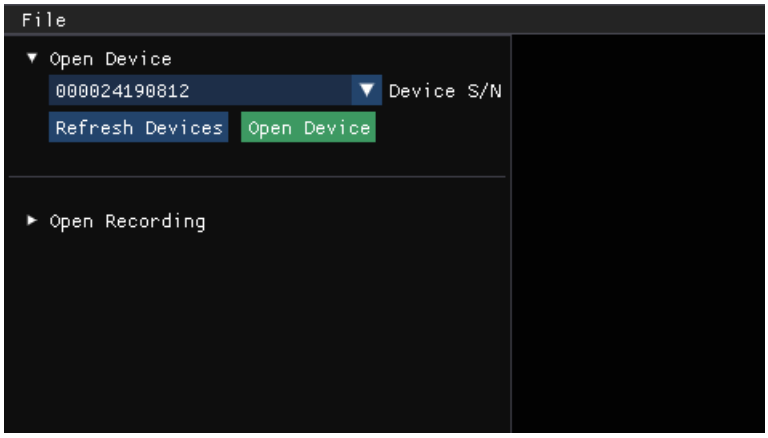
Azure Kinect Viewer is open source and can be used as an example for how to use the APIs.

Use viewer

The viewer can operate in two modes: with live data from the sensor or from recorded data (Azure Kinect Recorder).

Start application

Launch the application by running `k4aviewer.exe`.



Use the viewer with live data

1. In the **Open Device** section, select the **Serial Number** of the device to open. Then, select **Refresh**, if the device is missing.
2. Select the **Open Device** button.
3. Select **Start** to begin streaming data with the default settings.

Use the viewer with recorded data

In **Open Recording** section, navigate to the recorded file, and select it.

Check device firmware version

Access the device firmware version in the configuration window, as shown in the following image.

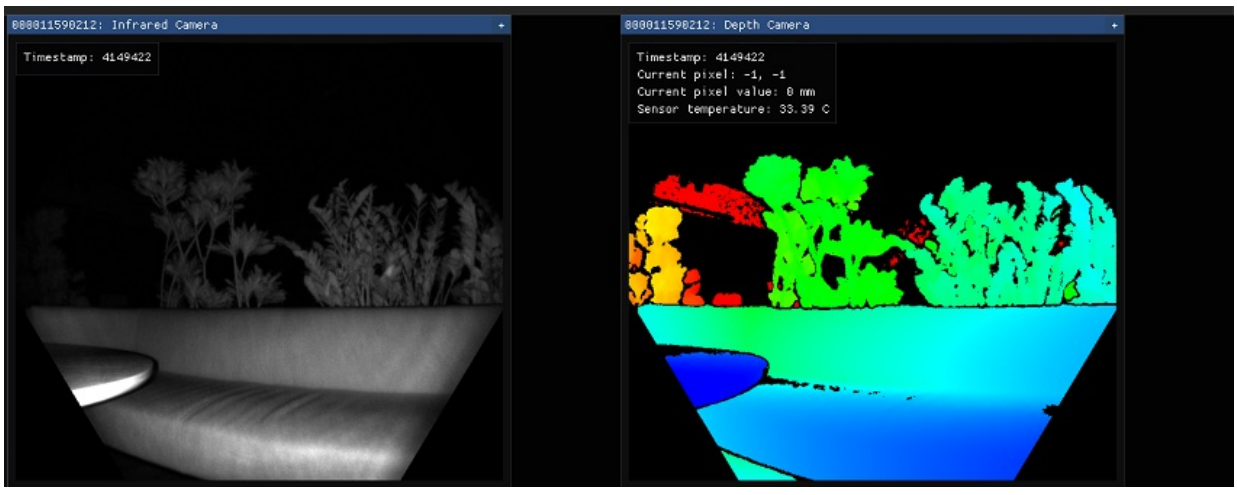
For example, in this case, the depth camera ISP is running FW 1.5.63.

Depth camera

The depth camera viewer will show two windows:

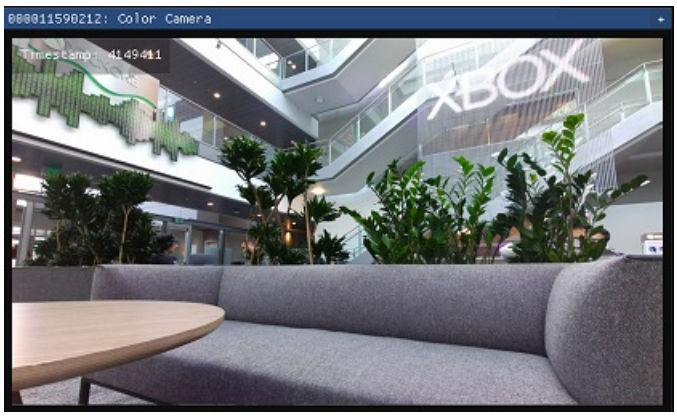
- One is called *Active Brightness* that is a grayscale image showing IR brightness.
- The second is called *Depth*, which has a colorized representation of the depth data.

Hover your cursor, at the pixel in the depth window, to see the value of the depth sensor, as shown below.

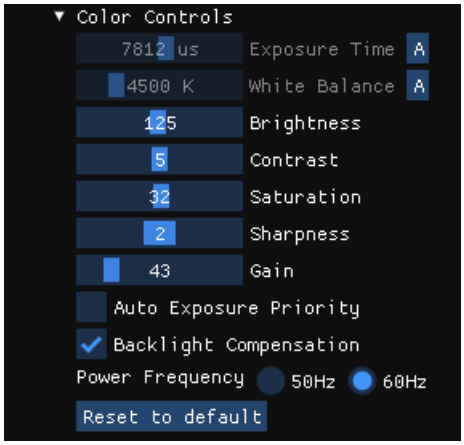


RGB camera

The image below shows the color camera view.



You can control RGB camera settings from the configuration window during the streaming.

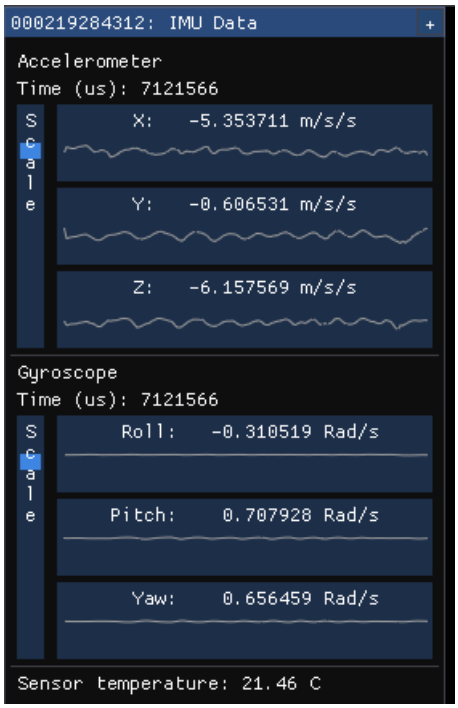


Inertial Measurement Unit (IMU)

The IMU window has two components, an accelerometer and a gyroscope.

The top half is the accelerometer and shows linear acceleration in meters/second^2. It includes acceleration from gravity, so if it's lying flat on a table, the Z axis will probably show around -9.8 m/s^2.

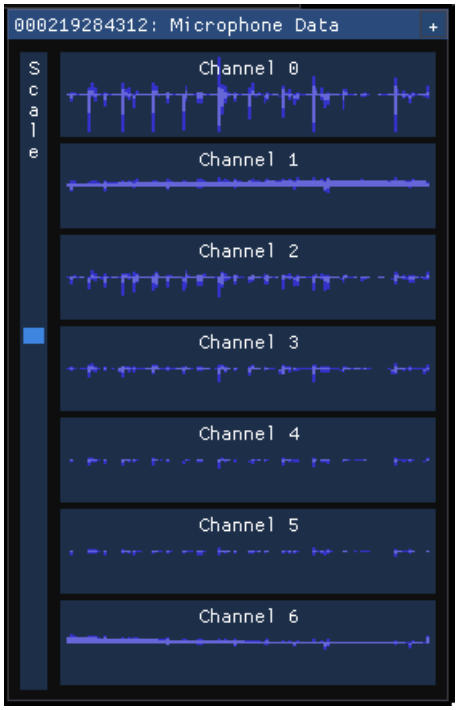
The bottom half is the gyroscope portion and shows rotational movement in radians/second



Microphone input

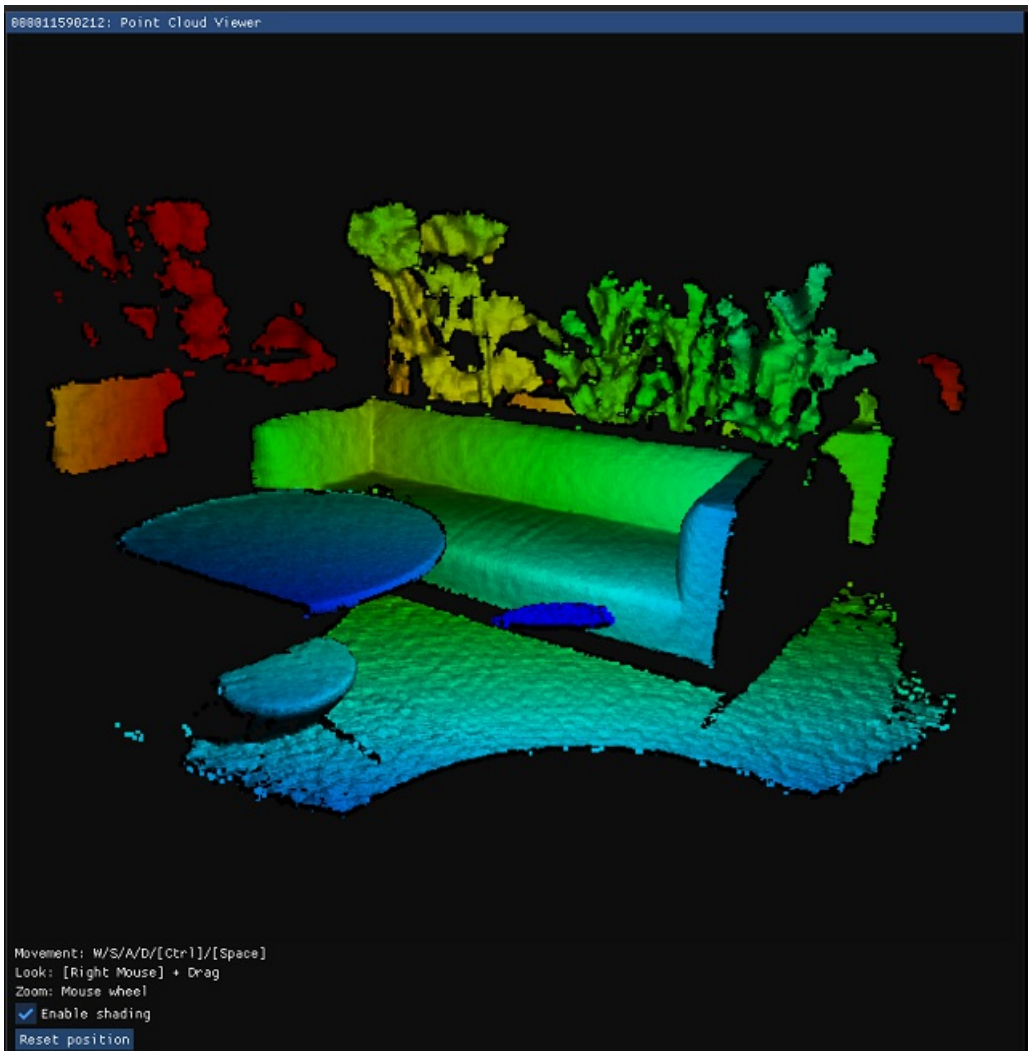
The microphone view shows a representation of the sound heard on each microphone. If there's no sound, the graph is shown as empty, otherwise, you'll see a dark blue waveform with a light blue waveform overlaid on top of it.

The dark wave represents the minimum and maximum values observed by the microphone over that time slice. The light wave represents the root mean square of the values observed by the microphone over that time slice.



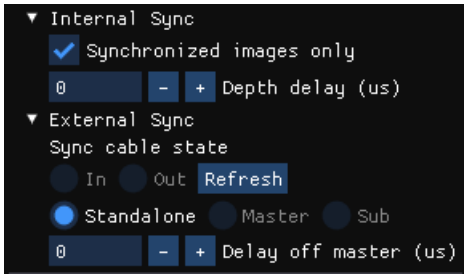
Point cloud visualization

Depth visualized in 3D lets you move in the image using instructed keys.



Synchronization control

You can use the viewer to configure the device as standalone (default), master, or subordinate mode when configuring multi-device synchronization. When changing configuration or inserting/removing synchronization cable, select **Refresh** to update.



Next steps

External synchronization setup guide

Azure Kinect DK recorder

11/12/2019 • 2 minutes to read • Edit Online

This article covers how you can use the `k4arecorder` command-line utility to record data streams from the sensor SDK to a file.

> **NOTE**
> Azure Kinect recorder doesn't record audio.

Recorder options

The `k4arecorder` has various command-line arguments to specify the output file and recording modes.

Recordings are stored in the Matroska .mkv format. The recording uses multiple video tracks for color and depth, and also additional information such as camera calibration and metadata.

```
k4arecorder [options] output.mkv

 Options:
  -h, --help              Prints this help
  --list                  List the currently connected K4A devices
  --device                Specify the device index to use (default: 0)
  -l, --record-length     Limit the recording to N seconds (default: infinite)
  -c, --color-mode        Set the color sensor mode (default: 1080p), Available options:
                            3072p, 2160p, 1536p, 1440p, 1080p, 720p, 720p_NV12, 720p_YUY2, OFF
  -d, --depth-mode        Set the depth sensor mode (default: NFOV_UNBINNED), Available options:
                            NFOV_2X2BINNED, NFOV_UNBINNED, WFOV_2X2BINNED, WFOV_UNBINNED, PASSIVE_IR, OFF
  --depth-delay           Set the time offset between color and depth frames in microseconds (default: 0)
                            A negative value means depth frames will arrive before color frames.
                            The delay must be less than 1 frame period.
  -r, --rate              Set the camera frame rate in Frames per Second
                            Default is the maximum rate supported by the camera modes.
                            Available options: 30, 15, 5
  --imu                   Set the IMU recording mode (ON, OFF, default: ON)
  --external-sync         Set the external sync mode (Master, Subordinate, Standalone default: Standalone)
  --sync-delay            Set the external sync delay off the master camera in microseconds (default: 0)
                            This setting is only valid if the camera is in Subordinate mode.
  -e, --exposure-control  Set manual exposure value (-11 to 1) for the RGB camera (default: auto exposure)
```

Record files

Example 1. Record Depth NFOV unbinned (640x576) mode, RGB 1080p at 30 fps with IMU. Press the **CTRL-C** keys to stop recording.

```
k4arecorder.exe output.mkv
```

Example 2. Record WFOV non-binned (1MP), RGB 3072p at 15 fps without IMU, for 10 seconds.

```
k4arecorder.exe -d WFOV_UNBINNED -c 3072p -r 15 -l 10 --imu OFF output.mkv
```

Example 3. Record WFOV 2x2 binned at 30 fps for 5 seconds, and save to output.mkv.

```
k4arecorder.exe -d WFOV_2X2BINNED -c OFF --imu OFF -l 5 output.mkv
```

TIP

You can use Azure Kinect Viewer to configure RGB camera controls before recording (e.g. to set manual white balance).

Verify recording

You can open the output .mkv file with Azure Kinect Viewer.

To extract tracks or view file info, tools such as `mkvinfo` are available as part of the MKVToolNix toolkit.

Next steps

Using recorder with external synchronized units

Azure Kinect DK firmware tool

11/12/2019 • 2 minutes to read • Edit Online

The Azure Kinect Firmware Tool can be used to query and update the device firmware of the Azure Kinect DK.

List connected devices

You can get a list of connected devices by using the -l option. `AzureKinectFirmwareTool.exe -l`

```
 == Azure Kinect DK Firmware Tool ==
Found 2 connected devices:
0: Device "000036590812"
1: Device "000274185112"
```

Check device firmware version

You can check the current firmware versions of the first attached device by using -q option, for example, `AzureKinectFirmwareTool.exe -q`.

```
 == Azure Kinect DK Firmware Tool ==
Device Serial Number: 000036590812
Current Firmware Versions:
    RGB camera firmware:      1.5.92
    Depth camera firmware:    1.5.66
    Depth config file:        6109.7
    Audio firmware:           1.5.14
    Build Config:             Production
    Certificate Type:         Microsoft
```

If there's more than one device attached, you can specify which device you want to query by adding the full serial number to the command, such as:

`AzureKinectFirmwareTool.exe -q 000036590812`

Update device firmware

The most common use of this tool is to update device firmware. Do the update by calling the tool using the `-u` option. A firmware update can take few minutes, depending on which firmware files must be updated.

For step-by-step firmware update instruction, see Azure Kinect firmware update.

`AzureKinectFirmwareTool.exe -u firmware\AzureKinectDK_Fw_1.5.926614.bin`

If there's more than one device attached, you can specify which device you want to query by adding the full serial number to the command.

`AzureKinectFirmwareTool.exe -u firmware\AzureKinectDK_Fw_1.5.926614.bin 000036590812`

Reset device

An attached Azure Kinect DK can be reset using -r option, if you must get the device into a known state.

If there's more than one device attached, you can specify which device you want to query by adding the full serial

number to the command.

```
AzureKinectFirmwareTool.exe -r 000036590812
```

Inspect firmware

Inspecting firmware allows you to get the version information from a firmware bin file before updating an actual device.

```
AzureKinectFirmwareTool.exe -i firmware\AzureKinectDK_Fw_1.5.926614.bin
```

```
 == Azure Kinect DK Firmware Tool ==
Loading firmware package ..\tools\updater\firmware\AzureKinectDK_Fw_1.5.926614.bin.
File size: 1228844 bytes
This package contains:
  RGB camera firmware:      1.5.92
  Depth camera firmware:    1.5.66
  Depth config files: 6109.7 5006.27
  Audio firmware:           1.5.14
  Build Config:             Production
  Certificate Type:         Microsoft
  Signature Type:           Microsoft
```

Firmware update tool options

```
 == Azure Kinect DK Firmware Tool ==
* Usage Info *
    AzureKinectFirmwareTool.exe <Command> <Arguments>

Commands:
    List Devices: -List, -l
    Query Device: -Query, -q
        Arguments: [Serial Number]
    Update Device: -Update, -u
        Arguments: <Firmware Package Path and FileName> [Serial Number]
    Reset Device: -Reset, -r
        Arguments: [Serial Number]
    Inspect Firmware: -Inspect, -i
        Arguments: <Firmware Package Path and FileName>

    If no Serial Number is provided, the tool will just connect to the first device.

Examples:
    AzureKinectFirmwareTool.exe -List
    AzureKinectFirmwareTool.exe -Update c:\data\firmware.bin 0123456
```

Next steps

Step-by-step instructions to update device firmware

Azure Kinect Sensor SDK download

11/12/2019 • 2 minutes to read • Edit Online

This page has the download links for each version of the Azure Kinect Sensor SDK. The installer provides all of the needed files to develop for the Azure Kinect.

Azure Kinect Sensor SDK contents

- Headers and libraries to build an application using the Azure Kinect DK.
- Redistributable DLLs needed by applications using the Azure Kinect DK.
- The Azure Kinect Viewer.
- The Azure Kinect Recorder.
- The Azure Kinect Firmware Tool.

Windows download link

Microsoft installer | GitHub source code

> **NOTE**
> When installing the SDK, remember the path you install to. For example, "C:\Program Files\Azure Kinect SDK 1.2". You will find the tools referenced in articles in this path.

You can find previous versions of Azure Kinect Sensor SDK and Firmware on GitHub.

Linux installation instructions

Currently, the only supported distribution is Ubuntu 18.04. To request support for other distributions, see this page.

First, you'll need to configure Microsoft's Package Repository, following the instructions here.

Now, you can install the necessary packages. The `k4a-tools` package includes the Azure Kinect Viewer, the Azure Kinect Recorder, and the Azure Kinect Firmware Tool. To install it, run

```
sudo apt install k4a-tools
```

The `libk4a<major>.<minor>-dev` package contains the headers and CMake files to build against `libk4a`. The `libk4a<major>.<minor>` package contains the shared objects needed to run executables that depend on `libk4a`.

The basic tutorials require the `libk4a<major>.<minor>-dev` package. To install it, run

```
sudo apt install libk4a1.1-dev
```

If the command succeeds, the SDK is ready for use.

Change log and older versions

You can find the change log for the Azure Kinect Sensor SDK here.

If you need an older version of the Azure Kinect Sensor SDK, find it here.

Next steps

Set up Azure Kinect DK

Download Azure Kinect Body Tracking SDK

1/24/2020 • 3 minutes to read • Edit Online

This document provides links to install each version of the Azure Kinect Body Tracking SDK.

Azure Kinect Body Tracking SDK contents

- Headers and libraries to build a body tracking application using the Azure Kinect DK.
- Redistributable DLLs needed by body tracking applications using the Azure Kinect DK.
- Sample body tracking applications.

Windows download links

VERSION	DOWNLOAD
1.0.0	msi nuget
0.9.5	msi nuget
0.9.4	msi nuget
0.9.3	msi nuget
0.9.2	msi nuget
0.9.1	msi nuget
0.9.0	msi nuget

Linux installation instructions

Currently, the only supported distribution is Ubuntu 18.04. To request support for other distributions, see this page.

First, you'll need to configure Microsoft's Package Repository, following the instructions here.

The `libk4abt<major>.<minor>-dev` package contains the headers and CMake files to build against `libk4abt`. The `libk4abt<major>.<minor>` package contains the shared objects needed to run executables that depend on `libk4abt` as well as the example viewer.

The basic tutorials require the `libk4abt<major>.<minor>-dev` package. To install it, run

```
sudo apt install libk4abt1.0-dev
```

If the command succeeds, the SDK is ready for use.

> **NOTE**
> When installing the SDK, remember the path you install to. For example, "C:\Program Files\Azure Kinect Body Tracking SDK 1.0.0". You will find the samples referenced in articles in this path. Body tracking samples are located in the body-tracking-samples folder in the Azure-Kinect-Samples repository. You will find the samples referenced in articles here.

Change log

v1.0.0

- [Feature] Add C# wrapper to the msi installer.
- [Bug Fix] Fix issue that the head rotation cannot be detected correctly: Link
- [Bug Fix] Fix issue that the CPU usage goes up to 100% on Linux machine: Link
- [Samples] Add two samples to the sample repo. Sample 1 demonstrates how to transform body tracking results from the depth space to color space Link; sample 2 demonstrates how to detect floor plane Link

v0.9.5

- [Feature] C# support. C# wrapper is packed in the nuget package.
- [Feature] Multi-tracker support. Creating multiple trackers is allowed. Now user can create multiple trackers to track bodies from different Azure Kinect devices.
- [Feature] Multi-thread processing support for CPU mode. When running on CPU mode, all cores will be used to maximize the speed.
- [Feature] Add `gpu_device_id` to `k4abt_tracker_configuration_t` struct. Allow users to specify GPU device that is other than the default one to run the body tracking algorithm.
- [Bug Fix/Breaking Change] Fix typo in a joint name. Change joint name from `K4ABT_JOINT_SPINE_NAVAL` to `K4ABT_JOINT_SPINE_NAVEL`.

v0.9.4

- [Feature] Add hand joints support. The SDK will provide information for three additional joints for each hand: HAND, HANDTIP, THUMB.
- [Feature] Add prediction confidence level for each detected joints.
- [Feature] Add CPU mode support. By changing the `cpu_only_mode` value in `k4abt_tracker_configuration_t`, now the SDK can run on CPU mode which doesn't require the user to have a powerful graphics card.

v0.9.3

- [Feature] Publish a new DNN model dnn_model_2_0.onnx, which largely improves the robustness of the body tracking.
- [Feature] Disable the temporal smoothing by default. The tracked joints will be more responsive.
- [Feature] Improve the accuracy of the body index map.
- [Bug Fix] Fix bug that the sensor orientation setting is not effective.
- [Bug Fix] Change the body_index_map type from K4A_IMAGE_FORMAT_CUSTOM to K4A_IMAGE_FORMAT_CUSTOM8.
- [Known Issue] Two close bodies may merge to single instance segment.

v0.9.2

- [Breaking Change] Update to depend on the latest Azure Kinect Sensor SDK 1.2.0.
- [API Change] `k4abt_tracker_create` function will start to take a `k4abt_tracker_configuration_t` input.
- [API Change] Change `k4abt_frame_get_timestamp_usec` API to `k4abt_frame_get_device_timestamp_usec` to be more specific and consistent with the Sensor SDK 1.2.0.
- [Feature] Allow users to specify the sensor mounting orientation when creating the tracker to achieve more accurate body tracking results when mounting at different angles.

- [Feature] Provide new API `k4abt_tracker_set_temporal_smoothing` to change the amount of temporal smoothing that the user wants to apply.
- [Feature] Add C++ wrapper k4abt.hpp.
- [Feature] Add version definition header k4abtversion.h.
- [Bug Fix] Fix bug that caused extremely high CPU usage.
- [Bug Fix] Fix logger crashing bug.

v0.9.1

- [Bug Fix] Fix memory leak when destroying tracker
- [Bug Fix] Better error messages for missing dependencies
- [Bug Fix] Fail without crashing when creating a second tracker instance
- [Bug Fix] Logger environmental variables now work correctly
- Linux support

v0.9.0

- [Breaking Change] Downgraded the SDK dependency to CUDA 10.0 (from CUDA 10.1). ONNX runtime officially only supports up to CUDA 10.0.
- [Breaking Change] Switched to ONNX runtime instead of Tensorflow runtime. Reduces the first frame launching time and memory usage. It also reduces the SDK binary size.
- [API Change] Renamed `k4abt_tracker_queue_capture()` to `k4abt_tracker_enqueue_capture()`
- [API Change] Broke `k4abt_frame_get_body()` into two separate functions: `k4abt_frame_get_body_skeleton()` and `k4abt_frame_get_body_id()`. Now you can query the body ID without always copying the whole skeleton structure.
- [API Change] Added `k4abt_frame_get_timestamp_usec()` function to simplify the steps for the users to query body frame timestamp.
- Further improved the body tracking algorithm accuracy and tracking reliability

Next steps

- Azure Kinect DK overview
- Set up Azure Kinect DK
- Set up Azure Kinect body tracking

Azure Kinect sensor SDK system requirements

11/12/2019 • 2 minutes to read • Edit Online

This document provides details about the system requirements needed to install the sensor SDK and successfully deploy your Azure Kinect DK.

Supported operating systems and architectures

- Windows 10 April 2018 release (x64) or later
- Linux Ubuntu 18.04 (x64) with OpenGLv4.4 or later GPU driver

The Sensor SDK is available for the Windows API (Win32) for native C/C++ Windows applications. The SDK isn't currently available to UWP applications. Azure Kinect DK isn't supported for Windows 10 in S mode.

Development environment requirements

To contribute to sensor SDK development, visit GitHub.

Minimum host PC hardware requirements

The PC host hardware requirement is dependent on application/algorithm/sensor frame rate/resolution executed on host PC. Recommended minimum Sensor SDK configuration for Windows is:

- Seventh Gen Intel® CoreTM i3 Processor (Dual Core 2.4 GHz with HD620 GPU or faster)
- 4 GB Memory
- Dedicated USB3 port
- Graphics driver support for OpenGL 4.4 or DirectX 11.0

Lower end or older CPUs may also work depending on your use-case.

Performance differs also between Windows/Linux operating systems and graphics drivers in use.

Body tracking host PC hardware requirements

The body tracking PC host requirement is more stringent than the general PC host requirement. Recommended minimum Body Tracking SDK configuration for Windows is:

- Seventh Gen Intel® CoreTM i5 Processor (Quad Core 2.4 GHz or faster)
- 4 GB Memory
- NVIDIA GEFORCE GTX 1070 or better
- Dedicated USB3 port

The recommended minimum configuration assumes K4A_DEPTH_MODE_NFOV_UNBINNED depth mode at 30fps tracking 5 people. Lower end or older CPUs and NVIDIA GPUs may also work depending on your use-case.

USB3

There are known compatibility issues with USB Host controllers. You can find more information on Troubleshooting page

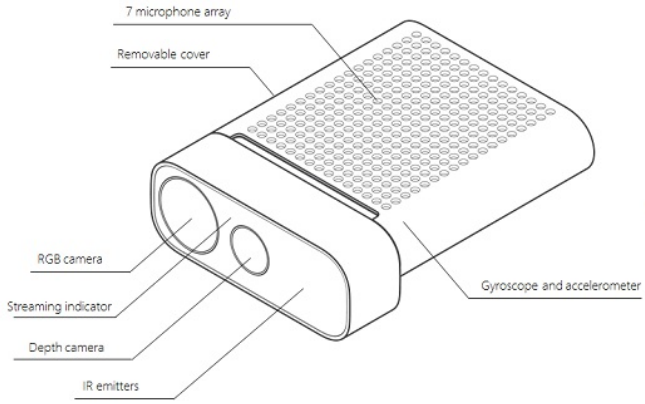
Next steps

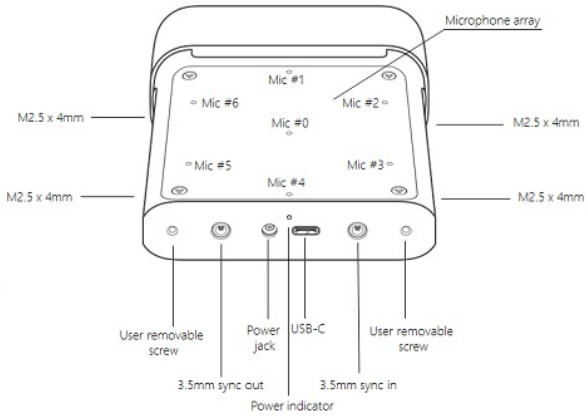
- Azure Kinect DK overview
- Set up Azure Kinect DK
- Set up Azure Kinect body tracking

Azure Kinect DK hardware specifications

2/16/2020 • 7 minutes to read • Edit Online

This article provides details about how Azure Kinect hardware integrates Microsoft's latest sensor technology into a single, USB-connected accessory.





Terms

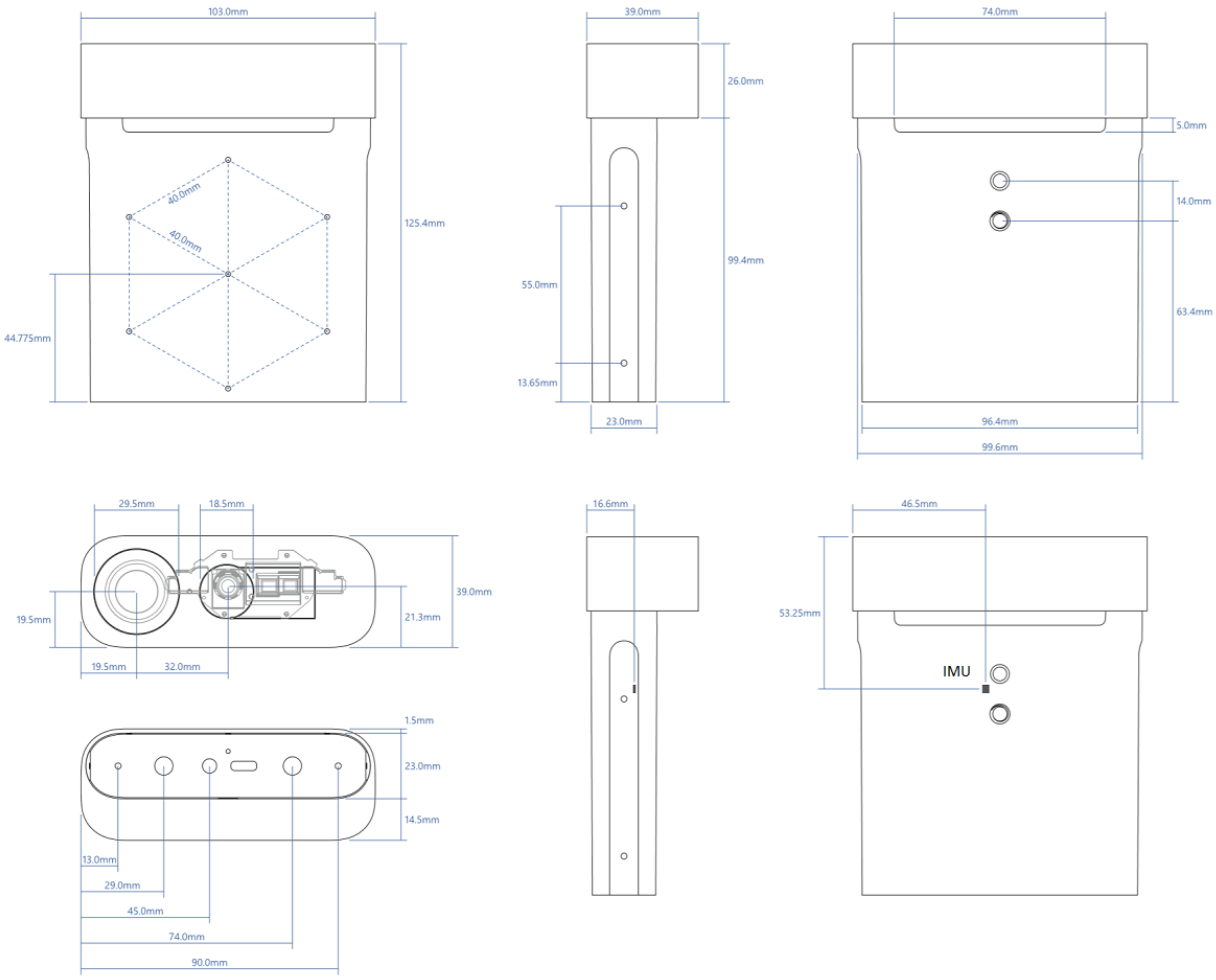
These abbreviated terms are used throughout this article.

- NFOV (Narrow field-of-view depth mode)
- WFOV (Wide field-of-view depth mode)
- FOV (Field-of-view)
- FPS (Frames-per-second)
- IMU (Inertial Measurement Unit)
- FoI (Field of Interest)

Product dimensions and weight

The Azure Kinect device consists of the following size and weight dimensions.

- **Dimensions**: 103 x 39 x 126 mm
- **Weight**: 440 g



Operating environment

Azure Kinect DK is intended for developers and commercial businesses operating under the following ambient conditions:

- **Temperature**: 10-25^{0}C
- **Humidity**: 8-90% (non-condensing) Relative Humidity

> **NOTE**
>
> Use outside of the ambient conditions could cause the device to fail and/or function incorrectly. These ambient conditions are applicable for the environment immediately around the device under all operational conditions. When used with an external enclosure, active temperature control and/or other cooling solutions are recommended to ensure the device is maintained within these ranges. The device design features a cooling channel in between the front section and rear sleeve. When you implement the device, make sure this cooling channel is not obstructed.

Refer to additional product safety information.

Depth camera supported operating modes

Azure Kinect DK integrates a Microsoft designed 1-Megapixel Time-of-Flight (ToF) depth camera using the image sensor presented at ISSCC 2018. The depth camera supports the modes indicated below:

MODE	RESOLUTION	FOI	FPS	OPERATING RANGE*	EXPOSURE TIME
NFOV unbinned	640x576	75°x65°	0, 5, 15, 30	0.5 - 3.86 m	12.8 ms
NFOV 2x2 binned (SW)	320x288	75°x65°	0, 5, 15, 30	0.5 - 5.46 m	12.8 ms
WFOV 2x2 binned	512x512	120°x120°	0, 5, 15, 30	0.25 - 2.88 m	12.8 ms
WFOV unbinned	1024x1024	120°x120°	0, 5, 15	0.25 - 2.21 m	20.3 ms
Passive IR	1024x1024	N/A	0, 5, 15, 30	N/A	1.6 ms

*15% to 95% reflectivity at 850nm, 2.2 μW/cm^2/nm, random error std. dev. ≤ 17 mm, typical systematic error < 11 mm + 0.1% of distance without multi-path interference. Depth provided outside of indicated range depending on object reflectivity.

Color camera supported operating modes

Azure Kinect DK includes an OV12A10 12MP CMOS sensor rolling shutter sensor. The native operating modes are listed below:

RGB CAMERA RESOLUTION (HXV)	ASPECT RATIO	FORMAT OPTIONS	FRAME RATES (FPS)	NOMINAL FOV (HXV) (POST-PROCESSED)
3840x2160	16:9	MJPEG	0, 5, 15, 30	90°x59°
2560x1440	16:9	MJPEG	0, 5, 15, 30	90°x59°
1920x1080	16:9	MJPEG	0, 5, 15, 30	90°x59°
1280x720	16:9	MJPEG/YUY2/NV12	0, 5, 15, 30	90°x59°
4096x3072	4:3	MJPEG	0, 5, 15	90°x74.3°
2048x1536	4:3	MJPEG	0, 5, 15, 30	90°x74.3°

The RGB camera is USB Video class-compatible and can be used without the Sensor SDK. The RGB camera color space: BT.601 full range [0..255].

> **NOTE**
>
> The Sensor SDK can provide color images in the BGRA pixel format. This is not a native mode supported by the device and causes additional CPU load when used. The host CPU is used to convert from MJPEG images received from the device.

RGB camera exposure time values

Below is the mapping for the acceptable RGB camera manual exposure values:

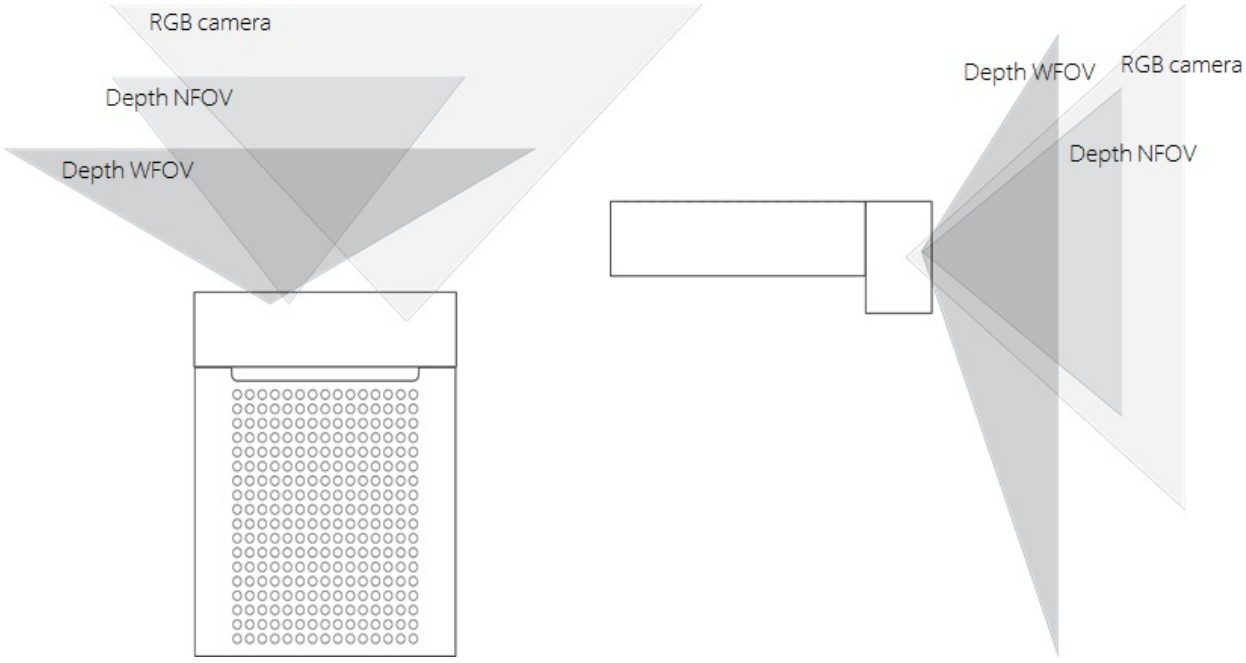
EXP	2^EXP	50HZ	60HZ
-11	488	500	500
-10	977	1250	1250
-9	1953	2500	2500
-8	3906	10000	8330
-7	7813	20000	16670
-6	15625	30000	33330
-5	31250	40000	41670
-4	62500	50000	50000
-3	125000	60000	66670
-2	250000	80000	83330
-1	500000	100000	100000
0	1000000	120000	116670
1	2000000	130000	133330

Depth sensor raw timing

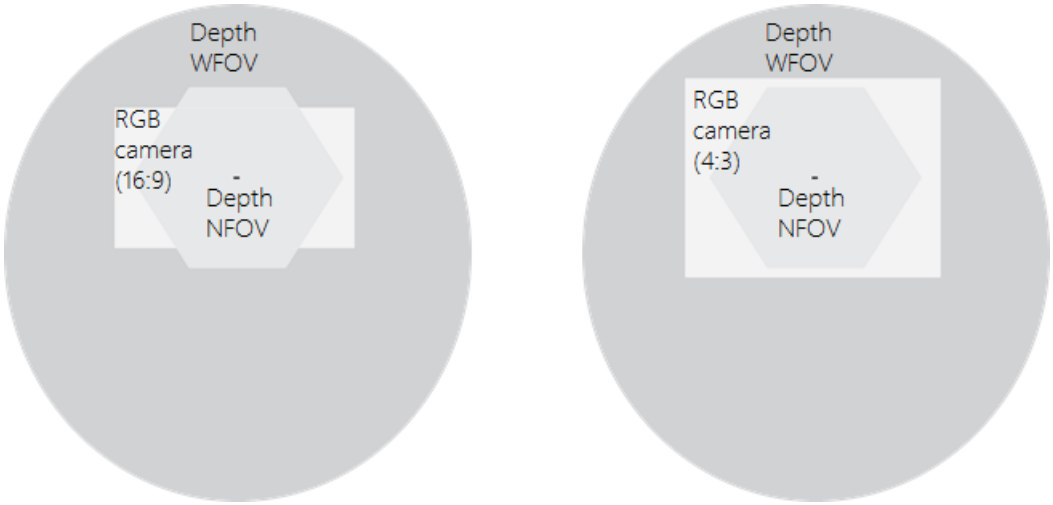
DEPTH MODE	IR PULSES	PULSE WIDTH	IDLE PERIODS	IDLE TIME	EXPOSURE TIME
NFOV Unbinned NFOV 2xx Binned WFOV 2x2 Binned	9	125 us	8	1450 us	12.8 ms
WFOV Unbinned	9	125 us	8	2390 us	20.3 ms

Camera field of view

The next image shows the depth and RGB camera field-of-view, or the angles that the sensors "see". This diagram shows the RGB camera in a 4:3 mode.



This image demonstrates the camera's field-of-view as seen from the front at a distance of 2000 mm.



NOTE

When depth is in NFOV mode, the RGB camera has better pixel overlap in 4:3 than 16:9 resolutions.

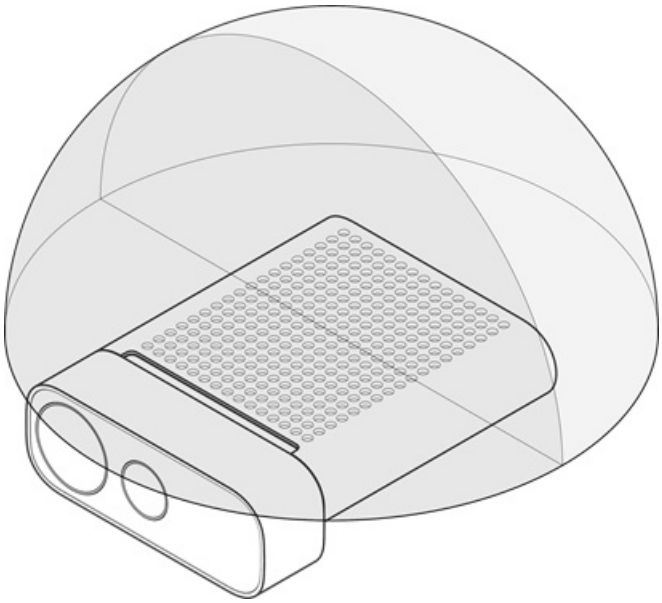
Motion sensor (IMU)

The embedded Inertial Measurement Unit (IMU) is an LSM6DSMUS and includes both an accelerometer and a gyroscope. The accelerometer and gyroscope are simultaneously sampled at 1.6 kHz. The samples are reported to the host at a 208 Hz.

Microphone array

Azure Kinect DK embeds a high-quality, seven microphone circular array that identifies as a standard USB audio class 2.0 device. All 7 channels can be accessed. The performance specifications are:

- Sensitivity: -22 dBFS (94 dB SPL, 1 kHz)
- Signal to noise ratio > 65 dB
- Acoustic overload point: 116 dB



USB

Azure Kinect DK is a USB3 composite device that exposes the following hardware endpoints to the operating system:

Vendor ID is 0x045E (Microsoft), Product ID table below:

USB INTERFACE	PNP IP	NOTES
USB3.1 Gen1 Hub	0x097A	The main hub
USB2.0 Hub	0x097B	HS USB
Depth camera	0x097C	USB3.0
Color camera	0x097D	USB3.0
Microphones	0x097E	HS USB

Indicators

The device has a camera streaming indicator on the front of the device that can be disabled programmatically using the Sensor SDK.

The status LED behind the device indicates device state:

WHEN THE LIGHT IS	IT MEANS
Solid white	Device is on and working properly.
Flashing white	Device is on but doesn’t have a USB 3.0 data connection.
Flashing amber	Device doesn't have enough power to operate.
Amber flashing white	Firmware update or recovery in progress

Power device

The device can be powered in two ways:

1. Using the in-box power supply. Data is connected by a separate USB Type-C to Type-A cable.
2. Using a Type-C to Type-C cable for both power and data.

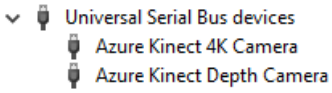
A Type-C to Type-C cable isn't included with the Azure Kinect DK.

> **NOTE**
>
> - The in-box power supply cable is a USB Type-A to single post barrel connector. Use the provided wall-power supply with this cable. The device is capable of drawing more power than two standard USB Type-A ports can provide.
> - USB cables do matter and we recommended to use high-quality cables and verify functionality before deploying the unit remotely.

> **TIP**
>
> To select a good Type-C to Type-C cable:
>
> - The USB certified cable must support both power and data.
> - A passive cable should be less than 1.5m in length. If longer, use an active cable.
> - The cable needs to support no less than >1.5A. Otherwise you need to connect an external power supply.

Verify cable:

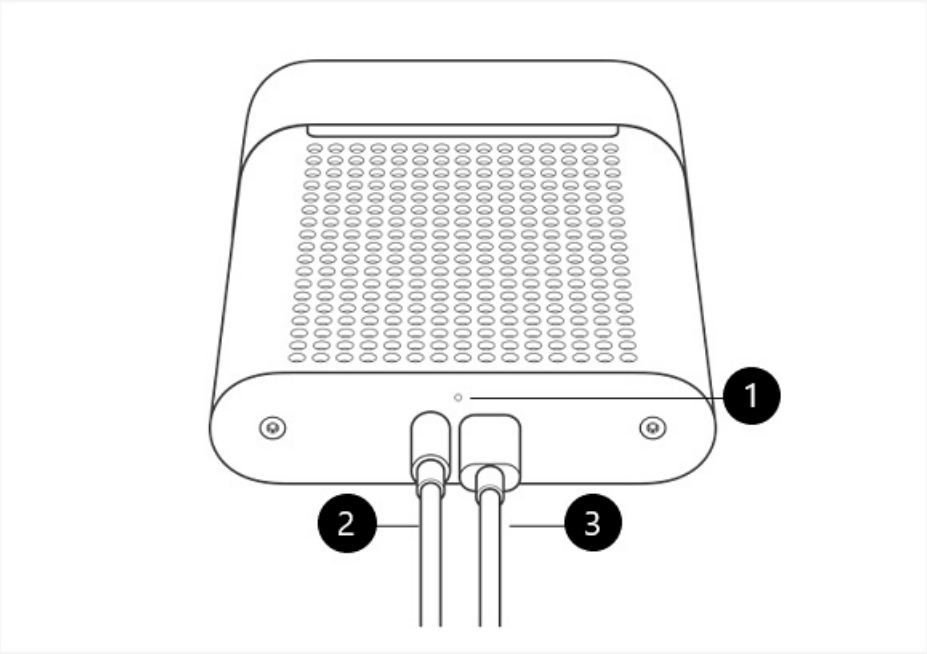
- Connect device via the cable to the host PC.
- Validate that all devices enumerate correctly in Windows device manager. Depth and RGB camera should appear as shown in the example below.



- Validate that cable can stream reliably on all sensors in the Azure Kinect Viewer, with the following settings:
  - Depth camera: NFOV unbinned
  - RGB Camera: 2160p
  - Microphones and IMU enabled

What does the light mean?

The power indicator is an LED on the back of your Azure Kinect DK. The color of the LED changes depending on the status of your device.



This figure labels the following components:

1. Power indicator
2. Power cable (connected to the power source)
3. USB-C data cable (connected to the PC)

Make sure that the cables are connected as shown. Then check the following table to learn what the various states of the power light indicate.

WHEN THE LIGHT IS:	IT MEANS THAT:	AND YOU SHOULD:
Solid white	The device is powered on and working correctly.	Use the device.
Not lit	The device is not connected to the PC.	Make sure that the round power connector cable is connected to the device and to the USB power adapter. Make sure that the USB-C cable is connected to the device and to your PC.
Flashing white	The device is powered on but doesn't have a USB 3.0 data connection.	Make sure that the round power connector cable is connected to the device and to the USB power adapter. Make sure that the USB-C cable is connected to the device and to a USB 3.0 port on your PC. Connect the device to a different USB 3.0 port on the PC. On your PC, open Device Manager (**Start** > **Control Panel** > **Device Manager**), and verify that your PC has a supported USB 3.0 host controller.

WHEN THE LIGHT IS:	IT MEANS THAT:	AND YOU SHOULD:
Flashing amber	The device doesn't have enough power to operate.	Make sure that the round power connector cable is connected to the device and to the USB power adapter. Make sure that the USB-C cable is connected to the device and to your PC.
Amber, then flashing white	The device is powered on and is receiving a firmware update, or the device is restoring the factory settings.	Wait for the power indicator light to become solid white. For more information, see Reset Azure Kinect DK.

Power consumption

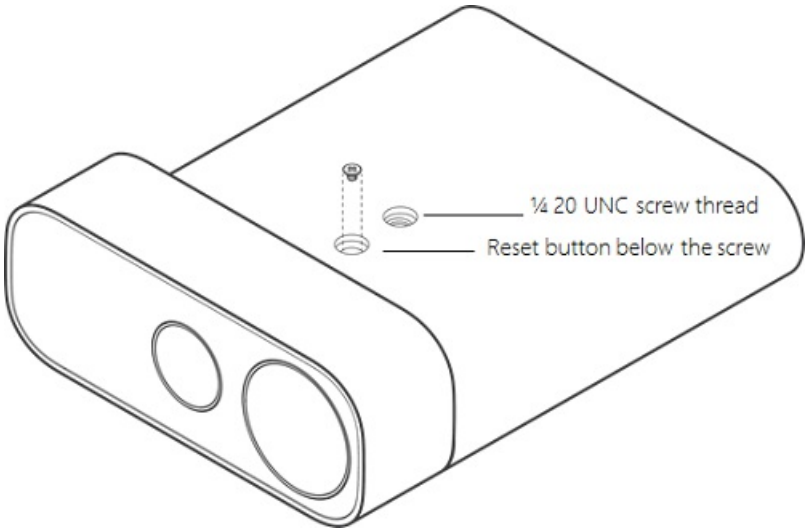
Azure Kinect DK consumes up to 5.9 W; specific power consumption is use-case dependent.

Calibration

Azure Kinect DK is calibrated at the factory. The calibration parameters for visual and inertial sensors may be queried programmatically through the Sensor SDK.

Device recovery

Device firmware can be reset to original firmware using button underneath the lock pin.



To recover the device, see instructions here.

Next steps

- Use Azure Kinect Sensor SDK
- Set up hardware

Synchronization across multiple Azure Kinect DK devices

1/19/2020 • 6 minutes to read • Edit Online

In this article, we will explore the benefits of multi device synchronization and its details.

Before you start, make sure to review Azure Kinect DK Hardware specification and the the multi-camera hardware set up.

There are a few important things to consider before starting your multi-camera setup.

- We recommend using a manual exposure setting if you want to control the precise timing of each device. Automatic exposure allows each color camera to dynamically change exposure, as a result it is impossible for the timing between the two devices to stay exactly the same.
- The device timestamp reported for images changes meaning to 'Start of Frame' from 'Center of Frame' when using master or subordinate modes.
- Avoid IR camera interference between different cameras. Use `depth_delay_off_color_usec` or `subordinate_delay_off_master_usec` to ensure each IR laser fires in its own 160us window or has a different field of view.
- Do ensure you are using the most recent firmware version.
- Do not repeatedly set the same exposure setting in the image capture loop.
- Do set the exposure when needed, just call the API once.

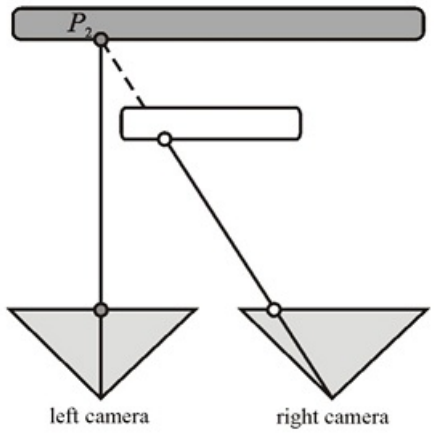
Why to use multiple Azure Kinect DK devices?

There are many reasons to use multiple Azure Kinect DK devices. Some examples are

- Fill in occlusions
- 3D object scanning
- Increase the effective frame rate to something larger than the 30 FPS
- Multiple 4K color images capture of the same scene, all aligned at the start of exposure within 100 us
- Increase camera coverage within the space

Solve for occlusion

Occlusion means that there is something you want to see, but can't see it due to some interference. In our case Azure Kinect DK device has two cameras (depth and color cameras) that do not share the same origin, so one camera can see part of an object that other cannot. Therefore, when transforming depth to color image, you may see a shadow around an object. On the image below, the left camera sees the grey pixel P2, but the ray from the right camera to P2 hits the white foreground object. As a result the right camera cannot see P2.



Using additional Azure Kinect DK devices will solve this issue and fill out an occlusion problem.

Set up multiple Azure Kinect DK devices

Make sure to review the multi-camera hardware setup article that describes different options for hardware setup.

Synchronization cables

Azure Kinect DK includes 3.5-mm synchronization ports that can be used to link multiple units together. When linked, cameras can coordinate the timing of Depth and RGB camera triggering. There are specific sync-in and sync-out ports on the device, enabling easy daisy chaining. A compatible cable isn't included in box and must be purchased separately.

Cable requirements:

- 3.5-mm male-to-male cable ("3.5-mm audio cable")
- Maximum cable length should be less than 10 meters
- Both stereo and mono cable types are supported

When using multiple depth cameras in synchronized captures, depth camera captures should be offset from one another by 160µs or more to avoid depth cameras interference.

> **NOTE**
> Make sure to remove the cover in order to reveal the sync ports.

Cross-device calibration

In a single device depth and RGB cameras factory calibrated. However, when multiple devices are used, new calibration requirements need to be considered to determine how to transform an image from the domain of the camera it was captured in, to the domain of the camera you want to process images in. There are multiple options for cross-calibrating devices, but in the GitHub green screen code sample we are using OpenCV methods There are multiple options for cross-calibrating devices, but in the GitHub green screen code sample we are using OpenCV method.

USB Memory on Ubuntu

If you are setting up multi-camera synchronization on Linux, by default the USB controller is only allocated 16 MB of kernel memory for handling of USB transfers. It is typically enough to support a single Azure Kinect DK, however more memory is needed to support multiple devices. To increase the memory, follow the below steps:

- Edit /etc/default/grub
- Replace the line that says GRUB_CMDLINE_LINUX_DEFAULT="quiet splash" with GRUB_CMDLINE_LINUX_DEFAULT="quiet splash usbcore.usbfs_memory_mb=32". In this example, we set the USB memory to 32 MB twice that of the default, however to can be set much larger. Choose a value that is

right for your solution.

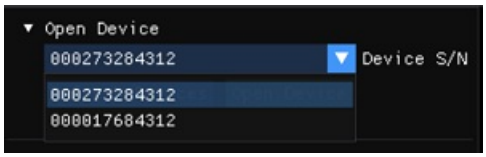
- Run sudo update-grub
- Restart the computer

Verify two Azure Kinect DKs' synchronization

After setting up the hardware and connecting the sync out port of the master to sync in of the subordinate, we can use the Azure Kinect Viewer to validate the devices setup. It also can be done for more than two devices.

> **NOTE**
> The Subordinate device is the one that connected to "Sync In" pin. The master is the one connected "Synch Out".

1. Get the serial number for each device.
2. Open two instances of Azure Kinect Viewer
3. Open subordinate Azure Kinect DK device first. Navigate to Azure Kinect viewer, and in the Open Device section choose subordinate device:



4. In the section "External Sync", choose option "Sub" and start the device. Images will not be sent to the subordinate after hitting start due to the device waiting for the sync pulse from the master device.

5. Navigate to another instance of the Azure Kinect viewer and open the master Azure Kinect DK device.

6. In the section "External Sync", choose option "Master" and start the device.

> **NOTE**
> The master device must always be started last the get precise image capture alignment between all devices.

When the master Azure Kinect Device is started, the synchronized image from both of the Azure Kinect devices should appear.

Avoiding interference from other depth cameras

Interference happens when the depth sensor's ToF lasers are on at the same time as another depth camera. To avoid it, cameras that have overlapping areas of interest need to have their timing shifted by the "laser on time" so they are not on at the same time. For each capture, the laser turns on nine times and is active for only 125us and is then idle for 1450us or 2390us depending on the mode of operation. As a result, depth cameras need their "laser on time" shifted by a minimum of 125us and that on time needs to fall into the idle time of the other depth sensors in use.

Due to the differences in the clock used by the firmware and the clock used by the camera, 125us cannot be used directly. Instead the software setting required to ensure sure there is no camera interference is 160us. It allows nine more depth cameras to be scheduled into the 1450us of idle time of NFOV. The exact timing changes based on the depth mode you are using.

Using the depth sensor raw timing table the exposure time can be calculated as:

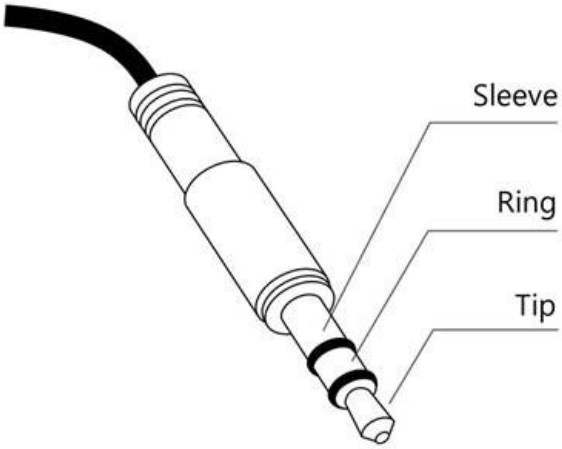
NOTE

Exposure Time = (IR Pulses * Pulse Width) + (Idle Periods * Idle Time)

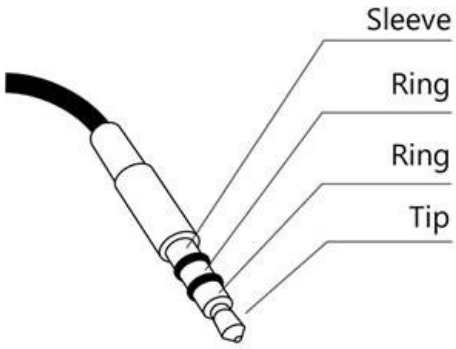
Triggering with custom source

A custom sync source can be used to replace the master Azure Kinect DK. It is helpful when the image captures need to be synchronized with other equipment. The custom trigger must create a sync signal, similar to the master device, via the 3.5-mm port.

- The SYNC signals are active high and pulse width should be greater than 8us.
- Frequency support must be precisely 30 fps, 15 fps, and 5 fps, the frequency of color camera's master VSYNC signal.
- SYNC signal from the board should be 5 V TTL/CMOS with maximum driving capacity no less than 8 mA.
- All styles of 3.5-mm port can be used with Kinect DK, including "mono", that is not pictured. All sleeves and rings are shorted together inside Kinect DK and they are connected to ground of the master Azure Kinect DK. The tip is the sync signal.



TRS Audio Plug



TRRS Audio Plug

Next steps

- Use Azure Kinect Sensor SDK
- Capture Azure Kinect device synchronization
- Set up hardware

Azure Kinect and Kinect Windows v2 comparison

6/26/2019 • 2 minutes to read • Edit Online

The Azure Kinect DK hardware and Software Development Kits have differences from Kinect for Windows v2. Any existing Kinect for Windows v2 applications will not work directly with Azure Kinect DK and will require porting to the new SDK.

Hardware

High-level differences between the Azure Kinect development kit and Kinect for Windows v2 are listed in the following table.

		AZURE KINECT DK	KINECT FOR WINDOWS V2
Audio	Details	7-mic circular array	4-mic linear phased array
Motion sensor	Details	3-axis accelerometer 3-axis gyro	3-axis accelerometer
RGB Camera	Details	3840 x 2160 px @30 fps	1920 x 1080 px @30 fps
Depth Camera	Method	Time-of-Flight	Time-of-Flight
	Resolution	640 x 576 px @30 fps	512 x 424 px @ 30 fps
		512 x 512 px @30 fps	
		1024x1024 px @15 fps	
Connectivity	Data	USB3.1 Gen 1 with type USB-C	USB 3.1 gen 1
	Power	External PSU or USB-C	External PSU
	Synchronization	RGB & Depth internal, external device-to-device	RGB & Depth internal only
Mechanical	Dimensions	103 x 39 x 126 mm	249 x 66 x 67 mm
	Mass	440 g	970 g
	Mounting	One ¼-20 UNC. Four internal screw points	One ¼-20 UNC

Find additional details in the Azure Kinect DK hardware document.

Sensor access

The following table provides low-level device sensor access capability comparison.

FUNCTIONALITY	AZURE KINECT	KINECT FOR WINDOWS	NOTES	
Depth	✔	✔		
IR	✔	✔		
Color	✔	✔	Color format supports differences, Azure Kinect DK supports these camera controls: Exposure, white balance, brightness, contrast, saturation, sharpness, and gain control	
Audio	✔	✔	Azure Kinect DK mics are accessed via Speech SDK or Windows native API	
IMU	✔		Azure Kinect DK has a full 6-axis IMU and Kinect for Windows only provides 1-axis	
Calibration data	✔	✔	OpenCV compatible camera model calibration	
Depth-RGB internal sync	✔	✔		
External Sync	✔		Azure Kinect DK allows programmable delay for external sync	
Share access with multiple clients		✔	The Azure Kinect Sensor SDK relies on WinUSB/libUSB to access device and does not have a service implemented to enable sharing device access with multiple processes	
Stream record / playback tool	✔	✔	Azure Kinect DK uses an open-source Matroska container-based implementation	

Features

The Azure Kinect SDK feature set is different from Kinect for Windows v2, as detailed below:

KINECT V2 FEATURE	KINECT V2 DATA TYPE	AZURE KINECT SDK/SERVICE
Sensor Data Access	DepthFrame	Sensor SDK - Retrieve images
	InfraredFrame	Sensor SDK - Retrieve images
	ColorFrame	Sensor SDK - Retrieve images
	AudioBeamFrame	Not currently supported
Body Tracking	BodyFrame	Body Tracking SDK
	BodyIndexFrame	Body Tracking SDK
Coordinate Mapping	CoordinateMapper	Sensor SDK - Image transformations
Face Tracking	FaceFrame	Cognitive Services: Face
Speech Recognition	N/A	Cognitive Services: Speech

Next steps

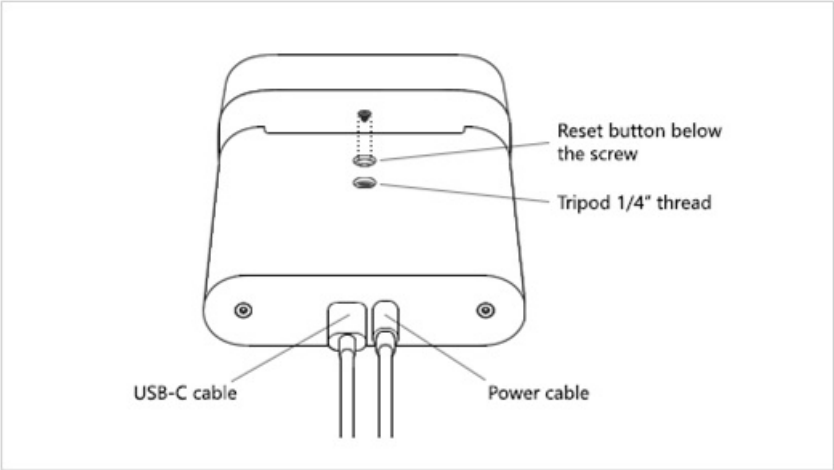
Kinect for Windows developer pages

Reset Azure Kinect DK

2/13/2020 • 2 minutes to read • Edit Online

You may encounter a situation in which you have to reset your Azure Kinect DK back to its factory image (for example, if a firmware update didn't install correctly).

1. Power off your Azure Kinect DK. To do this, remove the USB cable and power cable.



2. To find the reset button, remove the screw that's located in the tripod mount lock.
3. Reconnect the power cable.
4. Insert the tip of a straightened paperclip into the empty screw hole, in the tripod mount lock.
5. Use the paperclip to gently press and hold the reset button.
6. While you hold the reset button, reconnect the USB cable.
7. After about 3 seconds, the power indicator light changes to amber. After the light changes, release the reset button.

   After you release the reset button, the power indicator light blinks white and amber while the device resets.
8. Wait for the power indicator light to become solid white.
9. Replace the screw in the tripod mount lock, over the reset button.
10. Use Azure Kinect Viewer to verify that the firmware was reset. To do this, launch the Azure Kinect Viewer, and then select **Device firmware version info** to see the firmware version that is installed on your Azure Kinect DK.

Always make sure that you have the latest firmware installed on the device. To get the latest firmware version, use the Azure Kinect Firmware Tool. For more information about how to check your firmware status, see Check device firmware version.

Related topics

- About Azure Kinect DK
- Set up Azure Kinect DK
- Azure Kinect DK hardware specifications: Operating environment
- Azure Kinect Firmware Tool

- Azure Kinect Viewer
- Synchronization across multiple Azure Kinect DK devices

Azure Kinect support options and resources

11/12/2019 • 2 minutes to read • Edit Online

This article will walk you through various support options.

Community support

There are multiple ways to get your questions answered through public forums:

- StackOverflow, where you can ask questions or search through existing library of questions.
- GitHub, where you can ask questions, open new bugs or contribute to development of Azure Kinect sensor SDK.
- Provide feedback, where you can share your ideas for the future of the product and vote for an existing idea.

Assisted support

There are multiple ways to get an assisted support for Azure Kinect.

Development Azure Kinect on Azure

Azure subscribers can create and manage support requests in the Azure portal. One-on-one development support for Body Tracking, Sensor SDK, Speech device SDK, or Azure Cognitive Services is available for Azure subscribers with an Azure Support Plan associated with their subscription.

- Have an Azure Support Plan associated with your Azure subscription?Sign in to Azure portal to submit an incident.
- Need an Azure Subscription? Azure subscription options will provide more information about different options.
- Need a Support plan? Select support plan

Azure Kinect on-premises or other cloud services

For technical support using Sensor SDK and Body Tracking SDK on-premises, open a ticket for professional support on Microsoft support portal.

Azure Kinect DK device

Before contacting hardware support, make sure that you have set up and updated Azure Kinect DK. To test if the device is working, use the Azure Kinect viewer. Find out more on our Azure Kinect DK help page. You may also want to check out our known issues and troubleshooting.

Get help with a device or sensor functionality, firmware updates, or purchasing options.

For more information on support offerings, learn more at Microsoft support for business.

Next steps

Azure Kinect troubleshooting

Azure Kinect known issues and troubleshooting

1/23/2020 • 7 minutes to read • Edit Online

This page contains known issues and troubleshooting tips when using Sensor SDK with Azure Kinect DK. See also product support pages for product hardware- specific issues.

Known issues

- Compatibility issues with ASMedia USB host controllers (for example, ASM1142 chipset)
  - Some cases using Microsoft USB driver can unblock
  - Many PCs have also alternative host controllers and changing the USB3 port may help

For more Sensor SDK-related issues, check GitHub Issues

Collecting logs

Logging for K4A.dll is enabled through environment variables. By default logging is sent to stdout and only errors and critical messages are generated. These settings can be altered so that logging goes to a file. The verbosity can also be adjusted as needed. Below is an example, for Windows, of enabling logging to a file, named k4a.log, and will capture warning and higher-level messages.

1. `set K4A_ENABLE_LOG_TO_A_FILE=k4a.log`
2. `set K4A_LOG_LEVEL=w`
3. Run scenario from command prompt (for example, launch viewer)
4. Navigate to k4a.log and share file.

For more information, see below clip from header file:

```
/**
* environment variables
* K4A_ENABLE_LOG_TO_A_FILE =
*    0    - completely disable logging to a file
*    log\custom.log - log all messages to the path and file specified - must end in '.log' to
*                     be considered a valid entry
*    ** When enabled this takes precedence over the value of K4A_ENABLE_LOG_TO_STDOUT
*
* K4A_ENABLE_LOG_TO_STDOUT =
*    0    - disable logging to stdout
*    all else  - log all messages to stdout
*
* K4A_LOG_LEVEL =
*    'c'  - log all messages of level 'critical' criticality
*    'e'  - log all messages of level 'error' or higher criticality
*    'w'  - log all messages of level 'warning' or higher criticality
*    'i'  - log all messages of level 'info' or higher criticality
*    't'  - log all messages of level 'trace' or higher criticality
*    DEFAULT - log all message of level 'error' or higher criticality
*/
```

Logging for the Body Tracking SDK K4ABT.dll is similar except that users should modify a different set of environment variable names:

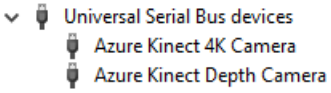
```
/**
* environment variables
* K4ABT_ENABLE_LOG_TO_A_FILE =
*    0    - completely disable logging to a file
*    log\custom.log - log all messages to the path and file specified - must end in '.log' to
*                     be considered a valid entry
*    ** When enabled this takes precedence over the value of K4A_ENABLE_LOG_TO_STDOUT
*
* K4ABT_ENABLE_LOG_TO_STDOUT =
*    0    - disable logging to stdout
*    all else  - log all messages to stdout
*
* K4ABT_LOG_LEVEL =
*    'c'  - log all messages of level 'critical' criticality
*    'e'  - log all messages of level 'error' or higher criticality
*    'w'  - log all messages of level 'warning' or higher criticality
*    'i'  - log all messages of level 'info' or higher criticality
*    't'  - log all messages of level 'trace' or higher criticality
*    DEFAULT - log all message of level 'error' or higher criticality
*/
```

Device doesn't enumerate in device manager

- Check the status LED behind the device, if it's blinking amber you have USB connectivity issue and it doesn't get enough power. The power supply cable should be plugged into the provided power adapter. While the power cable has a USB type A connected, the device requires more power than a PC USB port can supply. So, don't connect to it to a PC port or USB hub.
- Check that you have power cable connected and using USB3 port for data.
- Try changing USB3 port for the data connection (recommendation to use USB port close to motherboard, for example, in back of the PC).
- Check your cable, damaged or lower quality cables may cause unreliable enumeration (device keeps "blinking" in device manager).
- If you have connected to laptop and running on battery, it may be throttling the power to the port.
- Reboot host PC.
- If problem persists, there may be compatibility issue.
- If failure happened during firmware update and device has not recovered by itself, perform factory reset.

Azure Kinect Viewer fails to open

- Check first that your device enumerates in Windows Device Manager.

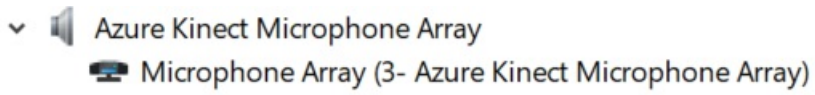


- Check if you have any other application using the device (for example, Windows camera application). Only one application at a time can access the device.
- Check k4aviewer.err log for error messages.
- Open Windows camera application and check of that works.
- Power cycle device, wait streaming LED to power off before using the device.
- Reboot host PC.
- Make sure you are using latest graphics drivers on your PC.
- If you are using your own build of SDK, try using officially released version if that fixes the issue.

- If problem persists, collect logs and file feedback.

Cannot find microphone

- Check first that microphone array is enumerated in Device Manager.
- If a device is enumerated and works otherwise correctly in Windows, the issue may be that after firmware update Windows has assigned different container ID to Depth Camera.
- You can try to reset it by going to Device Manager, right-clicking on "Azure Kinect Microphone Array", and select "Uninstall device". Once that is complete, detach and reattach the sensor.



- After that restart Azure Kinect Viewer and try again.

Device Firmware update issues

- If correct version number is not reported after update, you may need to power cycle the device.
- If firmware update is interrupted, it may get into bad state and fail to enumerate. Detach and reattach the device and wait 60 seconds to see if it can recover. If not then perform a factory reset

Image quality issues

- Start Azure Kinect viewer and check positioning of the device for interference or if sensor is blocked or lens is dirty.
- Try different operating modes to narrow down if issue is happening in specific mode.
- For sharing image quality issues with the team you can:

1. Take pause view on Azure Kinect viewer and take a screenshot or
2. Take recording using Azure Kinect recorder, for example, `k4arecorder.exe -l 5 -r 5 output.mkv`

Inconsistent or unexpected device timestamps

Calling `k4a_device_set_color_control` can temporarily induce timing changes to the device that may take a few captures to stabilize. Avoid calling the API in the image capture loop to avoid resetting the internal timing calculation with each new image. Instead call the API before the starting the camera or just when needing to change the value within the image capture loop. In particular avoid calling `k4a_device_set_color_control(K4A_COLOR_CONTROL_AUTO_EXPOSURE_PRIORITY)`.

USB3 host controller compatibility

If the device is not enumerating under device manager, it may be because it's plugged into an unsupported USB3 controller.

For the Azure Kinect DK on **Windows, Intel**, **Texas Instruments (TI)**, and **Renesas** are the *only host controllers that are supported*. The Azure Kinect SDK on Windows platforms relies on a unified container ID, and it must span USB 2.0 and 3.0 devices so that the SDK can find the depth, color, and audio devices that are physically located on the same device. On Linux, more host controllers may be supported as that platform relies less on the container ID and more on device serial numbers.

The topic of USB host controllers gets even more complicated when a PC has more than one host controller installed. When host controllers are mixed, a user may experience issues where some ports work fine and other do not work at all. Depending on how the ports are wired to the case, you may see all front ports having issues with

the Azure Kinect

Windows: To find out what host controller you have open Device Manager

1. View -> Devices by Type
2. With the Azure Kinect connected select Cameras->Azure Kinect 4K Camera
3. View -> Devices by Connection

To better understand which USB port is connected on your PC, repeat these steps for each USB port as you connect Azure Kinect DK to different USB ports on the PC.

Depth camera auto powers down

The laser used by the depth camera to calculate image depth data, has a limited lifespan. To maximize the life of the lasers, the depth camera will detect when depth data is not being consumed. The depth camera power downs when the device is streaming for several minutes but the host PC is not reading the data. It also impacts Multi Device Synchronization where subordinate devices start up in a state where the depth camera is streaming and depth frames are actively help up waiting for the master device to start synchronizing captures. To avoid this problem in Multi Device capture scenarios, ensure the master device starts within a minute of the first subordinate being started.

Using Body Tracking SDK with Unreal

To use the Body Tracking SDK with Unreal, make sure you have added `<SDK Installation Path>\tools` to the environment variable `PATH` and copied `dnn_model_2_0.onnx` and `cudnn64_7.dll` to `Program Files/Epic Games/UE_4.23/Engine/Binaries/Win64`.

Next steps

More support information

Use Azure Kinect Sensor SDK to record file format

11/12/2019 • 2 minutes to read • Edit Online

To record sensor data, the Matroska (.mkv) container format is used, which allows for multiple tracks to be stored. using a wide range of codecs. The recording file contains tracks for storing Color, Depth, IR images, and IMU.

Low-level details of the .mkv container format can be found on the Matroska Website.

TRACK NAME	CODEC FORMAT
COLOR	Mode-Dependent (MJPEG, NV12, or YUY2)
DEPTH	b16g (16-bit Grayscale, Big-endian)
IR	b16g (16-bit Grayscale, Big-endian)
IMU	Custom structure, see IMU sample structure below.

Using third-party tools

Tools such as `ffmpeg` or the `mkvinfo` command from the MKVToolNix toolkit can be used to view and extract information from recording files.

For example, the following command will extract the depth track as a sequence of 16-bit PNGs to the same folder:

```
ffmpeg -i output.mkv -map 0:1 -vsync 0 depth%04d.png
```

The `-map 0:1` parameter will extract track index 1, which for most recordings will be depth. If the recording doesn't contain a color track, `-map 0:0` would be used.

The `-vsync 0` parameter forces ffmpeg to extract frames as-is instead of trying to match a framerate of 30 fps, 15 fps, or 5 fps.

IMU sample structure

If IMU data is extracted from the file without using the playback API, the data will be in binary form. The structure of the IMU data is below. All fields are little-endian.

FIELD	TYPE
Accelerometer Timestamp (µs)	uint64
Accelerometer Data (x, y, z)	float[3]
Gyroscope Timestamp (µs)	uint64
Gyroscope Data (x, y, z)	float[3]

Identifying tracks

It may be necessary to identify which track contains Color, Depth, IR, and so on. Identifying the tracks is needed when working with third-party tools to read a Matroska file. Track numbers vary based on the camera mode and set of enabled tracks. Tags are used to identify the meaning of each track.

The list of tags below are each attached to a specific Matroska element, and can be used to look up the corresponding track or attachment.

These tags are viewable with tools such as `ffmpeg` and `mkvinfo`. The full list of tags is listed on the Record and Playback page.

TAG NAME	TAG TARGET	TAG VALUE
K4A_COLOR_TRACK	Color Track	Matroska Track UID
K4A_DEPTH_TRACK	Depth Track	Matroska Track UID
K4A_IR_TRACK	IR Track	Matroska Track UID
K4A_IMU_TRACK	IMU Track	Matroska Track UID
K4A_CALIBRATION_FILE	Calibration Attachment	Attachment filename

Next steps

Record and Playback