DriveWorks

PR-08397-5.0
NVIDIA CONFIDENTIAL | Prepared and provided under NDA

0.6 Development Guide

TABLE OF CONTENTS

About This Guide

Welcome to the development guide for NVIDIA® DriveWorks. This guide provides instructional material for developing applications based on DriveWorks samples and API. For reference material, see the *DriveWorks API Reference*.

Basic Hardware Requirements

The DriveWorks SDK is distributed as a compressed file (DEB) and has specific hardware and software dependencies.

Platform Prerequisites

On the hardware side, you will need one of the following:

- PC (x86 architecture) with a NVIDIA GPU (Maxwell based GPU minimum, Pascal based GPU recommended) and/or
- NVIDIA DRIVE™ PX 2 with the latest SDK flashed in the system.

Host System Prerequisites

- Linux Desktop or Linux x86/x64 Only

These are the basic prerequisites for Linux. For the version information for each software component, see the Release Notes.

- Ubuntu Linux 16.04 or 14.04 (out of the box installation)
- GCC >= 4.8.X && GCC <= 4.9.x
- cmake version >= 3.3

Note:	**By default, Ubuntu 14.04 installs cmake version 2.8. If you are using that Ubuntu version, you must upgrade cmake to 3.3 or later.**

- NVIDIA® CUDA® Toolkit 9.0

You may also need to install (using `apt-get install`) the following packages:

- libx11-dev
- libxrandr-dev
- libxcursor-dev
- libxxf86vm-dev
- libxinerama-dev
- libxi-dev
- libglu1-mesa-dev
- libglew-dev

Getting Started

Installing DriveWorks

NVIDIA DriveInstall automatically installs on the target and host systems the following items:

- NVIDIA DRIVE ™ 5.0 SDK
- CUDA Toolkit
- cuDNN
- DriveWorks
- Libraries upon which the above items depend

If you have successfully run DriveInstall, you can skip this section.

For guidance on manually installing DriveWorks on the host or target system, see the README file located adjacent to the installation files.

Installing DriveWorks on the DRIVE PX Platform

To install the DriveWorks package on the DRIVE PX platform

1. On the Linux system, navigate to:

   ```
   /root/apt-repos/binary-DWx
   ```

2. On the platform, enter:

   ```
   $ sudo dpkg -i driveworks-v0.6.<release_info>-drive-t186ref-5.0.5.0.deb
   ```

   This installs DriveWorks into the following folder, which is the main installation folder on the target.

   ```
   /usr/local/driveworks-0.6/
   ```

Installing DriveWorks on the Linux Host

To install DriveWorks on the Linux host

1. Copy the following file to your Linux system:

   ```
   driveworks-v0.6.<release_info>-linux-amd64-ubuntu1404.deb
   ```

2. Enter:

   ```
   $ sudo dpkg -i driveworks-v0.6.<release_info>-linux-amd64-ubuntu1404.deb
   ```

   This installs DriveWorks into the following folder, which is the main installation folder on the target.

   ```
   /usr/local/driveworks-0.6/
   ```

Building the Samples

The samples installed with DriveWorks are already compiled; however, if you customize samples, you will need to cross-compile them on your Linux host system. For more information, see “DriveWorks Samples” in *DriveWorks SDK Reference*.

DriveWorks Introduction

DriveWorks is designed to achieve the full throughput limits of the computer. This requires careful architecture of the end-to-end software pipeline

- Efficiently utilize the many processors inside Tegra
- Optimize data communication formats between engines
- Minimize data copies (zero copy exchange of buffers)
- Create and utilize the most efficient algorithms
- Optimize implementations

DriveWorks is an API and SDK for Autonomous Driving:

- SDK, Runtime, Tools, Reference Applications, Library Modules
- Run-time Pipeline Framework
    - Modules and framework to create computational pipelines from Sensors through Perception

The DriveWorks design philosophy is modular, optimized, and flexible.

Sensor Abstraction Layer

DriveWorks SDK provides a sensor abstraction layer that supports easy capturing of data from various sources, and it has been designed with the following goals in mind:

- Provide a common and simple unified interface to the sensors
- Provide both HW sensor abstraction as well as virtual sensors (for replay)
- Provide raw sensor serialization (for recording)
- Deal with platform and SW particularities
  - API/Processor Conversion/transfer: CUDA, GL, NvMedia, CPU
  - Make use of the additional SoC engines: H264 encoder/decoders, VIC, etc.
- Current paradigm is non-blocking functions and blocking with timeout
- Defined by EGL, CUDA and NvMedia paradigms and capabilities
- Goal is event-driven and non-blocking data-flow model to be light-weight and efficiently:
  - Schedule work ahead to hide latencies on triggering work for all our HW engines.
  - Minimize your use of threads to increase runtime determinism of the system.

Supported Sensors

The following list shows the sensors that DriveWorks supports. For the most up-to-date information on supported sensors, see the *Release Notes*.

- GMSL Cameras (NVIDIA DRIVE™ platform only)
  - Omnivision OV10635
  - Omnivision OV10640
  - Sekonix AR0231 (RGGB, RCCB sensors)
- USB Camera
  - Any Video4Linux supported devices
- PointGrey USB Cameras
  - Chameleon CM3-U3-31S4C-CS (Color sensor)
  - Chameleon CM3-U3-50S5M-CS (Mono sensor)
  - Grasshopper GS3-U3-50S5C-C (Color sensor)
- Stereo Camera
  - ZED
- CAN Bus
  - SocketCAN
  - Aurix Easy CAN
- GPS
  - Any NMEA-compatible sensor using a serial UART

  - Xsens MTi-G-700 (serial based NMEA protocol + USB proprietary)
  - Garmin 18x
  - NovAtel dGPU
- Lidar
  - Quanergy M8
  - IBEO Lux
  - Velodyne (VPL16, HDL32E, HDL 64-S3)
- Radar
  - Continental ARS430
  - Delphi_ESR_2.5
- IMU
  - Xsens MTi-G-700 (serial based NMEA Protocol + USB proprietary)

Sensor Abstraction Layer Basics

Sensor Querying

DriveWorks context allows querying all the supported sensors for each HW platform as well as the parameters available to configure them. Please note that the query returns supported sensors by DriveWorks and not if a particular sensor is actually physically attached to the platform. Sensors are specified using a factory pattern such as:

```
sensor_type.mode param1=value1,param2=value2,...
```

where `sensor_type.mode` specifies the protocol and the following specifies the various parameters:

```
param1=value1,param2=value2,...
```

For example, the code snipped below lists all sensors supported on the NVIDIA DRIVE platform and Linux host system. It also lists the parameters for each sensor.

```
DW_PLATFORM_OS_V4L
   dwPlatformOS currentPlatform;
    dwPlatformOS platform[] = {DW_PLATFORM_OS_LINUX, DW_PLATFORM_OS_V4L};
    CHECK_DW_ERROR(dwSAL_getPlatform(&currentPlatform, hal));
    // get information about available sensors on each platform
    for (size_t i = 0; i < sizeof(platform) / sizeof(dwPlatformOS); i++) {
        const char *name = nullptr;
        CHECK_DW_ERROR(dwSAL_getPlatformInfo(&name, platform[i], hal));
        std::cout << "Platform: " << name;
        if (platform[tformOS currentPlatform;
    dwPlatformOS platform[] = {DW_PLATFORM_OS_LINUX, DW_PLATFORM_OS_V4L};
    CHECK_DW_ERROR(dwSAL_getPlatform(&currentPlatform, hal));
    // get information about available sensors on each platform
    for (size_t i = 0; i < sizeof(platform) / sizeof(dwPlatformOS); i++) {
        const char *name = nullptr;
        CHECK_DW_ERROR(dwSAL_getPlatformInfo(&name, platform[i], hal));
```

```
        std::cout << "Platform: " << name;
        if (platform[i] == currentPlatform)
            std::cout << " - CURRENT";
      std::cout << ": " << std::endl;
        uint32_t numSensors = 0;
        CHECK_DW_ERROR(dwSAL_getNumSensors(&numSensors, platform[i], hal));
        for (uint32_t j = 0; j < numSensors; j++) {
            const char *protocol = "";
            const char *params = "";
            CHECK_DW_ERROR(dwSAL_getSensorProtocol(&protocol, j, platform[i], hal));
```

In the above code:

- CHECK_DW_ERROR is a macro that throws an exception if an error occurs and
- DW_PLATFORM_OS_V4L is the NVIDIA DRIVE platform.

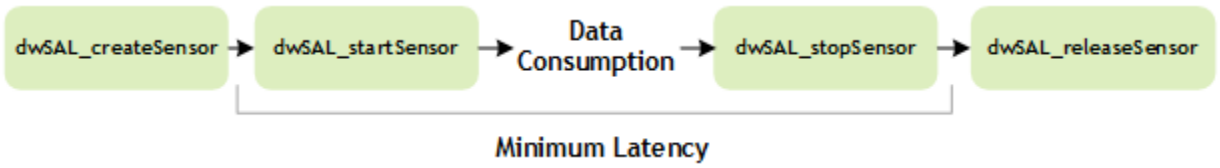
With the output being:

```
Platform: OS_LINUX:
  Sensor [0] : can.socket ? device=can0
  Sensor [1] : can.virtual ? file=/path/to/file.can
  Sensor [2] : camera.virtual ? video=filepath.h264[,timestamp=file.txt]
  Sensor [3] : camera.cpu ? device=0
  Sensor [4] : gps.uart ? device=/dev/ttyXXX[,
    baud={1200,2400,4800,9600,19200,38400,57600,115200}
    [,format=nmea0183]]
  Sensor [5] : gps.virtual ? file=filepath.gps
  Sensor [6] : imu.uart ? device=
    /dev/ttyXXX[,    baud={1200,2400,4800,9600,19200,38400,57600,115200}
    [,format=xsens_nmea]]
  Sensor [7] : imu.xsens ? frequency=100[,format=xsens_binary]]
  Sensor [8] : imu.virtual ? file=filepath.txt
  Sensor [9] : lidar.virtual ? file=filepath.bin
  Sensor [10] : lidar.socket ? ip=X.X.X.X,port=XXXX,
    device={QUAN_M81A, IBEO_LUX, VELO_VPL16, VELO_HDL32E, VELO_HDL64E},
    scan-frequency=XX.X
Platform: OS_DRIVE_V4L:
  Sensor [0] : can.socket ? device=can0
  Sensor [1] : can.aurix ? ip=10.0.0.1,bus={a,b,c,d,e,f}[,aport=50000,bport=60395]
  Sensor [2] : can.virtual ? file=/path/to/file.can
  Sensor [3] : camera.gmsl ? csi_port={ab,cd,ef},camera-count={1,2,3,4},
    camera-type={ov10635,c-ov10640-b1,ar0231,ar0231-rccb,ar0231-rccb-ssc,
    ar0231-rccb-bae,ar0231-rccb-ss3322,ar0231-rccb-ss3323},
    output-format={yuv,raw},output-image-attributes=0,slave=0
  Sensor [4] : camera.virtual ? video=
    filepath.h264[,timestamp=file.txt,start=0,length=-1]
  Sensor [5] : camera.cpu ? device=0
  Sensor [6] : gps.uart ? device=/dev/ttyXXX[,baud=
    {1200,2400,4800,9600,19200,38400,57600,115200}[,format=nmea0183]]
  Sensor [7] : gps.virtual ? file=filepath.gps
  Sensor [8] : imu.uart ? device=
    /dev/ttyXXX[,baud={1200,2400,4800,9600,19200,38400,57600,115200}
    [,format=xsens_nmea]]
  Sensor [9] : imu.xsens ? frequency=100[,format=xsens_binary]
  Sensor [10] : imu.virtual ? file=filepath.txt
  Sensor [11] : lidar.virtual ? file=filepath.bin
```

```
  Sensor [12] : lidar.socket ? ip=X.X.X.X,
    port=XXXX,device={QUAN_M81A, IBEO_LUX, VELO_VPL16, VELO_HDL32E, VELO_HDL64E},
    scan-frequency=XX.X
```

Sensor Life Cycle

Before you use any sensor in DriveWorks, you must create an instance of the sensor, then use a start-stop mechanism to collect data and finally release the sensor so all resources are freed.



The following functions support the life cycle shown above:

- `dwSAL_createSensor` is the function call that prepares the sensor for data delivery. This includes power up, establish connection, open channels, allocates FIFOs, etc... This function is expected to have a significant cost and should only be used during initialization. Sensor type and parameters are determined by using the above protocol and parameter strings.
- `dwSensor_start` is a low latency call that starts the capturing of sensor data.
- `dwSensor_stop` is a low latency call that will stop capturing data and will drain any data not consumed in preparation for the next start call.
- `dwSAL_releaseSensor` is a function call used to stop the sensor, disconnect it from DriveWorks SAL and release any allocated resources. It is expected to be high latency and should only be called at application termination.

Here is an example of initialization of a virtual camera for video replay and its release

```
// Initialize DriveWorks
dwInitialize(&sdk, DW_VERSION, &sdkParams);
// create SAL module of the SDK
dwSAL_initialize(&sal, sdk);
// create virtual camera interface
dwSensor cameraSensor = DW_NULL_HANDLE;
{
    dwSensorParams params;
    std::string parameterString = "file=/tmp/test.h264";
    params.protocol             = "camera.virtual";
    dwStatus result = dwSAL_createSensor(&cameraSensor, params, sal);
    if (result != DW_SUCCESS) {
        std::cerr << "Cannot create camera.virtual?"
                  << params.parameters << std::endl;
        exit(1);
    }
}
…
// release create SAL module of the SDK
dwSAL_releaseSensor(&cameraSensor);
// Release DriveWorks
dwRelease(&sdk);
```

Sensor Data Consumption

Once the sensor has started, it is possible to consume the data being acquired by using generic accessors or specialized accessors.

Generic accessors are used mainly for serialization purposes as they provide raw sensor data. The available function calls are `dwSensor_readRawData` to get access to the data memory pointers and `dwSensor_returnRawData` to return the pointers back to the sensor abstraction layer.

To access processed data one needs to use specialized function calls that will provide the data formatted appropriately depending on the sensor type:

Image sensors (cameras and virtual cameras)

The `dwSensorCamera_readFrame` function returns a handle to the last frame that the camera captured. A frame might contain a RAW or YUV frame. The `dwSensorCamera_getImageCUDA` or `dwSensorCamera_getImageCPU` functions extract the specified image type from the actual frame.

- dwSensorCamera_readFrame(…) followed by dwSensorCamera_getImageCUDA(…)
- dwSensorCamera_readFrame(…) followed by dwSensorCamera_getImageCPU(…)
- dwSensorCamera_returnFrame(…)

To start capturing the camera sensor at the framerate specified in the sensor properties, call `dwSensor_start`. To get the sensor properties, call `dwSensorCamera_getSensorProperties`.

A CSI frame drop occurs when sequential calls to `dwSensorCamera_readFrame` exceed the frame capture frequency.

Lidar sensors

- dwSensorLidar_readPacket(…)
- dwSensorLidar_returnPacket(…)

CAN, GPS, IMU sensors, and Radar,

These sensors do not have a return function because the messages are copied.

- dwSensorCAN_readMessage(…)
- dwSensorGPS_readFrame(…)
- dwSensorIMU_readFrame(...)
- dwSensorRadar_readPacket(…)

Sensor Data Timestamping

One of the critical aspects of data acquisition on systems with multiple sensors for real-time processing is to obtain accurate timestamping of the data. DriveWorks relies on the system time source of Tegra, in particular, the CLOCK_MONOTONIC time source. This time source adjusted to the UNIX epoch time is used internally to timestamp all sensors and events, and the application can get the current time via the function:

```
dwStatus dwContext_getCurrentTime(dwTime_t *time, dwContextHandle_t ctx);
```

Timestamps within the same context are guaranteed to be in sync. One thing to note is that this time source is relative to UNIX epoch time on Unix based platforms. Any NTP/PTP time corrections will influence this clock.

DriveWorks computes timestamps for each sensor when data arrives to Tegra, so sensor data returned always has a timestamp associated with it. In particular:

- GMSL Cameras: timestamping is done by the kernel driver triggered by the HW when full frame has been written into memory.
- SocketCAN: timestamping occurs by the kernel driver when the full message is captured from the HW by the driver. When HW timestamping is used, timestamping is performed at the HW level, by the TegraCAN hardware. The HW counters are synced to the SW clock.
- AurixCAN: messages through this interface are timestamped by the Aurix safety OS. gPTP must be running and Aurix+Tegra clock must be synced to make CAN messages timestamped by Aurix be in sync with DriveWorks timestamps.
- Serial port sensors (GPS): timestamping occurs when packets arrive to DriveWorks, at the SW level.
- Xsens IMU/GPS proprietary mode: clock of the Xsens devices are synchronized to DriveWorks clock, making timestamps to be in sync with DriveWorks timestamps.
- Lidar: Lidar generated point clouds provide a DriveWorks timestamp (taken when the Ethernet packet was read), the sensor provided timestamp at start of capturing the packet, and the duration of the packet capture. If the particular sensor does not provide timestamps, both host and sensor timestamps are identical.

Sensor Sharing

DriveWorks API supports sharing of the same HW sensor by multiple DriveWorks abstractions. For example, same GPS sensor can be created twice from the same hardware implementation (e.g. Xsens device or serial connection). Each DriveWorks sensor will have its own internal FIFO to keep the data ready for consumption. Sensor data of the shared sensors will be equal and can be matched by equal timestamps if required. The sample `sample_gps_logger` shows, in particular, how two sensors can be created from the same HW GPS sensor. For more information, see “GPS Location Logger Sample” in *DriveWorks SDK Reference.*

Point Grey Sensors

The following are prerequisites and information helpful in getting started with supported PointGrey USB cameras:

- Download the appropriate version of FlyCapture2 SDK from:

  https://www.ptgrey.com/support/

  Point Grey provides the following versions:

  - arm64 for Ubuntu 16.04
  - arm64 for Ubuntu 14.04
  - Linux Ubuntu 16.04
  - Linux Ubuntu 14.04
- Extract the package and perform the following steps:

  ```
  cd flycapture.<version>_<architecture>
  ```

```
sudo sh flycap2-conf
cd lib
sudo cp flycapture* /usr/lib/
cd C
sudo cp * /usr/lib
```

These commands configure the udev and copy the required shared library files to `/usr/lib`.

- Add the following lines to `/etc/sysctl.conf`:
  - net.core.rmem_default = 1048576
  - net.core.rmem_max = 10485760
  - net.core.wmem_default = 1048576
  - net.core.wmem_max = 10485760
  - net.core.netdev_max_backlog = 30000
  - net.ipv4.ipfrag_high_thresh = 8388608
  - net.ipv6.conf.all.disable_ipv6 = 1
  - vm.dirty_background_ratio = 5
  - vm.dirty_ratio = 80

This maximizes network and USB performance. These are persistent changes and are preserved between reboots.

- On every boot, the following must be executed:

```
sudo -s
echo 1000 > /sys/module/usbcore/parameters/usbfs_memory_mb
```

- Supported families and output formats: Currently only Chameleon and Grasshopper families of PointGrey cameras are supported. Given below is the list of verified cameras along with their sensor and output information:
  - CM3-U3-31S4C-CS: Color sensor; outputs is RGB.
  - CM3-U3-50S5M-CS: Mono sensor, output is grayscale.
  - GS3-U3-50S5C-C: Color sensor; output is RGB.
- USB 3.0 port settings: PointGrey USB cameras must be connected to Intel USB 3.0 controllers only. Connecting to any other USB port results in undefined behavior.
- USB 3.0 connector cable: Only PointGrey USB 3.0 connector cables must be used.
- Output image dimensions: Currently fixed to the largest supported image dimensions.

Sensor Timeout

The camera samples call the `dwSensorCamera_readFrame` function with a standard timeout value. If one of those samples fails with a DW_TIME_OUT status, you should increase the timeout value to be suitable for your camera.

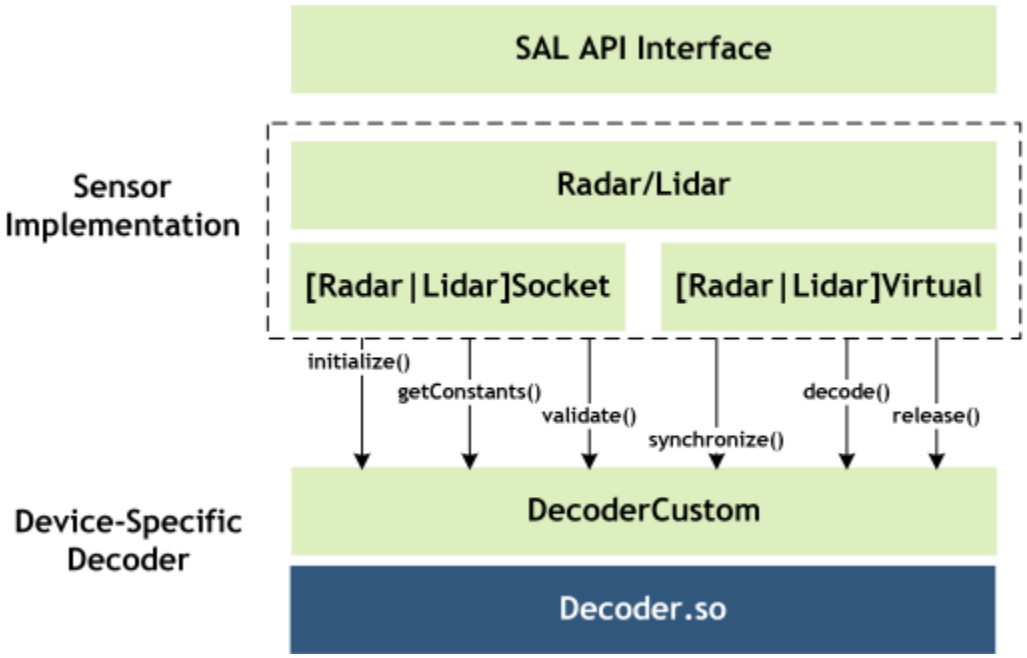
Integrating with Custom Sensors

Radar and Lidar Decoder Plugins

The DriveWorks sensor abstraction layer (SAL) implements radar and Lidar sensor functionality into two modular layers:

- Implementation—not pluggable. DriveWorks provides implementations for the supported sensor interfaces (e.g., Ethernet Radar).
- Device-specific decoder—handles parsing and interpretation of raw data.

To support customer radar and Lidar devices, DriveWorks provides plug-in interfaces for custom radar and Lidar decoders. The following diagram illustrates this architecture. The socket interface is for real data, and the virtual interface is for data recorded with the socket interface.



In the above illustration, functions such as `initialize()` correspond to the header functions `_dwStatus _dwRadarDecoder_initialize()` or `_dwLidarDecoder_initialize()`. The underscore (_) in the declaration names identify items that you must implement. DriveWorks uses the information in the JSON file to determine how to interact with your implementation.

The decoder header files for the radar and Lidar plugins are found at:

```
dw/sensors/plugins/radar/RadarDecoder.h
dw/sensors/plugins/lidar/LidarDecoder.h
```

For information about the plugin declarations, see *DriveWorks SDK Reference*.

You must implement the interfaces outlined in the header files below, and compile their plugin as a shared object file (.so).

After compiling your custom plugin, you must represent the Lidar/radar in a JSON file such as `recorder-config.json` or `recorder-qt-config.json`.

- Radar socket

```
"protocol": "radar.socket"
"params": "ip=X.X.X.X,
           port=XXXX,device=CUSTOM,
           decoder=<path_to_the_decode.so>",
```

- Lidar socket

```
"protocol": "lidar.socket"
"params": "device=CUSTOM,ip=X.X.X.X,
           port=XXXX,decoder=<path_to_the_decode.so>",
```

- Radar virtual

```
"protocol": "radar.virtual"
"params": "device=CUSTOM,
           decoder=<path_to_the_decode.so>",
```

- Lidar virtual

```
"protocol": "lidar.virtual"
"params": "device=CUSTOM,
           decoder=<path_to_the_decode.so>",
```

For information on the following, see Data Acquisition in this guide:

- Recording radar or Lidar data
- Consuming recorded radar or Lidar data

Limitations

The plugin interfaces currently support the following protocols:

- `radar.socket`
- `radar.virtual` (for files recorded with this decoder)
- `lidar.socket`
- `lidar.virtual` (for files recorded with this decoder)

Other Sensors

DriveWorks does not constrain the integration of any sensors. Any third-party sensor can be used in the DriveWorks modules, if their output is mapped to the sensor data API representation, such as: `dwCANMessage`, `dwGPSFrame`, `dwIMUFrame`, `dwLidarDecodedPacket`, `dwImageCUDA`, `dwImageGL`, `dwImageCPU`, and `dwImageNvMedia`. Use `dwContext_getCurrentTime` to time-stamp individual sensors.

Integrating with a Custom Board

Note:	**This feature is beta.**

The camera.gmsl sensor creator supports an additional parameter that you can use to specify a custom board. You describe the board with a JSON file that you pass with a statement such as `custom-board-json=/path/to/custom.json`. For example:

```
dwSensor cameraSensor = DW_NULL_HANDLE;
{
    dwSensorParams params;
    params.protocol    = "camera.gmsl";
    params.parameters  = "camera-type=ar0231-rccb, \
        custom-board-json=/path/to/custom_board.json";
    dwStatus result = dwSAL_createSensor(&cameraSensor, params, sal);
}
```

The JSON file contains definitions for the `ExtImgDev` structs to overwrite the hardcoded one in DriveWorks. If you use the API, you must create `camera.gmsl` sensor with a correct camera-type that best matches your camera, for example `ar0231-rccb-ss3322`, however the JSON file can overwrite default settings. A JSON file for `ar0231-rccb-ss3322` on DRIVE PX 2-TegraA looks like this:

```
{
  "ExtImgDevParam" : {
    "moduleName" : "ref_max9286_96705_ar0231rccbss3322",
    "resolution" : "1920x1208",
    "inputFormat" : "raw12",
    "interface" : "csi-ab",
    "i2cDevice" : 7,
    "desAddr" : 72,
    "brdcstSerAddr" : 64,
    "serAddr" : [0, 0, 0, 0],
    "brdcstSensorAddr" : 16,
    "sensorAddr" : [0, 0, 0, 0],
    "slave" : false,
    "enableEmbLines" : true,
    "reqFrameRate" : 30
  }
}
```

Recording and Replaying Sensors

Recording Sensors

See Data Acquisition in this guide.

Replaying Sensors

To replay sensor data from the DriveWorks recording, users can use the same APIs introduced above for creating a sensor object as virtual sensor for replaying. In general, the change is in:

```
dwSensorParams params
```

of sensor creation. Change the protocol part to the corresponding `sensorType.virtual` and the parameters part to file locations. Each sensor type's API definition has more details.

With the correct sensor parameter setting for recorded data as virtual sensor, call:

```
dwSAL_createSensor() and dwSensor_start()
```

to create and start the sensor object. Then, the following usage is the same as real-time sensors:

```
dwSensorType_readFrame()
```

Call the correct sensor APIs to get data frames with timestamps at recording time. Also, some sensor types require:

```
dwSensorType_returnFrame()
```

for returning the data frame to sensor object, for reusing the allocation memory.

To easily replay sensor data:

1. Try the default compiled samples with predefined sensor input data
2. Change the input parameter of the samples to the data you want to replay. The data should be collected by the recording tools in DriveWorks. (Each sensor sample has a detailed README for usage and explanation)
3. Read the sample codes for how to use DriveWorks APIs to replay data as virtual sensors. Read the API definition of each sensor type for details
4. Start to use DriveWorks APIs in the targeted application and platforms

Image Pipeline

Image Data Structures

Images are represented in DriveWorks with specialized structs, one for each supported API. The supported APIs are CPU, CUDA, OpenGL, and NvMedia. There are properties that are common to the images of all API types and some that are specific to an API.

The `dwImageProperties` structure summarizes properties common to all images. It contains this information:

- Type of the image, for example `DW_IMAGE_CPU`.
- Width and height.
- Pixel format which describes the format in which the image data is represented. For example, for an image with red, green, blue, and alpha channels, the format is `DW_IMAGE_RGBA`.
- Pixel type which describes the type of each pixel. For example, for an 8-bit unsigned integer, the type is `DW_TYPE_UINT8`.

The timestamp property is a common property for all images. The timestamp specifies the point in time when the Tegra processor captured the image.

Note:	**Due to technical limitations of the NVIDIA DRIVE PX 2 platform, the point in time when the Tegra processor captures the image may be slightly later than the time when the camera takes the picture.**

Storage in Memory

Images can be stored in memory in various formats. One dimension of this variation is interleaved vs planar storage for multi-channel images. For pitch-linear data storage, interleaved vs planar is defined by a combination of the image pixel format and the planeCount. For example, an interleaved RGB image has `planeCount` set to 1. A YUV420, which is typically represented planar, has `planeCount` set to 3.

Another dimension of memory representation that only applies for CUDA images only is that of pitch-linear vs block-linear memory. `dwImageCUDA` has an additional field layout that specifies what memory representation is used. Depending on the layout, either `array[]` for block-linear or `dptr[]` for pitch-linear memory holds the data.

Format Conversion and Image Streaming

DriveWorks provides a `dwImageFormatConvert` module that can be used to convert the format of an image from one type to another, e.g. from YUV to RGB. Another module that is useful are the `dwImageStreamer` functions that bridge API as well as thread and process boundaries.

Image Streamers

Image streamers are means to transform images from one API to another and/or from one process to another. The supported APIs are CPU, CUDA, OpenGL and NvMedia. The implementation of the image streamer uses mechanisms that allow mapping rather than copying to optimize latency and performance. This is achieved for

example through EGL streams. Further, the image streamer is designed to recycle data. That means images are allocated once at application init time and are recycled to improve both safety as well as performance.

Image streamers are setup for exactly one pair of input and output APIs as well as image formats (see `dwImageStreamer_initialize`). If streamers for multiple types or APIs are needed multiple streamers must be initialized.

Once a streamer has been created it is ready to take input and provide output. The flow is adapted from that of EGL streams. In particular, the producer provides one or multiple input images and posts them via the `dwImageStreamer_postAPI` functions. Once the image has been posted the producer must not access the image until it is returned. See further details later in this section.

The consumer can check for posted images via the `dwImageStreamer_receiveAPI` function. This function has a timeout argument which allows for the consumer to use this function to synchronize in a block or non-blocking way (e.g. timeout of few microseconds) with the producer. Once the consumer is done working with the image, it must return the image back to the producer to be recycled. This is achieved by the function `dwImageStreamer_returnReceivedAPI`.

Finally, the producer does check for returned frames from the consumer by the `dwImageStreamer_waitPostedAPI`. This function again has a timeout argument to control blocking or non-blocking data flow. Once the image is returned from the image streamer to the producer, the producer can recycle the image, e.g. write new image information into it and post it again to the consumer.

Note:	**By default, EGL processors always keep one frame available for the consumer. This means that the producer does not get the first image posted back directly after the consumer returned it. The first image is available via the waitPosted API, only after a second image has been posted by the producer.**

Image streamers are thread-safe and can be used in a multi-threaded and multi-process environment.

```
[Producer]
imagePool pool;
while(running) {
  // Check for returned images
  Image returnedImage;
  if( dwImageStreamer_waitPosted(&returnedImage, 50 (usecs)) == DW_STATUS_OK ) {
    put returnedImage back into pool;
  }
  // Post new image
  get image from pool;
  fill image with content;
  dwImageStreamer_postAPI(image)
}
[Consumer]
while(running) {
  // Check for new images
  Image* image;
  If(dwImageStreamer_receiveAPI(&image, 100000 (usecs) != DW_STATUS_OK)
  {
    // No new data available
    continue;
  }
  // Consume and return
  consume image;
```

```
  dwImageStreamer_returnReceivedAPI(image)
}
```

Table of Supported Image Streamer Inputs and Outputs

From (column) \ To (row)	CPU	GL	CUDA	NvMedia
CPU	-	X*	X*	X
GL	X*	-	X	X
CUDA	X	X*	-	X
NvMedia	X	X	X	X (Ideal for cross processing)
Note				
* Supported on iGPU only.				

Table of Streamable Image Formats and Types (DW_IMAGE_X and DW_TYPE_X)

From (column) \ To (row)	CPU	GL	CUDA	NvMedia
CPU	-	RGBA, R, UINT8	ALL	RGBA, R, YUV420 p/s, YUV422 p/s, RAW, UINT8, UINT16
GL	RGBA, UINT8	-	RGBA, UINT8	RGBA, UINT8
CUDA	ALL	RGBA, UINT8	ALL	RGBA, YUV420 p/s, YUV422 p/s, UINT8
NvMedia	RGBA, YUV420 p/s, YUV422 p/s, UINT8	RGBA, UINT8	RGBA, YUV420 p/s, YUV422 p/s, UINT8	RGBA, YUV420 p/p, YUV422 p/p, RAW, UINT8, UINT16

Format Converters

A format converter copies an image with certain properties to an image with different ones. The typical use case is to convert a YUV image as captured by the cameras into an RGBA image for display with OpenGL. Note that

they do not convert the API type of an image but only the pixel format, type, and layout. For details see the table below.

Currently the image converter has an implementation in CUDA and NvMedia. CUDA is available on all platforms and utilizes the GPU to do the conversion. See below table for detail of the currently supported conversions. On the Drive PX platform additionally an NvMedia version is available. Through NvMedia a dedicated HW block on the Tegra processor, the VIC engine, is used to perform the conversion. For the list of the supported conversions check the NvMedia documentation for NvMedia2DBlit that is shipped with the SDK.

Table for supported formats of the CUDA format converter

Specifics	From	To
-	any property and layout	same property and layout (simple copy)
any property	layout DW_IMAGE_CUDA_PITCH	layout DW_IMAGE_CUDA_BLOCK
any property	layout DW_IMAGE_CUDA_BLOCK	layout DW_IMAGE_CUDA_PITCH
type uint8, input planeCount 2/3	DW_IMAGE_YUV420	DW_IMAGE_RGBA
type uint8, output planeCount 3	DW_IMAGE_RGBA	DW_IMAGE_YUV420
type uint8, input planeCount 2/3, output planeCount 1/3 input type uint8, output type float16, input planeCount ⅔, output planeCount 3	DW_IMAGE_YUV420	DW_IMAGE_RGB
type uint8, input planeCount 1/3, output planeCount 3	DW_IMAGE_RGB	DW_IMAGE_YUV420
any type, input planeCount 1/3	DW_IMAGE_RGB	DW_IMAGE_RGBA
any type, output planeCount 1/3	DW_IMAGE_RGBA	DW_IMAGE_RGB
input type float16, output type uint8 (tone mapping)	DW_IMAGE_RGBA	DW_IMAGE_RGBA
input planeCount 2, output planeCount 3	DW_IMAGE_YUV420	DW_IMAGE_YUV420
type uint8	DW_IMAGE_RGBA	DW_IMAGE_R

DriveWorks Conventions

In this section describe all the standards and conventions followed

Coordinate Systems

The car and each sensor have right-handed coordinate systems attached to them. The following sections describe the conventions for each coordinate system.

Car Coordinate System

The origin of the car coordinate system, also called the *rig* coordinate system, is under the center of the rear axle and on the ground (at calibration time). The x-axis points forward, to the front of the car. The y-axis points to the left of the car and the z-axis points up.

Car Coordinate System

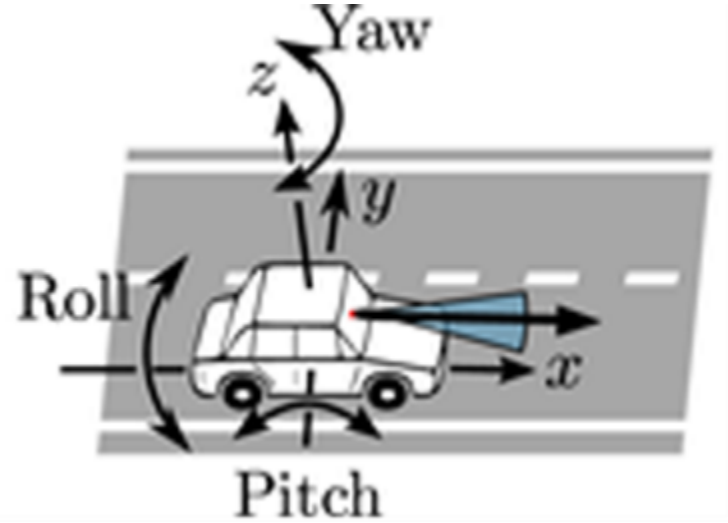


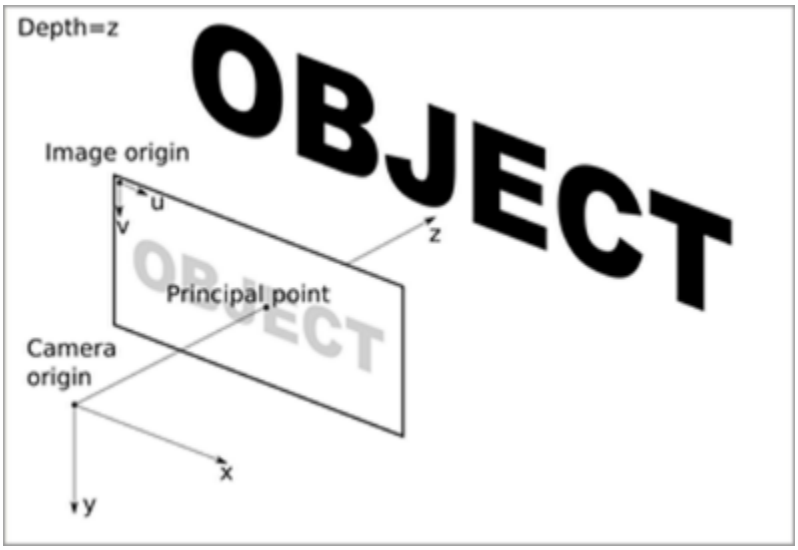
Image and Camera Coordinate Systems

The image coordinate system describes the position of points in the image space; and the camera coordinate system describes the corresponding point in 3D space. The intrinsics of the camera are used to transform a point between the two coordinate systems

The image coordinate system originates at the top-left of the image. The u- and v-axes, in pixel units, are aligned with the way pixels are stored in memory.

The camera coordinate system has its origin at the optical center of the camera. The x-axis points to the right of the image plane and the y-axis points to the bottom of the image plane. The z-axis points forward, along the optical axis. The x- and y-axes point in the same direction as the image u- and v-axes, respectively. The camera coordinate axes are in metric units.

Image and Camera Coordinate Systems



Radar coordinate system

The radar coordinate system is centered at the radar's geometric center and the system axis follow the AUTOSAR and ISO-8855 standard. The x-axis points forward, to the front of the car. The y-axis points to the left of the car and the z-axis points up.

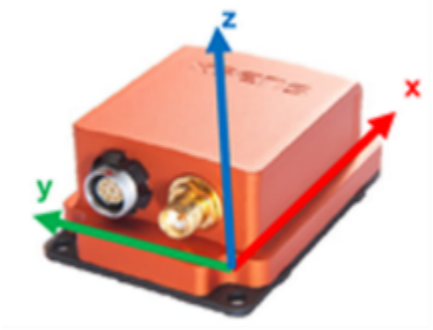
LIDAR Coordinate System

The LIDAR manufacturer defines the LIDAR coordinate system.

IMU Coordinate System

The x-axis points forward, to the front of the car. The y-axis points to the left of the car, and the z-axis points up. The origin is located inside the case of the sensor, at the accelerometer.

IMU Coordinate System



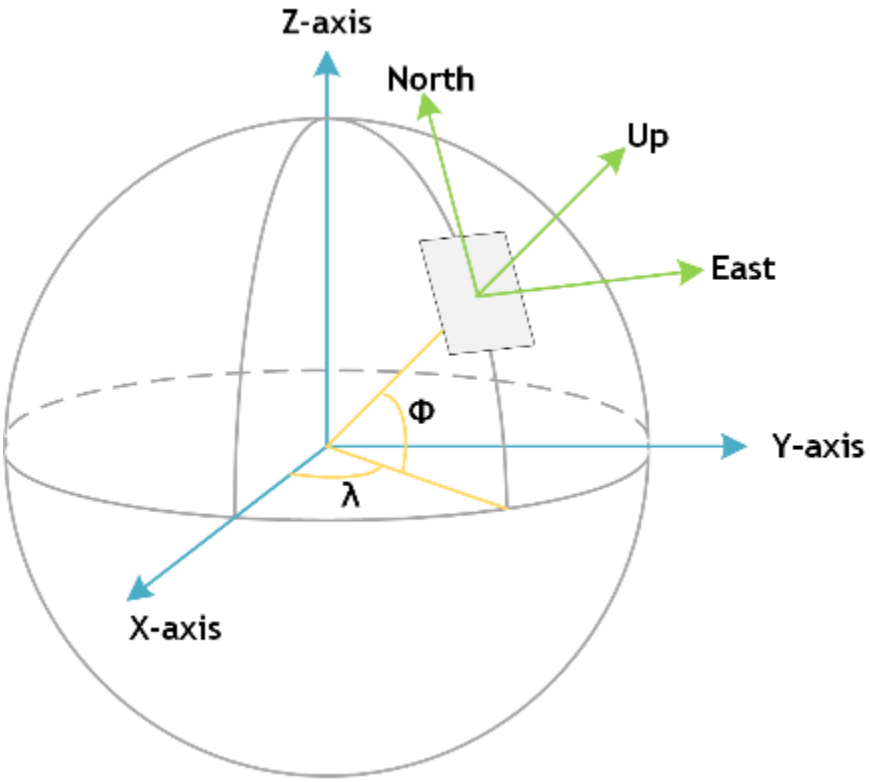
GPS and HD Maps Coordinate Systems

GPS sensors and HD maps usually describe positions in WGS84 coordinates (latitude, longitude, height).

To represent points on the earth surface in a Cartesian coordinate system, ENU (east-north-up) coordinates can be used. The origin is defined at an arbitrary point on the Earth surface. x, y and z axes defined the tangent plane at that origin point, such that (x,y,z) corresponds to (east, north, up).

The DriveWorks maps module provides helper functions to transform into local Cartesian coordinates.

WGS84 (λ,φ) and ENU Coordinate Systems



Modules

This section introduces the NVIDIA® DriveWorks modules, including details on how they work and how they must be used.

Image Processing Modules

Camera Color Correction

There are multiple cameras in the system and their captured images are likely to have different color tendencies. The color correction module uses one camera as a master camera and adjusts the remaining images according to the master camera's statistical data. The color matching method assumes that ground areas of each camera have similar color distributions.

The workflow of color correction module is:

1. Create the handle with rig.xml file containing camera's calibration data, so that the module can extract ground area from each camera's view.

```
dwRigConfigurationHandle_t rigConfig = DW_NULL_HANDLE;
dwRigConfiguration_initializeFromFile(&rigConfig, sdk, "rig.xml");
dwColorCorrectHandle_t cc = DW_NULL_HANDLE;
dwColorCorrectParameters ccParams{};
```

```
ccParams.cameraWidth = g_imageWidth;
ccParams.cameraHeight = g_imageHeight;
dwColorCorrect_initializeFromRig(&cc, sdk, rigConfig, &ccParams);
```

2. Send master camera's image to the handle to gather its statistic data

```
dwColorCorrect_setReferenceCameraView(frameCUDAyuv, cameraIdx, cc);
```

3. Send the rest camera's images to the handle, it will calculate their statistic data and match them to the master one.

```
dwColorCorrect_correctByReferenceView(frameCUDAyuv, cameraIdx, 0.8f, cc);
```

The third argument factor in the following controls how much the master camera will affect the current view. 1.f means 100% dependency on the master camera's color distribution; 0.f means no correction and use current camera's own color. Default value is 0.8f.

```
dwColorCorrect_correctByReferenceView(dwImageCUDA* pImage, uint32_t curCameraIdx, \
    float32_t factor, dwColorCorrectHandle_t obj)
```

The input image must be YUV420 `dwImageCUDA`, which is the default camera image format on a non-NVIDIA DRIVE™ system. You can use `dwImageStreamer` to convert `dwImageNvMedia` from NVIDIA DRIVE to `dwImageCUDA` easily.

Test sample can be run directly by ./sample_color_correction, the other arguments include:

```
--video1
--video2
--video3
--video4        The 4 video inputs
--rigFile       Rig file contains camera calibration data to extract ground area
--ref           Master camera index
--factor        Correction factor, 0 means no correction, 1 mean full correction
```

For more information on this sample, see *DriveWorks SDK Reference*.

Video Rectification

The video rectification sample illustrates how to use the rectification module to remove fisheye distortion from a video (undistortion). The main purpose of the module is to convert an image acquired with an input camera model by projecting it into an output camera module. In the sample the input camera model is an OmniCam with lense distortion, projected into a pinhole camera with no lense distortion.

The sample reads frames from a video input recorded from an omnicamera and calibration from the `rig.xml` file. It then performs rectification and displays both the original and rectified video side-by-side.

```
./sample_rectifier
```

To play a custom video and with a corresponding rig calibration file, the options `--video` and `--rig` can be used:

```
./sample_rectifier --video=<video file.h264> --rig=<rig.xml>
```

To use a rectifier object (`dwRectifierHandle_t`), initialize the sample with the input and output camera models as `dwCalibratedCameraHandle_t` to `dwRectifier_initialize`(). Then, given an input image, which is a pointer to `dwImageCUDA`, the function `dwRectifier_warp`() returns a pointer to `outputImage` and `dwImageCUDA`.

In the sample, the input camera model is created by the RigConfiguration object, by calling `dwCameraRig_initializeFromConfig`(). On the other hand, the output camera model must be created as a pinhole camera (in order to remove fisheye) with no distortion. In the following example, the focal length was chosen manually for best result at ¼ of the image resolution.

```
dwPinholeCameraConfig cameraConf = {}
cameraConf.distortion[0] = 0.f;
cameraConf.distortion[1] = 0.f;
cameraConf.distortion[2] = 0.f;
cameraConf.focalX = FOC_X;
cameraConf.focalY = FOC_Y;
cameraConf.u0 = static_cast<float32_t>(gCameraWidth/2);
cameraConf.v0 = static_cast<float32_t>(gCameraHeight/2);
cameraConf.width = gCameraWidth;
cameraConf.height = gCameraHeight;
```

To remove lense distortion from an image acquired with a pinhole camera, the rectifier must project from Pinhole to Pinhole with no distortion. In that case, the intrinsics of the camera must be known.

It is also possible to set an homography transformation by calling `dwRectifier_setHomography()`.

Image Signal Processor (ISP)

The software ISP converts RAW images from a specific camera into images with linear response curve.

Note:	**The above picture is from DriveWorks 0.2.1. Beginning with version 0.3, DriveWorks has improved adaptive tone mapping.**

The output format can be:

- Bayer pattern image,
- Demosaiced image,
- Tone-mapped image, or
- All the above.

Bayer image means a linear response image that has the same position varying color filter as the camera physical sensor. The demosaiced image is the converted Bayer image result, where each pixel has all color channels available (Red, Clear, and Blue), either via interpolation or downsampling. The tonemapped image finally scales the range of intensity value to produce a regular RedGreenBlue image. For optimization purposes, if only a subregion of the image is of interest, demosaicing can be run on that sub-region only.

The state of the software ISP pipeline controls the demosaic output. Use the following methods to control the pipeline state:

- dwSoftISP_setDemosaicMethod
- dwSoftISP_setDemosaicROI

Note:	**The Bayer image is always the full resolution and not affected by the demosaic state.**

Currently, the pipeline only supports RAW images from the ar0231 RCCB/RCCC cameras.

Maps Module

The maps module provides an API to access HD maps data from different backends.

DriveWorks currently supports:

- HERE maps
  - Cache file
  - Download with HERE maps connection parameters
- TomTom XML file

Data Format

The backend specific map data is translated into a common DriveWorks format. The main data structs holding the map data are

- Road Segments
- Lanes
- Lane Dividers
- Features

The basic elements are the Road Segments, all data can be accessed through them. A Road Segment represents a piece of road, containing several Lanes, Lane Dividers, and Features. Element geometries are represented as polylines of points in WGS84 coordinates. For Lanes, a polyline represents the centerline of the Lane. For Lane Dividers and Features, the geometry describes the shape of the object, for example a polyline modelling the painted marking on the road.

Features are a generic representation of objects encountered along the road, e.g. traffic signs, traffic lights, etc. A feature is described by its type and geometry.

Lane Dividers are grouped into Lane Divider Groups. Although usually there is only a single Lane Divider at a lane boundary, there are cases where lanes are separated with multiple dividers, for example a painted lane marking and a physical barrier.

A Lane has pointers to two Lane Divider Groups, representing the lane boundaries on both sides of the Lane. The Lane Divider Groups are part of the Road Segment, two neighboring Lanes may point to the same Lane Divider Group.

Connections

Road Segments and Lanes have connections to their predecessors and successors along the road, represented by the `dwMapsRoadSegmentConnection` and `dwMapsLaneConnection` structs. Connections can be on two sides of the element. `previous` connections are at the beginning of the geometry polylines, whereas next connections connect at the end. It is possible that connected elements have opposite geometry directions, meaning for a `next` connection, the end of a polyline connects to the end of the connected polyline (and similarly on the start side for `previous` connections). A `sameDirection` Boolean in the `Connection` struct stores the directions of connected elements relative to each other. A connection is represented by the ID of the connected element, and, if available, a pointer to it. If the connected element is currently not in memory, the pointer is a `nullptr`.

- Lanes, Lane Dividers, Lane Divider Groups, and Features also have pointers to their parent structs, as shown below:
- dwMapsLane -> dwMapsRoadSegment
- dwMapsLaneDivider -> dwMapsLaneDividerGroup
- dwMapsLaneDividerGroup -> dwMapsRoadSegment
- dwMapsFeature -> dwMapsRoadSegment

Attributes

Road Segments, Lanes, Lane Dividers, and Features store various attributes to describe their type and properties.

Main Data Structure	Type and Properties
Road Segment	Type (bridge, tunnel, etc.)
Lane	Type (shoulder, car pool, etc.) Driving direction, relative to the geometry polyline
Lane Divider	Type (dashed, solid, etc.) Material Color
Feature	Type (traffic sign, traffic light, etc.)

Road Segments are defined such that attributes within a Road Segment do not change. If an attribute of an element (Road Segment, Lane, or Lane Divider) changes along the road, a new Road Segment is created.

Map Initialization

There is an initialize function for each supported backend:

```
dwMaps_initializeHERE
dwMaps_initializeTomTom
```

The initializer reads existing map caches or xml files, translates the data into the internal DriveWorks format, and loads it into the system memory. The initializer returns a map handle that can be used to access the data through the query functions of the maps module.

The data buffers must be pre-allocated to a sufficient size at initialization. The number of expected Road Segments

defines the size of the buffers and must be estimated. The Road Segments are either created during initialization or they are created during the `dwMaps_update` call. As a result, the number of created Road Segments depends on:

- Content of the loaded data files and
- Size and content of the bounds during the update call.

There is also an initialize function to read serialized map data in DriveWorks format from file:

- dwMaps_initialize

Local Data Update

It is possible to update local cache files with map data of a map region defined by a bounding box of longitude and latitude coordinates:

```
dwStatus dwMaps_update(dwMapsBounds bounds, \
  dwBool download, \
  dwMapHandle_t mapHandle);
```

Currently only the HERE maps backend implements this function. If the download Boolean is set, and HERE maps connection parameters are provided during initialization, this function will download HERE maps data from the HERE servers and updates the local HERE maps cache file (named `HDMapCache.db` by default). If 'download' is false, the current local HERE maps cache file is used to update the DriveWorks map data in memory.

For the given bounds, the map data of the backend is translated to the DriveWorks format. A binary DriveWorks map cache file with the translated data can be stored using the serialization API. This binary file can be used at the next DriveWorks initialization time and will allow efficient loading of the data into memory.

Serialization

With the following methods, the map data in the Maps module can be serialized to and deserialized from a file in binary or xml format:

```
dwStatus dwMaps_serialize(
    const char *filename,
    dwMapsSerializationFormat format,
    dwConstMapHandle_t mapHandle);


dwStatus dwMaps_deserialize(
    const char *filename,
    dwMapHandle_t mapHandle);
```

To avoid dynamic memory allocations, the buffer sizes in the existing map handle do not change at deserialization. If the data from the file does not fit, only part of it is deserialized and an error is logged.

It is also possible to directly initialize a map handle from a DriveWorks data file:

```
dwStatus dwMaps_initialize(
    dwMapHandle_t *mapHandle,
    const char *filename,
```

```
    dwContextHandle_t contextHandle)
```

In that case, the internal buffers are sized to exactly fit the data from the file.

Map Query

Result Buffers

Maps module functions that return an array of elements write the returned elements into a fixed size buffer that is passed in as an argument. There are Buffer structs for various map elements:

```
dwMapsGeoPointBuffer
dwMapsPointBuffer
dwMapsPolyline3fBuffer
dwMapsLineBuffer
dwMapsLaneDividerLineBuffer
dwMapsFeatureBuffer
dwMapsLaneDividerBuffer
dwMapsRoadSegmentBuffer
```

A Buffer struct must be initialized before being passed to the function:

- buffer: pointer to the fixed size array
- maxSize: size of the fixed size array
- size: current number of valid elements in the array

Query functions

Road Segments

Road Segments provide access to all the map data.

The `getRoadSegments` function returns all Road Segments that overlap with a bounding box of longitude and latitude coordinates.

```
dwStatus dwMaps_getRoadSegments(
    dwMapsRoadSegmentBuffer *roadSegments,
    const dwMapsBounds *bounds,
    dwConstMapHandle_t mapHandle);
```

For convenience, there are a few more specific query functions:

```
getLaneDividers
getFeatures
```

Lane Dividers

```
dwStatus dwMaps_getLaneDividers(
```

```
    dwMapsLaneDividerBuffer *laneDividers,
    uint32_t typeFilter,
    const dwMapsBounds *bounds,
    dwConstMapHandle_t mapHandle);
```

`getLaneDividers` returns all Lane Dividers that overlap with a bounding box of longitude and latitude coordinates, filtered by their type. A logical combination of type flags can be used to specify which Lane Dividers are returned.

Features

```
dwStatus dwMaps_getFeatures(dwMapsFeatureBuffer *features,
                            uint32_t typeFilter,
                            const dwMapsBounds *bounds,
                            dwConstMapHandle_t mapHandle);
```

`getFeatures` works the same way as `getLaneDividers`.

Closest Lane

```
dwStatus dwMaps_getClosestLane(const dwMapsLane **closestLane,
                               dwMapsGeoPoint *closestPoint,
                               const dwMapsGeoPoint *p,
                               dwBool onlyDrivable,
                               dwBool ignoreHeight,
                               dwConstMapHandle_t mapHandle);
```

`getClosestLane` returns the closest Lane, and the closest point on it, to a given query point `p`. When the `onlyDrivable` flag is set to true, non-drivable lanes are not considered. `ignoreHeight` can be set if the height of the input point is unknown or inaccurate. In that case, the distances are compared, after projecting all points to height 0.

Note that the maps module also provides a map tracker that allows a robust selection of the current lane when driving along a road. It also incorporates orientation and logical connections and thus can give a better result than just picking the closest lane.

Map Tracker

The Map Tracker tracks a trajectory on the map and selects the lane that corresponds to the current pose of a trajectory. The current lane is updated based on position, orientation, and time. It is chosen from a list of candidate lanes that are reachable from the lane selected in the previous update of the Tracker. Considering connectivity helps to avoid erroneously selecting crossing or nearby lanes that are not actually reachable from the previous position. If there is no previous update or if the result is bad (further away than 10 meters), the search is extended to all nearby lanes ignoring connectivity. This extended search is used only for the first frame or for recovery in case of a tracking error.

The Map Tracker is initialized with an existing Map Handle:

```
dwStatus dwMapTracker_initialize(
    dwMapTrackerHandle_t *mapTrackerHandle,
    dwConstMapHandle_t map);
```

The current lane is being tracked by subsequently updating the Tracker with the current position, orientation, and time. Orientation is represented as a rotation matrix that transforms from local coordinates (forward-left-up) into

East-North-Up (ENU) coordinate system. For more information, see GPS and HD Maps Coordinate Systems in this guide.

```
dwStatus dwMapTracker_updateCurrentPose(
                      const dwMapsGeoPoint *position,
                      const float64_t *localToENURotation33,
                      dwTime_t timestamp,
                      dwBool ignoreHeight,
                      dwBool reset,
                      dwMapTrackerHandle_t mapTrackerHandle);
```

If the height at the current position is not known or inaccurate, set `ignoreHeight` to `True`.

The tracker can be reset manually, thus discarding the tracking result of the previous update. To obtain the result of the update, call:

```
dwStatus dwMapTracker_getCurrentLane(
                 const dwMapsLane **currentLane,
                 dwConstMapTrackerHandle_t mapTrackerHandle);
```

To identify the candidate lanes among which the current lane has been selected, call:

```
dwStatus dwMapTracker_getCurrentCandidateLanes(
                  dwMapsLaneBuffer *lanes,
                  dwConstMapTrackerHandle_t mapTrackerHandle);
```

Lane Tree

The Lane Tree module represents how the road continues from a given lane. The root of the Lane Tree is the current lane. The children of a tree node are its direct successor lanes at the end of the current Road Segment. Optionally, the successors of neighbor lanes that are reachable through a lane change are child nodes. This capability supports exploration of reachable lanes starting at a given lane, up to a given limit, either with or without lane changes. The limit can be either distance or driving time.

To create a Lane Tree

1. Initialize a Lane Tree Handle with an existing Map Handle as input:

   ```
   dwStatus dwMapsLaneTree_initialize(
                      dwMapsLaneTreeHandle_t *laneTreeHandle,
                      uint32_t maxLaneCount,
                      dwConstMapHandle_t map);
   ```

   The caller must estimate `maxLaneCount`. It is the maximum number of `dwMapsLane` objects that can be stored in the tree. The function uses it to pre-allocate the buffers used during the tree creation.

2. Use the Lane to create the tree:

   ```
   dwStatus dwMapsLaneTree_create(const dwMapsLane *lane,
          float32_t limit, dwMapsLaneTreeLimit limitType,
          dwBool doLaneChanges, dwMapsLaneTreeHandle_t laneTree);
   ```

To obtain the Lane tree result

There are three different ways to access the result:

- To access the tree structure through the root node, call:

```
dwStatus dwMapsLaneTree_get(
    dwMapsLaneTreeNode **root,
    dwMapsLaneTreeNodeBuffer *laneTreeNodes,
    dwConstMapsLaneTreeHandle_t laneTree);
```

- To get a list of all lanes in the tree, call the following:

```
dwStatus dwMapsLaneTree_getLaneCount(
    uint32_t *laneCount,
    dwConstMapsLaneTreeHandle_t laneTree);
```

  Followed by:

```
dwStatus dwMapsLaneTree_getLanes(dwMapsLaneBuffer *laneBuffer,
    dwConstMapsLaneTreeHandle_t laneTree);
```

- To get a sorted list of lanes from the root lane to each leaf of the Lane Tree, call:

```
dwStatus dwMapsLaneTree_getLeafCount(uint32_t *leafCount,
    dwConstMapsLaneTreeHandle_t laneTree);
```

  Followed by:

```
dwStatus dwMapsLaneTree_getLaneListToLeaf(
    dwMapsLaneBuffer *laneBuffer,
    uint32_t leafIndex,
    dwConstMapsLaneTreeHandle_t laneTree);
```

There is a helper function to stitch the geometry of connected lanes into one single polyline, see `dwMaps_stitchLaneGeometry` in the chapter below.

Lane Tree Helper Functions

Local Space Lane Divider Line Segments

```
dwStatus dwMaps_transformLaneDividersToLocalLines(
    dwMapsLaneDividerLineBuffer *lineSegments,
    const dwMapsLaneDividerBuffer *laneDividers,
    const dwMapsGeoPoint *localOrigin,
    const float64_t *localToENURotation33,
    const dwMapsLocalBounds *bounds,
    const dwVector3f *directionFilterVector,
    float32_t directionFilterAngleRadian,
    dwBool ignoreLaneDirection);
```

This helper function combines a few things to transform and filter the lane divider geometry. It does the following:

- Transforms from WGS84 coordinates into in a user-defined local Cartesian coordinate space.
- Transforms the polylines into a list of line segments.
- Filters the line segments by a bounding box defined in local space.
- Filters the line segments by direction.

Local Cartesian Coordinate System

The local coordinate system is defined by:

- Point in WGS84 coordinates
- Rotation matrix

The point defines the origin of the east-north-up (ENU) coordinate system on the tangent plane of the Earth spheroid. The basis vectors of the ENU space are:

```
(1,0,0) = east
(0,1,0) = north
(0,0,1) = up
```

The rotation matrix transforms from the local coordinate system into the ENU space. It defines the user local space of the returned coordinates. The basis vectors of the user local space can be interpreted as:

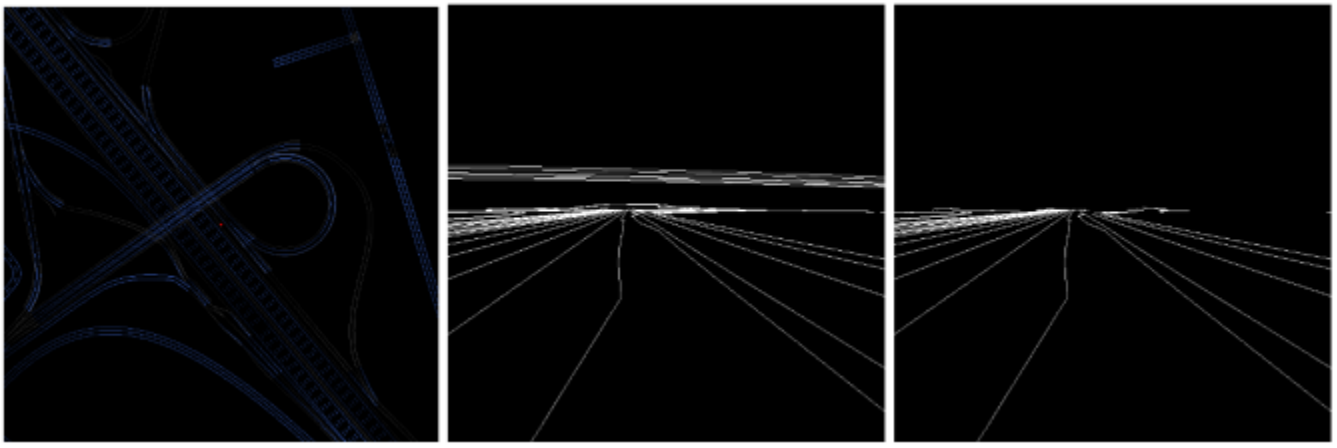
```
(1,0,0) = forward
(0,1,0) = left
(0,0,1) = up
```

This means that if the rotation matrix is an identity matrix, the local space is facing east.

There is also a helper function `dwMaps_computeLocalToENU` that creates the rotation matrix from a single bearing value.

Filtering

The direction filtering allows to discard all line segments that do not point into a desired direction. For example, providing a direction filter vector of (0,1,0) with an angle of 0.25*Pi will only return line segments that have an angle of less than 45 degrees compared to the viewing direction. This can be used to filter out bridges that cross the current Road Segment horizontally, as shown in the following images



Local Space Feature Line Segments

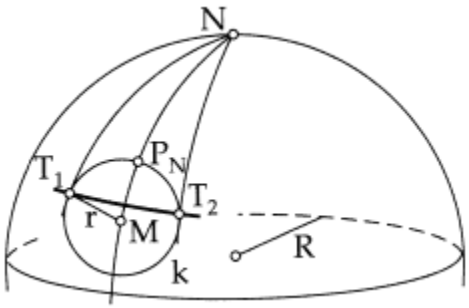
```
dwStatus dwMaps_transformRoadFeaturesToLocalSpace(
                  dwMapsPolyline3fBuffer *localPolylines,
                  dwMapsPointBuffer *pointBuffer,
                  const dwMapsFeatureBuffer *features,
                  const dwMapsGeoPoint *localOrigin,
                  const float64_t *localToENURotation33,
                  const dwMapsLocalBounds *localBounds);
```

This function has the same functionality as `dwMaps_transformLaneDividersToLocalLines`, but for features instead of lane dividers.

Compute Bounds

```
dwStatus dwMaps_computeBounds(dwMapsBounds *bounds,
                              const dwMapsGeoPoint *p,
                              float32_t radiusMeter);
```

The query functions require a bounding box in WGS84 coordinates to define the area of interest. The size of the longitude/latitude box for a given radius in meters varies with latitude, so it is not obvious how big the WGS84 bounding box must be to cover a desired radius in meters. The `computeBounds` function does this calculation. It returns the bounds that fully contain a given circle on the earth surface.



[Image source: http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates]

Compute Bearing

There is a helper function to compute the bearing (clock-wise angle from north) from a current and a target position:

```
dwStatus dwMaps_computeBearing(float64_t *bearingRadian,
    const dwMapsGeoPoint *position,
    const dwMapsGeoPoint *headingPoint);
```

Compute Local To ENU

```
dwStatus dwMaps_computeLocalToENU(
                        dwMatrix3d *localToENURotation33,
                        float32_t bearingRadian);
```

The coordinate space in the functions that transform into local space is defined by a position and a rotation matrix that transforms from local coordinate space into the ENU coordinate system. `computeLocalToENU` is a helper function that creates the rotation matrix from a bearing angle. It is a clockwise rotation around the z-axis by bearing angle.

Transform Polylines

```
dwStatus dwMaps_transformPolylines(
                      dwMapsPointBuffer *transformedPoints,
                      const dwMapsGeoPolyline *polylines,
                      uint32_t polylineCount,
                      const dwMapsGeoPoint *localOrigin,
                      const float64_t *localToENURotation33);
```

All map data polylines are defined in WGS84 coordinates by longitude angle, latitude angle and height above the earth spheroid surface. `transformPolylines` transforms an array of WGS84 coordinate polylines into a Cartesian local space, the same way it is done for the `getLaneDividerLinesLocal` query, however it just returns the polylines in local space (as opposed to returning filtered line segments).

Transform Point

```
dwStatus dwMaps_transformPoint(dwVector3f *transformedPoint,
                        const dwMapsGeoPoint *point,
                        const dwMapsGeoPoint *localOrigin,
                        const float64_t *localToENURotation33);
```

The `transformPoint` function does the same as `transformPolylines`, but for a single point only.

Interpolation Between Polylines

Interpolation between two Polylines

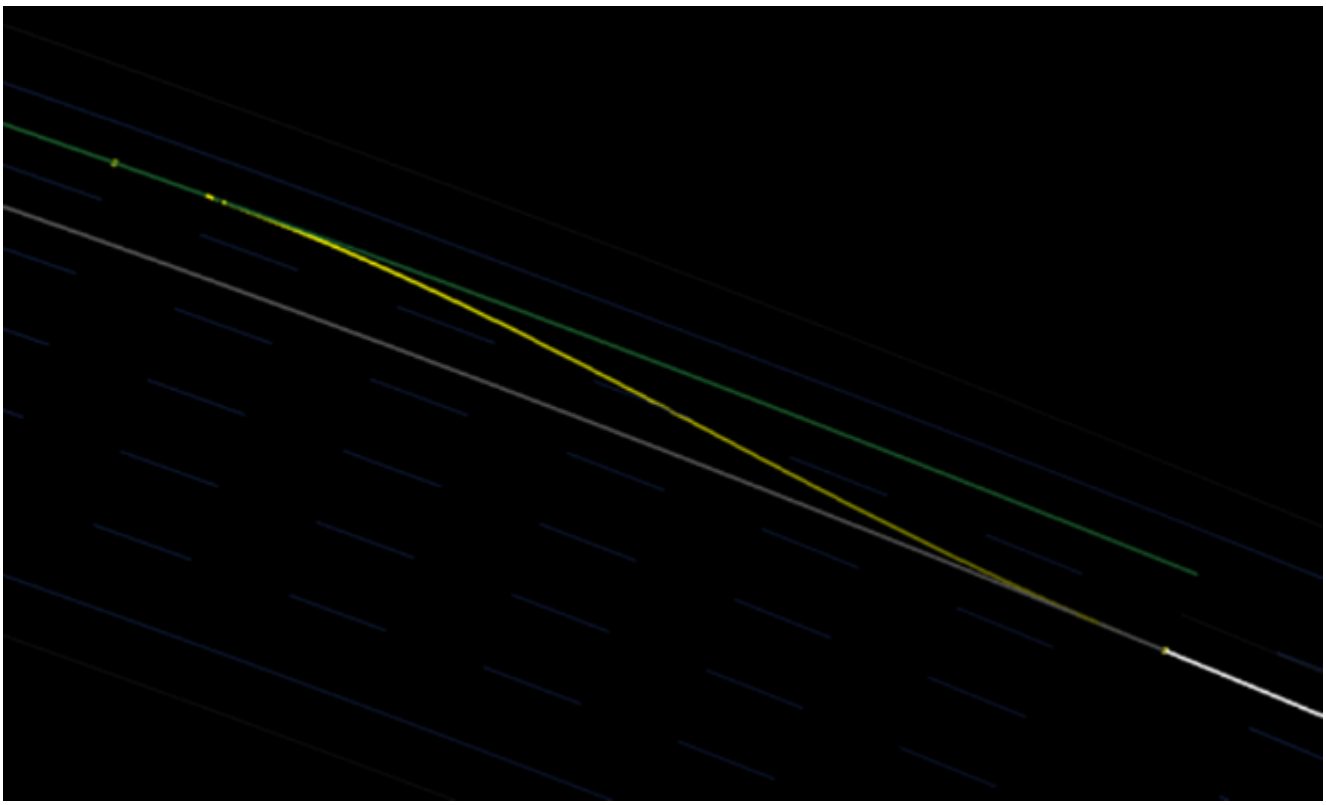
```
dwStatus dwMaps_interpolatePolylines(uint32_t *srcStartIndex,
                uint32_t *targetEndIndex,
                dwMapsGeoPointBuffer *interpolatedPoints,
                const dwMapsGeoPoint *srcPoints,
                uint32_t srcPointCount,
```

```
                const dwMapsGeoPoint *targetPoints,
                uint32_t targetPointCount,
                float32_t start,
                float32_t end,
                float32_t stepSize,
                float32_t (*interpolationFn)(float32_t, void*),
                void *interpolationFnContext);
```

This helper function provides interpolation between 2 polylines. It can be used to create a path that connects two parallel polylines, for example to model a lane change.

Input are:

- the source polyline
- the target polyline
- the start of the interpolation (distance in meters from first source polyline point)
- the end of the interpolation (distance in meters from the first target polyline point)
- the step size to define where interpolation points are evaluated
- a function callback to define the interpolation curve (linear by default)



In the above example image, the interpolation function is:

```
1.0 - 0.5*(cos(d * Pi) + 1.0);
```

It maps the input parameter d, which goes from 0.0 to 1.0 and represents the distance from interpolation start to interpolation target (horizontal distance in the example image), to the target range 0.0 to 1.0 that represents the weight between source and target polyline.

Neighbor Lanes

Given a dwMapsLane, it is possible to figure out how many lanes are left and right on the current road segment.

To query the number of lanes on a side, call:

```
dwStatus dwMaps_getNeighborLaneCount(uint32_t *laneCount,
                   uint32_t *laneCountAccessible,
                   const dwMapsLane *lane, dwMapsSide side,
                   dwBool sideRelativeToDrivingDirection);
```

There are two return values, the total number of lanes on that side, and the number accessible ones. Accessible lanes are the ones that can be reached from the current lanes through a lane change, i.e. all lane dividers in between can be legally crossed.

The side can be requested either relative to the polyline directions on the road segment, or relative to the driving direction on the input lane (these directions are not necessarily the same).

A neighbor lane can be access by calling:

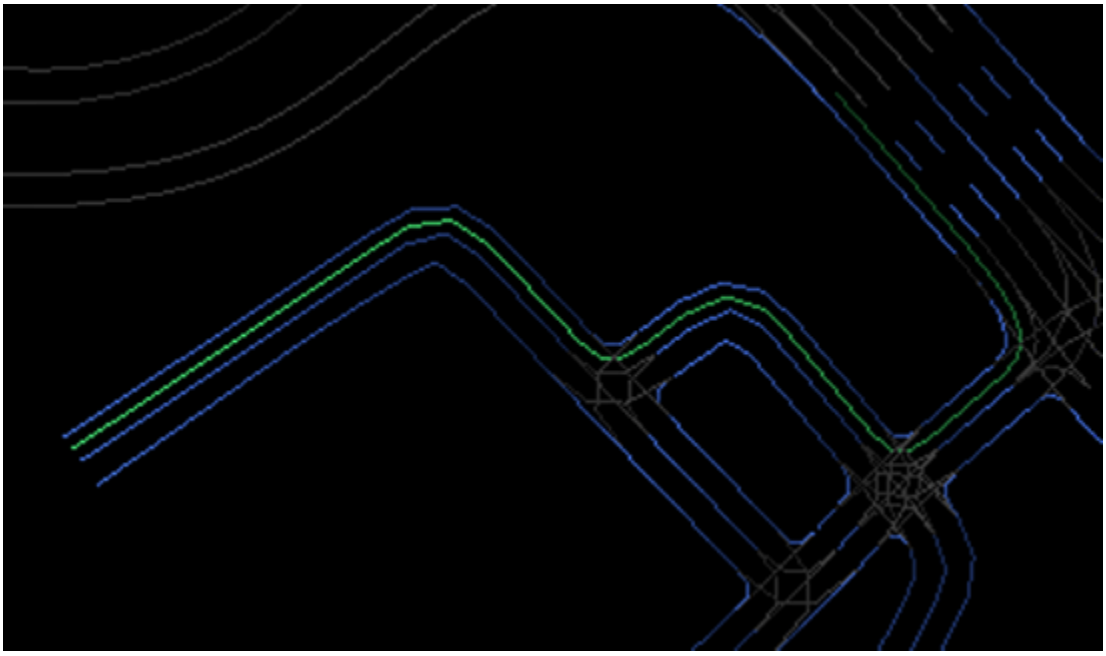
```
dwStatus dwMaps_getNeighborLane(const dwMapsLane **otherLane,
                   const dwMapsLane *currentLane,
                   dwMapsSide side, uint32_t offset,
                   dwBool sideRelativeToDrivingDirection);
```

offset = 1 returns the directly adjacent lane, offset = 2 the next one, etc.

Stitching of Lane Geometry

Given a list of connected dwMapsLane objects, this helper functions stitches the requested geometry (lane center line, left lane divider, right lane divider) into one connected polyline:

```
dwStatus dwMaps_stitchLaneGeometry(
                 dwMapsGeoPointBuffer *polyline,
                 const dwMapsLane *lanes, uint32_t laneCount,
                 dwMapsLaneGeometry geometrySelection);
```



Distance Calculations

The length of a polyline of WGS84 points can be computed with the helper function

```
dwStatus dwMaps_computePolylineLength(float32_t *length,
          const dwMapsGeoPoint *points, uint32_t pointCount);
```

There is also a helper function that computes the Euclidean distance between two WGS84 points:

```
dwStatus dwMaps_computeDistance(float32_t *distance,
    const dwMapsGeoPoint *p1,
    const dwMapsGeoPoint *p2);
```

Vehicle Module

Rig Module

Rig Configuration

The car is considered a *rig* with several rigidly-attached sensors. The dimensions of the car and the positions of these sensors relative to the car are important to DriveWorks accuracy. The rig characteristics are measured and estimated by a calibration process.

The rig configuration module allows the reading and enumeration of these pre-calibrated properties. The module obtains the rig configuration from an XML generated by the DriveWorks calibration tool. For more information, see *DriveWorks Calibration Tool Application Note* in the `doc/pdf/calibration` folder of this release.

An example of the XML structure is given below.

```
<DRIVEWORKS ver="1.0">
  <VEHICLE>
   <PROPERTY Value="1.455" Name="Height"/>
   <PROPERTY Value="2.912" Name="Wheelbase"/>
   <PROPERTY Value="0.68" Name="Wheel diameter"/>
   ...
  </VEHICLE>
  <SENSORS>
    <CAMERA Name="Front" Type="gmsl">
      <PROPERTY Value="ocam" Name="Model"/>
      <PROPERTY Value="1280" Name="width" Hint="intrinsic"/>
      <PROPERTY Value="800" Name="height" Hint="intrinsic"/>
      <PROPERTY Value="657.93319" Name="cx" Hint="intrinsic"/>
         ...
     </CAMERA>
     ...
   </SENSORS>
</DRIVEWORKS>
```

DriveWorks recently added support for a similar JSON rig configuration format. A rig configuration object (`dwRigConfigurationHandle`) can be initialized either from a file or directly from string (see `dwRigConfiguration_initializeFromFile`). Once the rig configuration is loaded successfully, individual properties can be queried about the rig and its sensors. Vehicle properties are represented by the `dwVehicle` struct (see `dwRigConfiguration_getVehicle`). Generic sensor properties can be obtained through the function `dwRigConfiguration_getSensorXXX`. Some sensors may have more specific properties, like the calibrated camera model (for example see `dwRigConfiguration_getPinholeCameraConfig`).

Camera Rig

The SFM and Stereo modules use a camera-only rig (`dwCameraRigHandle`) that contains only rigidly-attached calibrated cameras (`dwCalibratedCameraHandle`). This camera rig is a subset of the generic rig. It can be initialized independently or directly from a generic rig configuration (see `dwCameraRig_initializeFromConfig`).

The supported calibrated cameras are pinhole and OCam. Their calibration parameters are specified with the `dwPinholeCameraConfig` and `dwOCamCameraConfig` structs respectively. The camera models and its parameters are described in detail in *DriveWorks SDK Reference*. DriveWorks also offers the `dwCalibratedCamera_pixel2Ray` and `dwCalibratedCamera_ray2Pixel` functions to transform a point from image space to camera space and test the model.

Calibration

The calibration parameters for a sensor model have the following properties:

- Intrinsic (e.g., camera lens properties)

- Extrinsic (e.g., the pose relative to the rig)

Nominal values for these parameters are obtained by static calibration or from previous knowledge about sensor characteristics and mounting positions. Exact parameters are crucial for any effective usage of the sensor's measurements. However, nominal calibration parameters can vary over time, e.g. due to influences of the environment like temperature changes or mechanical stress. This module addresses these transitional variations.

Self-calibration is the process of correcting the nominal calibration parameters based on up-to-date sensor readings. This process compensates for transitional variations to enhance the availability of high quality parameters at any time.

In DriveWorks, the Calibration module provides a common interface to perform self-calibration of different sensor types. It is a lightweight service that is constantly running in the background to always provide the best estimations for calibration parameters.

For a particular sensor, a sensor calibration routine is registered with the calibration engine (`dwCalibrationEngine`) and then fed with recent sensor readings / processed data or detections. For instance, a camera calibration is requested via `dwCalibrationEngine_initializeCamera`, and data is provided to the calibration engine via `dwCalibrationEngine_addLaneDetections`. Internally, the calibration engine analyses the provided data and estimates the corrected calibration parameters for the particular sensor. At any time, the status of the calibration process (`dwCalibrationEngine_getCalibrationStatus`) and the latest calibration parameters can be queried (e.g., via `dwCalibrationEngine_getSensorToRigTransformation`).

Egomotion

The Egomotion module tracks and predicts a vehicle's pose, on the basis of a motion model, given measurements from multiple sensors. The motion model is selected at initialization time. During run-time, the module takes measurements as input and internally updates the current estimation of the vehicle pose. The module can be queried for vehicle motion between any two points in time.

The motion model is based on the Ackermann principle. The simplest model, selectable with `DW_EGOMOTION_ODOMETRY` type, estimates the vehicle motion based on odometry information only. Measurements like speed in meters per second and steering angle on the road must be passed to the module to perform estimation. The model assumes a vehicle of given length with a fixed rear axle and steerable front wheels, driving on a 2D plane. Predictions are done assuming constant steering angle and velocity during the time delta.

To select IMU-based motion estimation is selected, specify `DW_EGOMOTION_IMU_ODOMETRY` as the type of the motion model. In this mode, the module estimates the car motion and its orientation based on velocity, steering angle, and IMU measurements, such as gyroscope and linear accelerometer. The change in position is estimated with the same Ackerman principle as the simplified motion model. The orientation, however, is estimated using a complementary filter that fuses gyroscope and linear accelerometer measurements.

The Egomotion module internally maintains a history of poses to allow a query of relative poses between any two timestamps. The returned relative pose represents the relative motion of the vehicle, i.e. change of orientation and change of translation, that the vehicle performed from timestamp A to timestamp B in the local flat Euclidean space. If a timestamp is in the future, then a motion prediction is returned. This prediction allows the module to make assumptions about how the vehicle will move, given the last known state of the sensors.

The following shows the sequence in a typical application using the Egomotion module:

1. Read out odometry from CAN sensors.
2. Pass odometry with `dwEgomotion_addOdometry()`.
3. Read out gyroscope+linear accelerometer data.

4. Pass IMU frame with `dwEgomotion_addIMUMeasurement()`.
5. Every X milliseconds (say every 20-ms), update the trajectory estimation with `dwEgomotion_update()`.
6. At any point of time, query the relative motion of the vehicle between any two timestamps with `dwEgomotion_computeRelativeTransformation()`.
7. Repeat in the loop.

VehicleIO

The VehicleIO module interfaces with the vehicle, where one side can control vehicle actuation and provide information regarding the current state of the vehicle. The module can actuate and read from the vehicle by using the following types:

- `DW_VEHICLEIO_DATASPEED` specifies the use of a "Dataspeed" device as an intermediary protocol.
- `DW_VEHICLEIO_GENERIC` specifies any programmable interface device that translates the messages provided by the application into a format understandable by the vehicle.

Initialization

The `dwVehicleIO_initialize` function creates a `VehicleIO` instance. That functions takes the following parameters:

- `VehicleIO` handle: `dwVehicleIOHandle_t`
- Type of interface device: `dwVehicleIOType (DW_VEHICLEIO_DATASPEED/DW_VEHICLEIO_GENERIC)`

  Note: `DW_VEHICLEIO_GENERIC` is not available in this release, but has been added as a placeholder, for the next release.

- Handle to the vehicle: `dwVehicle`

  Note: Before creating a VehicleIO instance, you must initialize the rig configuration module using `dwRigConfiguration_initialize()` and then get the vehicle properties using `dwRigConfiguration_getVehicle()`. For more information, see Rig Configuration in this guide.

- Context Handle : `dwContextHandle_t`

Driving Mode

The `dwVehicleIO_setDrivingMode` function sets the driving mode. This function consumes the `dwVehicleIODrivingMode` enum as an argument, which is defined as

```
typedef enum dwVehicleIODrivingMode
{
/// Comfortable driving is expected (most conservative). Commands that leave
/// the comfort zone are treated as unsafe, which immediately leads to
/// VehicleIO being disabled.
DW_VEHICLEIO_DRIVING_LIMITED = 0x000,
/// Same as above, but unsafe commands are clamped to safe limits and
/// warnings are isssued. VehicleIO stays enabled.
```

```
   DW_VEHICLEIO_DRIVING_LIMITED_ND = 0x100,
/// Safety checks suitable for collision avoidance logic (right now same as
/// NO_SAFETY below).
   DW_VEHICLEIO_DRIVING_COLLISION_AVOIDANCE = 0x200,
/// VehicleIO will bypass all safety checks.
   DW_VEHICLEIO_DRIVING_NO_SAFETY = 0x300
} dwVehicleIODrivingMode;
```

Vehicle State Information

The `dwVehcileIO_consume` function parses received CAN messages. The resulting parsed messages generate certain reports, which can be gathered using the predefined callbacks.

The current vehicle state information can be retrieved with `dwVehicleIO_getVehicleState`, which returns the vehicle state in the `dwVehicleIOState` format.

Sending Vehicle Commands

The `dwVehicleIO_sendVehicleCommand` function sends a command to the vehicle via `VehicleIO`. The command is sent in the `dwVehicleIOCommand` format and is passed as one of the arguments. An additional argument is passed as a handle to the underlying sensor that `VehicleIO` uses to pass this command.

Selecting Driver Overrides

Signals that the driver uses to override Vehicle control are configurable and can be selected using `dwVehicleIO_selectDriverOverrides`. The driver can override vehicle control with any combination of throttle, steering, brake, and/or gear.

Running VehicleIO Sample

Test the application directly by running it with the following sample:

```
./sample_vehicleio
```

Alternatively, the sample application accepts custom inputs for the underlying sensor, rig and `vehicleIO` type:

```
./sample_vehicleio --driver=can.virtual --params=file=vehicleio/can.bin \
    --type=dataspeed
```

Sensor Fusion

Occupancy Grid

The occupancy grid module showcases a dense Bayesian occupancy grid given sensor input. Specifically, this dense view provides probabilities of a grid cell being free. When creating an occupancy grid, there are two types of input layers that can be added: point clouds and objects. In the initialization of each layer, a set of probabilities is specified. For point clouds, these probabilities correspond to the actual probability of a grid cell being free at the actual point, the probability that sensor origin is free space, and the probability of free space beyond the point. Point clouds are inserted based on the assumption the data came from a range sensor. Therefore, the point clouds are inserted by casting a ray from the sensor origin, to the point. For object lists, only the probability that a cell is

free at the object is specified. Additionally, each layer specifies the sensor to sensor rig transformation ensuring the points inserted are oriented correctly in the grid.

Creating an occupancy grid is done with the `dwOccupancyGrid_initialize` function and passing `dwOccupancyGridParameters` struct object as a parameter. The `dwOccupancyGridParameters` struct contains information about the size and the render color of the occupancy grid.

```
typedef struct dwOccupancyGridParameters
{
   /// The dimensions of the grid in length, width, and
   /// height.
   float3_32_t gridDimensionsMeters;
   /// The minimum grid coordinates in length, width, and
   /// height.
   float3_32_t gridMinMeters;
   /// The size of each grid cell in the occupancy grid,
   /// in meters.
   float32_t cellSizeMeters;
   /// The render color for rendering to an FBO.
   float4_32_t renderColor;
   /// Specifies whether to translate only the new points
   /// that are inserted.
   dwBool isScrollingMap;

} dwOccupancyGridParameters;
```

The renderColor affects the output color of the occupancy grid in general. For example, if the renderColor is set to red ({1.0f, 0.0f, 0.0f, 1.0f}), the probability that a cell is occupied will be based on that color. In other words, completely free would be completely red, unknown would be dark red ({0.5f, 0.0f, 0.0f, 1.0f}) and completely occupied would be black.

The occupancy grid can support up to OCCUPANCY_GRID_MAX_LAYER_COUNT, which is currently set to 16. A layer can be added with one of two functions, depending on the desired layer type: `dwOccupancyGrid_addRangeSensorLayer` for range sensor layers (point cloud input) or `dwOccupancyGrid_addObjectListLayer` for object list layers (polygon input). Each function expects initialization parameters for the layer:

```
typedef struct dwOccupancyGridRangeSensorParameters {
    /// The column-major orientation of the sensor for a given
    /// layer. This includes rotation and translation
    /// and will be used to apply to points
    /// on insertPointCloud.
    dwTransformation sensorToRigTransformation;
    /// The min valid distance in meters that this layer can
```

```
    /// register.
    float32_t minValidDistanceMeters;
    /// The max valid distance in meters that this layer can
    /// register.
    float32_t maxValidDistanceMeters;
    /// The probability that a grid cell is free at the sensor
    /// origin.
    float32_t probabilityFreeAtSensorOrigin;
    /// The probability that a grid cell is free at the sensor
    /// max valid distance.
    float32_t probabilityFreeAtSensorMaxDistance;
    /// The probability that a grid cell is free given a grid
    /// point hit.
    float32_t probabilityFreeAtSensorHit;
    /// The probability that a grid cell is free beyond the
    /// sensor hit to the max distance.
    float32_t probabilityFreeBeyondSensorHit;
    /// The min threshold value per grid cell.
    float32_t minAccumulatedProbability;
    /// The max threshold value per grid cell.
    float32_t maxAccumulatedProbability;
    /// Sets whether or not the layer is cumulative. If false
    /// insert point cloud will
    /// always reset previously inserted point clouds.
    dwBool isCumulative;
    /// Sets the ray width
    float32_t rayWidth;
    /// Sets the hit width
    float32_t hitWidth;
} dwOccupancyGridRangeSensorParameters;

typedef struct dwOccupancyGridObjectListParameters {
    /// The column-major orientation of the objects for a
    /// given layer. This includes
    /// rotation and translation and will be used to apply
    /// to objects on insert objects.
    dwTransformation objectListToRigTransformation;
```

```
    /// The probability that a grid cell is free given a grid
    /// point hit.
    float32_t probabilityFreeAtObject;
    /// The min threshold value per grid cell.
    float32_t minAccumulatedProbability;
    /// The max threshold value per grid cell.
    float32_t maxAccumulatedProbability;
    /// Sets whether or not the layer is cumulative. If false
    /// insert point cloud will
    /// always reset previously inserted point clouds.
    dwBool isCumulative;
    /// Sets the object line width
    float32_t objectLineWidth;
} dwOccupancyGridObjectListParameters;
```

Object list layers are useful if you have input that should not be inserted to the occupancy grid based on ray casting. This type of layer will insert the polygon lines without making assumptions about the free space in between the sensor origin and the polygon.

To insert data into the occupancy grid, use `dwOccupancyGrid_insertPointCloud` or `dwOccupancyGrid_insertObjectList`, depending on the layer type. The `dwOccupancyGrid_update` call will update the grid with the current transformation matrix that is passed in as a parameter. Typically, this is the world-to-rig matrix or the matrix that transforms world coordinates into rig coordinates.

Multiple layers rendered separately with `dwOccupancyGrid_renderLayer` or rendered combined with `dwOccupancyGrid` render. When using the combined function, it will add all the layers together in the log-space before rendering.

Occupancy Grid Sample

The occupancy grid sample application uses three sensors in combination: CAN data for the car position and orientation, LiDAR for point cloud data, and camera for visualization. The sample allocates a grid and allows the application to insert point clouds or object lists, which are polygons stored as a list of points, on a stationary-to-moving vehicle. You can observe the point clouds or objects being accumulated, and as the vehicle moves, the grid is updated according to the probabilities given in the initialization.

Running Occupancy Grid Sample

Test the application directly by running it:

```
./sample_occupancy_grid
```

Alternatively, the sample application accepts custom inputs for lidar, can, and camera data:

```
./sample_occupancy_grid --canFile=can.bin --dbcFile=can.dbc --lidarFile=lidar.bin \
    --videoFile=video.h264 --videoTimestampFile=video_time.txt --fps=30
```

The following is an explanation of each argument:

```
--canFile -              The recorded can data
--dbcFile -              DBC file for interpreting can
--lidarFile -            The recorded lidar data
--videoFile -            The recorded h264 video
--videoTimestampFile -   The timestamp file associated with the videoFile
--fps -                  The speed at which the sample plays
```

All inputs are assumed to be in the same format that the DriveWorks recording tool uses when recording. The videoFile must be h264 and must have an associated videoTimestampFile in order to be able to synchronize with the other sensors properly. Additionally, the sensors must be recorded at the same time so that they can be synchronized and played back together.

The expected output is a Bayesian occupancy grid with areas that are free colored white, areas that are not free colored black, and everything unknown colored gray. The colors represent the probability of a cell being free.

ICP Module

ICP module provides an API to align 3D points from a pair of lidar spins via Point-Plane Iterative Closest Points implementation. It's usage is shown in `sample_icp`. This module assumes a small translation and rotation between the two point clouds (relative to a prior pose provided), such as the lidar sweeps captured <2 meters apart from a moving vehicle.

Terminology in the API

The two point clouds input to the the ICP module are named `Target` and `Source` point clouds respectively. The output of the alignment is the transform that must be applied to Source points so that they align to the target point cloud. As this module uses an iterative approach, a `Convergence` criteria is used to test if iterations need to stop. `Cost` of the iteration is a measure of distance between point and their target planes (please refer to literature on ICP process).

Restrictions on the input data

ICP Module place the following major restrictions on data input to the module:

- The number of points may not exceed 32767 points in either source or target clouds.
- There is a small rotation/translation relation between the two point clouds, e.g. a translation of no more than 2-meters converges to acceptable results.
- Lastly, the point cloud data input via a pointer is arranged in such a way that points in close proximity to one another in 3D space are nearby in the laid out memory.

Iteration Stopping Criteria

Two criteria control the stopping point of the iteration in this module and consequently the total run time and the accuracy of the results. These criteria are defined in terms of:

- Rotation (in radians) and translation (in meters) tolerances or
- Maximum number of iterations

When the difference between results of two iterative steps is smaller than the rotation/translation tolerances, the

control returns to the application. The process terminates after a set number of iterations are complete, which is intended to produce a more uniform distribution of run times.

Alternatively, the application can provide a callback for the convergence test. This callback can compare two of the latest transforms and return a value that indicates whether the iterative process should conclude.

Result Statistics

ICP Module provides some basic statistics about the last iteration performed. The statistics are a quantitative indicator of the alignment between the point clouds.

```
typedef struct dwICPResultStats {
    uint32_t actualNumIterations;
    float32_t rmsCost;
    float32_t inlierFraction;
    uint32_t numCorrespondences;
}dwICPResultStats ;
```

- `rmsCost` is the main indicator of the alignment, giving the rms of the distance between points in source point clouds to the planes in the target point clouds.
- `inlierFraction` indicates the fraction of points that were considered for the ICP process. When the transform between two point clouds is small, this fractions tends to be large.
- `numCorrespondences` is the total number of point-to-plane correspondences used in in the optimization.
- `actualNumIterations` is the actual number of iterations used before terminating.

Initializing the algorithm

As mentioned, the target and source points are expected to have small transforms (< 1 meter, <10 degree) separating them. However, in case larger transforms are present, prior-information about the relative transform (e.g. from inertial / odometry based method) can be used to initialize the algorithm. This prior pose can be passed into the ICP module via the `dwICPIterationParams::initialSource2Target` argument.

Lidar Accumulator Module

This module provides the API access to the partial/full 3D Lidar sweep and 360-degree Lidar images for rotating beam Lidars. DriveWorks samples include `sample_lidar_accumulator`, which is described in an adjacent README file. If you plan to manually accumulate the Lidar packets for a full sweep, see the `sample_lidar_replay` sample.

Initialization

To initialize the module, call `dwLidarAccumulator_initialize`. This function consumes the following arguments:

- dwContextHandle
- dwLidarProperties
- filterWinSize

Note that `filterWinSize` specifies the horizontal smoothing for the Lidar sweep to reduce the horizontal jitter inherent to the Lidar rotating units. It must be set to even number ranges from 1 to 2, 4 and 8. When set to values

other than 1, the Lidar point whose 3D distance is the closest to the Lidar sensor is selected in the horizontal window.

Lidar Sweep

In addition to accumulating Lidar packets, the Lidar Accumulator module also accesses the organized Lidar point clouds. To add the Lidar packet to the accumulator, call `dwLidarAccumulator_addPacket`. To access the organized 3D Lidar data, call `dwLidarAccumulator_getSweep`. The data struct `dwLidarAccumulatorBuffer` includes the pointer to the organized Lidar coordinates and intensity values.

```
typedef struct {
   /// Total number of non-zero points
   uint32_t validCount;
   /// Total number of points in data
   uint32_t dataCount;
   /// Organized Lidar points in Cartesian coordinate XYZI
   /// It includes both zero and non-zero points.
   const dwVector4f *data;
   /// Host timestamps
   const dwTime_t *hostTimestamps;
   /// Sensor timestamps
   const dwTime_t *sensorTimestamps;
} dwLidarAccumulatorBuffer;
```

You can call `dwLidarAccumulator_returnSweep` when the operation on the organized Lidar point cloud is done.

Lidar Image

Apart from organized Lidar point cloud, the module offers the access to 360-degree cylindrical Lidar image whose pixel value can be either 3D radial distance, 2D radial distance or intensity value. You can specify one of the following in the API `dwLidarAccumulator_createImage`.

```
DW_LIDAR_IMAGE_TYPE_3D_DISTANCE_IMAGE
DW_LIDAR_IMAGE_TYPE_2D_DISTANCE_IMAGE
DW_LIDAR_IMAGE_TYPE_INTENSITY_IMAGE
```

The image memory ownership resides on the application. Once the sweep is ready, API dwLidarAccumulator_fillImage returns the 360-degree Lidar image. Sample Lidar images can be found below.

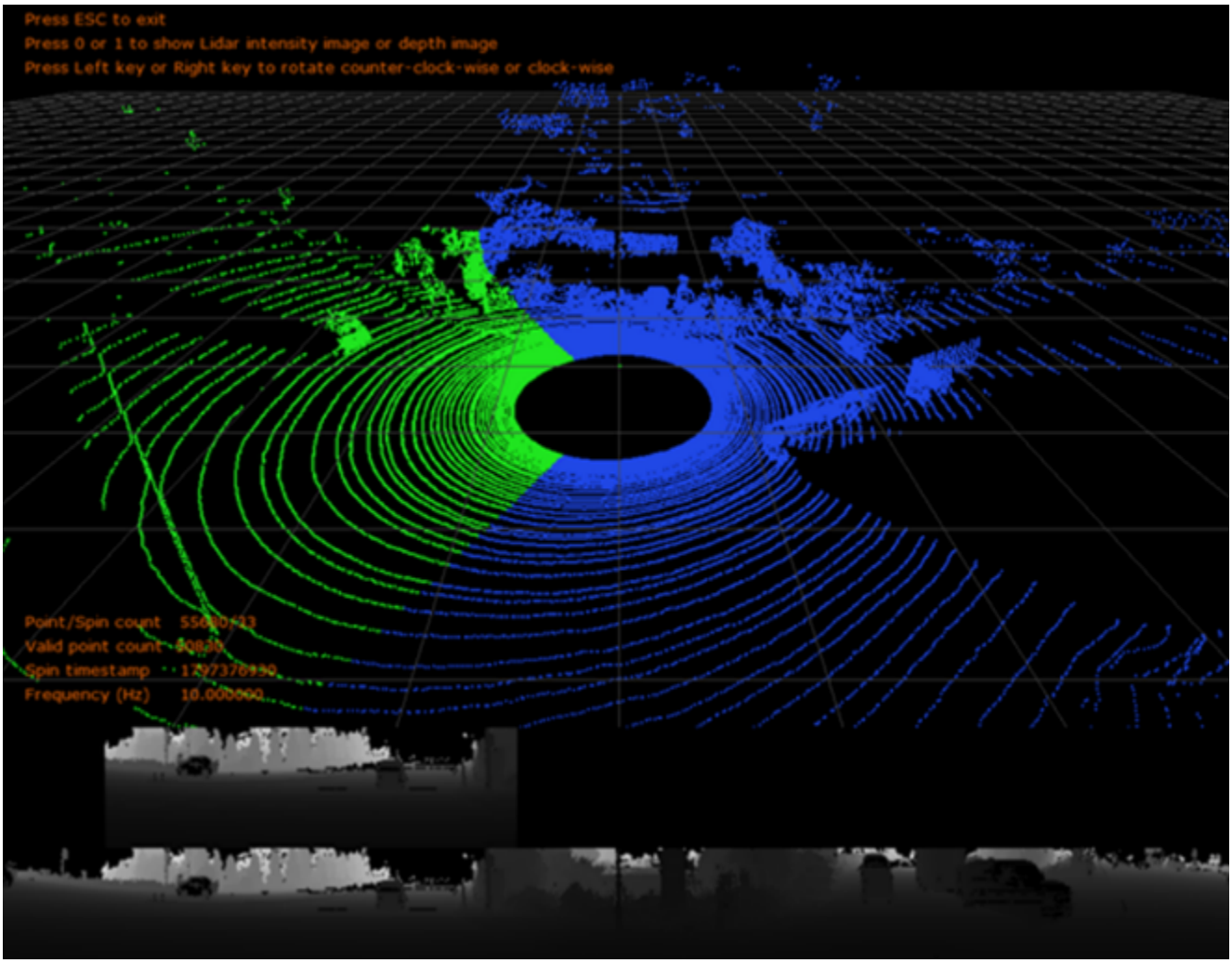
3D Distance Lidar Image

Intensity Lidar Image

Lidar Sweep Angle Setting

By default, the module returns full 360-degree Lidar sweep. If you instead plan to accumulate Lidar sweep in a particular angle range, use `dwLidarAccumulator_setAngleSpan`. This function allows you to specify the starting and ending angle of the range.

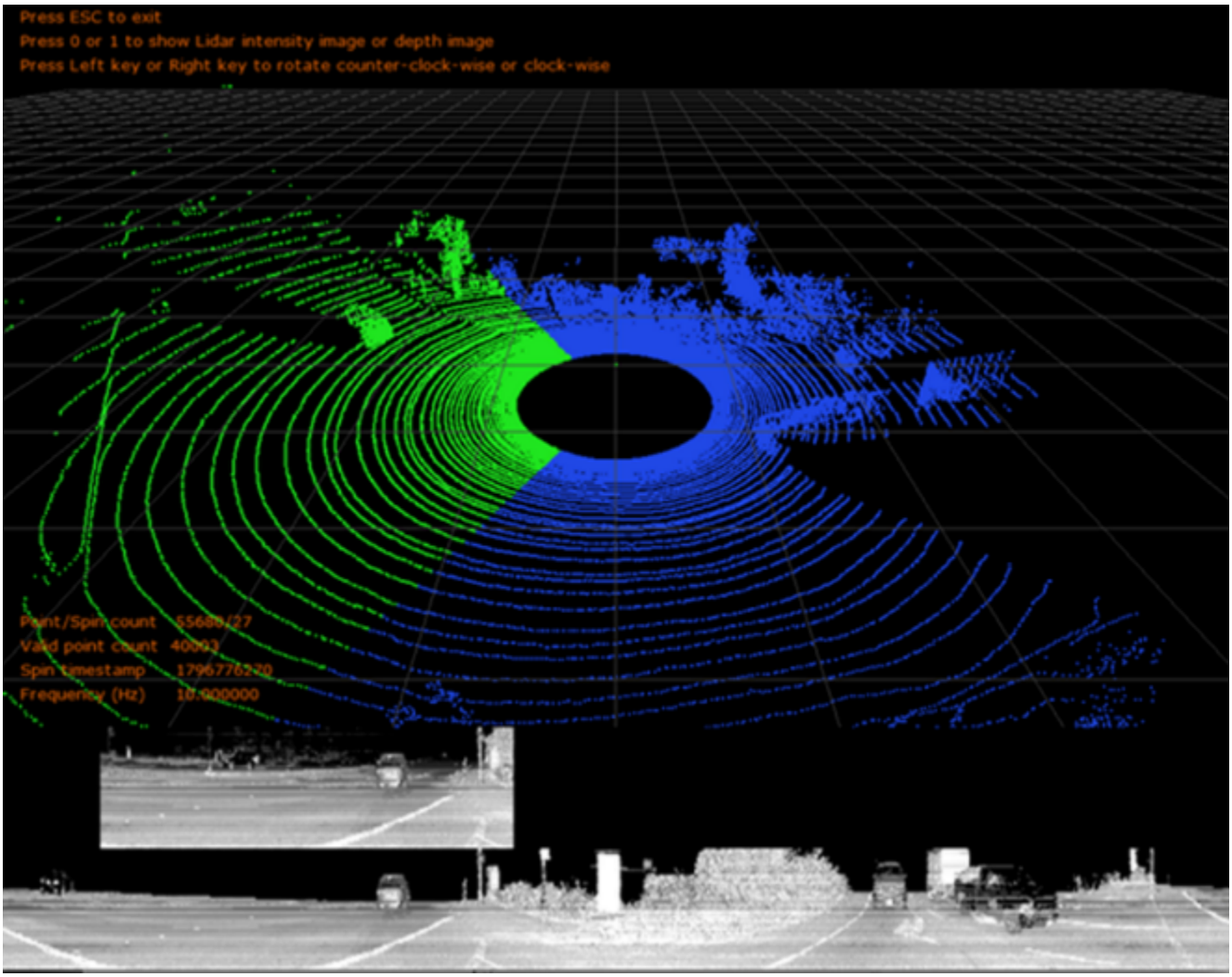
In the figure below, the green color point cloud represents the sector Lidar sweep. The second bottom image corresponds to the 3D distance Lidar image.



Lidar Scan Distance Setting

Like the Lidar sweep angle setting, the Lidar Accumulator module provides `dwLidarAccumulator_setDistanceSpan` to customize the range of distance in a 360-degree Lidar sweep. In

the figure below, the blue color point cloud specifies the Lidar data from a 5 to 20-meter distance, while the green ones cover all the distances.



Vision Processing Modules

2D Tracker Module

The 2D Tracker module can detect and track feature points between frames recorded by one camera. The detector is a Harris Corner detector and the tracker is a KLT tracker. To support the tracker, the module also provides a Gaussian Pyramid implementation.

In addition to detection and tracking, the module can manage lists of 2D features. All functionality is implemented as CUDA kernels and runs asynchronously on the GPU.

Pyramid

The input to the tracker are Gaussian Pyramids of single channel frames, e.g. the Y channel of an YUV image. Memory for it is allocated during `dwPyramid_initialize`, which also specifies the number of levels. Each level has the quarter resolution of the previous level, e.g. 1280x800, 640x400, 320x200 for a 3-level pyramid

and camera resolution of 1280x800. Each created pyramid is meant to be reused and is updated via the `dwPyramid_build` API.ad

Feature Tracker

The Feature Tracker has two functional pieces: the detector and the tracker. The user defines an upper bound on the number of features during initialization. This defines the size of allocations inside the tracker module. The runtime of the tracker can still vary per the actual number of features.

To track 2D feature points between two frames, `dwFeatureTracker_trackFeaturesAsync` takes as input two pyramids and list of 2D feature points as well as predicted locations for those points. The prediction can be the current position or can be computed by some motion model for the feature points. The output is the location and status for each input point, e.g. if the feature could be tracked successfully or not.

Calling `dwFeatureTracker_detectNewFeaturesAsync` runs a Harris Corner implementation on the GPU to detect new features and adds them to the end of the given list of feature points. For example, if the maximum feature count is 2000 and the list has 1500 features, currently only up to 500 features will be added, starting from index 1500 onwards. The decision if a new feature is added is dependent on proximity to features already in the list as well as a threshold on the corner response function.

Feature Lists

Feature lists manage an ordered list of 2d feature points. Each feature point has a 2D location that is represented as non-normalized floating point coordinates. Each feature also has a status flag indicating if whether has been successfully tracked.

Besides storing the features and serving as I/O to the tracker, the feature list also comes with basic housekeeping functionality. While the detector avoids creating new features close to existing features to avoid congestion, the tracker does not check for converging features. To avoid feature congestion during tracking, `dwFeatureList_proximityFilterAsync` ensures that only one feature remains in those areas.

To remove features from the list that have a status indicating that they were not successfully tracked the combination of `dwFeatureList_selectValid` and `dwFeatureList_compact` can be used. The output is a compacted feature list with only valid features as well as indices of location in the input list.

Putting It All Together

The following code snippet shows the general structure of a program that uses the 2D tracker to track features in a single camera. See the `sample_camera_tracker` for a complete implementation.

```
dwFeatureTracker_initialize(...);
dwFeatureList_initialize(&list, ...);
dwFeatureList_initialize(&listClean, ...);
dwPyramid_initialize(&pyramid, ...);
dwPyramid_initialize(&pyramidOld, ...);
while(true)
{
    std::swap(pyramid, pyramidOld);
    // CODE: Get frame
    // CODE: Extract luminance channel
```

```
    dwPyramid_build(..., pyramid);
    dwFeatureTracker_trackFeatures(list, pyramidOld, pyramid, ...);

    // Discard unwanted features
    dwFeatureList_proximityFilter(list);
    dwFeatureList_selectValid(..., list);
    dwFeatureList_compact(listClean, list, ...)
    std::swap(list, listClean);

    dwFeatureTracker_detectNewFeatures(list, pyramid, ...);
}
dwPyramid_release(&pyramidOld, ...);
dwPyramid_release(&pyramid, ...);
dwFeatureList_release(&listClean, ...);
dwFeatureList_release(&list, ...);
dwFeatureTracker_release(...);
```

2D Scaling Tracker Module

The 2D Scaling Tracker module tracks scaling features between frames recorded by one camera. Scaling feature contains both position and size information. A 2D bounding box is considered a scaling feature. The module supports scaling features up to 128x128. For those features with larger size, it will use the center 128x128 subregion for prediction. The module only does tracking work, user need to do detection by the help of other DW modules.

Besides tracking the module comes with functionality to manage lists of 2D scaling features. All functionality is implemented as CUDA kernels and runs asynchronously on the GPU.

Scaling Feature Tracker

The user defines an upper bound on the number of scaling features during initialization. This defines the size of allocations inside the scaling tracker module. The runtime of the scaling tracker can still vary per the actual number of scaling features.

The tracker doesn't do detections, it only removes the features failing to be tracked. If there are new features, user must update scaling feature list themselves.

To track 2D scaling feature points between two frames

(Assume the scaling feature list is updated)

`dwScalingFeatureTracker_trackAsync` takes as input the target frame to be tracked and list of 2D scaling feature points. The output is the location, the size, the scale change factor and status for each input scaling feature, e.g. if the feature could be tracked successfully or not.

`dwFeatureTracker_updateTemplateAsync` takes as input the tracked scaling features and the new template image to be tracked from. To track scaling features from Frame N-1 to Frame N, Frame N-1 is

the template frame while Frame N is the target frame. `dwScalingFeatureTracker_trackAsync` and `dwScalingFeatureTracker_trackAsync` must be called by pair to ensure template updating.

Scaling Feature Lists

Scaling Feature lists manage an ordered list of 2d scaling features. Each feature point has a 2D location, its 2D size, and status flag indicating if the feature has been successfully tracked or not. It also provides the scale factor to indicate the change of size and the template location/size information.

Besides storing the scaling features and serving as I/O to the tracker, the scaling feature list also comes with basic housekeeping functionality. Although the scaling tracker supports feature size larger than 128x128 by selecting the center part, it may lose precision slightly for large feature size. To avoid the too large scaling features, `dwScalingFeatureList_applySizeFilter` will mark all features with size larger than the given value as invalid.

To add new scaling features to the list, `dwScalingFeatureList_addEmptyFeatures` must be called so that the new added features can be assigned with correct initial properties automatically. Input is the number of new features to be added.

To remove scaling features from the list that have a status indicating that they were not successfully tracked the combination of `dwScalingFeatureList_selectValid` and `dwScalingFeatureList_compact` can be used. The output is a compacted scaling feature list with only valid features in the input list.

Putting It All Together

The following code snippet shows the general structure of a program that uses the 2D scaling tracker to track scaling features in a single camera. For a complete implementation, see the `sample_camera_scaling_tracker`.

```
dwScalingFeatureTracker_initialize(...);
dwScalingFeatureList_initialize(&list, ...);
while(true)
{
    // CODE: Get frame
    // CODE: Extract luminance channel
    // CODE: Update scaling feature list for new features
    dwScalingFeatureTracker_trackAsync(list, ...);
    dwScalingFeatureTracker_updateTemplateAsync(list, ...)

    // Discard unwanted features
    dwScalingFeatureList_applySizeFilter(list);
    dwFeatureList_selectValid(..., list);
    dwFeatureList_compact(listClean, list, ...)
    std::swap(list, listClean);

}
```

```
dwScalingFeatureList_release(&list, ...);
dwScalingFeatureTracker_release(...);
```

Box Tracker Module

2D Box Tracker module tracks rigid objects for a limited duration. The tracker contains a rich set of configuration parameters that address different tracking scenarios. To successfully track objects, you can modify parameters with contextual information and apply additional constraints for their specific application setup.

Initialization

DriveWorks includes a sample pipeline demonstrating tracking of multiple cars across multiple video frames. The following summarizes the sample's initialization steps:

```
std::vector<dwBox2D> initBoxes;
dwBoxTracker2DHandle_t boxTracker;

// CODE: initialize the Box Tracker parameter
// CODE: initialize the Box Tracker
// CODE: allocate initBoxes
// CODE: fill initBoxes with detected bounding boxes

// Feed initBoxes to Box Tracker
dwBoxTracker2D_add(initBoxes.data(), initBoxes.size(), boxTracker);
```

Initializing the Box Tracking Parameters

The DriveWorks `dwBoxTracker2D_initParams` function initializes the Box Tracker parameter with default values. You can revise the tracker parameters for their specific use case. For a definition of the `dwBoxTracker2DParams` fields, see *DriveWorks API Reference*.

Initializing the Box Tracker

After you have customized the tracking parameters, you must call the `dwBoxTracker2D_initialize` function to initialize the Box Tracker internal states and memory.

Initializing Initial 2D Object Locations

The Box Tracker module requires initial 2D object locations represented by 2D rectangles. You can feed detection results to the Box Tracker.

Applying Clustering

DriveWorks provides two methods for applying clustering to reduce redundant/adjacent "raw" detection results:

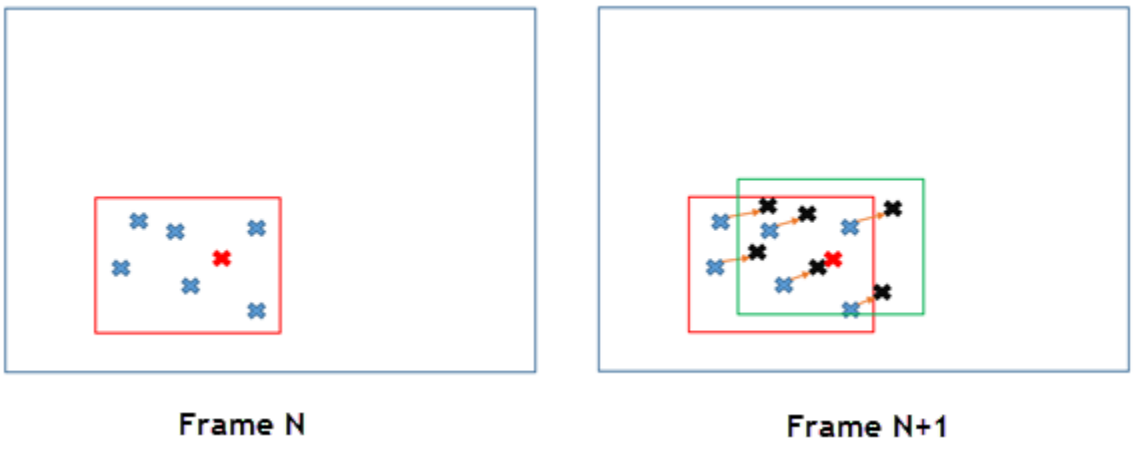
- With `dwBoxTracker2D_add`, which is internal to the Box Tracker module

- With `dwBoxTracker2D_addPreClustered`, which applies clustering outside Box Tracker. This function assumes you first cluster the bounding boxes and add it to Box Tracker afterwards.
- Manually specify the object locations to start with. For example:

```
int32_t x, y;
int32_t width, height;
dwBox2D userBox;

// CODE: specify top left of the bounding box (x, y)
// CODE: specify width and height of the bounding box (width, height)

userBox.x = x;
userBox.y = y;

userBox.width  = width;
userBox.height = height;

dwBoxTracker2D_add(userBox, ...);
```

Process

Given the initial bounding boxes, the Box Tracker module computes and associates 2D image features in adjacent image frames. The module uses such feature displacement to estimate the bounding box motion in the next frame. In the figure below, blue, and black crosses represent the successfully tracked 2D image features from frame N to frame N+1. Red cross failed to find association in frame N+1 is thus not used to update the red rectangle in frame N+1. For more information, see 2D Tracker Module in this chapter.



To predict the bounding box locations in the current frame using the tracked 2D image features, call the `dwBoxTracker2D_track` function. Tracking features in multiple frames could suffer from failure association such that the tracked feature number decreases. To avoid such a degradation, the DriveWorks sample Box Tracker application detects 2D image feature for each frame, after previous frame features are tracked. In this way, there are new features available for the next frame to use. Once the 2D feature detection is complete,

Box Tracker updates each tracked bounding box with the detected 2D image features. The DriveWorks `dwBoxTracker2D_updateFeatures` function offers the update mechanism.

The Box Tracker sample shows how to use `dwBoxTracker2D_track` and `dwBoxTracker2D_updateFeatures`.

To obtain the tracked bounding boxes, call `dwBoxTracker2D_get`. That function returns information such as:

- List of 2D rectangles and tracking states, i.e. confidence
- Box ID
- Total number of tracked frames
- 2d feature locations used to predict the bounding boxes

Structure from Motion (SFM)

The Structure from Motion module reconstructs the 3D structure of the scene given a moving camera rig. This is achieved by means of triangulation, e.g. geometric reasoning based on optics and multiple observations over time. The assumption made is that of a static world with a moving observer, e.g. changes in observation are only due to motion of the car and not of changes in the 3D position of the feature itself.

The structure is reconstructed as a point cloud and a series of rig poses, i.e. a 3D location for each tracked feature and the rotations and translations of the camera rig with respect to a fixed world reference frame. The module requires as inputs a list of tracked feature points and an initial estimate for the rig pose at each time instant. These inputs can be generated by the 2D tracker and the Egomotion module respectively.

The reconstructor object (`dwReconstructorHandle`) provides three main functionalities: triangulating 3D points from 2D tracked features, refining the rig pose, and predicting the pixel locations of 3D points in future frames (see `dwReconstructor_triangulateFeatures`, `dwReconstructor_estimatePose`, and `dwReconstructor_predictFeaturePosition`).

Triangulation

Triangulating points is the first step of the algorithm. 2D features must be tracked over several frames until they are observed with a wide-enough baseline to provide a stable triangulation. With the `dwReconstructorConfig` structure, you specify a baseline suitable for your application.

Determining When the Baseline is Wide-Enough

SFM determines when there is a wide-enough baseline by waiting until several sequential frames are observed, each of which has a wide-enough baseline to provide a stable triangulation. An additional reprojection check ensures a reduced number of outliers.

A wide-enough baseline is not ensured for the entire rig (i.e. the `minRigDistance` parameter is not currently used) but the baseline is ensured for each feature being triangulated when you use a combination of `minNewObservationAngleRad` and `minTriangulationEntries`.

Rig distance is not a good measure for triangulation accuracy because far-away features require more distance between observations than near features. The algorithm uses the angle between optical rays as a measure instead. An observation is only added if the angle between the new observation and the observations in the history is above the threshold (`minNewObservationAngleRad`). Moreover, a feature is only triangulated once the number of observations is above a threshold (`minTriangulationEntries`). Thus, the effective minimum observation angle before triangulation can be approximated by

`minTriangulationEntries*minNewObservationAngleRad`. This ensures a good-enough baseline for triangulation.

Updating History

The reconstructor object keeps a running history of the tracked features and where they have been observed at different points in time. For every frame, you must update this history by calling `dwReconstructor_updateHistory`. The algorithm calculates the observation baseline and only adds entries to the history if they contribute information for triangulation.

Getting Triangulation Information

After updating the history, features can be triangulated by calling `dwReconstructor_triangulateFeatures`. The triangulation uses the internal history accumulated over the previous frames. Only features that have accumulated enough information are triangulated. Once a feature is triangulated, if an entry in the history is detected as an outlier the status for that feature is marked as `DW_FEATURE_STATUS_INVALID`. Triangulated points are returned as a 3D homogeneous point in world coordinates, where the fourth element is zero if the triangulation is invalid.

Pose Refinement

The SFM module requires an initial pose estimate to perform triangulation. The camera rig pose is provided as a `dwTransformation`, i.e. a 4x4 matrix composed of a 3D rotation and translation. The name of the pose argument denotes the direction of the transformation. For example, a pose called rig2World can be used to transform a point in rig coordinates to world coordinates:

$$x_w = T_{r2w} x_r$$

where the points are in 3D homogeneous coordinates.

This pose is usually provided through odometry measurements (e.g. using the Egomotion module). However, once enough features have been triangulated this initial pose estimate can be refined by calling `dwReconstructor_estimatePose`. This function optimizes the pose by minimizing the reprojection error of 3D points with regards to the tracked features.

Feature Prediction

Most 2D trackers can greatly benefit from a good prediction of where a previously seen feature will be in the current frame. The SFM module can predict the position of most features given an estimation of the camera rig's pose (see `dwReconstructor_predictFeaturePosition`). The module provides three types of feature prediction according to how much is known about the feature.

- Triangulated points are directly reprojected onto the image using the estimated rig's pose, the rig to camera transformation, and the camera intrinsics.
- Features without triangulation that are below the horizon are temporarily assumed to lie on the ground plane and predicted to move according to the corresponding plane-induced homography.
- Features without triangulation that are above the horizon are temporarily assumed to be very far away from the car. Thus, only the relative rotation between the rig's previous pose and its current pose is considered. The features are predicted to move according to the corresponding 3D rotation-induced homography.

Putting It All Together

The following code snippet shows the general structure of a program that uses the SFM module. See the sample_sfm for a complete implementation of an SFM-enabled application.

```
dwReconstructor_initialize(...);
while(true)
{
    // CODE: Get frame
    // CODE: Estimate camera rig pose through odometry
    dwReconstructor_predictFeaturePosition(...);

    for each camera
    {
        // CODE: Build pyramid
        // CODE: Track features
    }

    dwReconstructor_estimatePose(...);
    dwReconstructor_updateHistory(...);

    for each camera
    {
        dwReconstructor_triangulateFeatures(...);
    }

    // CODE: Select features to discard
    dwReconstructor_compactFeatureHistory(...);
    dwReconstructor_compactWorldPoints(...);
}
dwReconstructor_release(...);
```

Stereo Module

The Stereo Module rectifies a pair of stereo images acquired from a calibrated stereo camera and computes a disparity map. The module is agnostic of the device that acquired the images and can be used with DriveWorks if the images are presented as two `dwImageCUDA` objects.

Stereo Rectifier

The `dwStereoRectifierHandle_t` object requires a calibrated `dwCameraRigHandle_t`. The calibrated object contains the intrinsics and extrinsics of the stereo camera pair, corresponding to Left and Right, with the Left camera being the center of the rig in the extrinsics parameters. From this rig configuration, the Stereo module extracts:

- Two `dwCalibratedCameraHandle_t` objects, each representing a calibrated pinhole camera. Together, these objects represent the intrinsics.
- Two `dwTransformation` objects.

Together, these objects represent the extrinsics.

The `data` folder provides an example of a calibrated rig, as represented in `dwCameraRigHandle_t`. That example was derived from the KITTI dataset calibration. Such calibration was, in turn, derived from OpenCV stereo calibration.

Afterwards, `dwStereoRectifier` consumes two unrectified `dwImageCUDA` stereo pairs and outputs two rectified `dwImageCUDA` pairs and the optimal ROI. This ROI includes an area inside the images that has only valid pixels (no interpolated data). The `sample_stereo_rectifier` sample shows how to perform such rectification on a KITTI video pair. In the sample, horizontal lines are rendered to show how the pixels in both rectified images lay on the same horizontal line. Moreover, a call to `dwStereoRectifier_getCropROI` returns a `dwRect` representing the ROI. In the sample, this is just used to render a green rectangle that shows where the most valid region is found. The user, at that point, can use the `dwRect` and call `dwImageCUDA_mapToROI` to get another `dwImageCUDA` that maps only to that ROI.

Disparity Computation

The stereo disparity algorithm consumes a pair of `dwPyramidHandle_t` objects built from a pair of rectified `dwImageCUDA` objects. To initialize a `dwStereoHandle_t` object, only `dwStereoParams` is required. Afterwards, during runtime, new pyramids must be built for the frame that came from the camera (or from `dwStereoRectifier` in case the camera does not perform rectification implicitly). Afterwards, the pyramids are given to `dwStereo_computeDisparity`. This function computes the disparity based on the `dwStereoParams` that have been set. It is then possible to call the `dwStereo_getDisparity` function and get the disparity map of the desired side (`DW_STEREO_SIDE_LEFT` or `DW_STEREO_SIDE_RIGHT`) and call the `dwStereo_getConfidence` function, which works in the same way. The confidence contains 8-bit values, where:

- 0 represents an Occlusion
- 1 represents an invalid disparity
- Other values represent a validity score where 255 is maximum and 2 is minimum

An *occlusion* is a region of the image that is visible in one stereo image but not on the other one. An invalid disparity is found on a region that contains no information and whose value has no confidence. In the sample, it is possible to highlight such areas. To obtain a real Occlusion in the confidence map, both disparity maps must be computed. If only one is computed, then the occlusion can be roughly found among low confidence pixels in the confidence map. Choosing to compute a disparity map with lower resolution than the input can greatly improve the runtime and smoothness of the result, but it sacrifices precision around edges and range.

Deep Neural Network Modules

The DNNs are trained with AR0231-RCCB, but with a good training set, the DNNs must be independent of

camera type. Given a big dataset for training, the tiny details that differentiate the cameras are averaged out as noise.

DNN Module

The DNN module implements functionality to run inference using deep neural networks, which were generated with an NVIDIA® TensorRT™ optimization tool.

Initialization

Initialize DNN module with TensorRT

There are two ways of initializing DNN module with TensorRT.

- Use the following function to provide the path to a serialized TensorRT model file generated with TensorRT_optimization tool:

```
dwStatus dwDNN_initializeTensorRTFromFile(
    dwDNNHandle_t *network,
    dwContextHandle_t context,
    const char *modelFilename);
```

- Use the following function to provide a pointer to the memory block where the serialized TensorRT model is stored.

```
dwStatus dwDNN_initializeTensorRTFromMemory(
    dwDNNHandle_t *network,
    dwContextHandle_t context,
    const char *modelContent,
    uint32_t modelContentSize)
```

Inference

`dwDNN` module offers two functions for running inference.

DNN models usually have one input and one output. For these kinds of models, the following function can be used for simplicity:

```
DW_API_PUBLIC dwStatus dwDNN_inferSIO(
    float32_t *d_output,
    float32_t *d_input,
    dwDNNHandle_t network);
```

This function expects a pointer to linear device memory where the output of inference is stored, a pointer to linear device memory where the input to DNN is stored and the corresponding `dwDNN` handle which contains the network to run. Please note that output must be pre-allocated with the correct dimensions based on the neural network model.

Input to DNN is expected to have NxCxHxW layout, where N stands for batches, C for channels, H for height and W for width.

Moreover, `dwDNN` module provides a more generic function, with which it is possible to run networks with multiple inputs and/or multiple outputs:

```
dwStatus dwDNN_infer(float32_t **d_output, float32_t **d_input, dwDNNHandle_t network);
```

This function expects an array of pointers to linear device memory blocks where the outputs of inference is stored, an array of pointers where the inputs of inference are stored and the corresponding `dwDNN` handle which contains the network to run.

In order to be sure that the inputs and outputs are given in the correct order, it is recommended to place the input and output data in their corresponding arrays at the indices based on the names of the blobs as defined in network description. The following functions return these indices:

```
dwStatus dwDNN_getInputIndex(uint16_t *blobIndex,
    const char *blobName,
    dwDNNHandle_t network);

dwStatus dwDNN_getOutputIndex(uint16_t *blobIndex,
    const char *blobName,
    dwDNNHandle_t network);
```

Furthermore, the following functions return the number of required inputs and outputs:

```
dwStatus dwDNN_getInputBlobCount(uint16_t *count, dwDNNHandle_t network);

dwStatus dwDNN_getOutputBlobCount(uint16_t *count, dwDNNHandle_t network);
```

In addition, dimensions of inputs and outputs are available via:

```
dwStatus dwDNN_getInputSize(dwBlobSize *blobSize,
    uint16_t blobIndex,
    dwDNNHandle_t network);

dwStatus dwDNN_getOutputSize(dwBlobSize *blobSize,
    uint16_t blobIndex,
    dwDNNHandle_t network);
```

DNN Metadata

Each DNN usually requires a specific pre-processing configuration, and it might, therefore, be necessary to include this information together with the DNN.

DNN Metadata contains pre-processing information relevant to the loaded network. This is not a requirement, but can be provided by the user together with the network by placing a certain json file in the same folder as the network with an additional ".json" extension.

For example, if the network is in path "/home/dwUser/dwApp/data/myDetector.dnn", DNN module will look for "/home/dwUser/dwApp/data/myDetector.dnn.json" to load DNN Metadata from.

The json file must have the following format:

```
{
    "dataConditionerParams" : {

        "meanValue" : [0.0, 0.0, 0.0],
        "splitPlanes" : true,
        "pixelScaleCoefficient": 1.0,
        "ignoreAspectRatio" : false,
        "doPerPlaneMeanNormalization" : false
```

```
    },
    "tonemapType" : "none",
    "__comment": "tonemapType can be one of {none, agtm}"
}
```

If the json file in question is not present in the same folder as the network, DNN Metadata is filled with default values. The default parameters would look like this:

```
{
    "dataConditionerParams" : {
        "meanValue" : [0.0, 0.0, 0.0],
        "splitPlanes" : false,
        "pixelScaleCoefficient": 1.0,
        "ignoreAspectRatio" : false,
        "doPerPlaneMeanNormalization" : false
    },
    "tonemapType" : "none",
    "__comment": "tonemapType can be one of {none, agtm}"
}
```

Note that whether DNN Metadata is used is a decision in the application level. The Metadata can be acquired by calling:

```
dwStatus dwDNN_getMetaData(dwDNNMetaData *metaData, dwDNNHandle_t network);
```

Data Conditioner Module

The Data Conditioner module shall be used in support of the DNN module in order to make the input data format compatible with the network input format. This can be done on a batch of images having the same size in parallel. In addition, the Data Conditioner module provides several pre-processing functionalities that can be used to further modify the input images, e.g., pixel value scaling, mean image subtraction, etc.

Initialization

In order to initialize the Data Conditioner module, it is first required to initialize the Data Conditioner parameters

```
dwStatus dwDataConditioner_initParams(dwDataConditionerParams *dataConditionerParams);
```

`dwDataConditionerParams` permits to set the following parameters:

- `float32_t meanValue[DW_MAX_IMAGE_PLANES]`: mean value to be subtracted from each input image pixel. Default is the 0-vector. This shall be used if the network has been trained with mean-centered data.
- `dwImageCUDA *meanImage`: mean image to be subtracted from each input image. meanImage is expected to be float16 or float32. The pixel format is required to be R or RGBA with interleaved channels. The dimensions of the mean image must meet the dimensions of the network input. Default is the null pointer. This is an alternative to meanValue, if a specific mean image is to be subtracted. Note: if both meanValue and meanImage are provided, both values are subtracted.
- `dwBool splitPlanes`: flag to indicate whether the image is in interleaved (false) or planar (true) format. Default is false.
- `float32_t scaleCoefficient`: Scale pixel intensities. Default is 1.0. It shall be used if the network

has been trained with images whose pixel values has been scaled to a specified range, e.g., in [0, 1]. If scaleCoefficient is 1.0, the output pixel intensities are always ranged between [0,255], regardless of the input pixel intensity range.

- `dwBool ignoreAspectRatio`: Indicates whether aspect ratio must be ignored during scaling operation. Default is false.
- `dwBool doPerPlaneMeanNormalization`: Indicates whether per-plane mean normalization must be performed. If true, mean value is computed for each plane on the image and this value is subtracted from pixel intensities of the corresponding plane.

Once the Data Conditioner parameters have been defined, the Data Conditioner object can be initialized with

```
dwStatus dwDataConditioner_initialize(dwDataConditionerHandle_t *obj,
const dwBlobSize *networkInputBlobSize,
const dwDataConditionerParams *dataConditionerParams,
cudaStream_t stream, dwContextHandle_t ctx);
```

The initialization requires only the network input blob size, which can be given through `networkInputBlobSize`. The user shall modify the batchsize in `networkInputBlobSize` to allow for a batch of images to be prepared in parallel.

Data Preparation

In order to actually perform the data preparation, i.e., to make the input data compatible with the network input and to apply the desired transformations, the following function must be called before calling the inference function of the DNN module

```
dwStatus dwDataConditioner_prepareData(float32_t *d_outputImage,
    const dwImageCUDA *const *inputImages,
    uint32_t numImages,
    const dwRect *roi,
    cudaTextureAddressMode addressMode,
    dwDataConditionerHandle_t obj)
```

`inputImages` contains pointers to the images of the batch that shall be prepared. The number of images in `inputImages` is given through `numImages` and shall not exceed the (possibly modified) batch size of the network. The `roi` parameter (region of interest) defines a specific region in all images to which the desired transformations as well as network inference shall be applied. `roi` is identified by the coordinates of the top left corner, and the width and the height of the rectangle. If full images are of interest, then the top left corner must be set as (0,0), width and height according to the images at hand. The internal resizing of `roi` to match the network input size is defined in such a way that no image content is lost, but undefined border values might be created. `addressMode` shall be used to set the fill-up strategy for the undefined border values. Two modes are allowed `cudaAddressModeBorder` and `cudaAddressModeClamp`.

In a nutshell, `dwDataConditioner_prepareData` crops and resizes the defined ROI from each input image in order to match the network input size, applies the desired transformations, and returns the transformed image batch in `d_outputImage`, which can be used as input to `dwDNN_infer` (see DNN Module Inference).

Finally, Data Conditioner transforms point coordinates, e.g., bounding box coordinates from the DNN inference, from the network input coordinate frame back to the original image coordinate frame by calling the function:

```
dwStatus dwDataConditioner_outputPositionToInput(float32_t *outputX,
    float32_t *outputY,
    float32_t inputX,
    float32_t inputY,
    const dwRect *roi,
```

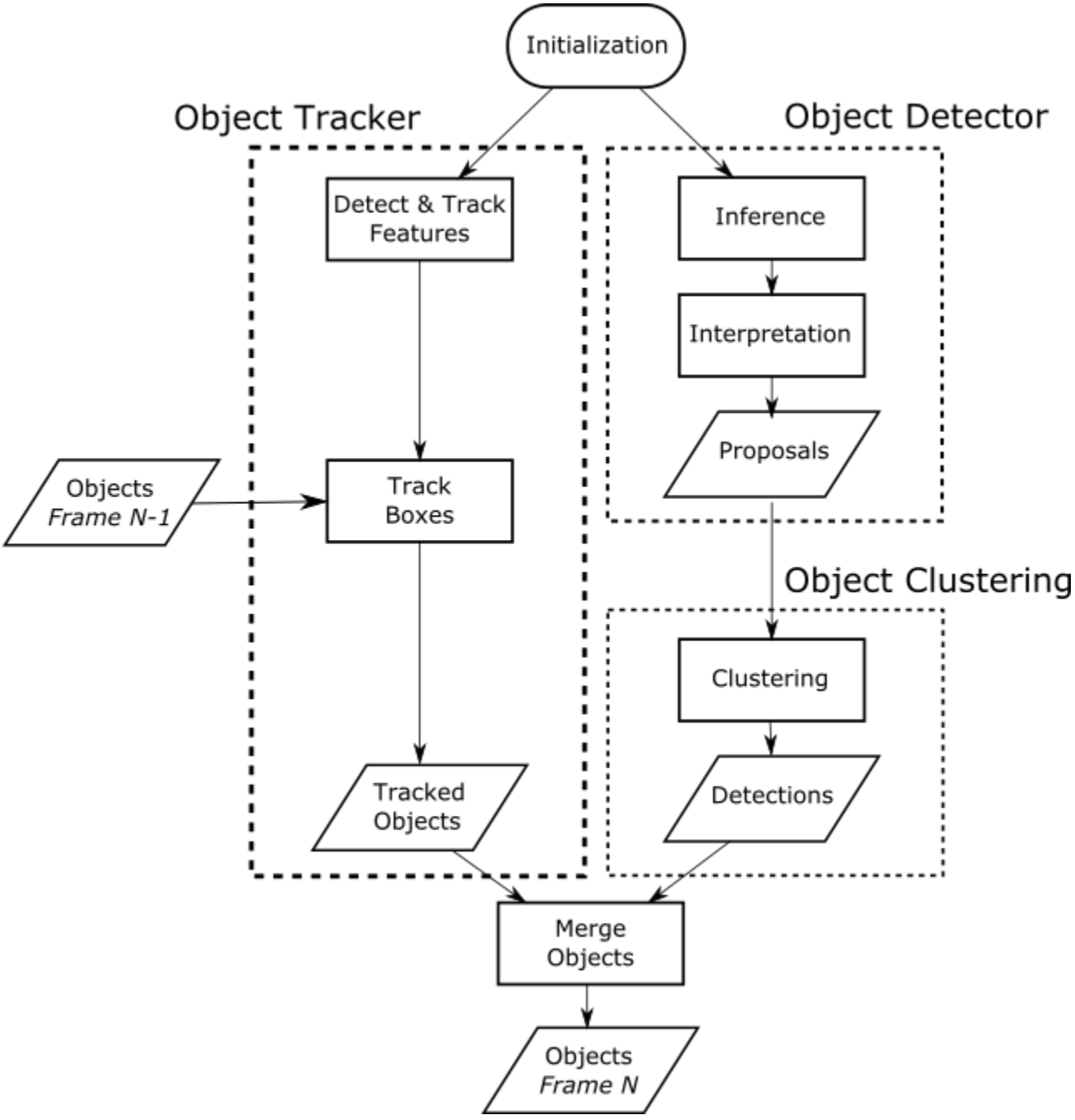
```
    dwDataConditionerHandle_t obj)
```

where the point to be transformed back is passed through `inputX` and `inputY` and is returned in `outputX` and `outputY`. The same `roi` passed to d`wDataConditioner_prepareData` shall be used.

Object Modules

Object module brings several functionalities together to provide the possibility to build a generic pipeline for detecting and tracking objects of several classes. Object module in DriveWorks consists of the following sub-modules: `dwObject`, `dwObjectDetector`, `dwObjectTracker`, `dwObjectClustering` and `dwDriveNet`. Each of these modules are explained below with example code snippets, and finally the integration of these modules to build an object detection application is shown.

The following diagram shows the roles of the object sub-modules in the object detection pipeline:



Object Detector

The Object Detector module implements functionality to load a detection network, apply transformations to the input such that the input image has the correct form for the loaded network, run inference using the loaded network and interpret output of the network to finally get the list of object proposals for the given image.

There are two ways to initialize an Object Detector module:

- Use a custom network or
- Use DriveNet.

Initialization with a Custom Network

In order to initialize Object Detector module, a DNN module must be initialized first with a TensorRT model. Moreover, Object Detector module requires some parameters to be defined so as to allocate enough resources, establish the pipeline for image preparation for the given DNN and to understand how to interpret the output of the network.

There are two different sets of parameters to be defined for ObjectDetector module:

- One group of parameters can be categorized under Detector parameters that specify where in an image and how to apply object detection.
- The other group is more relevant to the network's output interpretation:

```
dwStatus dwObjectDetector_initialize(
    dwObjectDetectorHandle_t *obj,
    dwContextHandle_t ctx,
    dwDNNHandle_t dnn,
    const dwObjectDetectorDNNParams *dnnParams,
    const dwObjectDetectorParams *detectorParams)
```

dwObjectDetectorParams: Detector Parameters

Below are the parameters to be defined:

```
uint32_t maxNumImages
```

Object Detector can be run on multiple images either sequentially or in parallel based on the batch size of the DNN. "maxNumImages" specifies the maximum number of images to be processed at one function call.

```
dwRect ROIs[DW_OBJECT_DETECTOR_MAX_IMAGES]
```

Region of interests (ROIs) determine which part of image must be fed as an input to the DNN. Therefore, a ROI must be provided for each image.

```
dwTransformation2D transformations[DW_OBJECT_DETECTOR_MAX_IMAGES]
```

Regardless of what the ROI is, the bounding boxes coming out of the network are in the input image coordinate system. If desired, however, the bounding boxes can be transformed into another coordinate system by providing a 2D transformation matrix.

```
dwBool enableFuseObjects
```

This flag enables objects coming from all the images to be in the same list. If, for example, multiple images from the same scene are to be processed, objects coming from different images can be transformed with corresponding transformation matrices into a common object coordinate system and can be fused into a single list with this flag set to `DW_TRUE`.

```
dwBool enableBoundingBoxClipping
```

This flag enables clipping of bounding boxes which are at borders of the image. By default, it is set to `DW_TRUE`.

dwObjectDetectorDNNParams: DNN parameters

```
float32_t boxScaleX1, boxScaleY1, boxScaleX2, boxScaleY2
```

A 2D object detection network usually consists of two output blobs: a blob that stores coverages and a blob that stores bounding boxes.

A coverage blob is a grid, and each cell in this grid contains a score telling how confident it is that it is part of an object in question. Each cell in this grid corresponds to a region in the input image. The size of the region that a cell refers to can be calculated as follows:

```
CellWidth = ImageWidth / GridWidth
CellHeight = ImageHeight / GridHeight
```

Hence, the center of a cell in the input image can be calculated as:

```
CellImageX = CellWidth * CellX
CellImageY = CellWidth * CellY
```

A bounding box blob is a grid having same width and height as the coverage blob with 4 channels for each object class. The first two channels refer to x and y coordinates of the upper left corner, whereas the last two channels refer to x and y coordinates of the lower right corner of the bounding box. The coordinates are usually estimated with respect to the center of the cell. Therefore, to get the coordinates in the network's input image coordinate system, the following transformations are to be applied:

```
UpperLeftX = CellImageX + Bbox[CellX, CellY][0]
UpperLeftY = CellImageY + Bbox[CellX, CellY][1]
LowerLeftX = CellImageX + Bbox[CellX, CellY][2]
LowerLeftY = CellImageY + Bbox[CellX, CellY][3]
```

In some cases, during training, the bounding box coordinates with respect to the center of cells are scaled by a factor. Therefore, the more generic way of calculating the coordinates is as follows:

```
UpperLeftX = CellImageX + (Bbox[CellX, CellY][0] * boxScaleX1)
UpperLeftY = CellImageY + (Bbox[CellX, CellY][1] * boxScaleY1)
LowerLeftX = CellImageX + (Bbox[CellX, CellY][2] * boxScaleX2)
LowerLeftY = CellImageY + (Bbox[CellX, CellY][3] * boxScaleY2)


uint32_t maxProposalsPerClass
```

Maximum proposals per class can be set using this parameter. This allows the Object Detector module to pre-allocate resources to account for the maximum number of proposals that must be interpreted from the output of the network. Note that if this number is exceeded, the proposals are truncated at this number regardless of their detection score.

```
const char *coverageBlobName
const char *boundingBoxBlobName
```

These strings help Object Detector module to find which output blob refers to coverage blob and which output blob refers to bounding box blob.

```
const char *bottomVisBlobName;
const char *horizontalVisBlobName;
```

```
const char *orientationYawBlobName;
```

If the network is capable of predicting the orientation of an object, these blob names must determine which blobs refer to which property.

- Bottom Visibility Blob

  This blob predicts the degree of visibility of the bottom of an object. The value is 0.0 if the bottom of the object is completely occluded and 1.0 if the bottom of the object is completely visible.

- Horizontal Visibility Blob

  This blob predicts the degree of horizontal visibility of an object. The value is 0.0 if the object is completely occluded from left or right side and 1.0 if the object is completely visible.

- Yaw Orientation Blob

  This blob predicts the yaw orientation of an object. The value is in [-Π, Π] from lateral axis. For example:

  - yaw orientation is 0 if object faces the same direction as the host car.
  - yaw orientation is +-Πif object faces towards the camera.
  - yaw orientation is Π/2if object faces the rights side.

```
dwBool enable2_5D
```

This flag enables prediction of object orientation and visibility. This requires the network to provide all the blobs mentioned above.

```
float32_t coverageThreshold[DW_OBJECT_MAX_CLASSES]
```

For each class, it is possible to define a coverage threshold which is utilized to eliminate the object proposals that have lower coverage score than the given threshold.

dwDataConditionerParams dataConditionerParams

Data conditioner parameters specify network specific transformations to be applied on each image before inference. For more information, see Data Conditioner Module.

Initialization with DriveNet

Object Detector module can be initialized with the NVIDIA proprietary deep neural network to perform object detection:

```
dwStatus dwObjectDetector_initializeFromDriveNet(
dwObjectDetectorHandle_t *obj, dwContextHandle_t ctx, dwDriveNetHandle_t drivenet, cons
```

This kind of initialization requires dwDriveNet handle. Please see DriveNet Module for more details.

As opposed to initialization with a custom network, it requires only detector parameters as input. For more information on dwObjectDetectorParams, see Initialization with a Custom Network in this section.

Inference - Object Detection

At first, the input image must be prepared for DNN and inference must be run on the given images. To achieve this, call:

```
dwStatus dwObjectDetector_inferDeviceAsync(const dwImageCUDA *const *imageArray,
    uint32_t numImages,
    dwObjectDetectorHandle_t obj);
```

`imageArray` contains pointers to the images where the Object Detection will take place. The number of images in the `imageArray` may not exceed the maximum number of images set during initialization.

This function is run on device. Run this function asynchronously if a cudaStream has been set using `dwObjectDetector_setCudaStream` function.

After the inference has been accomplished, as mentioned earlier, Object Detector module interprets the output of the network on host by the following function:

```
dwStatus dwObjectDetector_interpretHost(uint32_t numImages,
    dwObjectDetectorHandle_t obj);
```

`numImages` is the number of images that was provided during dwObjectDetector_inferDeviceAsync. This function synchronizes the NVIDIA® CUDA® stream that the inference takes place in.

Once the interpretation is done, object proposals can be received for each class in each image using the following function:

```
dwStatus dwObjectDetector_getDetectedObjects(dwObject *objectList,
    size_t *numObjects,
    uint32_t imageIdx,
    uint32_t classIdx,
    dwObjectDetectorHandle_t obj);
```

`objectList` must be allocated by the user with enough memory to store the maximum number of proposals. This number can be obtained from `dwObjectDetectorDNNParams::maxProposalsPerClass` or `dwDriveNetDNNParams::maxProposalsPerClass` provided at initialization.

Object Clustering

A single object might cause DNN to result in several object proposals around the same area. Although these all refer to the same object, during the interpretation of DNN, this is not known. Object Clustering Module implements the functionality to apply clustering of objects proposals based on their bounding boxes and user-defined clustering parameters in order that only one bounding box assigned to a single object.

Initialization

Initialization of Object Clustering is achieved by the following function call:

```
dwStatus dwObjectClustering_initialize(dwObjectClusteringHandle_t *obj,
    dwContextHandle_t ctx,
    const dwObjectClusteringParams *clusteringParams)
```

The initialization of Object Clustering module requires the following clustering parameters to be defined.

```
float32_t epsilon;
```

Like in any clustering methods, in Object Clustering module, there are bounding boxes which are defined to be “core” which represent the cluster. Epsilon is the maximum distance for a box to be considered an element of a cluster to the core of that cluster.

```
uint32_t minBoxes;
```

Defines the minimum number of boxes required to form a dense region. `minBoxes` and `minSumOfConfidences` are checked conjunctively.

```
float32_t minSumOfConfidences
```

Defines the minimum sum of confidences required to form a dense region. `minBoxesandminSumOfConfidences` are checked conjunctively.

```
uint32_t maxProposals;
```

This parameter defines the maximum number of proposals the Object Clustering module should expect as input at once. This allows the module to pre-allocate resources at initialization time.

```
uint32_t maxClusters;
```

This parameter defines the maximum number of clusters the Object Clustering module should output. If the number of clusters exceeds this parameters, regardless of the cluster's confidence, the list is truncated to meet this requirement.

```
dwObjectClusteringAlgorithm algorithm;
```

Currently, there is one sort of clustering algorithm available. `DW_CLUSTERING_DBSCAN`, as the name suggests, applies DBSCAN clustering algorithm on the object proposals, where the distance between two objects is computed as 1.0-"intersection over union" of their bounding boxes; i.e. smaller IOU is larger distance.

```
dwBool enableATHRFilter;
float32_t thresholdATHR;
```

Area-to-hit-ratio is used to filter out object clusters which are, based on `thresholdATHR`, unlikely to be objects if `enableATHRFilter` is set to `DW_TRUE`.

Area-to-hit-ratio is computed with the following formula:

```
ATHR = sqrt(area(cluster.box)) / nClusterMembers
```

```
dwBool enable2_5D
```

This flag indicates whether 2.5D fields (Horizontal Visibility, Bottom Visibility and Yaw Orientation) must be taken into account during clustering.

```
float32_t minHeight
```

MinHeight is used to filter out clusters having less height than the indicated value.

Clustering

Object clustering is achieved by the following function call:

```
dwStatus dwObjectClustering_cluster(dwObject *clusters, size_t *numClusters,
```

```
    const dwObject *detections,
    size_t numDetections,
    dwObjectClusteringHandle_t obj);
```

The `clusters` argument points to an array of clusters which is allocated by the user having the size of maxClusters elements. This array is filled up by the ObjectClustering module after the clustering is performed. “numClusters” is set to the number of actual clusters as a result of clustering algorithm.

The `detections` argument points to an array of object proposals with size `numDetections` that are to be clustered.

Object Tracker

The Object Tracker module implements the functionality to track features in a frame using 2D Tracker module and using these features to track bounding boxes of the objects from one frame to another.

Initialization

The following function call initializes the Object Tracker module:

```
dwStatus dwObjectTracker_initialize(dwObjectTrackerHandle_t *obj,
    dwContextHandle_t ctx,
    const dwImageProperties *imageProperties,
    const dwObjectFeatureTrackerParams *featureTrackerParams,
    const dwObjectTrackerParams *objectTrackerParamsArray,
    uint32_t numClasses);
```

Initialization of Object Tracker module requires to set up a certain list of parameters which define the way the features are generated and tracked, and how the bounding boxes are updated, whether they must be kept or lost.

`dwObjectFeatureTrackerParams` contain the feature generation and tracking parameters that are already explained in 2D Tracker Module. Below are the explanations of the parameters contained in `dwObjectTrackerParams`:

```
uint32_t maxFeatureCountPerBox
```

This parameter determines the maximum number of features to track for each bounding box. The rest of the features inside that bounding box are ignored.

```
float32_t maxBoxImageScale
```

With this parameter, it is possible to eliminate objects that get too large throughout their lifetime. The objects that have larger width or height than the maximum allowed are dropped. Maximum allowed width and height of the object are:

```
maxWidth = maxBoxImageScale * imageWidth

maxHeight = maxBoxImageScale * imageHeight

float32_t minBoxImageScale
```

With this parameter, it is possible to eliminate objects that get too small throughout their lifetime. The objects that have smaller width than the minimum allowed are dropped. Minimum allowed width of the object is:

```
minWidth = minBoxImageScale * imageWidth
```

```
minHeight = minBoxImageScale * imageHeight
float32_t confRateDetect
```

During the lifetime of an object, if the object is detected again, the confidence of the object must be incremented. How much the confidence of the new detection should affect the confidence of the object is determined by this parameter in the following way:

```
conf = conf + conf_detection * confRateDetect
float32_t confRateTrackMax
float32_t confRateTrackMin
```

During the lifetime of an object, as the object is not being detected, it is becoming less certain that it is an actual object of interest. To represent this situation, the confidence of the object is decreased each frame that the object is not detected. The confidence update is as follows:

```
confRate = confRateTrackMin +
  (confRateTrackMax - confRateTrackMin) *
  (1.0 - sqrt(featureCount/maxFeatureCount))
conf = conf - confRate
```

Therefore, `confRateTrack` is expected to be a negative value.

```
float32_t confThreshDiscard
```

With this parameter, it is possible to eliminate objects that get too low in confidence during their lifetime. If the confidence is lower than confThreshDiscard, the object is no longer tracked.

```
uint32_t maxNumObjects
```

This parameter defines the maximum number of objects the Object Tracker module should track. This allows the module to pre-allocate resources at initialization time. If number of objects exceed this number, the object list is truncated.

```
dwBool enablePriorityTracking
```

A single feature may belong to multiple bounding boxes. If enablePriorityTracking is set to `DW_TRUE`, however, it is possible to assign a feature only once (i.e. to a single bounding box) and render it as used so that it is not assigned to any other. The owner of the feature is the one with the lowest index in the object list given as input to `dwObjectTracker_boxTrackHost`.

Therefore, it is user's responsibility to sort the object list such that the object with the highest priority has the lowest index.

```
uint32_t numClasses
```

This parameter defines the number of classes of objects that are to be tracked by the Object Tracker module. The objects belonging to different classes are tracked separately. This parameter allows the module to pre-allocate resources at initialization time.

Tracking

In Object Tracker module, there are two possible ways to track the objects: either tracking based on features

that are detected and tracked within Object Tracker module or tracking based on features that are detected and tracked outside this module.

If the features must be detected and tracked within Object Tracker module, then it is required to call the following function on each frame before tracking bounding boxes:

```
dwStatus dwObjectTracker_featureTrackDeviceAsync(const dwImageCUDA *image,
    dwObjectTrackerHandle_t obj)
```

This function runs asynchronously on device.

The objects can then be tracked with the following function call:

```
dwStatus dwObjectTracker_boxTrackHost(dwObject *trackedDetections,
    size_t *numTrackedDetections,
    const dwObject *previousDetections,
    size_t numPreviousDetections,
    uint32_t classIdx,
    dwObjectTrackerHandle_t obj);
```

This function is provided with the objects from previous frame to get the tracked positions of the objects in the current frame using the features tracked inside Object Tracking module.

If it is desired that the objects must be tracked based on external features, the following function can be utilized:

```
dwStatus dwObjectTracker_boxTrackHostExternalFeatures(dwObject *trackedDetections,
    size_t *numTrackedDetections,
    const dwObject *previousDetections,
    size_t numPreviousDetections,
    const dwFeatureListPointers *previousFeatures,
    const dwFeatureListPointers *currentFeatures,
    uint32_t classIdx,
    dwObjectTrackerHandle_t obj);
```

This function not only expects the list of objects from previous frame, but also list of features from previous frame and current frame.

It is also required to provide `classIdx` so that the corresponding tracker parameters are used for the given set of objects.

Merging

The objects list being tracked from the previous frame and the objects that are detected in the current frame need to be merged so that the final list of objects for the current frame can be obtained. The merging takes place in such a way that if a new detection refers to an object that has been tracked from the previous frame, this new detection is merged into that object, and the object is then assumed to be “detected again”. If a new detection refers to a completely new object, it is simply added to the list iff the maximum number of objects is not exceeded.

The merging is achieved by the following function call:

```
dwStatus dwObject_merge(dwObject *mergedObjects,
    size_t *numMergedObjects, size_t maxNumMergedObjects,
    const dwObject * const* objectLists,
    const size_t *numObjectsPerList,
    size_t numLists,
    float32_t thresholdIOU,
```

```
    float32_t maxMatchDistance,
    dwContextHandle_t ctx)
```

This function expects a list of object lists which will be merged into a single list `mergedObjects`.

In this scenario, one of the object lists will be the list of objects that has been tracked from the previous frame and the other will be the list of objects that has just been detected. The decision whether a new detection refers to the same object as one of the tracked objects is made based on the following two parameters:

```
float32_t thresholdIOU
```

If the intersection over union (IOU) value of two bounding boxes (e.g. tracked box and detected box) is higher than this threshold, these two bounding boxes are assumed to refer to the same object.

```
float32_t maxMatchDistance
```

Maximum match distance around the closest tracked box to search for a candidate matching box. Distance here is defined as 1 – IOU. Within this margin, the box with the longest track history is preferred and will be selected as the candidate matching box. The candidate still must pass the thresholdIOU criteria to be considered a positive match for the new box.

DriveNet

The DriveNet module is a DNN module that automatically loads NVIDIA proprietary deep neural network for object detection, which is provided to Object Detector module. For more information on how to provide DriveNet handle to Object Detector module, see Object Detector in this guide.

Initialization

DriveNet module not only creates a DriveNet handle, but also creates a list of Object Clustering handles that are properly configured for this specific deep neural network. These handles must be used for clustering the detections to get the best results.

Note that these handles are owned by the application and must be released by the application when they are not needed anymore.

The following parameters are required to be defined at initialization:

```
uint32_t maxProposalsPerClass
```

Maximum number of proposals (i.e. detections before clustering) per class. This value can only be set at initialization.

```
uint32_t maxClustersPerClass
```

Maximum number of clusters per class. This value can only be set at initialization.

```
dwDriveNetNetworkPrecision networkPrecision
```

This parameter determines at which precision DriveNet must be loaded. This parameter can affect the speed of the network greatly.

DriveNet module can be initialized with the following function:

```
dwStatus dwDriveNet_initialize(
```

```
dwDriveNetHandle_t *drivenetHandle,
    dwObjectClusteringHandle_t **objectClusteringHandles,
    const dwDriveNetClass **objectClasses,
    uint32_t *numObjectClasses,
    dwContextHandle_t ctx,
    const dwDriveNetParams *drivenetParams)
```

DriveNet provides at initialization as outputs an Object Clustering handle per object class, number of object classes that DriveNet can detect and the list of object classes that DriveNet can detect. The object classes are returned as dwDriveNetClass. Upon desire, the classes can be obtained as string via dwDriveNet_getClassLabel.

DriveNet comes with meta data which contains preprocessing configuration for the network. ObjectDetector module automatically reads this meta data information and configures DataConditioner accordingly. However, tone mapping type required by DriveNet module can be obtained from this meta data to set up SoftISP accordingly to get the best results.

Lane Detection

The lane detection module streams a H.264 video and detects lane markings on the road. The pipeline of this module includes loading NVIDIA proprietary lane detection network called LaneNet, running inference using the loaded network, post-processing the network output to get an array of lane marking objects, and finally rendering the detected lane markings on the input frame.

The detected lane markings are visualized by polylines in different colors, which annotate their position relative to the lane vehicle is driving in (ego-lane). Specifically, the defined position types include ego-lane left, ego-lane right, left adjacent lane, and right adjacent lane. Note that, in current release, LaneNet is not trained to distinguish different lane marking appearance types.

The module uses 60° Field of View (FOV) Sekonix Camera Module (SS3323) with an AR0231 RCCB sensor to record videos. To use as an input to the sample, the recording is converted into a H.264 video in RCB color space. Results have shown LaneNet also performs well on H.264 RGB videos.

Note:	**Lane detection sample directly resizes video frames to the network input resolution. To get the best performance, it is suggested to use videos with similar aspect ratio to the demo video (960×604). For other details, please refer to the camera sensor section.**

Initialization

A LaneNet based lane detection module is initialized with the following function call:

```
dwStatus dwLaneDetector_initializeLaneNet(dwLaneDetectorHandle_t
*obj,
    uint32_t frameWidth,
    uint32_t frameHeight,
    dwContextHandle_t ctx);
```

The initialization of lane detection module requires the designation of camera frame width and height.

Process

After successfully initialized a lane detection module, a camera frame is processed in 2 sequential steps: inference and post-processing. The inference step pre-processes a camera frame per LaneNet input specs and runs per

frame inference to compute the likelihood map of lane markings. This is performed asynchronously on GPU by calling the `dwLaneDetector_processDeviceAsync` function.

To extract lane markings from the network output, call `dwLaneDetector_interpretHost` after `dwLaneDetector_processDeviceAsync`. This function is run on CPU and post-processes input likelihood map into individual lane markings defined by `dwLaneMarking` as sets of sorted image points. More specifically, `dwLaneDetector_interpretHost` binarizes a likelihood map into clusters of lane markings, employs several image processing steps to sample points from lane clusters, and assigns them with lane position types. `dwLaneDetector_getLaneDetections` gets the per frame lane detection results in `dwLaneDetection`.

You can set LaneNet detection threshold by `dwLaneDetectorLaneNet_setDetectionThreshold`. Any likelihood value above the threshold is considered a lane marking pixel. By default, the value is 0.3, which provides the best accuracy based on NVidia's test data set. Reduce the threshold value if lane polylines flicker or cover shorter distances but may subject to more false detections.

Currently, the lane detection module is optimized for 60° and 120° horizontal FOV cameras. By default, LaneNet FOV is 60-degrees. You can set other FOV values with `dwLaneDetectorLaneNet_setHorizontalFOV`.

Tools

This section covers the tools provided with the DriveWorks release It is not a manual for each tool but some introductions to what the tool does, etc.

Recording Tools

DriveWorks provides a command-line and GUI recording tool that save data to disk files. Both tools use an underlying recording tool library. For practical help on recording data, see Data Acquisition in this guide.

Recording Tool Library

The recording library is capable to handling most supported DriveWorks sensors, excluding virtual sensors. For identifying the list of supported sensors, see the Sensor Querying section of this Development Guide.

The DriveWorks recording tool library is configurable with a JSON file.

A sample JSON configuration file is given below, and explanations of some key fields is annotated inline.

The JSON file is written this way.

```
{
    "version": "0.8",
```

This is the version number of the config file. If this version number differs from the in-code version number, the recorder will throw an exception.

```
   "path": ".",
```

This is where the recording tool stores captured data. Each time a new capture session is invoked, a new folder is created at this path, with the time and location (if available) in the folder name.

```
   "file-buffer-size": 2097152,
```

This is the size of the write buffer used in conjunction with the `fwrite()` API. The buffer size dictates how much data is accumulated before being flushed to disk. It can be useful in tuning CPU usage during recording.

```
   "camera": {
        "separate-thread": true,
        "record-thread-priority": 0,
        "write-file-pattern": "video_*",
        "sensors": [
          {
                "protocol": "camera.gmsl",
                "params": "camera-type=ar0231-rccb,csi-port=ab,camera-count=4",
                "channel-names": [
```

```
                                    "first",
                                    "second",
                                    "third",
                                    "fourth"
                                ]
            }
        ]
    },
```

Each sensor group is set up identically. The following are important parameters common to every sensor type:

1. **sensors** - Multiple sensors are added by creating a new object in the sensors array section of each. Each sensor is described by the DriveWorks protocol and parameters string. For more information, see Sensor Querying in this guide.
2. **separate-thread** - Dictates whether a new thread will spawn for this sensor group. If false, this sensor group will share a capture thread with other sensor groups.
3. **record-thread-priority** - Specifies the priority of the recording thread. The range of this value is [0, 20] and directly corresponds to Linux nice values for thread scheduling.
4. **write-file-pattern** - Indicates the prefix for the recorded files in this group. This string must end with an asterisk, which is replaced by the channel number or channel-name of the sensor being recorded.
5. **channel-names** - Specifies the name to be given for this sensor. For sensors that control multiple sources (such as camera), channel-names array can be specified. This will replace the * in the write-file-pattern when the file is saved to disk. If no channel-name is specified, the *in write-file-pattern is replaced with the channel number.

Apart from the common parameters above, camera sensors have additional specific parameters specified within the DriveWorks parameter string:

1. **output-format** - specifies the output format of the recorded video file. Can be one of the following:
   a. raw - RAW format
   b. lraw - Lossless RAW format
   c. h264 - H.264 encoded format
   d. raw+h264 - RAW + H.264 encoded format
   e. lraw+h264 - Lossless RAW + H.264 encoded format
2. **frame-skip-count** - allows one to record camera frames at a lower rate than default. For example, a value of frame-skip-count=1 would record at 15fps if the camera hardware runs at 30fps.
3. **required-framerate** - for certain kinds of camera, e.g. ar0231-rccb, the camera framerate can be controlled by required-framerate option. Valid values for now are 20, 30 and 36.
4. **async-record** - enables the camera to record asynchronously, on a separate thread than the camera was read from.

```
"can" : {
     "separate-thread": false,
     "write-file-pattern": "can_*.bin",
```

```
        "sensors" : [
            {
                "protocol": "can.socket",
                "params": "",
                "channel-name": "0"
            }
        ]
    },

"gps" : {
        "separate-thread": false,
        "write-file-pattern": "gps_*.txt",
        "sensors" : [
            {
              "protocol": "gps.uart",
              "params": "device=/dev/ttyACM0",
              "channel-name": "0"
            }
        ]
    },
"imu" : {
        "separate-thread": false,
        "write-file-pattern": "imu_*.bin",
        "sensors" : [
            {
                "protocol": "imu.xsens",
                "params": "device=X,frequency=100",
                "channel-name": "0"
            }
```

]

```
    },

"lidar" : {
        "separate-thread": false,
        "write-file-pattern": "lidar_*.bin",
        "sensors" : [
```

```
            {
                "protocol": "lidar.virtual",
                "params": "file=./lidar_sample.bin",
                "channel-name": "0"
            }
        ]
    }

"radar" : {
     "write-file-pattern": "radar_*",
     "sensors" : [
             {
             "protocol": "radar.virtual",
             "params": "file=./radar_sample.bin",
             "channel-name": "0"
             }
     ]
 }
```

The sensors above are described by the common parameters described earlier.

Command Line Recording Tool

Running the Tool

For guidance on preparing to run and running the GUI-based recording tool, see Data Acquisition in this guide.

Command-Line Options

The DriveWorks command line recording tool is configurable with a JSON file. Additionally, it can be started offscreen for command-line only use with the

```
./recorder --offscreen=1
```

option.

By default, the JSON file that is read into the tool is `recorder-config.json`. This file is generated when the tool is run. A different file can be specified with the

```
./recorder --config-file=<config-file-path>
```

command-line option.

For information about the JSON config file description, see Recording Tool Library in this guide.

Note: This tool creates output files that, per default, are put into the current working directory. Hence, write permissions to the current working directory are necessary. For convenience, NVIDIA suggests adding the tools folder to the binary search path of the system and to execute from your home directory.

GUI-Based Recording Tool

The GUI recording tool provides an interface that provides visual information about what is being recorded.

For information on using the interface and configuring distributed recording, see "DriveWorks Tools" in *DriveWorks SDK Reference*.

For guidance on preparing to run and running the GUI-based recording tool, see Data Acquisition in this guide.

Replayer Tool

The replayer tool provides the following:

- Replays the sensor data captured by Recording Tool, with the same JSON file for sensor configuration
- Displays the data for the first channel in each sensor type
- Simple UI for quick sanity check on recorded data

For more information on this tool, see "DriveWorks Tools" in *DriveWorks SDK Reference*.

Troubleshooting

DriveWorks Cannot Create Sensors

Issue

Replayer errors such as the following may be the result of problems with the JSON file.

```
Driveworks exception thrown: DW_FAILURE: LidarVirtual: unable to open Lidar file
Cannot create lidar sensor: DW_FAILURE
Driveworks exception thrown: DW_SAL_CANNOT_CREATE_SENSOR: CANVirtual: cannot open file
Cannot create CAN sensor: DW_SAL_CANNOT_CREATE_SENSOR
```

To troubleshoot

- Ensure the JSON file is valid for the recorder.
- Ensure that, for each sensor, the JSON file contains at least one `channel_names` definition.

```
"channel-names": [
    "[<channelName>]"
]
```

Where <channelName> is a value such as camera0. A comma (,) must separate "channel-names" from the previous definition.

Replayer uses the "channel-names" definition to form a valid filename.

- If you also supply replayer with the --show-camera option, ensure that your JSON file "channel-names" definition includes the specified camera channel.

Determine if one of the sensors is broken. You can do this by feeding each offline file to the corresponding DriveWork sensor sample. If one of the files fails, you know there is a problem with the sensor.

TensorRT Optimization Tool

This tool enables optimization of a given Caffe model using TensorRT.

Run this tool by executing:

```
s./tensorRT_optimization
```

Required Arguments

The following arguments are required:

- --prototxt: Absolute path to a Caffe deploy file.
- --caffemodel: Absolute path to a Caffe model file that contain weights.
- --outputBlobs: Names of output blobs combined with a comma.

Optional Arguments

- --iterations: Number of iterations to run to measure speed
- --batchSize: Batchsize of the model to be generated
- --half2: The network runs in paired fp16 mode. NOTE: Requires platform to support native fp16.
- --inputBlobs: Names of input blobs combined with a comma
- --out: Name of the optimized model file
- --int8: Run in INT8 mode
- --calib: INT8 calibration file name

Note: **This tool creates output files that, by default, are put into the current working directory. Hence, write permissions to the current working directory are necessary. For convenience, NVIDIA suggests that you:**

- Include the tools folder in the binary search path of the system.
- Execute from your home directory.

For more information about INT8 calibration, see NVIDIA TensorRT documentation at

https://developer.nvidia.com/tensorrt

Data Acquisition

The NVIDIA DRIVE™ PX 2 device supports data acquisitions from multiple sensors and other NVIDIA DRIVE PX 2 devices with other sensors. The data is then stored on a USB Drive or to another data storage device connected through Ethernet. The data from each sensor is synchronized with data from other sensors by adding a time stamp to the data.

The DriveWorks SDK contains sample applications to capture, synchronize, and play back the data captured from the sensors. You can use these sample applications to test the data logging capabilities and then use the included source code to create your own custom application.

Overview

The device acquires data from the connected sensors, processes the data, and then stores the data in a connected storage device.

The following shows the data acquisition capabilities:

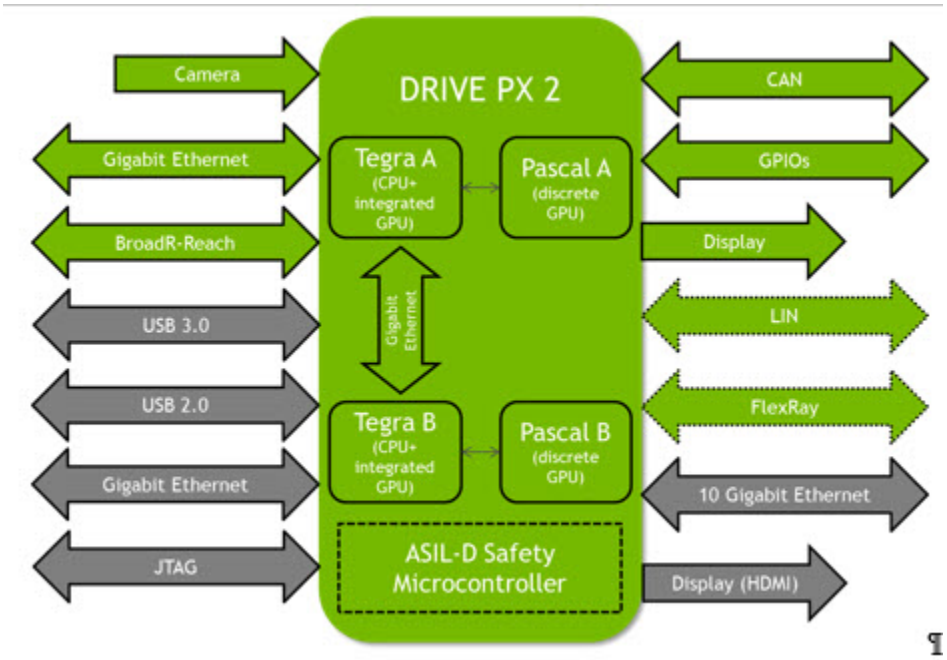
Each sensor acquires and transfers the data to one of the NVIDIA® Tegra® processors on the device. Depending on sensor type, the data may first go through a post-processing step (ex. encoding for camera) prior to disk serialization. It is important to ensure the storage device's bandwidth and capacity are sufficient for the sensor set.

Supported Sensors

For a list of the supported sensors, see Supported Sensors in this guide.

Supported Interfaces

The device supports the following interfaces:



Sensor Interfaces

- BroadR-Reach
- Controller Area Network (CAN)
- FlexRay
- General Purpose Input/Output (GPIO)
- Gigabit Ethernet (GbE)
- Gigabit Multimedia Serial Link (GMSL) Camera
- Local Interconnect Network (LIN)

Display and Cockpit Computer Interfaces

- FPDLink III
- GMSL
- HDMI

Development and Debug Interfaces

- 10 Gb and 1 Gb Ethernet
- HDMI
- Universal Serial Bus (USB 2 and USB 3)

For more information about flashing your device, see *NVIDIA DRIVE 5.0 Linux PDK Development Guide*.

Acquiring Data

The following are the steps to log data in the device.

Prerequisites

Before you can use the DriveWorks sample recorder or DriveWorks sample sensor applications, you must install DriveWorks on the NVIDIA DRIVE PX 2 platform. For more information, see Getting Started in this guide.

Step 1: Verify the Sensors Are Collecting Data

In this step, you will connect the sensor to the device and then verify that the sensor is collecting data.

Warning:	**Turn off the device before connecting or disconnecting a sensor.**

You can determine whether the sensor is correctly connected and collecting data by starting the appropriate sample application specified in the following table. The samples are on the target in:

```
/usr/local/driveworks/bin/
```

Sensor	Sample Application
Camera	sample_camera_gmsl
	sample_camera_multiple_gmsl
GPS	sample_gps_logger
Lidar	sample_lidar_replay
Radar	sample_radar_replay
IMU	sample_imu_logger
CAN	sample_canbus_logger

For more information about these applications, see *DriveWorks SDK Reference*.

Step 2: Configure the Device to Acquire Data

After verifying that the sensors are collecting data, you can configure the device to collect and store the data. The first time the recording tools run, they generate a JSON config file.

For `recorder`, the file is:

```
/driveworks/tools/recorder-config.json
```

For `recorder-qt`, the file is:

```
/driveworks/tools/recorder-qt-config.json
```

Modify and save the configuration file with the configuration settings for the sensors connected to the device. For more information, see Examples in this chapter.

Step 3: Start the Recording Application and Acquire the Data

After you modify the configuration file to collect and store the data, you can begin logging the data from the sensors. You can use the command-line or GUI tool:

- `tools/recorder`
- `tools/recorder-qt`

Both tools collect the data from the sensors, synchronizes the data by adding time stamps, and saves the data to the storage device.

Recording SocketCAN Data

If you are recording data from SocketCAN, you must start the `recorder` or `recorder-qt` tools with root privileges. For example:

```
# sudo tools/recorder-qt
```

The recorder tools use an IOCTL command to enable HW timestamping for the TegraCAN interface. Without the sudo preface, the recorder tools raise an error.

Examples

This section shows how to log data from the connected sensors. The applications can check that the sensors can collect data and saves the data to the Tegra System on Chip (SOC).

Prerequisites

To complete the procedures in this chapter, you must have the following:

- NVIDIA DRIVE PX 2 device flashed with an operating system.
- DriveWorks SDK installed on the device.
- Supported sensors.

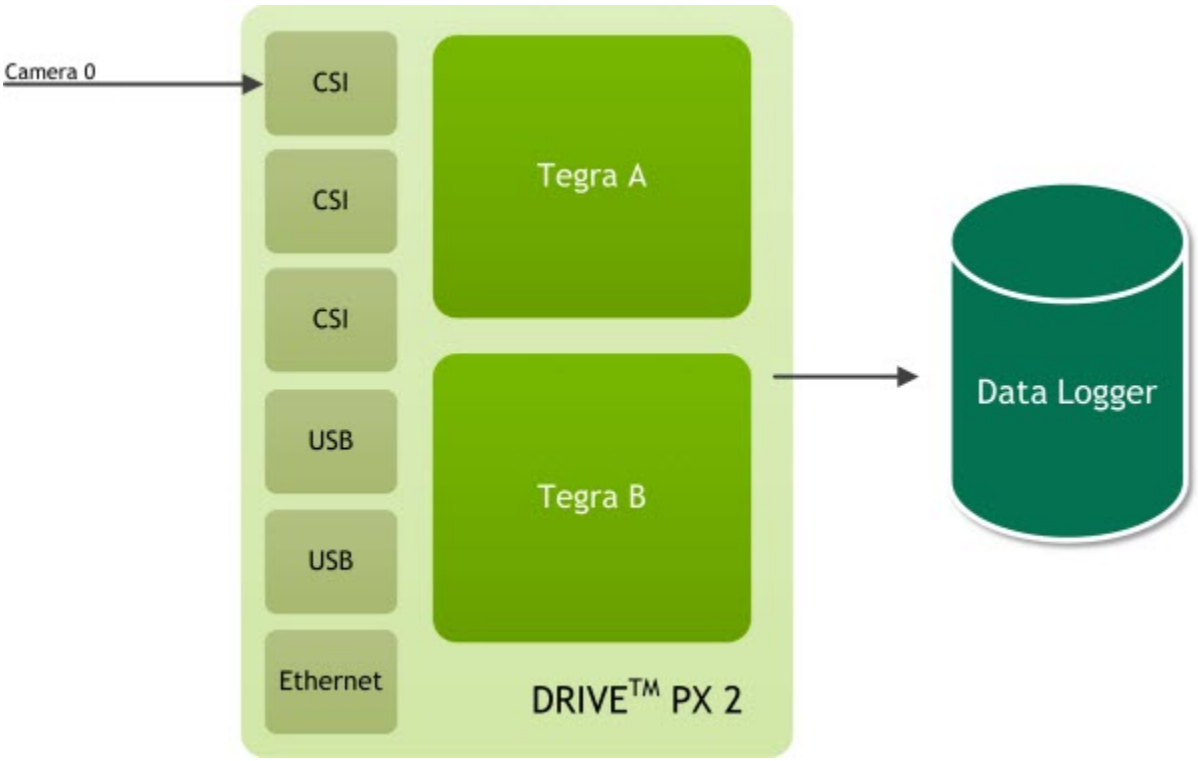
Camera Sensor Data Acquisition

This section explains how to use the DriveWorks recorder to acquire data from one or more camera sensors.

Recording from a Single Camera Sensor

This procedure shows how to record data from a single (AR0231) camera sensor.

Diagram: One camera connects to ab



To record data from a single camera sensor

1. With the device powered off, connect the camera sensor to port A0 of a camera group.

Warning:	Turn off the device before connecting or disconnecting a sensor.

2. Turn on the device and open a terminal window.
3. On the target, navigate to:

   ```
   /usr/local/driveworks/bin/
   ```

4. Run the following command to verify that your camera sensor is correctly connected to the device:

   If the camera type is c-ov10649-b1 and the CSI port is the default ab, execute:

   ```
   ./sample_camera_gmsl
   ```

   Otherwise, execute:

   ```
   ./sample_camera_gmsl --camera-type=<camera_type>
   ```

   Where `<camera_type>` is the camera sensor.

   For example, if your camera sensor is AR0231-RCCB, run the following command:

   ```
   ./sample_camera_gmsl --camera-type=ar0231-rccb
   ```

A window appears with video if the camera sensor is correctly connected to the device and the drivers are running.

Close the window to stop the camera sensor application.

5. Go to the following location in the DriveWorks folder:

```
/driveworks/tools/
```

6. Run the following command to create the configuration file:

```
# sudo ./recorder
```

A window appears with the following:

```
Press spacebar to begin recording.
```

Close the window to generate the configuration file. The following is the configuration file:

```
/driveworks/tools/recorder-config.json
```

7. Change the following section in the configuration file to log the data from the camera sensor.

```
"camera": {
"write-file-pattern":"video_*.h264",
"sensors":[
    {
"protocol": camera.gmsl
"params": "camera-type=ar0231-rccb,
                csi-port=ab,
                camera-count=1,
                async-record=1,
                fifo-size=12",
      "channel-names": [
        "first"
      ]
    }
  ]
}
```

The `async-record` and `fifo-size` parameter entries are optional.

8. Save the modified configuration file.
9. Run the following command to log the data from the camera sensor:

```
# sudo ./recorder
```

A window appears with the following:

```
Press spacebar to begin recording.
```

Press the spacebar to start acquiring data from the camera sensor. The following message appears when you start acquiring data.

```
Press spacebar to stop recording.
```

Press the spacebar to stop acquiring data.

Warning:	**Stop acquiring data and then close the window. If you close the window before you stop acquiring data, you will corrupt the acquired data.**

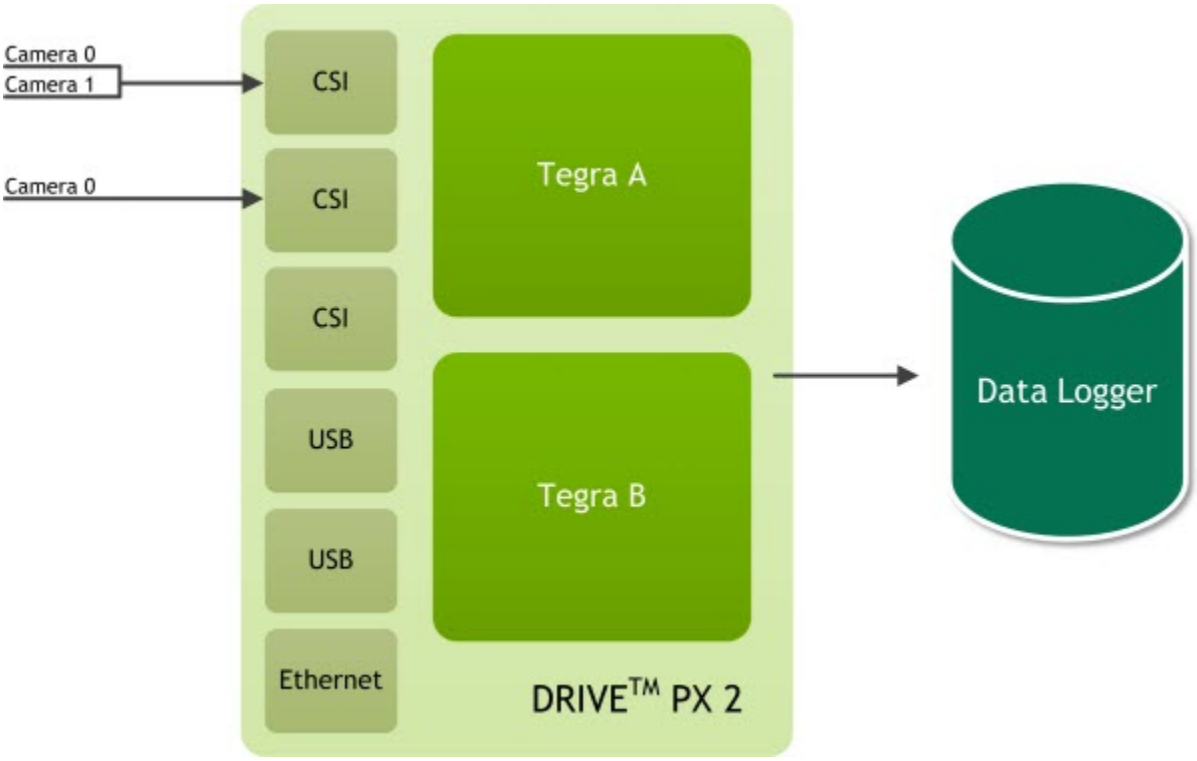
The sensor data is in the same folder as the recorder sample application.

Recording from Three Camera Sensors

This procedure demonstrates how to record data from four AR0231 camera sensors. The cameras connected to the following ports:

Camera Group	Port Name	Camera Type	Ports
A	ab	AR0231-RCCB	Port 0 Port 1
B	cd	AR0231-RCCB	Port 0

Diagram: Two cameras connect to ab and one to cd



Warning:	**Turn off the device before connecting or disconnecting a sensor.**

To acquire data from three camera sensors

1. Turn on the device and open a terminal window.
2. On the target, navigate to:

```
/usr/local/driveworks/bin/
```

3. Run `sample_camera_multiple_gmsl` to verify that your camera sensor is correctly connected to the device. The following command is for cameras connected at 0 and 1 on port-ab and 0 on port-cd:

```
./sample_camera_multiple_gmsl --type-ab=ar0231-rccb --type-cd=ar0231-rccb \
--selector-mask=00110001
```

Warning:	**Cameras must be connected to the port in ascending order (0, 1, 2, 3).**

A window appears with video if the camera sensor is correctly connected to the device.

Close the window to stop the camera sensor application.

4. On the target, locate and edit:

```
/driveworks/tools/recorder-config.json
```

Modifications to this file require super-user privileges. Also, the JSON file is created when you first run the DriveWorks sample recorder application.

5. Change the following section in the configuration file to log the data from the camera sensor.

```
"camera": {
  "separate-thread": true,
  "write-file-pattern": "video_*",
  "sensors": [
    {
      "protocol": "camera.gmsl",
      "params": "camera-type=ar0231-rccb,csi-port=ab,
                 camera-count=2,
                 camera-mask=0011,async-record=1,
                 fifo-size=12",
      "channel-names": [
        "first",
        "second"
      ]
    },
    {
      "protocol": "camera.gmsl",
      "params": "camera-type=ar0231-rccb,
                 csi-port=cd,
                 camera-count=1,
                 camera-mask=0001",
      "channel-names": [
        "third"
      ]
    }
```

```
  ]
}
```

Save the modified configuration file.

6. Run the following command to log the data from the camera sensor:

```
# sudo ./recorder
```

A window appears with the following:

```
Press spacebar to begin recording.
```

Press the spacebar to start acquiring data from the camera sensor. The following message appears when you start acquiring data.

```
Press spacebar to stop recording.
```

Press the spacebar to stop acquiring data.

Warning:	**Stop acquiring data and then close the window. If you close the window before you stop acquiring data, you will corrupt the acquired data.**

The sensor data is in the same folder as the recorder sample application.

Recording from Six Cameras

This procedure demonstrates how to record data from six AR0231 camera sensors. The cameras connected to the following ports:

Camera Group	Port Name	Camera Type	Ports
A	ab	AR0231-RCCB	Port 0 Port 1 Port 2 Port 3
B	cd	AR0231-RCCB	Port 0 Port 2

To acquire data from six camera sensors

1. Follow steps 1-4 in Recording from Three Camera Sensors.
2. Change the following section in the configuration file to log the data from the camera sensor.

```
"camera": {
  "separate-thread": true,
  "write-file-pattern": "video_*",
  "sensors": [
    {
```

```
      "protocol": "camera.gmsl",
      "params": "camera-type=ar0231-rccb,
                 csi-port=ab,
                 camera-count=4,
                 output-format=raw,
                 async-record=1,
                 fifo-size=12",
      "channel-names": [
        "first",
        "second",
        "third",
        "fourth"
      ]
    },
    {
      "protocol": "camera.gmsl",
      "params": "camera-type=ar0231-rccb,
                 csi-port=cd,
                 camera-count=2,
                 output-format=raw,
                 async-record=1,
                 fifo-size=12",
      "channel-names": [
        "fifth",
        "sixth"
      ]
    }
  ]
}
```

3. Follow step 6 in Recording from Three Camera Sensors.

Recording at a Framerate Other Than 30 FPS

For certain kinds of camera, e.g., ar0231-rccb, the camera framerate can be controlled by the required-framerate option.

To record camera frames at a rate other than the default 30 FPS, specify values for `frame-record-count` and `frame-skip-count` in the parameters, as shown in the following table.

For example, a value of frame-skip-count=1 records at 15 FPS if the camera hardware runs at 30 FPS. Setting record-frame-count=2 with frame-skip-count=1 records two (2) frames and skips 1, and then repeats.

Framerate	JSON Parameter
15 FPS	frame-record-count=1 frame-skip-count=1
10 FPS	frame-record-count=1 frame-skip-count=1 required-framerate= 20
Applies to: ar0231-rccb only 20 FPS on	required-framerate= 20
30 FPS	No change needed. This is the default rate.

36 FPS	required-framerate=36

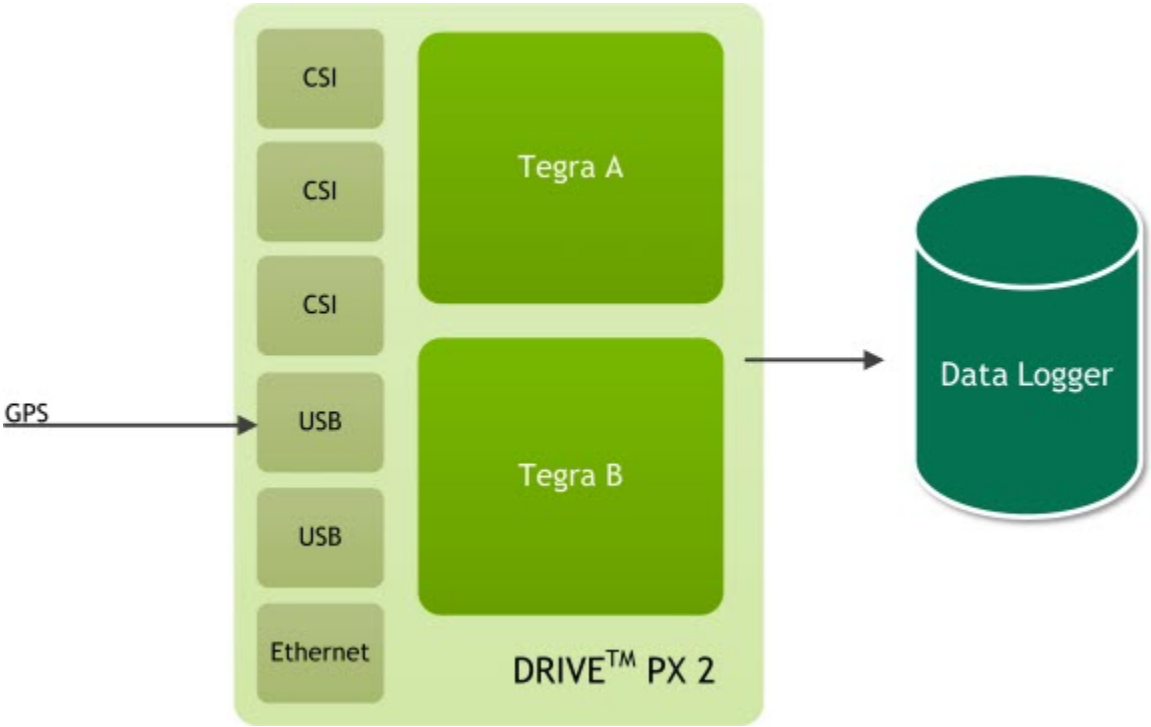
For example, the following JSON configuration specifies a framerate of 36 FPS.

```
"camera": {
  "separate-thread": true,
  "write-file-pattern": "video_*",
  "sensors": [
    {
      "protocol": "camera.gmsl",
      "params": "camera-type=ar0231-rccb-ssc,
                 csi-port=ab,
                 camera-count=2,
                 output-format=raw,required-framerate=36",
      "channel-names": [
        "first",
        "second"
      ]
    }
  ]
}
```

Where the values for `params` are concatenated into a single line:

GPS Data Acquisition

This section shows how to acquire data from a GPS sensor.



To determine GPS settings for the JSON file

1. With the device powered off, connect the GPS sensor to the device per vendor documentation. For example, Garmin connection instructions are on page 8 at:

   ```
   http://static.garmin.com/pumac/GPS_18x_Tech_Specs.pdf
   ```

2. Determine on which serial port the device is enumerated:

   ```
   dmesg | grep ttyUSB
   ```

3. Determine the baud rate at which the GPS device transmits data. This information can be obtained from vendor documentation. For example, Garmin LVC GPS 18x uses 4800 baud.
4. Set the baud rate to the serial port. For example:

   ```
   stty -F /dev/ttyUSB0 4800
   ```

5. Verify that GPS data is being received:

   ```
   cat /dev/ttyUSB0 4800
   ```

6. Update the GPS settings in the JSON file with the serial port and baud rate.

To acquire data from a GPS sensor

1. With the device powered off, connect the GPS sensor to the device.

   Warning: **Turn off the device before connecting or disconnecting a sensor.**

2. Turn on the device and open a terminal window.
3. On the target, locate and edit:

   ```
   /driveworks/bin/
   ```

4. Run the following command to verify that your GPS sensor is correctly connected to the device:

   ```
   ./sample_gps_logger --driver=gps.uart --params=device=/dev/ttyUSB0,baud=4800
   ```

   A window appears that displays the data acquired if the GPS sensor is correctly connected to the device and the drivers are running.

   Close the window to stop the camera sensor application.

5. Go to the following location in the DriveWorks folder:

   ```
   /driveworks/tools/
   ```

6. Run the following command to create the configuration file:

   ```
   # sudo ./recorder
   ```

   A window appears with the following:

   ```
   Press spacebar to begin recording.
   ```

Close the window to generate the configuration file. The following is the configuration file:

```
/driveworks/tools/recorder-config.json
```

Modifications to this file require super-user privileges. Also, the JSON file is created when you first run the DriveWorks sample recorder application.

7. Change the following section in the configuration file to log the data from the camera sensor. For following example is for the Xsens GPS.

```
"gps" : {
  "write-file-pattern": "xsens_imu*",
  "sensors" : [
    {
      "protocol": "gps.uart",
      "params": "device=/dev/ttyUSB0,baud=115200",
      "channel-name": "0"
    }
  ]
},
```

Save the modified configuration file.

8. Run the following command to log the data from the camera sensor:

```
# sudo ./recorder
```

A window appears with the following:

```
Press spacebar to begin recording.
```

Press the spacebar to start acquiring data from the camera sensor. The following message appears when you start acquiring data.

```
Press spacebar to stop recording.
```

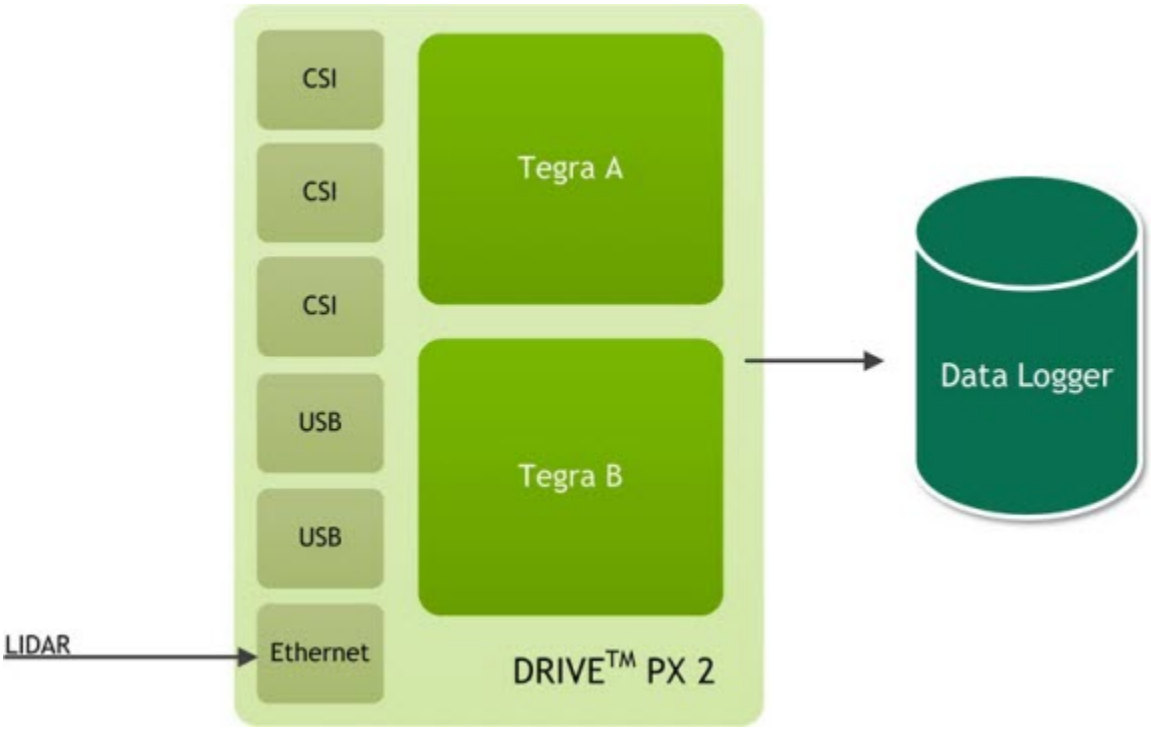
Press the spacebar to stop acquiring data.

Warning:	**Stop acquiring data and then close the window. If you close the window before you stop acquiring data, you will corrupt the acquired data.**

The sensor data is in the same folder as the recorder sample application.

Lidar Data Acquisition

This section shows how to acquire data from a Lidar sensor.



Determining the Lidar IP Address and Port

To determine the IP address and port for a Quanergy LIDAR

1. Connect the Lidar Ethernet port to eth0 port of Tegra A.
2. Connect the Quanergy to a router switch and connect this router to Tegra A eth0. The IP of router will be in range 192.168.x.x.
3. Find the IP address by running:

   ```
   # nmap -sn 192.168.1.0/25
   ```

   If Quanergy is detected, you will see an IP address associated with Quanergy system.

4. To obtain the port number, consult the Quanergy technical specification.

The following is an example of `recorder-config.json` with a Quanergy LIDAR.

```
"lidar" : {
  "separate-thread": false,
  "write-file-pattern": "lidar_*",
  "sensors" : [
    {
      "protocol": "lidar.socket",
      "params": "device=QUAN_M81A,
                 ip=192.168.1.8,
                 port=4141,
                 scan-frequency=10",
      "channel-name": "0"
    }
  ]
```

```
}
```

To determine the IP address for a Velodyne LIDAR

1. Connect the LIDAR to the eth0 port of Tegra A.
2. Obtain the default IP address and port number from the Velodyne technical specification.
3. Set up the IP of eth0 with:

   ```
   # ifconfig eth0 <lidar_address>
   ```

   By default, the Lidar IP address is `192.168.1.201`.

   Ensure this IP does not conflict with the AURIX, Tegra A, or Tegra B IPs.

The following is an example of `recorder-config.json` with a Velodyne 32 LIDAR.

```
"lidar" : {
  "separate-thread": false,
  "write-file-pattern": "lidar_*",
  "sensors" : [
    {
      "protocol": "lidar.socket",
      "params": "device=VELO_HDL32E,
                 ip=192.168.1.201,
                 port=2368,scan-frequency=10",
      "channel-name": "1"
    }
  ]
}
```

The value for params above is:

```
"device=VELO_HDL32E,ip=192.168.1.201,port=2368,scan-frequency=10"
```

For Velodyne 16, the device name is `VELO_VLP16`.

For Velodyne 64, the device is `VELO_HDL64E`.

Acquiring Data from a Lidar Sensor

After you obtain the Lidar IP address and port, you can obtain Lidar data.

To acquire data from a Lidar sensor

1. With the device powered off, connect the LIDAR sensor to the device.

Warning:	**Turn off the device before connecting or disconnecting a sensor.**

2. Turn on the device and open a terminal window.
3. On the target, locate and edit:

   ```
   /driveworks/bin/
   ```

4. Run the following command to verify that your LIDAR sensor is correctly connected to the device. For example:

```
./sample_lidar_replay --device=VELO_HDL32E --ip=192.168.1.201 --port=2368 \
    --scan-frequency=10
```

   A window appears that displays a point cloud if the LIDAR sensor is correctly connected to the device and the drivers are running.

   Close the window to stop the camera sensor application.

5. Go to the following location in the DriveWorks folder:

```
/driveworks/tools/
```

6. Run the following command to create the configuration file:

```
# sudo ./recorder
```

   A window appears with the following:

```
Press spacebar to begin recording.
```

   Close the window to generate the configuration file. The following is the configuration file:

```
/driveworks/tools/recorder-config.json
```

   Modifications to this file require super-user privileges. Also, the JSON file is created when you first run the DriveWorks sample recorder application.

7. Change the following section in the configuration file to log the data from the camera sensor.

```
"lidar" : {
  "write-file-pattern": "lidar_*",
  "sensors" : [
    {
      "protocol": "lidar.socket",
      "params": "ip=<ip_address>,port=<port>,device=<device_name>,frequency=10",
      "channel-name": "0_front"
    }
  ]
}
```

   Where:

   - `<ip_address>`—is the IP address of the LIDAR sensor.
   - `<port>`—is the port for the Lidar sensor.

   Save the modified configuration file.

8. Run the following command to log the data from the camera sensor:

```
# sudo ./recorder
```

A window appears with the following:

```
Press spacebar to begin recording.
```

Press the spacebar to start acquiring data from the LIDAR sensor. The following message appears when you start acquiring data.

```
Press spacebar to stop recording.
```

Press the spacebar to stop acquiring data.

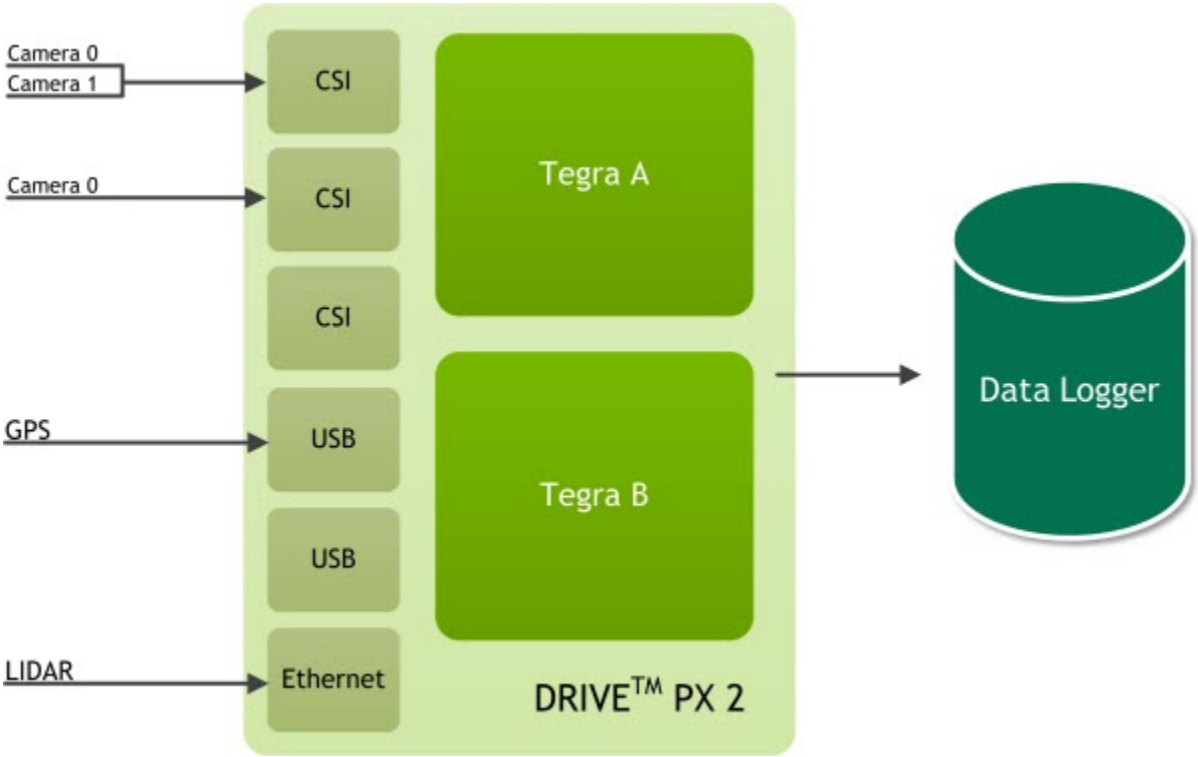
Warning:	**Stop acquiring data and then close the window. If you close the window before you stop acquiring data, you will corrupt the acquired data.**

The sensor data is in the same folder as the recorder sample application.

Multiple Sensor Data Acquisition

This section shows you how to acquire data from multiple sensors connected to the device. This section assumes that you know how to verify that each sensor is working and that you have a configuration file.

Diagram: Two cameras connect to ab and one to cd, one GPS to USB, and the LIDAR to Ethernet



To acquire data from multiple cameras, an HDL-32 LIDAR, and a GPS Sensors

This procedure shows you how to log the data from the following sensors:

- Three AR0321 camera sensors
- One HDL-32 LIDAR sensor
- One GPS Sensor

The cameras are connected to the following ports:

Camera Group	Port Name	Camera Type	Ports
A	ab	AR0231-RCCB	Port 0 Port 1
B	cd	AR0231-RCCB	Port 0

Warning: **Turn off the device before connecting or disconnecting a sensor.**

1. Change the following sections in the configuration file to log the data from the sensors:

```
"camera": {
  "separate-thread": true,
  "write-file-pattern": "video_*",
  "sensors": [
    {
      "protocol": "camera.gmsl",
      "params": "camera-type=ar0231-rccb,csi-port=ab,camera-count=2,camera-mask=0011",
      "channel-names": [
        "first",
        "second"
      ]
    },
    {
      "protocol": "camera.gmsl",
      "params": "camera-type=ar0231-rccb,csi-port=cd,camera-count=1,camera-mask=0001",
      "channel-names": [
        "third"
      ]
    }
  ]
},
"gps" : {
  "write-file-pattern": "xsens_imu*",
  "sensors" : [
    {
      "protocol": "gps.uart",
      "params": "device=/dev/ttyUSB0,baud=115200",
```

```
      "channel-name": "0"
    }
  ]
},
"lidar" : {
  "write-file-pattern": "lidar_*",
  "sensors" : [
    {
      "protocol": "lidar.socket",
      "params": "ip=<ip_address>,port=<port>,device=<device_name>,frequency=10",
      "channel-name": "0_front"
    }
  ]
}
```

Where

`<ip_address>`—is the IP address of the LIDAR sensor.

`<port>`—is the port for the LIDAR sensor

Save the modified configuration file.

2. Run the following command to log the data from the camera sensor:

```
# sudo ./recorder
```

A window appears with the following:

```
Press spacebar to begin recording.
```

Press the spacebar to start acquiring data from the camera sensor. The following message appears when you start acquiring data.

```
Press spacebar to stop recording.
```

Press the spacebar to stop acquiring data.

Warning:	**Stop acquiring data and then close the window. If you close the window before you stop acquiring data, you will corrupt the acquired data.**

The sensor data is in the same folder as the recorder sample application.

Sensor Data Quality

To sanity-check the Lidar and Camera data

1. Use the recorder tool to capture Lidar and camera data on someone walking past.
2. Use the replayer tool on the captured data to verify that:

- Data from the different sensors are roughly in sync and
- Lidar data makes sense (i.e. the shape of the person walking past).

To sanity-check the Lidar data

- For Lidar, compare the Lidar plot coming from the PDK with the reference plots obtained from the Windows machine or any other Linux x86 machine, at the same location.

To sanity-check GPS data

- If you are testing in a Lab, ensure the GPS produces the same coordinates over time. It should be roughly the same because it is not moving.

To determine if a sensor is broken

- Feed each offline file to the corresponding DriveWork sensor sample. If one of the files fails, you know there is a problem with the sensor.
- Read the documentation provided by the sensor manufacturer.

Frequently Asked Questions

Where can I get answers to my DriveWorks questions?

On the DRIVE Platforms forum, you can search through discussions on questions posed by the community. If you can also ask your own questions. You must create an account before you can access this information.

https://devtalk.nvidia.com/default/board/182/drive-platforms/

How to achieve the maximum USB 3.0 write throughput for data acquisition?

NVIDIA testing easily achieved USB3.0-SSD write speeds between 300- to 400- megabytes /second, depending on the brand of the USB 3.0 SSD. NVIDIA has tested extensively with Samsung 850/750 Evo SSD with ext4 format.

For guidance on optimizing data storage to your USB disk, see Data Acquisition in this guide.

What are the networking settings needed for the best results for Lidar capture ?

For guidance on optimizing data storage to your USB disk, see Data Acquisition in this guide.

Legal Information

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, ”MATERIALS”) ARE BEING PROVIDED ”AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OR CONDITION OF TITLE, MERCHANTABILITY, SATISFACTORY QUALITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT, ARE HEREBY EXCLUDED TO THE MAXIMUM EXTENT PERMITTED BY LAW.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, Jetson, NVIDIA DRIVE, Tegra, and TensorRT are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

The Android robot is reproduced or modified from work created and shared by Google and is used according to terms described in the Creative Commons 3.0 Attribution License.

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ARM, AMBA, and ARM Powered are registered trademarks of ARM Limited. Cortex, MPCore and Mali are trademarks of ARM Limited. All other brands or product names are the property of their respective holders. ”ARM” is used to represent ARM Holdings plc; its operating company ARM Limited; and the regional subsidiaries ARM Inc.; ARM KK; ARM Korea Limited.; ARM Taiwan Limited; ARM France SAS; ARM Consulting (Shanghai) Co. Ltd.; ARM Germany GmbH; ARM Embedded Technologies Pvt. Ltd.; ARM Norway, AS and ARM Sweden AB.

Copyright

© 2017 by NVIDIA Corporation. All rights reserved.

Open Source and Third-Party Software Licenses

This NVIDIA product contains third party software that is being made available to you under their respective open source software licenses. Some of those licenses also require specific legal information to be included in the product. This chapter provides such information.