SmartAR™ SDK Overview

Ver 1.1

Change History

Version	Division of Work	Item	Changed Contents	Date
1.0	Created	-	Document created	2016/01/27
1.1	Changed	-	Added some description about license certification	2016/03/28

Table of Contents

1 Overview

Preface

SmartAR™ SDK is a multiplatform AR (Augmented Reality) application development environment targeted for Android and iOS. Application developers can create AR applications for multiplatform using SmartAR™ easily by using this SDK. This document explains the overview of SmartAR™ SDK.

Main Functions and Purpose

SmartAR™ SDK contains class groups such as SarRecognizer class for recognition process, SarCameraDevice class which is the abstraction of the platform's camera devices, SarSensorDevice class which is the abstraction of the platform's sensor devices, SarCameraImageDrawer class for rendering camera image. The classes provide common interface for both Android and iOS.

Figure1 is showing the typical structure of SmartAR™ SDK and application using the SDK. By implementing core part of the application containing this SDK and OpenGL, developer can efficiently develop application for multiplatform.

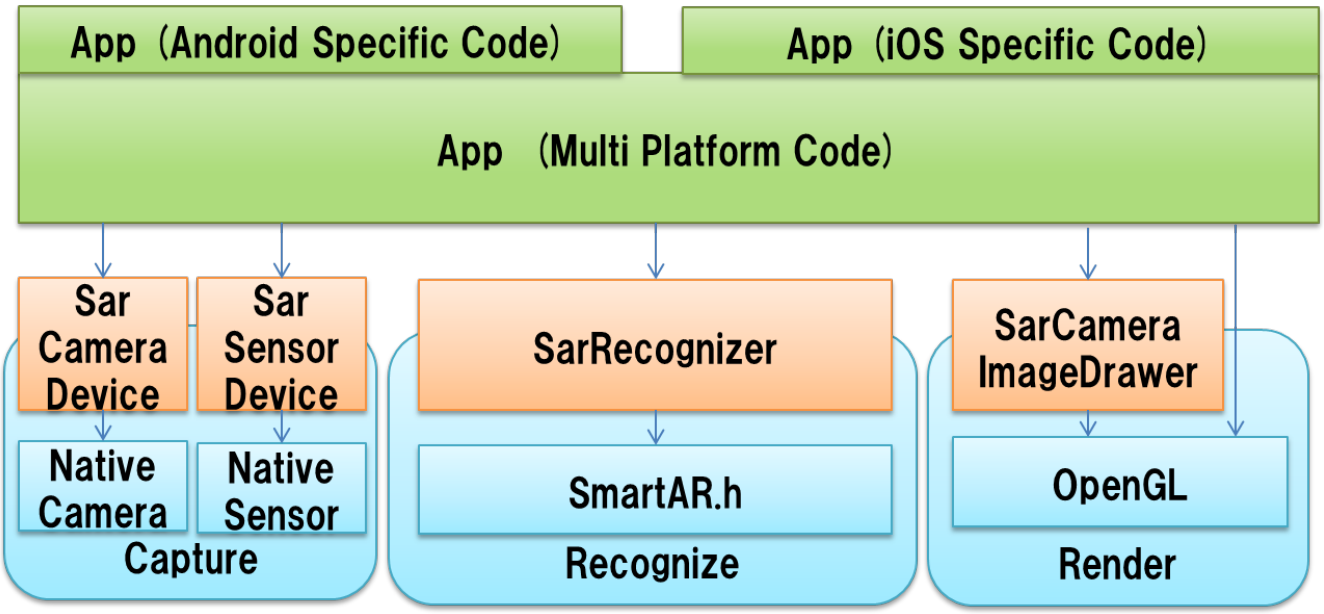

Figure1 Configuration Diagram of SmartAR™ SDK and Application

Typical application using SmartAR™ SDK processes as shown in Figure2.

Application receives camera image and sensor information from SarCameraDevice and SarSensorDevice, then pass these to SarRecognizer for recognition process.
After receiving recognition result from SarRecognizer, AR contents are rendered upon camera image based on the recognition result to realize AR function.

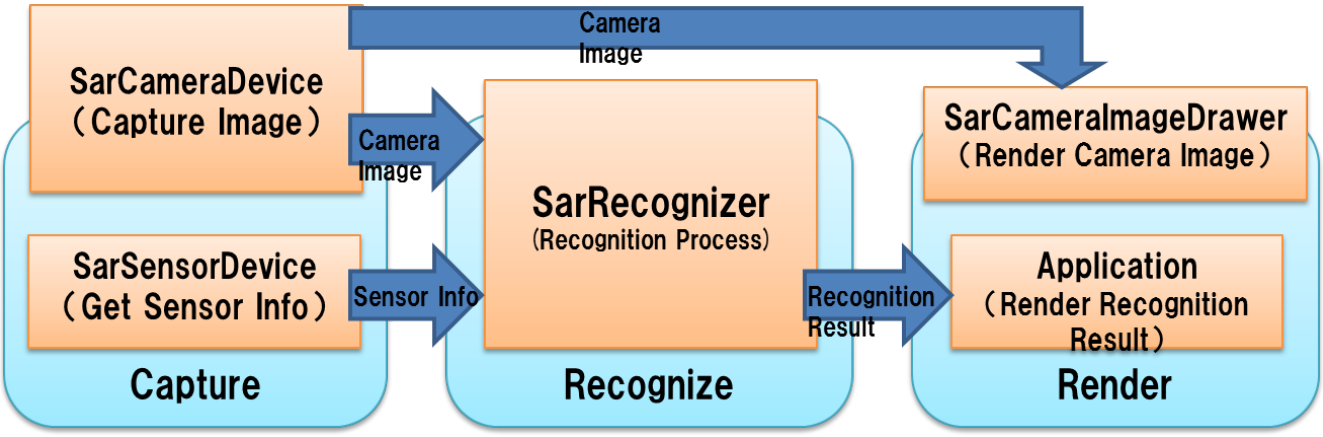

Figure 2 Process Stream of SmartAR™ SDK

SarRecognizer has 2 processing modes, TargetTracking and SceneMapping.
The overview of these modes is as follows.

TargetTracking Mode

The TargetTracking mode is the mode that recognizes planar objects in camera images and to estimate the "pose" of device running SmartAR™ (position and orientation). You can also set natural images of your choice as targets for recognition. For more information on the TargetTracking mode's processing, refer to the "Structure of the TargetTracking Mode's Processing" section in Chapter 3.

SceneMapping Mode

The SceneMapping mode is the mode that estimates the 3D structure of an unknown environment and the pose (position and orientation) of a device running SmartAR™ inside that 3D structure based on changes in the camera images and on measurements taken by the motion sensors as the device moves across space. It realizes a technology called Simultaneous Localization and Mapping (SLAM). For more information on the SceneMapping mode's processing, refer to the "

Structure of the SceneMapping Mode's Processing" section in Chapter 3.

Supported Environment

SmartAR™ SDK supports the following platforms

OS	Version
Android	4.0.3 and above
iOS	8.0 and above

Embedding into a Program

The following files are required in order to use SmartAR™ SDK.

File	Description
SarCameraDevice.h	Header file defines SarCameraDevice and related classes
SarCameraImageDrawer.h	Header file defines SarCameraImageDrawer class
SarCommon.h	Header defines basic constants and classes
SarPlatformDef.h	Header used internally by the SDK. Application developer is not expected to use this file.
SarRecognizer.h	Header file defines SarRecognizer and related classes
SarSensorDevice.h	Header file defines SarSensorDevice and related classes
SarScreenDevice.h	Header file defines SarScreenDevice class
SarSmart.h	Header file defines SarSmart class
SarUtility.h	Header provides utility functions
libsmart.a	Static link library file for iOS
libsmartar.so	Dynamic link library for Android
smartar.jar	jar file for Android
libSmartAR.so	Shared library for Linux (Core part)

Please refer to “SmartAR™ SDK-Reference.pdf” and include header files as necessary.

Library file to link is different between platforms.
When running on Android, add smartar.jar to project and load libsmartar.so from program as follows.

Example) Loading libsmartar.so

```
public class Sample {
    ～(omitted)～
    static {
        System.loadLibrary("smartar");
    }
    ～(omitted)～
}
```

When running on iOS, add libsmartar.a to project.

Updating SmartAR™ SDK files

If you will update to latest version SDK, please overwrite header and binary files.

Libraries

Each file has the following roles.

	AR core part	API part	Device control part
Android	libsmartar.so		smartar.jar
iOS	libsmartar.a		

Library supports camera, and sensor using the API provided by the Android, iOS.

Sample Program

Sample program of SmartAR™ SDK for Android and iOS are available.
There are two types of sample program.
“sample_api_simple” is a program that shows the running configuration of the minimum required.
“sample_api_full” is a program that allows the execution of each API that is included in the library.

dictool

This is a tool to create dictionary of recognition target from natural image.
For more information refer “How to use dictool” in the next chapter.

About License Certification

The certified license is required to use full function of SmartAR™ SDK.
In order to certify your license, include your license file to your app's resource by placing the license file in assets folder (in case of Android), or by adding the license file to the project (in case of iOS). Then designate the file path when you initialize SarSmart.
You can use SmartAR™ SDK without certified license, but the following restrictions apply when you do so.

- The picture must be taken using SarCameraDevice for recognition.
- SarCameraImageDrawer must be used to draw camera image taken by SarCameraDevice.
- Watermark will always appear on top of the camera image.
- Screen shot cannot be taken using image capture function included in the libraries.

Please refer "SmartARSDK-Reference_e.pdf" for more details.

Sample programs certify license with “license.sig” as license file. (Because “license.sig” is dummy file, SmartARTM SDK can’t be certified.)
You can refer “SDK/Sample/mobile_common/sample_simple/sample_simple.cc” for initialize process of SarSmart.

Example)

```
SarSmart* smart = new SarSmart(nativeContext, nativeEnv, "license.sig");
```

2 Using the SDK

Overview of TargetTracking Mode's Processing

TargetTracking mode detects natural image registered as recognition target from the camera image. When the natural image is detected, estimate the pose (position and orientation) of the device running SmartAR™. As long as the natural image exists within the camera image and is recognized properly the pose of the device is estimated at real time.

When the recognition of the natural image is successful, the member isRecognized_ of the obtained result from SarRecognizer (SarRecognitionResult) is true.

The TargetTracking Mode's Calling Procedure

Recognition can be processed both single and multithreaded.
Usually for device has multicore the multithreaded method can provide better performance.
The pseud code of each mode is shown as follows.

*** Pseud code of single threaded recognition process**

```
void main_thread()
{
	// Initialization
	SarSmart* smart = new SarSmart();
	recognizer = new SarRecognizer(smart);

	target = new SarLearnedImageTarget(stream);
	recognizer->setTargets(target);

	cameraDevice = new SarCameraDevice(smart);
	cameraDevice->setVideoImageListener(myVideoImageListener);
	 cameraDevice->getCameraDeviceInfo(&cameraDeviceInfo);
	 recognizer->setCameraDeviceInfo(&cameraDeviceInfo);

	 sensorDevice = new SarSensorDevice(smart);
	 sensorDevice->setSensorListener(mySensorListener);
	 sensorDevice->getSensorDeviceInfo(&sensorDeviceInfo);
	 recognizer->setSensorDeviceInfo(&sensorDeviceInfo);

	cameraDevice->start();
	 sensorDevice->start();
}

// Receive sensor information
class MySensorListener : public SarSensorListener
{
public:
	 virtual void onSensor(sensorState)
	 {
		sensorStates.add(sensorState);
```

```
    }
};

//Receive image and execute recognition process
class MyVideoImageListener : public SarCameraImageListener
{
public:
    virtual void onImage(imageHolder)
    {
        SETUP_REQUEST(imageHolder, sensorStates, request);

        recognizer->run(request);
    }
};

//Finalization
void end_app()
{
    cameraDevice->stop();
    sensorDevice->stop();

    delete cameraDevice;
    delete sensorDevice;

    delete recognizer;
    delete target;
    delete smart;
}

//Rendering
void render_thread()
{
    cameraImageDrawer = new SarCameraImageDrawer(smart);
    cameraImageDrawer->start();
    while (...)
    {
        cameraImageDrawer->draw(image);

        recognizer->getResults(&results, maxResults);
        if (results[0].isRecognized_)
        {
            DRAW_AR_CONTENTS();
        }
    }
    cameraImageDrawer->stop();
    delete cameraImageDrawer;
}
```

*** Pseud code of multithreaded recognition process**

```
// Initialization
void start_app()
{
    SarSmart* smart = new SarSmart();
    recognizer = new SarRecognizer(smart);
```

```
        recognizer->setWorkDispatchedListener(myWorkDispatchedListener);

        target = new SarLearnedImageTarget(stream);
        recognizer->setTargets(target);

        cameraDevice = new SarCameraDevice(smart);
        cameraDevice->setVideoImageListener(myVideoImageListener);
         cameraDevice->getCameraDeviceInfo(&cameraDeviceInfo);
         recognizer->setCameraDeviceInfo(cameraDeviceInfo);

         sensorDevice = new SarSensorDevice(smart);
         sensorDevice->setSensorListener(mySensorListener);
         sensorDevice->getSensorDeviceInfo(&sensorDeviceInfo);
         recognizer->setSensorDeviceInfo(sensorDeviceInfo);

        cameraDevice->start();
         sensorDevice->start();
        START_WORKER_THREAD();
}

// Receive sensor information
class MySensorListener : public SarSensorListener
{
public:
        virtual void onSensor(sensorState)
        {
                sensorStates.add(sensorState);
        }
};

//Receive image and recognition request
class MyVideoImageListener : public SarCameraImageListener
{
public:
        virtual void onImage(imageHolder)
        {
                SETUP_REQUEST(imageHolder, sensorStates, request);

                recognizer->dispatch(request);
        }
};

//Finalization
void end_app()
{
        FINISH_WORKER_THREAD();
        sensorDevice->stop();
        cameraDevice->stop();

        delete cameraDevice;
        delete sensorDevice;

        delete recognizer;
        delete target;
        delete smart;
}
```

```
//Start recognition process
class MyWorkDispatchedListener : public SarWorkDispatchedListener
{
public:
      virtual void onWorkDispatched()
      {
            RAIZE_EVT_TO_WORKER_TRHEAD();
      }
};

//Execute recognition process
void worker_thread()
{
      while (...)
      {
            WAIT_EVT();

            recognizer->runWorker();
      }
}

//Rendering
void render_thread()
{
      cameraImageDrawer = new SarCameraImageDrawer(smart);
      cameraImageDrawer->start();
      while (...)
      {
            cameraImageDrawer->draw(image);

            recognizer->getResults(&results, maxResults);
            if (results[0].isRecognized_)
            {
                  DRAW_AR_CONTENTS();
            }
      }
      cameraImageDrawer->stop();
      delete cameraImageDrawer;
}
```

(1) Initialization

Generate instance of SarSmart, SarRecognizer and SarLearnedImageTarget class and initialize them. Then call SarRecognizer::setTargets() to register SarLearnedImageTarget to SarRecognizer. For multithread, register trigger of worker threads SarWorkDispatchedListener to SarRecognizer.

(2) Execute recognition process

For single thread, recognition is done by SarRecognizer::run() synchronously on one thread. For multithread SarRecognizer::dispatch() is called to call a thread to dispatch recognition requests, and SarRecognizer::runWorker() is called to process requested tasks on multiple worker threads.

SarRecognitionRequest structure for SarRecognizer::run() or SarRecognizer::dispatch() as argument contains input data is needed for recognition processing.
The member image_ of SarRecognitionRequest need to be SarImage class contains image data. The format of SarImage class can be SAR_IMAGE_FORMAT_L8, SAR_IMAGE_FORMAT_YCBCR420 and SAR_IMAGE_FORMAT_YCRCB420.
By setting sensor information from SarSensorDevice class to member sensorStates_, following capability to the device's movement is improved.

When recognition is successful the member isRecognized_ of SarRecognitionResult structure obtained by SarRecognizer::getResult() or SarRecognizer::getResults() is true.

(3) Finalization

Destruct instance of SarSmart, SarRecognizer and SarLearnedImageTarget.

Overview of SceneMapping Mode's Processing

SceneMapping mode estimates the device running SmartAR™ 's pose (position and orientation) within a space and the 3D structure of the space by using camera image change and sensor information.
Application can render AR contents without specific marker image exist in the camera image.
Because the 3D structure is estimated from the movement parallax difference and sensor information, the device must be moved in the space.

The definition of coordinate system has multiple options. This can be changed by setting initialization mode to SarRecognizer.

SAR_SCENE_MAPPING_INIT_MODE_TARGET	Center of natural image in camera image as point of origin
SAR_SCENE_MAPPING_INIT_MODE_HFG	Center of horizontal plain in camera image as point of origin
SAR_SCENE_MAPPING_INIT_MODE_VFG	Center of vertical plain in camera image as point of origin
SAR_SCENE_MAPPING_INIT_MODE_SFM	Center of dominant plain estimated from 3D structure as point of origin

For the structure of SceneMapping mode and detail of coordinate definition of each initialization mode please refer chapter 3 "Reference Information".

When any problem occurs during recognition process of SceneMapping mode, SarRecognizer may change to a mode that localization is impossible. In this case application needs to response according to each use case, for example inform user or reset SarRecognizer. Therefore it is recommended that application check the value of sceneMappingState_ member of SarRecognitionResult structure obtained from SarRecognizer is not SAR_SCENE_MAPPING_STATE_LOCALIZE_IMPOSSIBLE.

The SceneMapping Mode's Calling Procedure

Calling procedure of SceneMapping mode is similar to TargetTracking mode.
However as shown as follows, initialization mode is need to be set at initialization of SarRecognizer.

Example) Initializing with"SAR_SCENE_MAPPING_INIT_MODE_TARGET"

```
SarRecognizer* recognizer = new SarRecognizer(
                smart,
                SAR_RECOGNITION_MODE_SCENE_MAPPING,
                SAR_SCENE_MAPPING_INIT_MODE_TARGET);
```

Each initialization mode is explained as follows.

SAR_SCENE_MAPPING_INIT_MODE_TARGET

Initialize with recognition target (natural image) extracted by recognition target dictionary building tool (dictool) beforehand. For world coordinate definition please refer to "Coordinate System" of chapter 3.

In this initialization mode, after loading dictionary with SarLearnedImageTarget class, register the SarLearnedImageTarget to SarRecognizer.

Note
Other than SarLearnedImageTarget, SarSceneMappingTarget and SarCompoundTarget can also be registered to SarRecognizer as SarTarget. For more information please refer to "SmartAR™ SDK-Reference.pdf"

SAR_SCENE_MAPPING_INIT_MODE_HFG

This is HFG (Horizontal from Gravity) initialization using acceleration sensor. World coordinate is defined from horizontal plain, which is estimated based on the gravity direction given by acceleration sensor. For world coordinate definition please refer to "Coordinate System" of chapter 3.
After SarRecognizer's recognition process is started, as long as enough initialization points (tracked points within 2D image) exist in the camera image, initialization is done immediately.

Note
It is required to set sensor information to SarRecognitionRequest structure when using this mode

SAR_SCENE_MAPPING_INIT_MODE_VFG

This is the VFG (Vertical from Gravity) initialization mode. It uses motion sensors. A vertical plane is calculated based on the gravity direction detected by the motion sensors and on the premise that the camera is directly confronting the vertical plane. For world coordinate definition please refer to "Coordinate System" of chapter 3.
At initialization, the camera and the vertical plane need to be directly confronting each other. If, after recognition process of SarRecognizer is started, the image is found to show sufficient initialization points, initialization will be completed immediately.

Note
It is required to set sensor information to SarRecognitionRequest structure when using this mode

SAR_SCENE_MAPPING_INIT_MODE_SFM

This is the SFM (Structure from Motion) initialization mode. It uses the camera's motion parallax. Based on camera images from two viewpoints with different focus position, this mode calculates a 3D structure composed of initialization points. Then, it calculates the dominant plane of the section photographed at the center of the camera image based on the 3D structure, and defines the world coordinate system based on the dominant plane. For world coordinate definition please refer to "Coordinate System" of chapter 3.
After recognition process of SarRecognizer is started, the library will check whether the image shows sufficient initialization points. If sufficient initialization points are shown, the library will continue tracking the initialization points until parallax is sufficient. When parallax becomes sufficient, the library will complete the initialization process.

Note
For example, the library will not be initialized if parallax is insufficient, such as when the camera is rotated in the same spot. If initialization points are insufficient - that is, if a featureless object is photographed - initialization will not be completed.
If no motion sensor value is passed the photographed object will be processed by assuming that it is a flat plane.

SAR_SCENE_MAPPING_INIT_MODE_DRY_RUN

This is a test mode for checking how many initialization points there are inside an image. It will not define a world coordinate system.
After recognition process is started, SarRecognizer will continue checking the number and location of initialization points within the image. Obtain the number and position of initialization points with numInitPoints_ and iniPoints_ member of SarRecognitionResult structure obtained by SarRecognizer::getResults() and SarRecognizer::getResult(). Use SAR_SCENE_MAPPING_INIT_MODE_HFG, SAR_SCENE_MAPPING_INIT_MODE_VFG or SAR_SCENE_MAPPING_INIT_MODE_SFM when using initialization points for initialization.

Note
Actual initialization cannot be performed in this mode. SmartAR™ SDK will not generate initialization points in excess of SAR_MAX_NUM_INITIALIZATION_POINTS.

How to Use dictool

dictool is a tool for generating dictionary data (xxxx.dic) to be used with SmartARTM SDK from the images of recognition targets. Target objects can be detected and tracked by having SmartAR™ SDK load generated dictionary data. One dictionary data will be necessary for each recognition target.
At present, the recognition targets that can be used with SmartAR™ SDK are flat printed materials (photographs, magazines, posters, etc.). Depending on the texture of the recognition target's image, some targets will be easier to identify, and SmartAR™ SDK 's performance and usability may vary considerably. For cases allowing processing the texture or selecting the recognition target, refer to the "Points to Bear in Mind to Select a Good Recognition Target" item.

Runtime Environment

- 32-bit/64-bit Windows
- SSE2-compatible processor

Usable Image Formats

- PNG format : 24bit color/ 8bit gray
- BMP format : 24bit color/ 8bit gray
- PGM(P5) format : 8bit gray
- PPM(P6)format: 24bit color

*Use images whose shorter side is 200 pixels or more. A warning will be displayed if the size is too small.

How to Create a Dictionary

(1) Prepare the image file of a recognition target (refer to the "Points to Bear in Mind to Select a Good Recognition Target" item).
(2) Print the image on a sheet with a printer. In doing so, take care to ensure that there is no variance between the image's aspect ratio and the printout's.
(3) Measure the lateral size (in metric units) of the printed image (*physicalWidth*).
(4) Execute dictool with the command below. Refer to "Build Options" below concerning build options. Dictionary data (xxxx.v9.dic) will be created in about 1 minute.
> dictool.exe build -image smartar01.pgm -physicalWidth 0.19
(5) When the dictionary data is created, the dictionary will be evaluated, and the score of target recognition performance will be displayed. Also, refer to the "Error Messages" item concerning evaluation.

Example of Display after executing dictool:

```
> dictool.exe build -image smartar01.pgm -physicalWidth 0.19
        Build dictionary:
        input          : smartar01.pgm
        physicalWidth : 0.190000 [m]
        serialID       : 0
        vendor         : 0-0
        dicfile        : smartar01.pgm.v9.dic
        autoresize (576, 768) -> (150, 200)
        points: 33 31
```

store the dictionary file to smartar01.pgm.v9.dic'
.....
the score of this target = 82.90 [0.00:bad - 100.00:excellent]

The "score" that is output when a dictionary is created is a simplified approximation relating to the recognition rate of the input recognition target's image. Use dictionary evaluation options for a more accurate evaluation (refer to the "How to Evaluate Dictionaries" item). Good recognition performance can be expected with a score of about 70 or more. If the score is extremely low (50 or less), SmartAR™ SDK may not recognize the target correctly. For cases allowing processing or selecting the recognition target, refer to the "Points to Bear in Mind to Select a Good Recognition Target" item. If an error occurs, refer to the "Error Messages" item.

Build Options

With dictool, you can use the following build options:

Option	Format	Description
-image	-image <*image.pgm*>	Specifies the input image to be recognized. Supports PGM, PPM, BMP and PNG formats
-help	-help	Displays help
-physicalWidth	-physicalWidth <*float*>	Specifies the recognition target's "physical" lateral size in metric units; this is not its size in pixels. This value is used for scaling the coordinate values output when SmartAR™ SDK recognizes a target. The default value is 1.0 [m]
-outimage	-outimage <*string*>	Saves as an image the feature point information stored in the dictionary data detected from the input image. The format is PPM. Refer to the "Usable Image Formats" item on how to view the results
-mask	-mask <*maskfile.pgm*>	Specifies the mask image for specifying the valid area to be recognized within the recognition target image. The mask image must be of the same size as the recognition target image. Areas with a pixel value of 255 will be used for dictionary creation. Supports the PGM format

Masking

By inputting a mask image (PGM format), you can create a dictionary using only a specific area of the input image. The mask image must be of the same size as the input image. Areas with a pixel value of

255 within the mask image will be used as recognition target areas, while other areas will not be used as recognition targets.
This masking functionality is intended to be used to exclude a single part of an input image in cases where you wish to accurately register the shape of a recognition target that is not rectangular, such as a coaster. Performance may drop with a large excluded area within the registered image area. Check the recognition score.

Usage Example of the Masking Functionality

```
> dictool.exe build -image smartar01.pgm -physicalWidth 0.19 -mask smartar01.mask.centeronly.pgm
        Build dictionary:
                input          : smartar01.pgm
               physicalWidth : 0.190000 [m]
                mask           :   smartar01.mask.centeronly.pgm
                serialID       : 0
                vendor         : 0-0
                dicfile        : smartar01.pgm.v9.dic
        autoresize (576, 768) -> (150, 200)
        points: 33 31
        store the dictionary file to 'smartar01.pgm.v9.dic'
.....................................................................................
        the score of this target = 83.40 [0.00:bad - 100.00:excellent]
```

Points to Bear in Mind to Select a Good Recognition Target

SmartAR™ SDK identifies and tracks recognition targets by using the feature points extracted during dictionary creation. It chooses parts with high contrast and forming corners (corner points) inside grayscale images (textures) as feature points. Depending on the texture, some targets will be easier to identify, and SmartAR™ SDK 's performance and usability may vary considerably. Refer to the "How to Evaluate Dictionaries" and "How to View Feature Point Information" items.

Suitable Targets for Recognition

- Flat, even printed materials (photographs, magazines, posters, etc.)
- Objects with many complex textures
- Objects with many corners (corner points)
- Objects with high contrast when turned into gray images

Unsuitable Targets for Recognition

- Objects with few textures, or simple objects (plain walls, company logos or characters, etc.)
- Objects with localized textures
- Objects with low texture contrast
- Objects on which the same texture is repeated
- Objects whose aspect ratio is greatly unbalanced
- Reflecting objects (mirrors, laminated objects)

How to Evaluate Dictionaries

Dictionaries created with dictool can also be evaluated with dictool. The dictionary evaluation obtained here is an approximation of the extent to which recognition targets with various orientation and placed in various environments can be recognized. Good recognition performance can be expected

with a score of about 70% or more. In order to evaluate a dictionary, after choosing the -i option, as shown below, input the dictionary file's name and execute dictool.

```
>dictool.exe evaluate -i smartar01.png.v9.dic
        input : smartar01.pgm.v9.dic
    .....
    the score of this target = 81.37 [0.00:bad - 100.00:excellent]
```

When the evaluation is complete, the dictionary's performance will be displayed in "the score of this target =".

How to View Feature Point Information

Save the feature point image with the feature points used for recognition by selecting the -outimage option when creating a dictionary. It would be difficult to pinpoint detection and tracking performance based on this feature point image; however, it may serve as reference for corrections in cases where recognition target detection/tracking performance is extremely poor, or for comparisons with other recognition targets.

Feature Points

- Red circle : feature points used for detecting recognition targets
- Green square : feature points used for tracking recognition targets

The more uniformly feature points are distributed, the more stable detection and tracking will be.
If the contrast of the area surrounding these feature points is low, detection rate and tracking performance will be poor.
If feature points for detection (red circle) are distributed unevenly, that location may be concealed or become unclear, thus causing detection performance to decline greatly.

Error Messages

- "The size of the input image, (<*width*>, <*height*>), is too small as a target image"
  The size of the input image is too small. Use an image of the appropriate size, making reference to "Masking" item.
- "The aspect of the input image, (<*width*>, <*height*>), is out of range [0.25 - 4.0]"
  The input image's aspect ratio is unbalanced. Ensure that the ratio of the longer side to the shorter side is 4:1 or less.

Version compatibility of recognition target dictionaries

Dictionary compatibility is not guaranteed when SDK version changes.

3 Reference Information

Terminology

Term	Description
Corner point	Angular points within an image that are used when recognizing a registered recognition target
Natural image	A general image where no special patterns, such as markers, are used
Recognition target	A planar object to be recognized and tracked
Pose	Combines position and orientation. Represents the camera's position and orientation in a 3D space with 6 degrees of freedom
Scene map	3D structure of an estimated environment. It is composed of multiple landmarks.
Landmarks	Point on the surface of an environment's 3D structure whose 3D position is estimated. They are tracked by selecting a corner point within the image, and their 3D position is estimated from parallax.

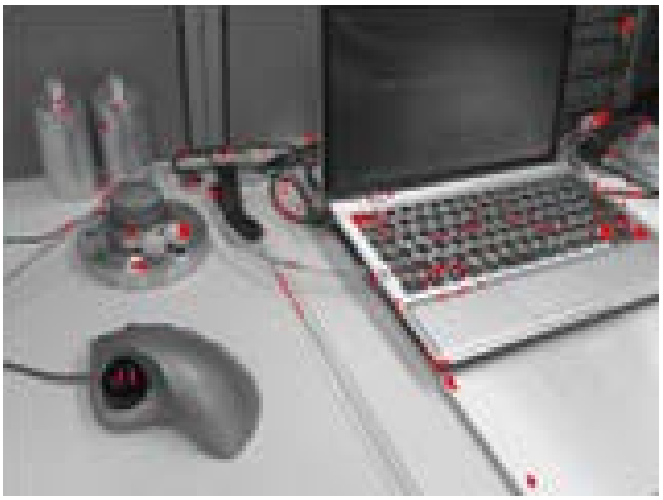

Figure 1 Corner points within an image

Structure of the TargetTracking Mode's Processing

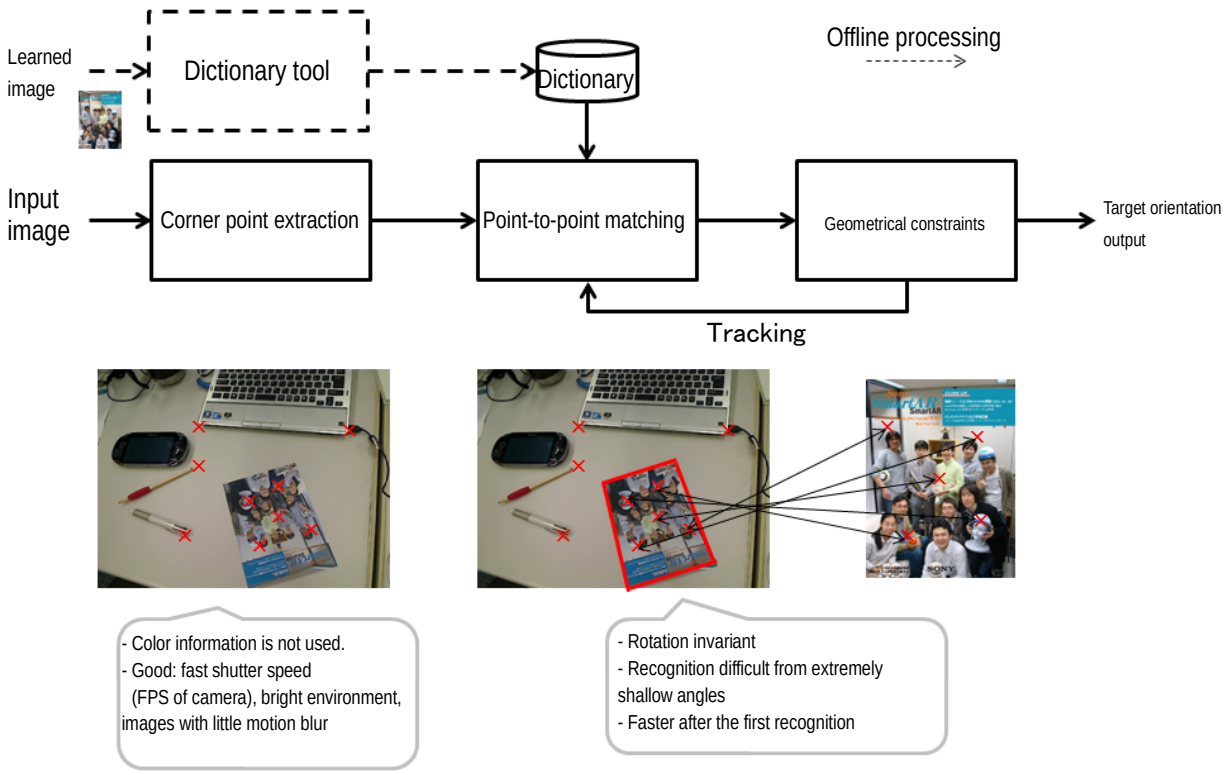

Figure 2 Diagram of the TargetTracking Mode's Processing

Input Image

A grayscale image is input.

Corner Point Detection

The corner points to be used for recognition are detected from the input image.

Point-to-Point Matching

The dictionary files for recognition targets registered to SarRecognizer are matched with detected corner points, finding their correspondence relationships.

Geometrical Constraints

The geometrical consistency of the correspondence relationships found through point-to-point matching is verified (positional relationships such as up-down, left-right, etc.).

Output

The results of target recognition are output. If recognition is successful, it will be possible to obtain the camera's pose in the coordinate system having the recognition target's center as its origin. Once

recognition is successful, the library will enter tracking state, and processing speed from the next frame onwards will become faster.

For Better TargetTracking Mode Performance

Conditions for Obtaining Good Recognition Results

The TargetTracking mode will run better under the following conditions.

- Print Of planar shape with no irregularities
- Including more complex textures
- Including more corner points
- Strong contrast when the gray image

Comparison with SmartAR™ SDK Sample Program

The sample programs distributed with SmartAR™ SDK are made to ensure that SmartAR™ SDK performs fully. Try comparing SmartAR™ SDK's operation with that of sample programs. If initializing with a natural image, run the sample program by replacing its dictionary with a dictionary you have created to achieve a more accurate comparison and to find out more easily whether any issues are caused by the program or by the dictionary.

Note
If the performance drops significantly in the replacement of the dictionary, please compare the aspect ratio of the dictionary and the input image.

Structure of the SceneMapping Mode's Processing

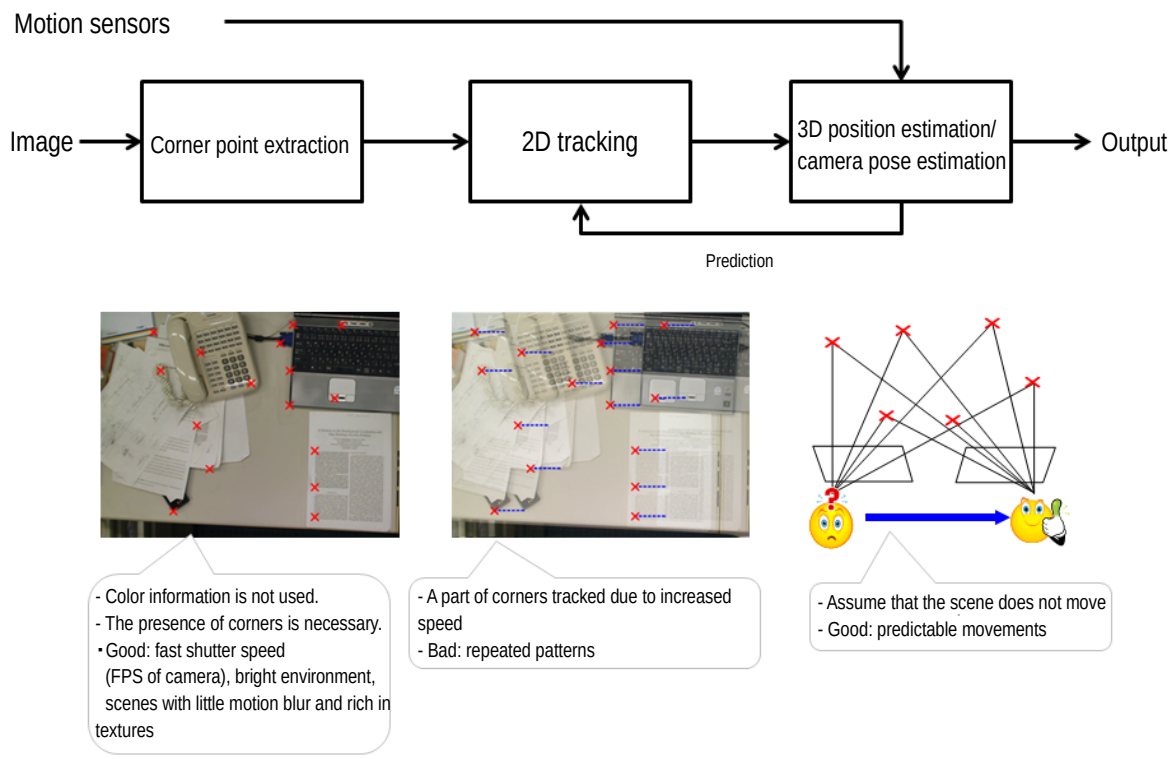

Figure 3 Diagram of the SceneMapping Mode's Processing

Image and Motion Sensor Data

A VGA-size grayscale image and the motion sensor measurement values obtained from SarSensorDevice are input.

Corner Point Extraction

Corner points are extracted from a camera image to add new landmarks to the scene map.

2D Tracking

Landmarks in the library appearing on the camera image are searched for within the image.

3D Position Estimation/Camera Pose Estimation

The library estimates a camera pose where the landmarks in the scene map appear in 2D-tracked positions in the camera image, and simultaneously estimates the 3D position of landmarks in the scene map.

Output

The library outputs the camera's pose (position and orientation) based on the scene map.

For Better SceneMapping Mode Performance

Conditions for Obtaining Good Recognition Results

The SceneMapping mode will run better under the following conditions.

- Unchanging environments
- Environments rich in corner points appearing on the camera image
- Fast shutter speed
- Bright environments
- Images with little motion blur
- Environments with few repeated patterns
- Smooth and predictable movements of device running SmartAR™

Points to Bear in Mind for Processing

Given that the analysis of continuous images carried out by the SceneMapping mode becomes more difficult the longer the interval between the images, calculation volume for the analysis may increase, or the accuracy of calculation results may decrease.
To ensure the SceneMapping mode's full performance, it is important to continue processing images as fast as possible.

Comparison with SmartAR™ SDK Sample Program

The sample programs distributed with SmartAR™ SDK are made to ensure that SmartAR™ SDK performs fully. Try comparing SmartAR™ SDK's operation with that of sample programs. If initializing with a natural image, run the sample program by replacing its dictionary with a dictionary you have created to achieve a more accurate comparison and to find out more easily whether any issues are caused by the program or by the dictionary. If operation with the sample differs greatly, find out and correct the cause, making reference to the following:

Checking the Camera's Frame Rate

Check that the camera is running at sufficient frame rate. Check that the time between image capture and its input to SarRecognizer::dispatch() is as short as possible. If sensor information is used, make sure the value read from sensor is fed to SarRecognizer::dispatch() as soon as possible.

Increasing the Camera's Frame Rate

Design buffer and thread carefully to make captured image be fed to SarRecognizer::dispatch() as soon as possible.

Checking for Chronological Gaps between the Results of the Analysis and Superimposed Display

The SarRecognitionResult that is output is calculated based on the time of the input image, and the input image's timestamp is therefore attached to it. Output to the screen is created by superimposing CG on the background of the camera image. Check the difference between the background camera image's timestamp and the timestamp of the SarRecognitionResult used for CG drawing. Ideally, this difference should be small or 0, but it is not essential.

Correcting Chronological Gaps between Analysis Results and Superimposed Display

In order to correctly match a camera image with CG on the screen, you will need to use either or both of the methods below. Consider this so as to match the game's features.

(a) Propagate SarRecognitionResult to the time of the latest camera image, and output to the screen upon superimposing CG with the propagated *result*.

(b) Buffer the camera image corresponding to SarRecognitionResult, and, when the *result* is output, output to the screen upon superimposing CG on the corresponding camera image.

In the sample program we have adopted method (b).
With this method in principal CG and camera image has no gap.
When using (a) actual chronological gap between the movement of the device and the camera image is smaller, but the propagated result may propagate gap as well. This may result in friction and gap of rendered CG.

Checking status of SarRecognizer::dispatch() and SarRecognizer::runWorker()

Use profiler to profile the program. Make sure that SarRecognizer::dispatch() and SarRecognizer::runWorker() are called continuously.

Improve working of SarRecognizer::dispatch() and SarRecognizer::runWorker()

If necessary set CPU mask and priority of thread to fit the task.

Coordinate System

All coordinate systems are right-handed. Position and orientation are represented by position vectors and orientation quaternions. Position vectors are camera-centered position vectors in a scene-fixed coordinate system. Orientation quaternions represent rotation of the device-fixed coordinate system from the scene-fixed coordinate system.
Different world coordinate systems are defined depending on how the SceneMapping mode is initialized, as shown below:

- Initialization with a natural image
- Initialization with HFG
- Initialization with VFG
- Initialization with SFM

Definition of the World Coordinate System during Initialization with a Natural Image

The world coordinate system at the time of initialization with a natural image (henceforth, the recognition target) is defined with the recognition target's center as its origin, the direction from the recognition target's back to its front as axis Z, and the direction from the recognition target's left to right when seen from the front as axis X.

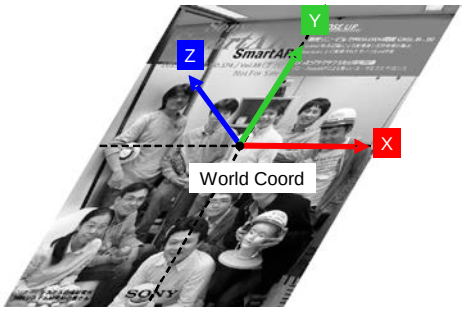

Figure 4 Definition of the Coordinate Axis when Initializing with a Natural Image

Definition of the World Coordinate System during HFG Initialization

The world coordinate system during HFG initialization is defined with the point on the horizontal plane appearing at the camera image's center as its origin, the direction opposite to that of gravity as axis Z, and the direction of the vector projected on the horizontal plane from the camera to the origin as axis Y.

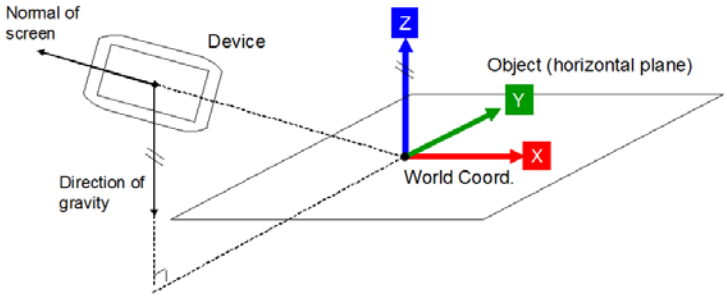

Figure 5 Definition of the Coordinate Axis during HFG Initialization

Definition of the World Coordinate System during VFG Initialization

The world coordinate system during VFG initialization is defined with the point on the vertical plane appearing at the camera image's center as its origin, the vertical plane's normal line as axis Z, and the direction opposite to that of gravity as axis Y.

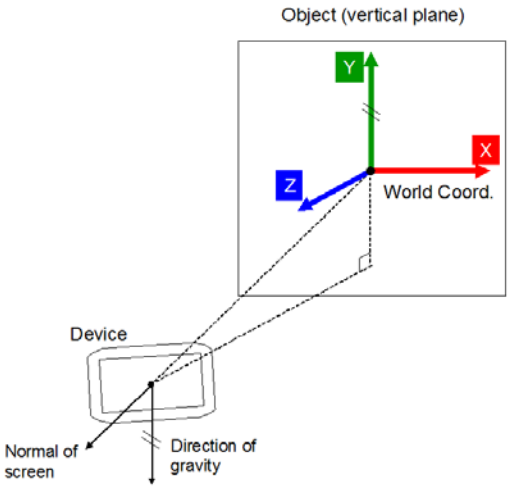

Figure 6 Definition of the Coordinate Axis during VFG Initialization

Definition of the World Coordinate System during SFM Initialization

The world coordinate system during SFM initialization is defined by calculating the dominant plane of the camera image's center area from a 3D structure, with the point on the dominant plane appearing at the camera image's center as the origin, the dominant plane's normal line as axis Z, and the direction of the vector projected on the dominant plane from the camera to the origin as axis Y.

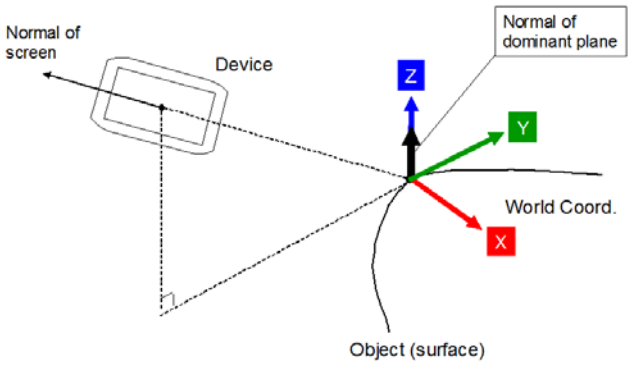

Figure 7 Definition of the Coordinate Axis during SFM Initialization