CTR
AR Library Programming Manual

2014/04/17

Version 1.3

The content of this document is highly confidential and should be handled accordingly.

CTR-06-0155-001-D
Released: June 3, 2014

Table of Contents

Code

Tables

Figures

1 Introduction

This document explains how to use the AR library.

The main purpose of the AR library is to detect several types of markers in real time. When detecting a marker, the library recognizes the marker's type and determines its position and orientation. This library assumes that the markers it detects are graphical cards (AR Cards).

1.1 Glossary

This section explains the terminology that this document uses to discuss augmented reality and the AR library.

- **AR**
  An abbreviation of "augmented reality."
- **AR Application**
  An application that, among other things, composites 3D models with images of real space in response to markers it identifies and to characteristics of those markers such as their position. The AR library assumes that AR Cards are the markers and is provided to enable applications to display 3D models according to the position and orientation of AR Cards in an image.
- **AR Card**
  A graphical card with a design that can be identified by the AR library. (Six representative cards are bundled with the CTR system.) You can create your own original AR Cards for your application. For more information on creating AR Cards, see the *Development Guide for Applications That Use AR Cards*.
- **Marker**
  A symbol used to identify an AR Card. The AR library uses a card's design as its marker.
- **Marker Template**
  Data that is a numeric representation of an AR Card's design; this data is used to identify a marker.
- **Marker Database**
  A database that registers multiple marker templates you want to identify.
- **Marker Coordinate System**
  A coordinate system whose origin is placed at the center of a detected marker.
- **Camera Coordinate System**
  A coordinate system that represents the width and height of an input image as numbers in the range (-1.0, +1.0), and whose origin is placed at the center of that input image.

2 Building

This chapter explains the requirements for building an application that uses the AR library. The code samples contain code that must be placed in the OMakefile you use with the CTR-SDK build system.

2.1 Adding the Include Path

You must add `$(CTRMW_NAR_ROOT)/include/mw/nar` to the compiler's include path.

Code 2-1 Adding the Include Path

```
INCLUDES += $(CTRMW_NAR_ROOT)/include/mw/nar
```

2.2 Adding Compiler Flags

You must add `-DNAR_CTR__` to the compiler flags. Note that there is a single underscore between "`NAR`" and "`CTR`" and two underscores after "`CTR`."

Code 2-2 Adding Compiler Flags

```
CCFLAGS += -DNAR_CTR__
```

2.3 Adding Library Files

Pass `libmw_nar.*.a` (where `*` is either `fast` or `small`) as a library file to the linker.

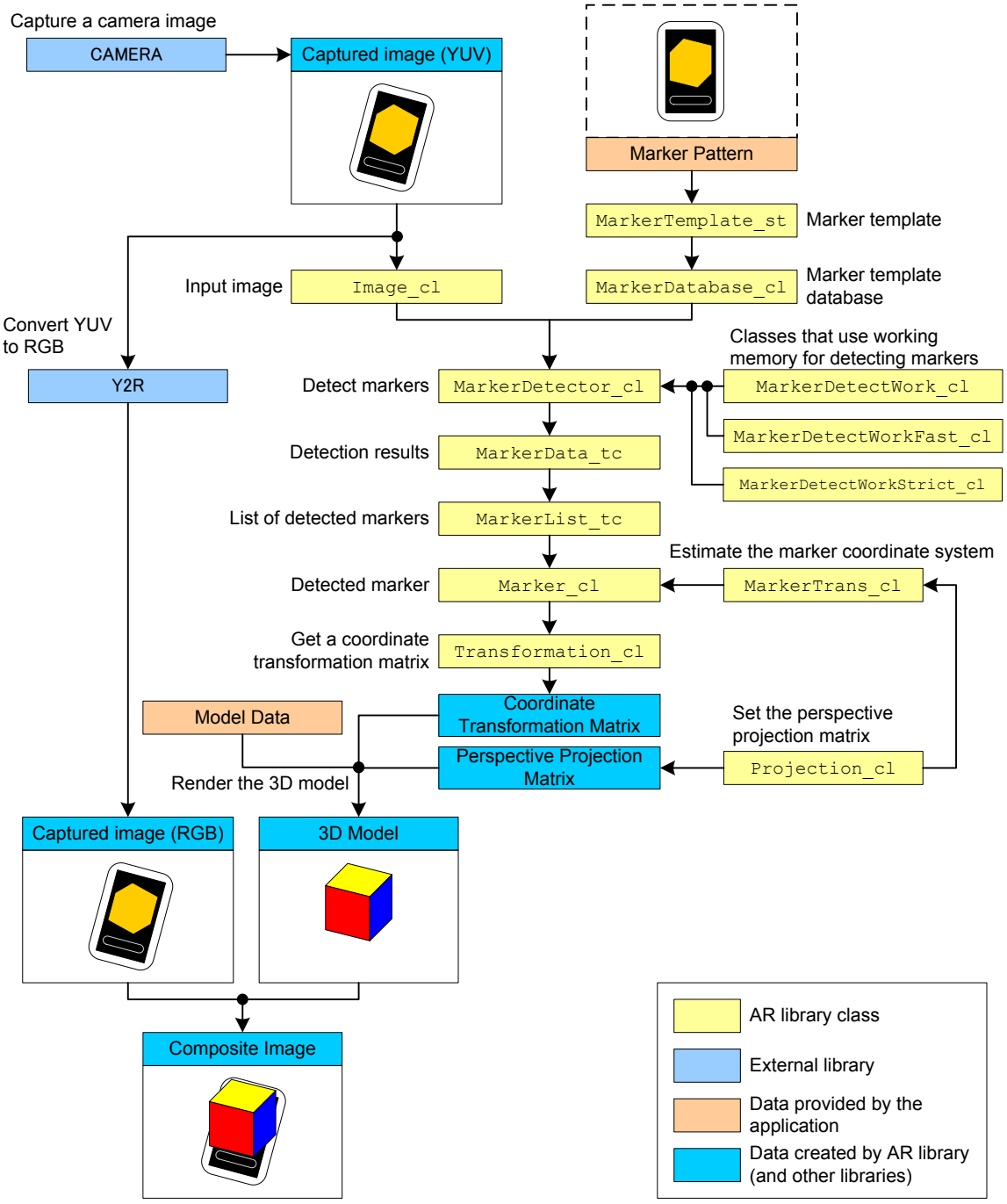
Code 2-3 Adding Library Files

```
LIBFILES = $`(addprefix $(CTRMW_NAR_ROOT)$(DIRSEP)libraries$(DIRSEP)$(config.
getTargetSubDirectory true)$(DIRSEP), libmw_nar)
```

3 How to Use the Library

This chapter explains how to implement applications that use the AR library and traces the flow the library uses to process data. The following figure shows the AR library classes used by applications and a process flowchart.

Figure 3-1 AR Library Classes and Process Flowchart



The following table shows how processes are split into stages and summarizes the AR library classes associated with each stage. The stages are explained in subsequent sections.

Table 3-1 Process Stages and Related Classes

Process Stage	Related Classes	Overview
Preparing the Marker Template	• `MarkerTemplate_st` • `MarkerDatabase_cl`	Registers marker information in a database.
Registering the Input Image	• `Image_cl`	Registers an image in which to detect markers.
Detecting the Markers	• `MarkerDetector_cl` • `Image_cl` • `MarkerDatabase_cl` • `MarkerData_tc` • `MarkerDetectWork_cl` • `MarkerDetectWorkFast_cl` • `MarkerDetectWorkStrict_cl`	Detects markers registered in the database within the input image.
Getting the Detection Results	• `MarkerData_tc` • `MarkerList_tc` • `Marker_cl`	Gets individual detection results from a list.
Using the Detection Results	• `MarkerTrans_cl` • `Marker_cl` • `Projection_cl` • `Transformation_cl`	Gets a coordinate transformation matrix for displaying 3D models from the detection results.

3.1 Required Header Files

The AR library's header file is `nar.h`. Classes and everything else defined by the AR library belong to the `mw::nar` namespace.

Code 3-1 Including the Header File

```
#include "nar.h"
```

For the most part, the AR library does not rely on other libraries. Two exceptions, however, are the MATH and CAMERA libraries. The MATH library is required to get perspective and transformation matrices and the CAMERA library is required to use camera images as input.

3.2 Initialization

The AR library has no initialization process. It also has no explicit finalization process.

When you need to process something with the AR library, create instances of the necessary classes.

3.3 Preparing the Marker Template

A marker template contains information used to identify markers in an input image. The template defines the size and design of a marker. Marker templates are defined by the `mw::nar::MarkerTemplate_st` structure.

Code 3-2 Marker Template Structure

```
struct mw::nar::MarkerTemplate_st
{
   s32                id;
   f32                aspectRatio;
   f32                sideLength;
   MarkerPattern_st      pattern;
   MarkerTemplate_st *    p_Next;
};
```

The following table describes the values to specify in each member variable of the structure.

Table 3-2 Member Variables of the Marker Template Structure

Member Variable	Description
`id`	An ID used to identify the marker. You can specify any value of 0 or greater.
`aspectRatio`	The marker's aspect ratio. Specify the width of the marker divided by `sideLength`.
`sideLength`	The marker's height. By defining this value with the marker's actual dimensions, you can display a 3D model of nearly the same size on top of the marker. In the sample demos, this is specified as the actual length of 7.36 cm.
`pattern`	Pattern data representing the marker's design; this is used to detect the marker. Use the sample demo `marker_maker` to create pattern data.
`p_Next`	This is used by the library. Applications must not change this value.

Marker Template for the "?" Card

The "?" Card is an AR Card that does not require any special license and can be freely used by applications.

You can use the marker template for the "?" Card, which is included with the sample demos, to implement an AR application that uses "?" Cards. To match the size of a 3D model to a "?" Card, use a `sideLength` value of 7.36 cm, which is the actual measured length of the marker's long side.

3.3.1 Marker Database

The marker database registers multiple marker templates that might be detected. The marker database is defined by the `mw::nar::MarkerDatabase_cl` class.

Code 3-3 Marker Database Class

```
class mw::nar::MarkerDatabase_cl
{
   MarkerDatabase_cl(f32 th = 0.5f);
   bool Register(mw::nar::MarkerTemplate_st &r_Template);
   bool IsRegistered(const mw::nar::MarkerTemplate_st &cr_Template);
   bool Unregister(mw::nar::MarkerTemplate_st &r_Template);
   void UnregisterAll();
   void SetThreshold(f32 th);
   f32 GetThreshold() const;
}
```

The constructor argument *th* is the threshold value for detecting markers. If a marker is detected with a score at or above the threshold value, it is treated as a match. Specify a threshold value in the range (-1.0, +1.0). As this value approaches +1.0, images are checked more strictly to be considered a match for the marker template. In contrast, if you specify a value of -1.0 even a negative image (with its color inverted) will be considered to be a match.

You can change the threshold value with the `SetThreshold` function. You can get the current threshold value with the `GetThreshold` function.

Note: If only one marker template is registered in the database and the threshold is at its default value of 0.5, an unregistered AR Card might be detected as a match (a false positive). Run tests using various threshold values to determine an appropriate threshold setting.

The `Register` function registers a marker template in the marker database. This function returns `true` when it registers a template successfully and `false` when it fails. A failure indicates the template is already registered, the number of marker templates that can be registered in the database has been exceeded, or some other reason. Use the `IsRegistered` function to determine whether a marker template is already registered in the database.

Note: You may be able to reduce the number of false positives by registering markers that are erroneously detected as matches when the threshold value is low. However, increasing the number of marker templates in the database also increases the time required to detect a marker.

The `Unregister` and `UnregisterAll` functions unregister marker templates from the marker database. The former unregisters a specific marker template and the latter unregisters all marker templates.

3.4 Registering the Input Image

An *input image* is an image in which you want to detect a marker. Markers can be detected in YUV422 packed linear format images. Images output by the CAMERA library can be used as input, therefore, without modification.

The AR library uses the `mw::nar::Image_cl` class to handle input images.

Code 3-4 Class That Handles Input Images

```
class mw::nar::Image_cl
{
    Image_cl(u16 w, u16 h, u32* p_Work, u16* p_DebugImg = 0);

    void SetImage(const u16* cp_Image);
    u16  GetHeight() const;
    u16  GetWidth() const;
    u8   GetXStep() const;
    u8   GetYStep() const;
    void SetXStep(u8 x);
    void SetYStep(u8 y);
    u16  GetColor(u32 x, u32 y) const;
    void SetBorder(mw::nar::Border_en b);
    mw::nar::Border_en GetBorder();
    void SetDebugImage(u16* p_Image);
}
```

Specify the width and height (in pixels) of the input image with the constructor's ***w*** and ***h*** arguments, respectively. Pass a pointer to the working memory for detection processing with ***p_Work***; this working memory must be a `u32` array no smaller than the value calculated by the `NAR_IMAGE_WORK_SIZE_4TH` macro. Use ***p_DebugImg*** to display debugging information. For more information about debugging, see section 3.4.2 Debugging Information.

Use the `SetImage` function to register an input image (YUV422 linear format). Pass the starting address of the buffer storing the input image into ***cp_Image***. The input images are usually unmodified camera images obtained by the CAMERA library, so we recommend the alignment of the buffer's starting address to be a multiple of 4 that is at least 64 bytes.

Note: Do not destroy or overwrite the buffer (registered with the `Image_cl` class) that stores the input image currently being processed until you have finished detecting markers in that image.

Use the `SetXStep` and `SetYStep` functions to set the number of horizontal and vertical pixels to be processed each time you look for markers in an input image. For both directions these step intervals must be a minimum of one pixel wide. You can use the `GetXStep` and `GetYStep` functions to get the current settings. The default settings are defined as `msc_DefaultXStep` (8) and `msc_DefaultYStep` (16). Although you can increase these settings to speed up processing input images, it may then no longer be possible to detect small markers.

The `GetColor` function gets the color of the input image at the coordinates specified by ***x*** and ***y*** (the reference point). This gets a color in RGBA5551 format.

You can use the `SetBorder` and `GetBorder` functions to specify the border color (the color of the blank frame surrounding the marker) and to get the current setting. By specifying the border color, you can choose to have the AR library detect either black markers within a white border or white markers within a black border. Choose the border color to pass to the ***b*** argument from the enumerated type defined by `mw::nar::Border_en`.

Table 3-3 Specifying the Border Color (mw::nar::Border_en)

Definition	Description
`e_BorderWhiteBlack`	Black background within a white border: The marker design uses a black background. This is the default setting.
`e_BorderBlackWhite`	White background within a black border: The marker design uses a white background.

3.4.1 Camera Settings When Camera Images Are Used as Input

AR applications normally combine rendered models with camera images. This section explains recommended settings and precautions related to using CTR camera images as the input images.

3.4.1.1 512x384 Pixels Is the Recommended Camera Resolution

We recommend a camera resolution of 512x384 pixels (`nn::camera::SIZE_DS_LCDx4`). Although this resolution's width and height both have more pixels than the LCD resolution, its width has the same number of pixels as an image that is converted to RGB with the Y2R library and then treated as a texture of size 512x512.

3.4.1.2 Set a High Brightness

We recommend setting a brightness of at least +3 to make markers show up clearly in camera images.

3.4.1.3 Set a Low Exposure

Setting a negative exposure value reduces blurring in the camera image. However, setting a low exposure darkens the entire image, so some color adjustment (such as brightening) is necessary when you display the image.

3.4.1.4 Avoid Effects That Make Large Changes to the Hue

The AR library identifies a marker's design from the color of sample points on the marker. The marker will not be detected properly, therefore, when effects cause large changes to the hue (such as inverting the colors for a “negative” effect or applying a sepia tone). For example, if a sepia effect is applied to the entire image, every AR Card bundled with the CTR system have a high probability of being identified as a "?" Card in that image because its cube silhouette will appear yellow.

3.4.2 Debugging Information

You can have the library generate debugging information by using the ***p_DebugImg*** argument to the `Image_cl` class's constructor to set an image region in which to display debugging information. If you do not need debugging information, do not specify any value other than `NULL` for ***p_DebugImg*** in the

constructor or for the argument to the `SetDebugImage` function. Generation of debugging information is disabled for release builds (when `BUILD=Release`).

Debugging information is generated as a texture image. Because the color format of this texture image is block-format RGBA5551, the texture image must be at least as large as the input image and its formats must be `GL_RGBA_NATIVE_DMP` and `GL_UNSIGNED_SHORT_5_5_5_1`. The amount of memory (in bytes) needed to display debugging information in an image region is yielded by the formula: [texture height × texture width × 2].

Note: Debugging information might not be displayed correctly if the input image and the texture do not have the same width.

The `Debug_cl` class manages the debugging information that is generated, and `Debug_cl::Switch_e` defines the type of debugging information. The **RGBA** column below uses RGBA5551 values to indicate the color with which the debugging information is rendered.

Table 3-4 Types of Debugging Information

Definition	Debugging Information	RGBA
`me_ViewLineCandidates`	Edge candidates. Lines are drawn around the edges of the marker.	(10, 31, 10, 1) (31, 31, 10, 1)
`me_ViewCheckMesh`	Mesh for evaluating a marker. This mesh draws lines that split the marker into sixteenths both vertically and horizontally.	(10, 31, 10, 1)
`me_ViewEdgeSlope`	The slope. Straight lines are drawn parallel to the edges of the marker.	(31, 3, 3, 1) (3, 31, 31, 1)
`me_ViewSample`	Sample points calculated using the least squares method. Points are rendered where marker edges intersect.	(31, 15, 0, 1)

3.5 Detecting the Markers

Although you use the `Marker_Detector_cl` class's `Detect` function to detect markers, you cannot create an instance of the `Marker_Detector_cl` class. Call `MarkerDetector_cl::Detect` directly.

Code 3-5 Marker Detection Class

```
class mw::nar::MarkerDetector_cl
{
public:
   static s32 Detect(
      mw::nar::Image_cl& r_Image, mw::nar::MarkerDatabase_cl& r_DB,
      mw::nar::MarkerData_ac& r_Data,
      mw::nar::MarkerDetectWork_ac& r_Work);
};
```

Pass into ***r_Image*** the instance of the `Image_cl` class that set the input image. After you have finished detecting markers in that image, you can use the `SetImage` function to register a new input image in the instance that you just passed into this function and then re-use that instance to detect more markers. As a result, you do not have to re-create an instance of the `Image_cl` class each time the input image changes, as long as the size of the input image remains the same.

Pass ***r_DB*** an instance of the `MarkerDatabase_cl` class in which you have registered the marker templates for the markers that you want to detect.

The ***r_Data*** argument takes an instance of the `MarkerData_ac` class—an abstract class—to accept detection results. To create the instance that you will actually pass as an argument, specify the *number of markers it is possible to detect* to the concrete class `MarkerData_tc`. If you are detecting markers in a series of input images from the same camera, pass the same instance of this class every time.

The ***r_Work*** argument takes an instance of the `MarkerDetectWork_ac` class—an abstract class—to handle working memory for detection processing. You actually pass an instance of the `MarkerDetectWork_cl`, `MarkerDetectWorkFast_cl`, or `MarkerDetectWorkStrict_cl` concrete class. During detection, `MarkerDetectWork_cl` references the average value of three points around each sample point. `MarkerDetectWorkFast_cl` is faster because it references only the sample point, but it is not quite as accurate at detecting markers. You must therefore be careful to prevent detection errors when you use `MarkerDetectWorkFast_cl`. `MarkerDetectWorkStrict_cl` also references the average value of three points around each sample point during detection, and additionally uses the vanishing point to divide up detected marker regions. As a result, this is the most accurate detection method but it causes a decrease in processing speed. You can use instances of these objects as you see fit for your particular application, but we recommend using `MarkerDetectWorkStrict_cl` unless you have a specific reason to do otherwise.

This function returns the number of markers that it detected. Get information on the detected markers through the instance passed to ***r_Data***.

3.6 Getting the Detection Results

The instance of the `MarkerData_tc` class that accepted the current detection results also maintains the previous detection results.

Code 3-6 Class That Accepts Detection Results

```
template < s32 N >
class mw::nar::MarkerData_tc : public mw::nar::MarkerData_ac
{
public:
   virtual mw::nar::MarkerList_ac& GetrDetectingMarkerList();
   virtual mw::nar::MarkerList_ac& GetrLastMarkerList();
private:
```

```
    mw::nar::MarkerList_tc< N >  ma_MarkerLists[ 2 ];
};
```

The `GetrDetectingMarkerList` function gets the current list of detected markers and the `GetrLastMarkerList` gets the previous list of detected markers.

Instances of the `MarkerList_tc` class for maintaining lists of detected markers are automatically generated when an instance of the `MarkerData_tc` class is created.

Code 3-7 Class That Maintains a List of Detected Markers

```
template < s32 N >
class mw::nar::MarkerList_tc : public mw::nar::MarkerList_ac
{
public:
    u16    GetMarkerNum();
    virtual s32 GetMarkerNumWithID(s32 id);
    virtual mw::nar::Marker_cl* GetpMarker(u32 idx);
private:
    mw::nar::Marker_cl ma_Marker[ N ];
};
```

The `GetMarkerNum` function gets the number of detected markers in the list. You can use the `GetMarkerNumWithID` function to get the number of markers that have the ID specified by ***id*** from all the detected markers in the list.

To get individual markers in the list, specify an index to the `GetpMarker` function. `NULL` is returned when you specify an index greater than or equal to the number of detected markers.

3.7 Using the Detection Results

To display models at the positions of individual detected markers, you need to understand both the marker coordinate system and the camera coordinate system.

Marker Coordinate System

The marker coordinate system is used to display 3D models or other images on top of markers detected in the input image. This coordinate system is defined in terms of the marker: the positive x-axis points across the marker from its left to its right, the positive y-axis points through the marker from its underside to its face side, and the positive z-axis points across the marker from its top to its bottom. The origin is roughly at the center of the marker, but it may be shifted slightly (for example, when there is an error detecting the outline of the marker). You can calculate the coordinates of the four corners of the marker from the marker template's `sideLength` and `aspectRatio`.

Figure 3-2 Marker Coordinate System Axes

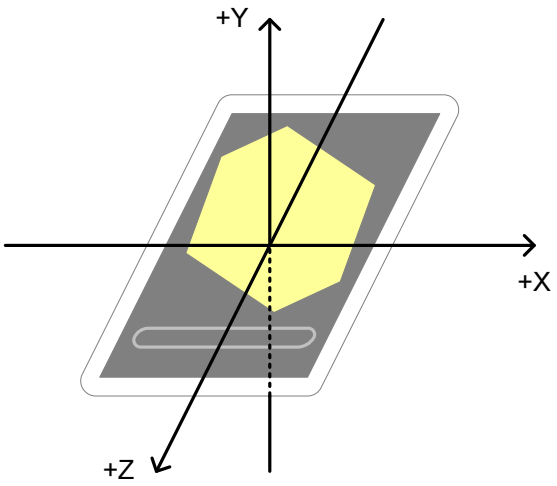
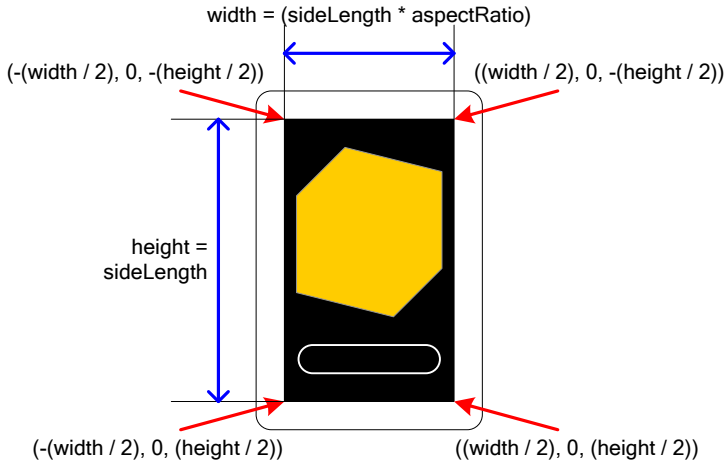


Figure 3-3 Coordinates of the Four Corners of the Marker (in the Marker Coordinate System)



Camera Coordinate System

The camera coordinate system has its origin at the center of the input image and uses numbers in the range -1.0 to +1.0 to represent the width and height of the image. The upper-left corner and lower-right corner of the input image have the coordinates (-1.0, -1.0) and (+1.0, +1.0), respectively.

3.7.1 Detected Marker Information

The `Marker_cl` class maintains information on the individual markers that are detected.

Code 3-8 Class That Maintains Information on Individual Detected Markers

```
class mw::nar::Marker_cl
{
public:
    s32 GetID() const;
    f32 GetScore() const;
    f32 GetSideLength() const;
    f32 GetAspectRatio() const;
    void SuppressShake(f32 sim, f32 far);
    bool IsEstimated() const;
    mw::nar::Transformation_cl& GetrTransformation();
    const mw::nar::Vec3F_st& GetrNormal();
};
```

The `GetID` function gets the ID of the detected marker. The ID obtained by this function is the same value specified by `id` in the marker template of this marker.

The `GetScore` function gets the detection score of the marker. This score approaches +1.0 as the detected design more closely resembles the marker template's design.

The `GetSideLength` and `GetAspectRatio` functions get the length of the left side and the aspect ratio, respectively, of the detected marker. These values are the same as the `sideLength` and `aspectRatio` specified in the marker template.

The `SuppressShake` function corrects the marker's position by minimizing the effect of shaking hands and trembling. To do so, this function finds the distance (in pixels) between the current and previous positions of the detected marker on the input image, and *calculates the square of this distance*. If the result is less than or equal to ***sim***, the marker's position is corrected to its previous one. If the result is less than or equal to ***far***, the marker's position is corrected to the average of its current and previous positions.

The `IsEstimated` function returns a value indicating whether the process of estimating the marker coordinate system has finished. In other words, this function returns `true` either when the `MarkerTrans_cl` class's `Estimate` function has finished estimating the marker coordinate system, or when correction by the `SuppressShake` function has determined that the current marker coordinate system is identical to the previous marker coordinate system. Once the process of estimating the coordinate system has finished, you can use the `GetrTransformation` function to get a transformation matrix that transforms from marker coordinates to camera coordinates. You can also use the `GetrNormal` function to get the marker's normal vector in camera coordinates.

3.7.2 Estimating the Marker Coordinate System

The marker coordinate system is estimated from the position and orientation of its marker. The `MarkerTrans_cl` class estimates the marker coordinate system. Use the `MarkerTrans_cl` class's `Estimate` function to estimate the marker coordinate system after you have first used the `Marker_cl` class's `IsEstimated` function to check that the process of estimating the marker coordinate system has not finished.

Code 3-9 Class That Estimates the Marker Coordinate System

```
class mw::nar::MarkerTrans_cl
{
public:
   MarkerTrans_cl();
   bool Estimate(mw::nar::Marker_cl& r_Marker,
               const mw::nar::Projection_cl& cr_Proj) const;
   void SetMildRotation(bool b, f32 t = 0.5f);
   void SetAccelerationOnMildRotation(bool b, f32 t = 0.25f);
};
```

You need an instance of the `Marker_cl` class (which has marker information) and an instance of the `Projection_cl` class (which handles the perspective projection matrix) to run the `Estimate` function. You can get an instance of the `Marker_cl` class from the detection results. Your application is responsible for creating an instance of the `Projection_cl` class.

Code 3-10 Class That Handles the Perspective Projection Matrix

```
class mw::nar::Projection_cl
{
public:
   Projection_cl();
   Projection_cl(f32 w, f32 h, f32 near, f32 far, f32 aov = 66.0);

   void Set(f32 w, f32 h, f32 near, f32 far, f32 aov = 66.0);
   void GetMTX44(nn::math::MTX44& r_Mtx) const;
   void GetTrimmed(nn::math::MTX44& r_Mtx,
          f32 l, f32 r, f32 b, f32 t, f32 n, f32 f, f32 aov = 66.f) const;
};
```

If you create an instance of the `Projection_cl` class using its default no-arguments constructor, you must call the `Set` function to set properties of the input image such as its width and height. The constructor and the `Set` function take the same arguments: ***w*** and ***h*** specify the width and height of the input image; ***near*** and ***far*** specify the distance to the near and far planes; and ***aov*** specifies the

angle of view (in degrees). The input image is usually captured by a camera, so the ***aov*** value is obtained from the camera's calibration data.

You can get the perspective projection matrix with the `GetMTX44` function. If you need a trimmed perspective projection matrix—for example, when the screen display and the input image have different ratios—get it with the `GetTrimmed` function. Specify the `GetTrimmed` function's ***l***, ***r***, ***b***, and ***t*** argument values in camera coordinates, in which the untrimmed x and y coordinates fall in the range (-1.0, +1.0).

Interpolating Rotation of the Coordinate System

You can use the `MarkerTrans_cl` class's `SetMildRotation` and `SetAccelerationOnMildRotation` member functions to specify how to interpolate coordinate system rotation during the process of estimating the marker coordinate system.

If you specify ***b*** to be `true` in the `SetMildRotation` function, then whenever the previously estimated and current coordinate systems are judged to have similar rotation, the previous coordinate system is applied to interpolate the current coordinate system by the ratio (ranging from 0.0 to 1.0) specified by ***t***. In other words, the current coordinate system is used when ***t*** is 0.0 and the previous coordinate system is used when ***t*** is 1.0. The `Marker_cl` class's `SuppressShake` function will not make any corrections while this interpolation is in progress. Interpolation is skipped when ***b*** is `false`.

The `SetAccelerationOnMildRotation` function takes the same ***b*** and ***t*** arguments as the `SetMildRotation` function, but is different because it uses the similarity between the current and previous motion of the marker position to determine whether to perform interpolation. This interpolation method allows the coordinate system to quickly track rotations that are judged to have a near-uniform speed and disregard shaking hands and other transient vibrations.

3.7.3 Converting from Marker Coordinates to Camera Coordinates

You can use the `Transformation_cl` class's `GetMTX34` member function to get a matrix that converts marker coordinates into camera coordinates.

Code 3-11 Class That Converts Marker Coordinates into Camera Coordinates

```
class mw::nar::Transformation_cl
{
public:
   void GetMTX34(nn::math::MTX34& r_Mtx) const;
};
```

The application does not create an instance of the `Transformation_cl` class, but instead uses the `GetrTransformation` member function of the detected marker's class (`Marker_cl`) to get an instance as shown in the following sample code.

Code 3-12 Sample Code to Get a Transformation Matrix from a Detected Marker

```
nn::math::MTX34 mtx34;
mw::nar::Marker_cl* pMarker;
pMarker = markerData.GetrDetectingMarkerList().GetpMarker(0);
pMarker->GetrTransformation().GetMTX34(mtx34);
```

3.8 Working with the Stereo Cameras

This section provides precautionary information related to using the stereo cameras installed on the CTR system to create an AR application that supports stereoscopic 3D images.

3.8.1 Camera Settings

This section explains recommended settings, precautions, and other information related to using stereo camera images as the input images.

3.8.1.1 Turn Off the Noise Filter

We recommend that you turn off the noise filter when using the stereo cameras.

3.8.1.2 Synchronize the Stereo Cameras' V-Sync

If the V-Sync timing is not synchronized between the stereo cameras, the images captured from the left and right cameras may have subtle differences that affect the stereoscopic image. We recommend that you call the `nn::camera::SynchronizeVsyncTiming` function to synchronize the V-Sync between the left and right cameras before beginning to get camera images, and call this function again whenever a large V-Sync timing discrepancy develops.

3.8.1.3 Synchronize the Stereo Cameras' Brightness

We recommend that you implement your application in a way that keeps the brightness of the left and right images as close as possible. Ways to do this include: calling the `nn::camera::SetBrightnessSynchronization` function to synchronize the stereo cameras' brightness; and taking parallax into account when you set the region on which automatic adjustments to the white balance and exposure will be based.

3.8.1.4 Avoid Variable Frame Rates

The V-Sync timing of the stereo cameras is very likely to lose synchronization when you use a variable frame rate.

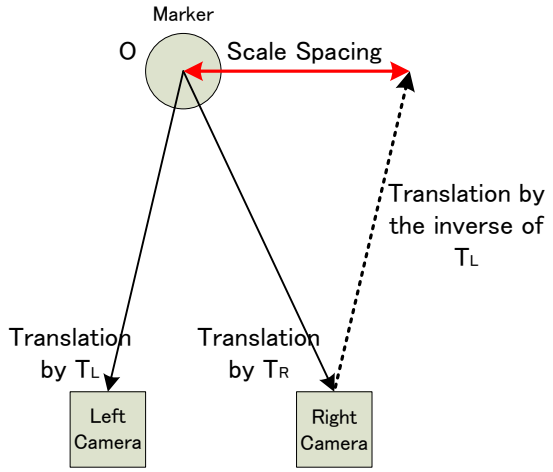
3.8.2 Measuring the Distance to the Marker

Carry out a zero-vector transform using a transformation matrix from the marker coordinate system to the camera coordinate system to get the marker position in the camera coordinate system. The marker position Z-axis value obtained from the transform is the distance from the camera to the marker in the camera coordinate system. Use this value as the ratio of the distance between the stereo cameras in the camera coordinate system (i.e., the scale spacing) to the actual distance between the stereo cameras in order to determine the actual distance.

Calculate the scale spacing as follows.

Taking the transformation matrix as the translation from the origin, multiply this by the inverse of the transformation matrix used for the camera on the other side to translate the origin from the marker by the camera's placement width amount. In other words, for a marker position of O, and transformation matrices obtained from the left and right camera image captures of T_L and T_R, the scale spacing would be a length of $OT_R(T_L)^{-1}$ (left and right can be swapped).

Figure 3-4 Calculating the Scale Spacing



Provided that the optical axis for the left and right cameras is completely horizontal, you only need to calculate the scale spacing once. However, if the optical axis is not horizontal due to stereo camera placement error, the scale spacing will change even after calibrating, depending on the distance to the marker. Also, if there are differently sized markers, the scale spacing value can change greatly. Note that extreme variations in the scale spacing value can lead to unexpected differences in all processing and displays that are based on it.

Note: The horizontal translation from the calibration data (`translation`) includes the actual stereo camera placement width, in *pixels*. The placement width value (`distanceCameras`), however, is specified in *millimeters*.

3.8.3 Reducing the Processing Load for Marker Detection

When getting the scale spacing, detect the marker from the captured image for one side, then translate the transformation matrix from the marker coordinate system in the X direction by just the scale spacing amount to derive the transformation matrix for the other side. This allows you to finish marker detection with just one image capture, thereby reducing the processing load. When using this method, Nintendo recommends re-measuring the scale spacing at regular intervals, as this also helps accurately place a 3D model at the location of the marker.

In addition, use whether the marker is to the left or right of the center of the captured image to determine whether to use the captured image from the left or right camera (i.e., to select the main camera) and thereby keep the marker fully within the field of view.

4 Description of the Sample Demos

This chapter explains the AR library sample demos in `$(CTRMW_NAR_ROOT)/sampledemos`, the classes used by these demos, implementation notes, and so on.

4.1 Sample Demos

The following table shows all of the sample demos that are currently provided.

Table 4-1 Sample Demos

Sample Demo	Overview
`simple`	This demo detects the "?" marker in camera images and renders a cube on top of it.
`stereo`	This demo extends the `simple` demo with support for stereo cameras.
`marker_maker`	This demo creates pattern data to be used in a marker template.
`multi_marker`	This demo recognizes multiple markers and renders a cube on top of each one.

4.2 Common Definitions Between Sample Demos

The `nar_common` folder has definitions of classes and structures that are common to all the sample demos.

Table 4-2 Common Definitions Between Sample Demos

Files	Description
• narDemoCamera.h • narDemoCamera.cpp	These files define the CameraProcess_cl class that controls the CAMERA library. This class only captures images and handles errors; the camera(s) must be configured separately.
• narDemoY2r.h • narDemoY2r.cpp	These files define the Y2RProcess_cl class that controls the Y2R library, which converts YUV to RGB, and the Y2RJob_cl class for conversion requests. Except for the input image size, all the conversion settings used are listed below. • Input format: INPUT_YUV422_BATCH • Output format: OUTPUT_RGB_16_555 • Output data sequence: BLOCK_8_BY_8 • Output data rotation: ROTATION_NONE • Conversion coefficients: Return value of nn::camera::GetSuitableY2rStandardCoefficient() • Alpha value: 0xFF
• narDemoPhoto.h • narDemoPhoto.cpp	These files define the Photo_cl class that maintains and renders the camera images converted to RGB, as well as the debugging images.
• hatenaMarkerTemplate.h • hatenaMarkerTemplate.cpp	These files define the marker template structure g_hatenaMarker for "?" markers.
• cube.h • cube.cpp	These files define the cube that is rendered on top of a marker and the Cube_cl class that renders it.

4.3 Common Behavior Between Sample Demos

When you create a CCI file using a Debug or Development build and then run it in the debugger, you can press the X Button to show or hide debugging information.

You can use the following buttons to toggle whether to render the corresponding type of debugging information while debugging information is being displayed.

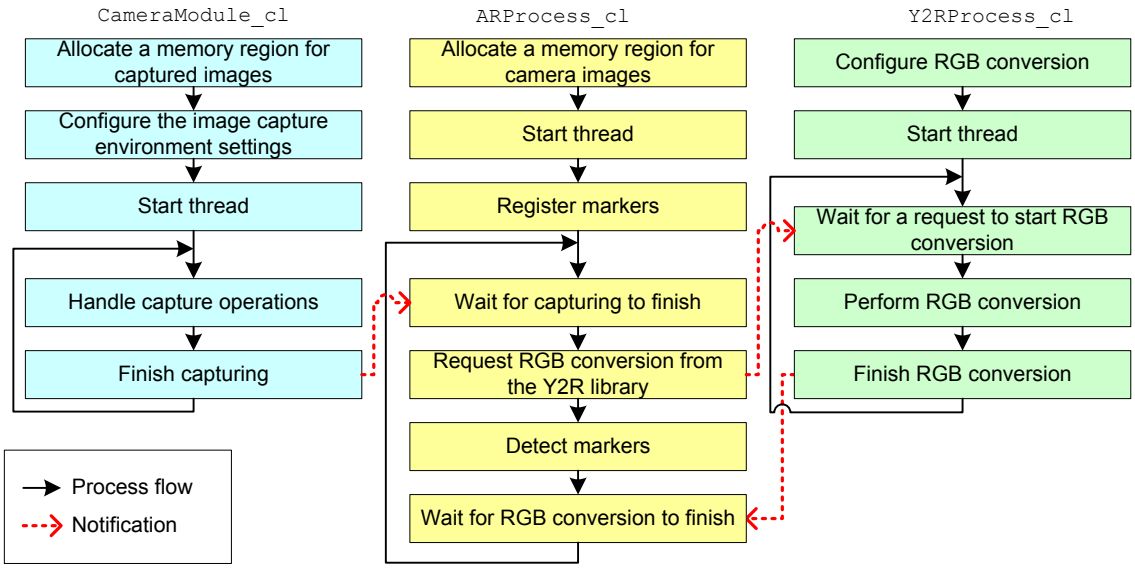
- Up on the +Control Pad: Edge candidates
- Left on the +Control Pad: Mesh for evaluating a marker
- Right on the +Control Pad: Slope

4.4 simple Demo

This demo uses a marker database in which only a "?" marker has been registered. It detects a marker and renders a cube on top of it.

In this and other sample demos, several threads other than the main thread are running. These threads are defined by three classes: CameraModule_cl, ARProcess_cl, and Y2RProcess_cl. The following figure shows how they work together.

Figure 4-1 Processing Outside of the Main Thread



The main thread displays information that becomes displayable after the `ARProcess_cl` class's thread has finished detecting markers and converting the captured camera images into RGB (such as the captured images converted to RGB and marker detection results).

Edit the `Initialize` function in the `CameraModule_cl` class if you want to detect markers when some visual effect has been applied to the camera images, or if you otherwise want to change the settings for the environment in which images are captured.

To detect your own original markers, edit the first part of the `ARProcess_cl` class's `process` function where it registers marker templates with the marker database.

The sizes of both the camera image and the texture are defined in `narDemoConfig.h`.

4.5 stereo Demo

This demo extends the `simple` demo with support for the stereo cameras. Both demos have the same basic process flow, but this one has changed the `CameraModule_cl` and `ARProcess_cl` classes to support images captured from two cameras. Also, the `StereoSetting` structure performs calculations needed for stereoscopic display.

4.6 marker_maker Demo

This demo creates the pattern data that is defined in marker templates. You can use the Y Button to switch between marker detection mode and marker registration mode.

The demo initially launches in marker detection mode. Because the marker template for "?" Cards has already been registered with the marker database, the demo detects "?" Cards and renders cubes on top of them. The lower screen shows the scores of the detected markers.

In marker registration mode, regions thought to be markers in an input image are displayed as marker candidates on the lower screen. Up to five marker candidates are displayed; these are assigned indices 0–4 in order from left to right. The number displayed after **Marker Pattern** is the index of the selected marker candidate. Press Left and Right on the +Control Pad to change the index and select a marker candidate to register.

Press the A Button to register a marker. Once a marker is registered, the demo will detect it in marker detection mode. Also, when a marker is registered, its pattern data is displayed on the debugger console. You can copy the displayed data into the `pattern` member variable of a marker template structure and use it with no modification necessary.

To use markers with a black border around a white region (where the marker design has a white background) instead of markers with the default white border around a black region (where the marker design has a black background), uncomment the line `narImage.SetBorder(mw::nar::e_BorderBlackWhite);` in the `ARProcess_cl` class's `process` function and then rebuild the demo.

Note: The AR Games built-in application uses template data that was created by the `marker_maker` demo in a bright environment, which was assumed to be similar to the environment where users would actually play.

4.7 multi_marker Demo

This demo detects markers using a marker database in which multiple marker templates have been registered. This is based on the `simple` demo and includes registered marker templates for the six AR Cards bundled with the CTR system.

5 Example of Practical Use in an Application

This chapter introduces a particular implementation of the library in an application.

5.1 Identifying Numerous AR Cards

NARLib is a library made for the purpose of detecting multiple types of markers in real time, so if the number of registration markers increases, its recognition accuracy gets worse and it becomes difficult to process them in real time. This example introduces a way for the application to detect numerous markers in real time.

5.1.1 Using the Colorbit Library

The Colorbit Middleware SDK for CTR package provided separately (hereafter referred to as the Colorbit library) can be used to identify numerous markers with high accuracy. The Colorbit library recognizes combined colored cell markers and identifies the corresponding IDs. By using Colorbit for recognition in addition to the recognition done with the `MarkerDetector_cl::Detect` function in NARLib, the accuracy of marker recognition can be improved.

However, the Colorbit library does not have a means of picking out the Colorbit portion of input images, so it is necessary for the developer to get the Colorbit portion from the camera image. By using `mw::nar::MarkerAnalizableData_tc`, the class used when detecting markers with NARLib, you can get the coordinates of the sampling points of the detected marker and the Colorbit coordinates. Also, if you create the design of the Colorbit portion along with the marker determination mesh from Table 3-4 Types of Debugging Information, you can create a Colorbit that is more easily extracted during marker detection.

Colorbit cells can be placed as desired, but it is best for the cells not to be placed where they would overlap with sampling points. The cell color changes if the number assigned to the cell is changed, which affects marker recognition.

For more information about how to use the Colorbit library, see the *Colorbit SDK Programming Manual* and *Development Guide for Applications That Use Colorbit.*

5.1.2 Real-Time Detection Processes

As the number of markers to register becomes greater, the database for detecting the markers also becomes larger. With a larger database; it takes longer to detect markers and makes it difficult to track markers in real time. To solve this problem, the database is separated into a database for detecting new markers (the *detection database*) and a database for tracking the markers that have already been detected (the *tracking database*), each of which is processed in a separate thread. This enables you to track markers in real time even if there are many of them being registered in the database.

The process for detecting new markers detects markers included in camera images from the detection database. The combined search using Colorbit (as described in the previous section)

enables you to detect markers with a high degree of accuracy. The markers detected this way are registered in the tracking database and used during the tracking process. At that point, the marker images obtained from the camera images are used as markers to register in the tracking database, which means that the tracking database is created in a way that conforms to the environment where the game is being played.

The process for tracking markers detects markers included in the tracking database from the camera images. Because only markers that have already been detected are included in the database, markers can be tracked very rapidly.

By running these processes in separate threads, you can detect markers coming in with new camera images while also rapidly tracking the markers that have already been detected.

5.1.3 Process for Improving Accuracy During Tracking

When tracking, recognition with Colorbit is usually not performed so that processing can be handled quickly. However, if a particular marker's recognition score continues to be low when tracking, Colorbit recognition is used to verify whether that marker is the correct one. Also, if the verification result is that the marker is recognized as the correct one, the marker is re-registered in the tracking database. This makes it possible to prevent the recognition rate from declining due to the effects of the environment while still tracking markers quickly.

5.2 Recognizing Markers for Posters

If an AR marker that has been affixed to a wall as a poster is recognized, the character for the 3D model is shown standing perpendicularly to the wall. To prevent this, when the marker printed on the poster is recognized, you must correct it so that the character stands horizontally against the wall. The following approaches are some of the ways to determine whether the marker that was recognized is one for a poster.

The first approach uses Colorbit. Include a Colorbit in the marker for the poster, and embed a flag in it indicating that this is a poster. When recognizing markers, use Colorbit recognition in addition to determine whether the target is a poster.

The second approach uses the CTR system's gyroscope. Get the system tilt using the gyroscope, and use the system tilt to guess whether the marker is oriented horizontally or vertically. If it seems that the orientation is vertical, recognize that the marker is a poster. Because the gyroscope is attached to the bottom of the CTR systems bottom screen, the camera cannot be used to accurately acquire orientation.

In this way, if the marker is recognized as being printed on the poster, you can make the character face forward by turning the front of the 3D model toward the camera.

If the character is shown horizontally with respect to the marker, the model could be shown from its bottom angle. To suppress this effect, when you attempt to view the model from a position slightly below the model, process it so that the character is standing on the card surface.

Revision History

Version	Revision Date	Category	Description
1.3	2014/04/17	Changed	• Table 3-1 Process Stages and Related Classes, 3.5 Detecting the Markers Added a description of `MarkerDetectWorkStrict_cl`.
1.2	2013/01/30	Added	• 5 Example of Practical Use in an Application
		Changed	• 4.2 Common Definitions Between Sample Demos Changed the `narDemoY2r.cpp` conversion coefficient to the return value of the `nn::camera::GetSuitableY2rStandardCoefficient` function.
1.1	2011/07/29	Added	• 3.8.2 Measuring the Distance to the Marker • 3.8.3 Reducing the Processing Load for Marker Detection
		Changed	• 3 How to Use the Library Figure 3-1 Corrected typos (JP document only) • 3.3 Preparing the Marker Template Changed wording. • 3.4.2 Debugging Information Table 3-4 Corrected typos (JP document only) • 4.4 simple Demo Changed wording.
1.0	2011/05/18	—	Initial version.