CTR

AR ライブラリ プログラミングマニュアル

2014-04-17

Ver. 1.3

任天堂株式会社発行

本ドキュメントの内容は、機密情報であるため、
厳重な取り扱い、管理を行ってください。

目次

コード

表

図

1 はじめに

本ドキュメントは、AR ライブラリの利用方法について説明したものです。

AR ライブラリの主な目的は、数種類のマーカーの検出（種類や位置、向きの判定）をリアルタイムに処理することです。また検出するマーカーには、グラフィカルなカード（ARカード）を使用することを前提としています。

1.1 用語

この節では、本ドキュメントに記述されている、AR や AR ライブラリに関係する用語を説明します。

AR

「Augmented Reality（拡張現実）」の略です。

AR アプリケーション

識別したマーカーや位置情報に対応する 3D モデルを現実空間の画像に合成するなどの処理を行うアプリケーションです。AR ライブラリは、ARカードを使用することを前提に、画像内にある ARカードの位置や向きに合わせて 3D モデルを表示するアプリケーションのために用意されています。

ARカード

CTR に同梱されている 6 枚のカードのように、AR ライブラリで識別可能なデザインが施されたグラフィカルなカードです。アプリケーション独自の ARカードを制作することができます。ARカードの制作については「ARカードを使ったアプリケーション開発ガイド」を参照してください。

マーカー

ARカードを識別するための目印のことです。AR ライブラリでは、カードデザインがマーカーとなっています。

マーカーテンプレート

マーカーを識別する際に使用する、ARカードのデザインを数値化したデータです。

マーカーデータベース

識別したい複数のマーカーテンプレートを登録するデータベースです。

マーカー座標系

検出されたマーカーの中心を原点とする座標系です。

カメラ座標系

入力された画像の幅と高さを −1.0～ +1.0 の範囲の値で表した、画像の中心を原点とする座標系です。

2 ビルド

この章では、AR ライブラリを使用するアプリケーションのビルドに必要な情報を説明します。なお、コード例は CTR-SDK のビルドシステムを使用した場合に OMakefile に記述すべき内容です。

2.1 インクルードパスの追加

コンパイラのインクルードパスに「$(CTRMW_NAR_ROOT)/include/mw/nar」を追加する必要があります。

コード 2-1 インクルードパスの追加

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

2.2 コンパイラフラグの追加

コンパイラに渡すフラグに「-DNAR_CTR__」を追加する必要があります。「NAR」と「CTR」の間のアンダーバーは 1 つ、「CTR」のあとのアンダーバーは 2 つです。

コード 2-2 コンパイラフラグの追加

```
CCFLAGS += -DNAR_CTR__
```

2.3 ライブラリファイルの追加

リンカに渡すべきライブラリファイルは「libmw_nar.*.a」(「*」は「fast」または「small」)です。

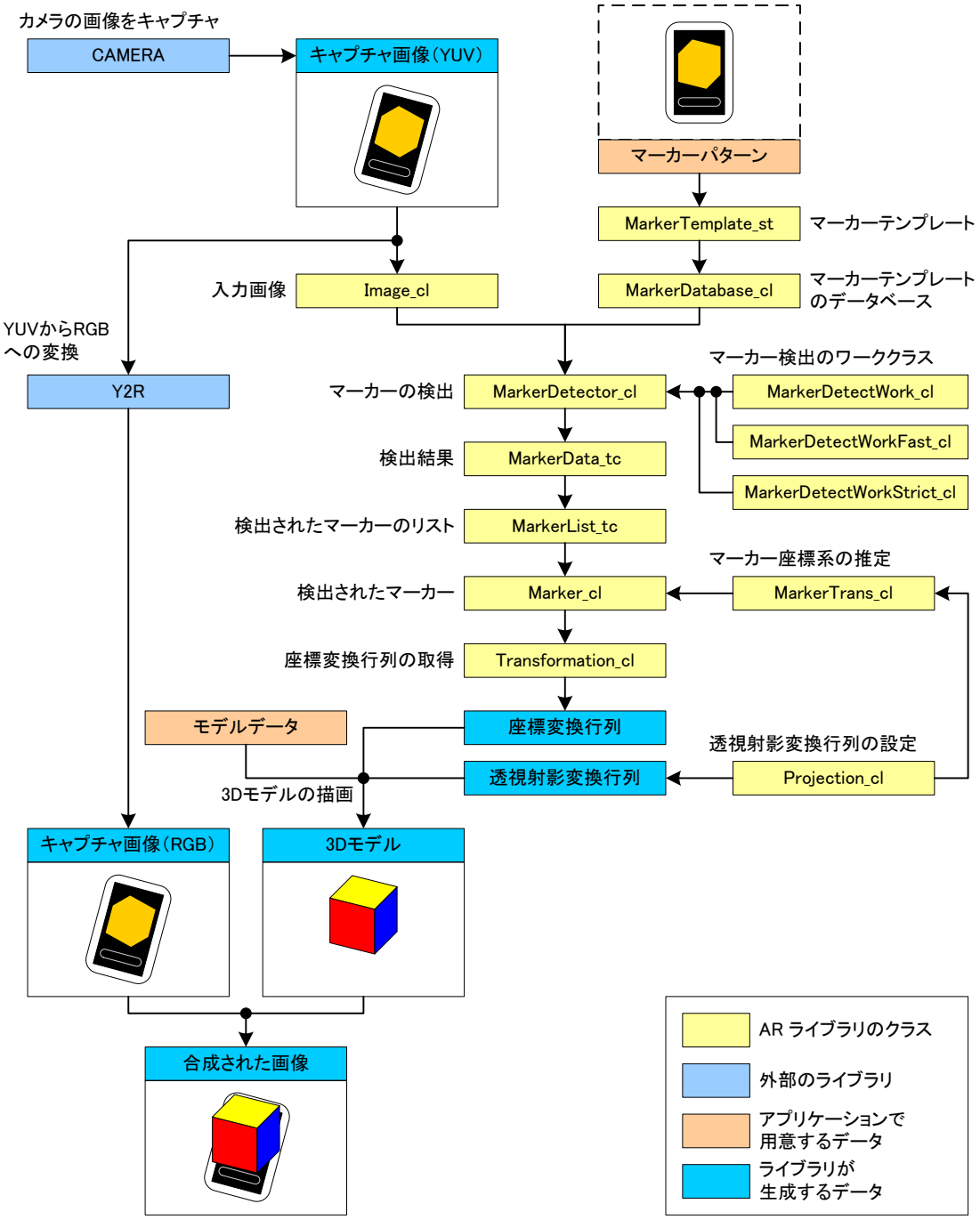
コード 2-3 ライブラリファイルの追加

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

3 ライブラリの使用方法

この章では、AR ライブラリを使用したアプリケーションの実装方法を処理の流れに沿って説明します。以下の図はアプリケーションで使用する AR ライブラリのクラスと処理の流れを図示したものです。

図 3-1 AR ライブラリのクラスと処理の流れ



これらの処理をいくつかの段階に分けて説明します。下表は処理の段階分けと、各段階に関連する AR ライブラリのクラスをまとめたものです。

表 3-1 処理の段階分けと関連するクラス

処理の段階	関連するクラス	概要
マーカーテンプレートの準備	MarkerTemplate_st MarkerDatabase_cl	マーカーの情報をデータベースに登録します。
入力画像の登録	Image_cl	マーカーを検出する画像を登録します。
マーカーの検出	MarkerDetector_cl Image_cl MarkerDatabase_cl MarkerData_tc MarkerDetectWork_cl MarkerDetectWorkFast_cl MarkerDetectWorkStrict_cl	入力画像からデータベースに登録されているマーカーを検出します。
検出結果の取得	MarkerData_tc MarkerList_tc Marker_cl	検出結果のリストから個々の検出結果を取得します。
検出結果の利用	MarkerTrans_cl Marker_cl Projection_cl Transformation_cl	3D モデルを表示するための座標変換行列を検出結果から取得します。

3.1 必要なヘッダファイル

AR ライブラリのヘッダファイルは「nar.h」です。AR ライブラリで定義されるクラスなどはすべて、「mw::nar」という名前空間に属しています。

コード 3-1 ヘッダファイルのインクルード

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

射影行列や変換行列を取得する際に MATH ライブラリや、カメラの画像を入力に使用する際に CAMERA ライブラリなどが必要になる以外は、AR ライブラリを使用するために別途ライブラリを必要としません。

3.2 初期化

AR ライブラリには初期化処理はありません。また、明示的に行う終了処理もありません。

AR ライブラリでは処理を行う際に、必要なクラスのインスタンスを生成して使用します。

3.3 マーカーテンプレートの準備

マーカーテンプレートは、入力された画像からマーカーを検出するために使用される情報で、マーカーの大きさやデザインを定義したものです。マーカーテンプレートは mw::nar::MarkerTemplate_st 構造体で定義されています。

コード 3-2 マーカーテンプレート構造体

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

構造体の各メンバに指定する値を説明します。

表 3-2 マーカーテンプレート構造体のメンバ

メンバ	説明
id	マーカーを識別するための IDです。0 以上の値を自由に指定することができます。
aspectRatio	マーカーの縦横比です。マーカーの幅を sideLength の値で除算した値を指定します。
sideLength	マーカーの高さです。この値を実際の寸法で定義することで、ほぼ実寸大の 3D モデルをマーカー上に表示することができます。ちなみに、サンプルデモでは実際の長さ 7.36 cm を指定しています。
pattern	マーカーを検出するために、マーカーのデザインをパターン化した情報です。パターンデータの作成には、サンプルデモの marker_maker を使用します。
p_Next	ライブラリが使用します。アプリケーションで値を変更しないでください。

ハテナカードのマーカーテンプレート

ハテナカード(「？」の描かれている ARカード)は、特別に使用許諾を必要としない、アプリケーションで自由に使用することができる ARカードです。

ハテナカードを使用する AR アプリケーションの実装には、サンプルデモに収録されているハテナカードのマーカーテンプレートを利用することができます。なお、3D モデルの大きさをハテナカードの大きさと合わせるために、マーカーテンプレートの sideLength の値にはマーカーの長辺の実測値(cm 単位)である 7.36 を使用しています。

3.3.1 マーカーデータベース

マーカーデータベースは、検出される可能性のある複数のマーカーテンプレートを登録しておくデータベースです。マーカーデータベースは mw::nar::MarkerDatabase_cl クラスで定義されています。

コード 3-3 マーカーデータベースのクラス

```
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;
}
```

コンストラクタの引数 *th* はマーカー検出のしきい値です。マーカー検出時のスコアがしきい値以上であれば一致と判定されます。しきい値は −1.0～+1.0 の範囲で指定し、+1.0 に近いほどマーカーテンプレートと画像が一致するかどうかの判定が厳しくなります。逆に −1.0 を指定した場合はネガフィルムのように色が反転していても一致と判定するようになります。

しきい値の設定は SetThreshold() で変更することができます。現在のしきい値は GetThreshold() で取得することができます。

> **補足**: データベースに 1 つだけマーカーテンプレートを登録している場合、しきい値が初期値(0.5)の状態では登録されていない ARカードを誤検知する可能性があります。様々な撮影環境でテストを行って調整し、適切なしきい値を設定するようにしてください。

Register() はマーカーデータベースへのマーカーテンプレートの登録を行います。登録に成功した場合、この関数は true を返します。すでにデータベースに登録されているマーカーテンプレートと同じものを登録しようとした場合や、データベースに登録可能なマーカーテンプレートの数を超えた場合など、登録に失敗した場合、この関数は false を返します。マーカーデータベースにマーカーテンプレートがすでに登録されているかどうかは IsRegistered() で確認することができます。

> **補足**: 似ているマーカー(しきい値が低い状態で誤検知するマーカー) をマーカーデータベースに登録しておくことで、誤検知を抑制できる可能性があります。ただし、登録されたマーカーの数が増えると、検出処理にかかる時間は長くなります。

Unregister() と UnregisterAll() はマーカーデータベースに登録されているマーカーテンプレートの登録解除を行います。前者は指定されたマーカーテンプレートの、後者は登録されているマーカーテンプレートすべての登録を解除します。

3.4 入力画像の登録

入力画像とはマーカーを検出したい画像のことです。検出に使用することのできる画像はピクセルフォーマットがYUV422 のリニアフォーマットですので、CAMERA ライブラリが出力するカメラのキャプチャ画像そのままを入力画像に使用することができます。

AR ライブラリでは、入力画像を mw::nar::Image_cl クラスで扱います。

コード 3-4 入力画像を扱うクラス

```
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);
}
```

コンストラクタの引数 *w* と *h* にはそれぞれ、入力画像の幅と高さをピクセル単位で指定します。*p_Work* には検出処理のワークメモリへのポインタを渡しますが、ワークメモリは u32 型の配列でサイズが NAR_IMAGE_WORK_SIZE_4TH マクロで算出した値以上でなければなりません。*p_DebugImg* はデバッグ情報の表示に使用します。デバッグ情報については「3.4.2 デバッグ情報」を参照してください。

入力画像（YUV422、リニアフォーマット）の登録は SetImage() で行います。入力画像が格納されているバッファの先頭アドレスを *cp_Image* に渡してください。通常、入力画像には CAMERA ライブラリで取得したカメラのキャプチャ画像をそのまま使用しますので、バッファの先頭アドレスのアラインメントは 64 Byte 以上の 4 の倍数であることが推奨されています。

> **注意**： マーカー検出処理が完了するまでは、処理中の Image_cl クラスに登録した入力画像が格納されているバッファを破棄したり、バッファの内容を書き換えたりしないでください。

SetXStep() と SetYStep() で入力画像からマーカーを探し出す際に、横方向と縦方向に何ピクセル（1 以上）ごと処理を行うかを設定することができます。現在の設定は GetXStep() と GetYStep() で取得することができます。デフォルトの設定は msc_DefaultXStep（8）と msc_DefaultYStep（16）で定義されています。設定値を大きくすることで処理速度は向上しますが、入力画像上のマーカーが小さい場合に検出できなくなる可能性があります。

GetColor() は引数 x と y で指定された入力画像の座標（参照点）の色を取得します。取得する色のフォーマットはRGBA5551 です。

SetBorder() と GetBorder() で縁（マーカーを囲む余白部分）の色の指定と現在の設定の取得を行うことができ

ます。縁の色を指定することで、AR ライブラリで検出するマーカーを白縁に黒地のマーカー、または黒縁に白地のマーカーから選択することができます。引数 *b* に渡す縁の色は mw::nar::Border_en で定義されている列挙子から選択します。

表 3-3 縁の色の指定(mw::nar::Border_en)

定義	説明
e_BorderWhiteBlack	白縁に黒地(マーカーのデザイン部分の背景色は黒。デフォルト設定)
e_BorderBlackWhite	黒縁に白地(マーカーのデザイン部分の背景色は白)

3.4.1 入力画像に使用する際のカメラの設定

通常、AR アプリケーションはカメラの画像と合成することになります。ここでは、入力画像に CTR のカメラの画像を使用する際に推奨する設定や注意点などを説明します。

カメラの解像度は 512 x 384 ピクセルを推奨

カメラの解像度には nn::camera::SIZE_DS_LCDx4 (512 x 384 ピクセル) を推奨します。この解像度は LCD の解像度よりも幅、高さともにピクセルが多く、Y2R ライブラリで RGB に変換した画像をテクスチャとして扱う際に、そのサイズを 512 x 512 にすることで幅のピクセル数が同じになります。

鮮明度は高く設定

カメラの画像にマーカーがくっきりと映るように、鮮明度を +3 以上に設定することを推奨します。

露光は低く設定

露光をマイナス値に設定すると、カメラの画像のブレを抑えることができます。ただし、露光を低く設定した場合は画像全体が暗くなりますので、表示の際に明るくするなどの色調整が必要になります。

色味が大きく変化するエフェクトは不適切

AR ライブラリはサンプルポイントの色からマーカーのデザインを識別するため、色味が大きく変化するエフェクト(ネガポジ反転やセピア調など)をかけるとマーカーの検出が正しく行われません。例えば、画像全体が黄色くなるセピア調のエフェクトをかけると、キューブシルエットが黄色く映るために、同梱されている ARカードのどれもがハテナカードと識別される可能性が高くなります。

3.4.2 デバッグ情報

Image_cl クラスのコンストラクタで、引数 *p_DebugImg* にデバッグ情報表示画像領域を設定することで、デバッグ情報の生成をライブラリに行わせることができます。デバッグ情報が必要でなければ、コンストラクタの *p_DebugImg* や SetDebugImage() の引数に NULL 以外を設定しないでください。また、リリースビルド(BUILD=Release) では無効化されます。

デバッグ情報はテクスチャイメージとして生成されます。テクスチャイメージはカラーフォーマットが RGBA5551 のブロックフォーマットで作成されるため、サイズを入力画像以上のサイズ、フォーマットを GL_RGBA_NATIVE_DMP と GL_UNSIGNED_SHORT_5_5_5_1 の組み合わせにする必要があります。デバッグ情報表示画像領域に必要なメモリサイズは(テクスチャの幅 * テクスチャの高さ * 2) Byte です。

> **補足**: 入力画像の幅とテクスチャの幅が異なる場合、デバッグ情報が正しく表示できない可能性があります。

生成されるデバッグ情報は Debug_cl クラスで管理されており、情報の種類は Debug_cl::Switch_e に定義されています。「RGBA」の列は、そのデバッグ情報がどのような色で描かれるかを RGBA5551 の 値で示しています。

表 3-4 生成されるデバッグ情報の種類

定義	生成されるデバッグ情報	RGBA
me_ViewLineCandidates	輪郭の候補。 マーカーのエッジ部分に線が引かれます。	(10, 31, 10, 1) (31, 31, 10, 1)
me_ViewCheckMesh	マーカー判定用メッシュ。 マーカーを縦方向、横方向ともに 16 分割する線が引かれます。	(10, 31, 10, 1)
me_ViewEdgeSlope	直線の傾き。 マーカーのエッジに沿った直線が引かれます。	(31, 3, 3, 1) (3, 31, 31, 1)
me_ViewSample	最小二乗法で算出したサンプルポイント。 マーカーのエッジの交点に点が描画されます。	(31, 15, 0, 1)

3.5 マーカーの検出

マーカーの検出処理は Marker_Detector_cl クラスの Detect() で行いますが、Marker_Detector_cl のインスタンスは生成できません。直接、MarkerDetector_cl::Detect() を呼び出してください。

コード 3-5 マーカー検出のクラス

```
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);
};
```

r_Image には、入力画像を設定した Image_cl クラスのインスタンスを渡します。検出処理が完了したあとは、渡したインスタンスに新しい入力画像を SetImage() で登録して次の検出処理に使用することができます。そのため、入力画像のサイズに変化がなければ、入力画像が変わるたびに Image_cl クラスのインスタンスを再作成する必要はありません。

r_DB には、検出したいマーカーのマーカーテンプレートを登録した MarkerDatabase_cl クラスのインスタンスを渡します。

r_Data には、検出結果を受け取る MarkerData_ac クラスのインスタンスを渡しますが、このクラスは抽象クラスです。実際に渡すインスタンスは、その実装クラスの MarkerData_tc に**検出可能なマーカー数**を指定して生成します。同じカメラからの入力画像に対しては、毎回同じインスタンスを渡すようにしてください。

r_Work には、検出処理でワークメモリを扱う MarkerDetectWork_ac クラスのインスタンスを渡しますが、このクラスは抽象クラスです。実際に渡すインスタンスは、その実装クラスの MarkerDetectWork_cl、MarkerDetectWorkFast_cl または MarkerDetectWorkStrict_cl のインスタンスです。

MarkerDetectWork_cl は検出時にサンプルポイントの周囲 3 点の平均値を参照するのに対し、MarkerDetectWorkFast_cl はサンプルポイントの 1 点のみを参照するので処理速度は向上しますが検出精度が少し低下します。そのため、MarkerDetectWorkFast_cl を利用する場合はマーカーの誤検知が起こらないように注意を払う必要があります。MarkerDetectWorkStrict_cl は検出時に周囲 3 点の平均を参照し、また検出したマーカーの領域を分割する際に消失点を利用した分割を行うため、処理速度は低下しますが最も精度の高い検出を行うことができます。これらのインスタンスは、それぞれアプリケーションの用途に応じて使い分けて頂いて構いませんが、特に理由が無ければ MarkerDetectWorkStrict_cl を使用することを推奨します。

返り値には検出されたマーカーの個数が返されます。検出されたマーカーの情報は *r_Data* に渡したインスタンスを介して取得します。

3.6 検出結果の取得

検出結果を受け取った MarkerData_tc クラスのインスタンスには、今回の検出結果に加え、前回の検出結果も保持されています。

コード 3-6 検出結果を受け取るクラス

```
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 ];
};
```

GetrDetectingMarkerList() は今回検出されたマーカーのリストを、GetrLastMarkerList() は前回検出されたマーカーのリストを取得します。

検出されたマーカーのリストを保持している MarkerList_tc クラスのインスタンスは MarkerData_tc クラスのインスタンスを生成する際に自動的に生成されます。

コード 3-7 検出されたマーカーのリストを保持するクラス

```
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 ];
};
```

GetMarkerNum() はリストに含まれている検出されたマーカーの個数を取得します。GetMarkerNumWithID()

では、その中でも *id* に指定された ID を持つマーカーの個数を取得することができます。

リストに含まれている個々のマーカーは、インデックスを指定して `GetpMarker()` で取得します。検出された数以上のインデックスを指定した場合は `NULL` が返されます。

3.7 検出結果の利用

検出された個々のマーカーの位置にモデルを表示するには、マーカー座標系とカメラ座標系の 2 つの座標系について理解する必要があります。

マーカー座標系

マーカー座標系は、入力画像の中から検出したマーカー上に 3D モデルなどを表示するために使用する座標系です。マーカーの左から右方向が X 軸の正方向、マーカーの裏から表方向が Y 軸の正方向、マーカーの上から下方向が Z 軸の正方向となり、輪郭の検出誤差などによって多少ずれる可能性がありますが、原点はマーカー表面のほぼ中央です。なお、マーカーの四隅の座標値は、マーカーテンプレートの `sideLength` と `aspectRatio` から計算することができます。

図 3-2 マーカー座標系の軸

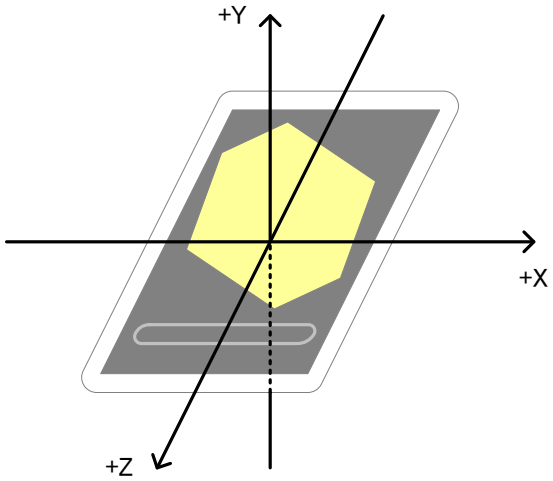
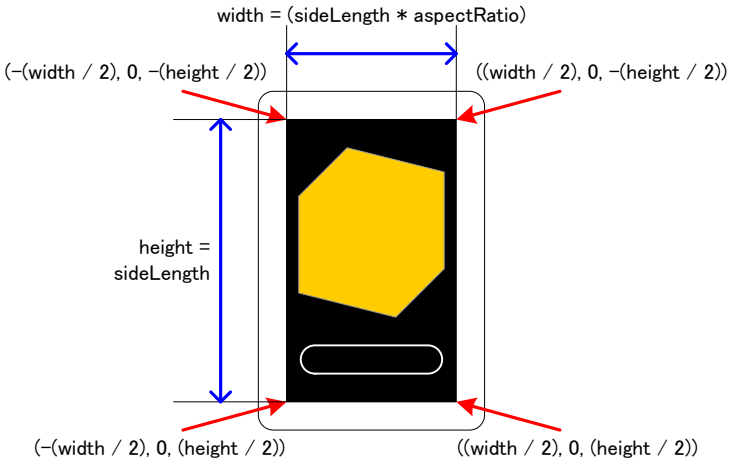


図 3-3 マーカーの四隅の座標値(マーカー座標系)



カメラ座標系

カメラ座標系は入力画像の幅と高さそれぞれを −1.0 ～ +1.0 の範囲で表した、画像の中心が原点となる座標系です。入力画像の左上隅の座標値は（−1.0, −1.0）、右下隅の座標値は（+1.0, +1.0）です。

3.7.1 検出されたマーカーの情報

検出された個々のマーカーの情報は Marker_cl クラスに保持されています。

コード 3-8 検出されたマーカーの情報を保持するクラス

```
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();
};
```

GetID() は検出されたマーカーの ID を取得します。取得する ID はマーカーテンプレートの id に指定されている値です。

GetScore() は検出時のスコアを取得します。このスコアが +1.0 に近いほど、検出したデザインがマーカーテンプレートのデザインと似ていることを示します。

GetSideLength() と GetAspectRatio() はそれぞれ、検出されたマーカーの左辺の長さと縦横比を取得します。取得する値はマーカーテンプレートに指定されている sideLength と aspectRatio の値そのままです。

SuppressShake() で手ブレなどの震えを抑制し、マーカー位置を補正します。前回検出時のマーカー位置との**入力画像上での距離（ピクセル単位）の二乗**を計算し、*sim* 以下ならば前回検出時のマーカー位置に補正し、*far* 以下ならば前回検出時のマーカー位置との平均に補正します。

IsEstimated() はマーカー座標系の推定が完了しているかどうかを返します。つまり、MarkerTrans_cl クラスの Estimate() でマーカー座標系の推定が完了している場合や、SuppressShake() による補正で前回のマーカー座標系と同一であると判断された場合は true が返されます。推定が完了していれば、GetrTransformation() でマーカー座標系からカメラ座標系への変換行列を取得することや、GetrNormal() でカメラ座標系でのマーカーの法線ベクトルを取得することができるようになります。

3.7.2 マーカー座標系の推定

マーカーの位置と向きから、マーカー座標系を推定します。マーカー座標系の推定は MarkerTrans_cl クラスで行われます。まず、Marker_cl クラスの IsEstimated() でマーカー座標系の推定が完了しているかを確認し、完了していなければ MarkerTrans_cl クラスの Estimate() でマーカー座標系を推定してください。

コード 3-9 マーカー座標系の推定を行うクラス

```
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);
};
```

Estimate() の実行には Marker_cl クラス(マーカーの情報)のインスタンスと Projection_cl クラス(透視射影行列を扱う)のインスタンスが必要です。Marker_cl クラスのインスタンスは検出結果から取得することができます。Projection_cl クラスのインスタンスはアプリケーションで生成しなければなりません。

コード 3-10 透視射影行列を扱うクラス

```
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;
};
```

引数のないコンストラクタで Projection_cl クラスのインスタンスを生成した場合は、Set() で入力画像の幅や高さなどを設定しなければなりません。Set() で指定する引数とコンストラクタの引数は同じで、*w* と *h* には入力画像の幅と高さを、*near* と *far* にはニア平面とファー平面までの距離を、*aov* には対角画角(単位は degree)を指定します。通常、入力画像にはカメラのキャプチャ画像を使用しますので、*aov* にはカメラのキャリブレーションデータから得た対角画角を指定することになります。

透視射影行列は GetMTX44() で取得することができます。画面表示と入力画像の比率が異なるなど、トリミングされた透視射影行列が必要な場合は GetTrimmed() で取得してください。GetTrimmed() で取得する際の引数 *l*、*r*、*b*、*t* は、トリミングされていない状態の x と y の座標値の範囲を −1.0 ～ +1.0 とする値(カメラ座標系)で指定します。

座標系の回転の補間

マーカー座標系を推定する際に、座標系の回転を補間する処理のかかり具合を MarkerTrans_cl クラスのメンバ関数 SetMildRotation() と SetAccelerationOnMildRotation() で指定することができます。

SetMildRotation() では *b* に true を渡すと、前回の座標系の回転と今回の回転が似ていると判断されたときに、引数 *t* の割合(0.0 ～ 1.0) で前回の座標系を使用した補間が行われます。つまり、*t* に 0.0 を指定すると今回の座標系となり、1.0 ならば前回の座標系となります。なお、この補間が行われている間は Marker_cl クラスの SuppressShake() による補正は行われません。また、*b* に false を渡した場合は補間処理が行われません。

SetAccelerationOnMildRotation() の引数 *b* と *t* への指定は SetMildRotation() と同じですが、補間がマーカー位置の動きが前回と今回で似ているかどうかで行われる点が異なります。この補間により、手ブレなどの震えではなく、一定に近い速度で回転していると判断されたときに素早く座標系を追従することができます。

3.7.3 マーカー座標系からカメラ座標系への変換

マーカー座標系からカメラ座標系へ変換する行列は Transformation_cl クラスのメンバ関数 GetMTX34() で取得することができます。

コード 3-11 マーカー座標系からカメラ座標系へ変換する行列を扱うクラス

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

Transformation_cl クラスのインスタンスはアプリケーションで作成することはなく、以下のコード例のように、検出されたマーカーのクラス（Marker_cl クラス）のメンバ関数 GetrTransformation() で取得します。

コード 3-12 検出されたマーカーから変換行列を取得するコードの例

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

3.8 ステレオカメラとの連動

ここでは、CTR に搭載されているステレオカメラと連動して、立体視に対応した AR アプリケーションを作成する際の注意点などを説明します。

3.8.1 カメラの設定

ここでは、入力画像にステレオカメラのキャプチャ画像を使用する際に推奨する設定や注意点などを説明します。

ノイズフィルタをオフに設定

ステレオカメラを使用するときは、ノイズフィルタをオフにすることを推奨します。

ステレオカメラの VSync の同期

ステレオカメラの VSync の同期がずれていると、左右のカメラでキャプチャされた画像に微妙な違いが生じ、立体視に影響を与える可能性があります。そのため、nn::camera::SynchronizeVsyncTiming() を呼び出して左右のカメラの VSync を同期させ、ずれが大きくなったときには再度呼び出すように実装することを推奨します。

ステレオカメラの明るさの同期

ステレオカメラの明るさを nn::camera::SetBrightnessSynchronization() で同期させる、ホワイトバランスや露光の自動調整の基準となる領域の設定に視差を考慮するなど、なるべく左右の画像の明るさが同じになるように実装

することを推奨します。

可変フレームレートは不適切

可変フレームレートに設定すると、高い確率でステレオカメラの VSync の同期がずれてしまいます。

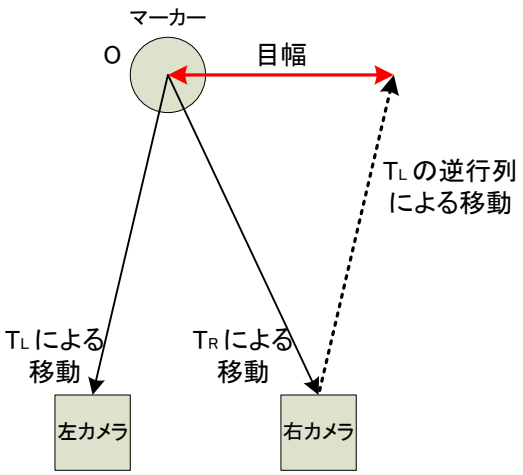
3.8.2 マーカーまでの距離の計算方法

マーカー座標系からカメラ座標系への変換行列でゼロベクトルを変換すると、そのカメラ座標系でのマーカーの位置を取得することができます。変換で得られたマーカー位置の Z 軸の値はカメラ座標系でのカメラからマーカーまでの距離ですので、カメラ座標系でのステレオカメラの取り付け幅（以降、目幅と呼びます）と実際のステレオカメラの取り付け幅との比を利用することで、実際の距離に変換することができます。

目幅は以下の方法で計算することができます。

変換行列を原点の移動ととらえれば、もう片方のカメラで使用する変換行列の逆行列を掛けることで原点がマーカーからカメラの取り付け幅分ずれた位置に移動します。つまり、マーカー位置を O、左右のカメラのキャプチャ画像で取得した変換行列を T_L と T_R とすると $OT_R(T_L)^{-1}$ の長さ（左右の入れ替えが可能）が目幅となります。

図 3-4 目幅の計算



左右のカメラの光軸が完全に平行であれば目幅の計算は 1 度で済みますが、ステレオカメラには取り付け誤差による光軸のズレがあり、キャリブレーションを行っても、マーカーまでの距離によっては目幅が変化してしまいます。また、異なる大きさのマーカーが混在する状況では、目幅の値が大きく変化する可能性があります。目幅の値が極端に変化することで、目幅を使用する処理や表示が意図しない変動を起こす可能性があることに注意してください。

> **補足**： キャリブレーションデータの水平方向の並進量（`translationX`）には、実際のステレオカメラの取り付け幅（ピクセル単位）が含まれています。一方、取り付け幅の設計値（`disatanceCameras`）はミリメートル単位で格納されています。

3.8.3 マーカー検出処理の負荷軽減

目幅を取得している場合は、片方のキャプチャ画像からマーカーを検出し、マーカー座標系からの変換行列を X 軸方向に目幅分だけ平行移動することで、もう片方の変換行列を求めることができます。これによりマーカーの検出処理が 1 回で済みますので、負荷の軽減に繋がります。この方法を利用する場合は、正確にマーカー位置に 3D モデルを表

示するためにも、目幅の再計測を定期的に行うことを推奨します。

さらに、マーカーがキャプチャ画像の中央より左右どちらにあるかを利用して、マーカー検出に左右どちらのキャプチャ画像を使用するかを決定（メインカメラの選択）すれば左右への見切れに強くなります。

4 サンプルデモの解説

この章では、$(CTRMW_NAR_ROOT)/sampledemos に収録されている AR ライブラリを使用したサンプルデモと、デモで使用されているクラスや実装の注意点などを解説します。

4.1 収録されているサンプルデモ

現在、以下のサンプルデモが収録されています。

表 4-1 収録されているサンプルデモ

サンプルデモ	概要
simple	カメラで撮影した画像からハテナマーカーを検出し、その位置に立方体を描画します。
stereo	simple デモをステレオカメラに対応させたものです。
marker_maker	マーカーテンプレートで使用するパターンデータを作成することができます。
multi_marker	複数のマーカーを認識し、それぞれの位置に立方体を描画します。

4.2 サンプルデモ共通の定義

サンプルデモで共通に使用されているクラスや構造体の定義は nar_common フォルダに収録されています。

表 4-2 サンプルデモ共通の定義

ファイル	説明
narDemoCamera.h narDemoCamera.cpp	CAMERA ライブラリの制御を行う CameraProcess_cl クラスが定義されています。 キャプチャやエラーのハンドリングのみを行いますので、カメラの設定などは別途行う必要があります。
narDemoY2r.h narDemoY2r.cpp	YUV から RGB に変換する Y2R ライブラリの制御を行う Y2RProcess_cl クラスと、変換要求の Y2RJob_cl クラスが定義されています。 入力される画像サイズ以外の変換設定は以下のように設定されます。 入力フォーマット: INPUT_YUV422_BATCH 出力フォーマット: OUTPUT_RGB_16_555 出力データの並び順: BLOCK_8_BY_8 出力データの回転: ROTATION_NONE 変換係数: nn::camera::GetSuitableY2rStandardCoefficient() の返り値 アルファ値: 0xFF
narDemoPhoto.h narDemoPhoto.cpp	RGB に変換された画像などの保持と描画を行う、Photo_cl クラスが定義されています。
hatenaMarkerTemplate.h hatenaMarkerTemplate.cpp	ハテナマーカーのマーカーテンプレートの構造体 g_hatenaMarker が定義されています。
cube.h cube.cpp	マーカーの位置に描画される立方体の定義と描画を行う、Cube_cl クラスが定義されています。

4.3 サンプルデモ共通の動作

Debug ビルドまたは Development ビルドで作成された CCI ファイルをデバッガで実行した場合、X ボタンでデバッグ情報の表示と非表示を切り替えることができます。

デバッグ情報を表示する場合、以下のキーで対応する情報を描画するかどうかを切り替えることができます。

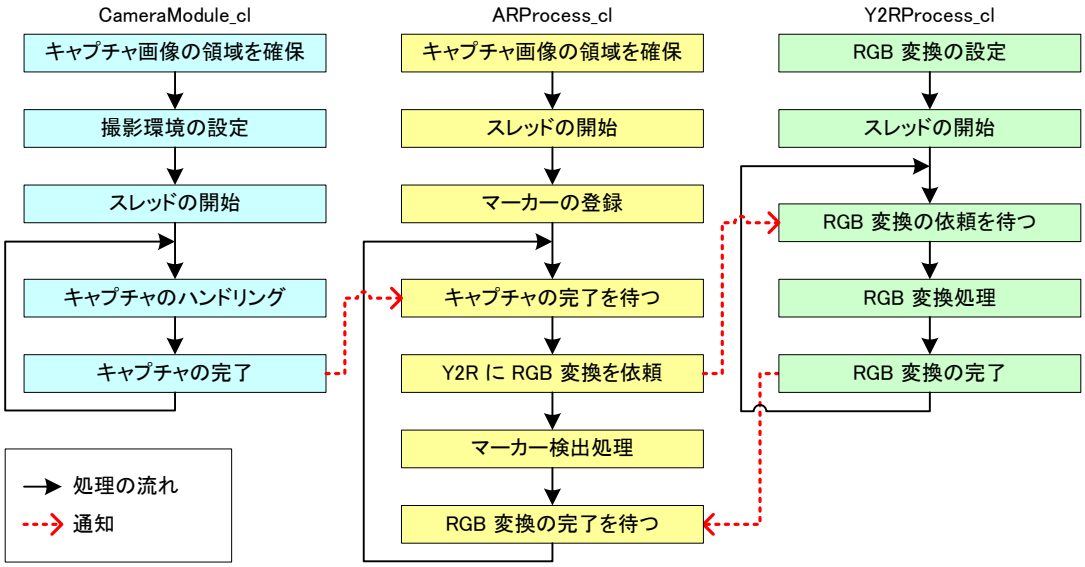
- 十字キー上: 輪郭の候補
- 十字キー左: マーカー判定用メッシュ
- 十字キー右: 直線の傾き

4.4 simple デモ

ハテナマーカーのみが登録されているマーカーデータベースを使ってマーカーの検出を行い、検出されたマーカーの位置に立方体を描画します。

このデモに限らず、サンプルデモではメインスレッドのほかに複数のスレッドが動作しています。スレッドは `CameraModule_cl` クラス、`ARProcess_cl` クラス、`Y2RProcess_cl` クラスの 3 つのクラスそれぞれで定義され、以下の図のように連携しています。

図 4-1 メインスレッド以外の連携



メインスレッドでは `ARProcess_cl` クラスのスレッドでのマーカー検出処理とキャプチャ画像の RGB 変換処理が完了したあとに表示可能となる情報(RGB に変換されたキャプチャ画像、マーカーの検出結果など)を表示します。

エフェクトがかけられた状態での検出を行いたいなど、撮影環境の変更を行う場合は `CameraModule_cl` クラスの `Initialize()` を編集してください。

独自に制作されたマーカーを検出させたい場合は、`ARProcess_cl` クラスの `process()` の冒頭で行われているマーカーデータベースへのマーカーテンプレートの登録部分を編集してください。

カメラ画像のサイズやテクスチャのサイズは `narDemoConfig.h` に定義されています。

4.5 stereo デモ

simple デモをステレオカメラに対応させたデモです。基本的な処理の流れは同じですが、CameraModule_cl クラスや ARProcess_cl クラスが 2 つのカメラからのキャプチャ画像に対応するように変更されています。また、StereoSetting 構造体で立体視のための計算が行われています。

4.6 marker_maker デモ

マーカーテンプレートに定義するパターンデータの作成を行うことができます。マーカー検出モードとマーカー登録モードの 2 つのモードを Y ボタンで切り替えることができます。

最初はマーカー検出モードで起動し、ハテナマーカーのマーカーテンプレートがマーカーデータベースに登録されていますので、ハテナマーカーの検出と立方体の描画が行われます。なお、下画面にはマーカー検出時のスコアが表示されます。

マーカー登録モードでは、入力画像内にあるマーカーと思われる領域がマーカー候補として下画面に表示されます。マーカー候補は最大で 5 候補表示され、左から順に 0～ 4 のインデックスが割り当てられています。「Marker Pattern」のうしろの数字は選択中のマーカー候補のインデックスを示しています。登録するマーカー候補を選択するには、十字キーの左右でインデックスを変更してください。

マーカーの登録は A ボタンで行います。登録されたマーカーはマーカー検出モードで検出されるようになります。また、登録時にはデバッガのコンソールにマーカーのパターンデータが表示されます。表示されるデータは、マーカーテンプレート構造体の pattern メンバにコピーしてそのまま使用することができます。

マーカーの縁の色をデフォルトの白縁に黒地（マーカーのデザイン部分の背景色は黒）ではなく、黒縁に白地（マーカーのデザイン部分の背景色は白）に変更する場合は、ARProcess_cl クラスの process() でコメントアウトされている「narImage.SetBorder(mw::nar::e_BorderBlackWhite);」のコメントを解除してビルドしてください。

> **補足**: 「ARゲームズ」で使用されているテンプレートデータは、ユーザが実際に遊ぶ環境を想定した明るい環境で marker_maker デモが作成したデータです。

4.7 multi_marker デモ

複数のマーカーテンプレートが登録されたマーカーデータベースを検出処理に使用するデモです。simple デモをベースにしており、CTR に同梱されている 6 枚の ARカードのマーカーテンプレートが登録されています。

5 アプリケーションでの応用例

本章では、実際にアプリケーションで実装された工夫について紹介します。

5.1 多数の AR カードを識別する方法

NARLib は数種類のマーカーをリアルタイムに検出することを目的としているライブラリのため、登録マーカー数を増加させると認識精度が悪くなったり、リアルタイムに処理を行うことが難しくなったりします。そこで、多数のマーカーをリアルタイムに検出するために、アプリケーションで行われた工夫について紹介します。

5.1.1 Colorbit ライブラリの利用

多数のマーカーを高精度に識別するために、別途提供されている Colorbit Middleware SDK for CTR（以下、Colorbit ライブラリ）を利用することができます。Colorbit ライブラリは色のついたセルを組み合わせたマーカーを認識し、対応する ID を識別するライブラリであり、NARLib の MarkerDetector_cl::Detect() による認識の他に、Colorbit でも認識することにより、マーカーの識別精度を向上させることができます。

しかし、Colorbit ライブラリには入力画像から Colorbit 部分を取り出す仕組みがないため、開発者がカメラ画像から Colorbit 部分を切り出す必要があります。これは、NARLib でマーカーを検出する際に使用するクラスとして、 mw::nar::MarkerAnalizableData_tc を利用することで、検出したマーカーのサンプリング点の座標を取得でき、Colorbit の座標を取得することができます。また、「表 3-4 生成されるデバッグ情報の種類」のマーカー判定用メッシュに合わせて Colorbit 部分のデザインを作成すると、マーカー検出時に切り出しやすい Colorbit を作成できます。

Colorbit のセルの配置は任意に設定することができますが、その際セルの配置された場所とサンプリング点は重ならないようにすることが望ましいです。セルに割り振る番号が変更されるとセルの色が変わり、マーカーの認識に影響があるためです。

Colorbit ライブラリの詳細な使用方法は「Colorbit SDK Programming Manual」、「Colobit を使ったアプリケーションガイド」を参照してください。

5.1.2 リアルタイムに検出する処理

登録するマーカーの数が多くなると、マーカーを検出するためのデータベースも大きくなるため、マーカーの検索に時間がかかり、リアルタイムにマーカーを追跡することが難しくなります。これを解決するため、新規にマーカーを検出するためのデータベース（検索データベース）と、すでに検出済みのマーカーを追跡するためのデータベース（追跡データベース）を分離し、それぞれ別スレッドで処理を行うことで、多数のマーカーをデータベースに登録していてもリアルタイムに追跡できるようにしています。

新規にマーカーを検出する処理では、カメラ画像に含まれているマーカーを検索データベースから検索します。前項で説明した Colorbit と組み合わせて検索することで、高精度にマーカーを検出することができます。ここで検出されたマーカーは追跡データベースに登録され、追跡処理時に利用されます。この時、追跡データベースに登録するマーカーとして、カメラ画像から取得したマーカー画像を使用することで、アプリを遊んでいる環境に即した追跡データベースを作成しています。

マーカーを追跡する処理では、カメラ画像から追跡データベースに含まれているマーカーを検索します。すでに検出されているマーカーのみがデータベースに含まれているので、高速に追跡を行うことができます。

上記の処理をそれぞれ別スレッドで行うことで、新しくカメラ画像に入って来たマーカーを検出しつつ、すでに検出されているマーカーを高速に追跡することができるようになります。

5.1.3 追跡時の精度を向上させる処理

追跡時は、より高速に処理を行うため基本的には Colorbit による認識を行っていません。しかし、追跡時にあるマーカーの認識スコアが低い状態がしばらく続いた場合、Colorbit の認識を使用して、そのマーカーが正しいものかどうかを検証します。また、その検証の結果正しいマーカーだと認識された場合、認識時のカメラ画像を使用して、マーカーを追跡データベースに登録し直します。これによって、環境の影響による認識率の低下を防ぎつつ、高速にマーカーを追跡することができるようになります。

5.2 ポスター用のマーカーの認識

ポスターとして壁に貼った AR マーカーを認識させる場合、3D モデルのキャラクターが壁に垂直に立つように表示されます。これを防ぐために、ポスターに印刷されたマーカーを認識した際には、キャラクターが壁に水平に立つように修正する必要があります。認識したマーカーがポスターのものかどうかを判定するために、下記のような方法が考えられます。

1 つ目は、Colorbit を利用する方法です。ポスター用のマーカーに Colorbit を含め、そこにポスターであることを示すフラグを埋め込むようにします。マーカー認識時に Colorbit による認識も行い、ポスターかどうかを判定します。

2 つ目は、CTR 本体のジャイロセンサーを利用する方法です。ジャイロセンサーを利用して得た本体の傾きから、マーカーが水平と垂直どちらに置かれているのかを推測し、垂直に置かれていると推定された場合はポスターであると認識しています。なお、ジャイロセンサーは CTR 本体の下画面側にとりつけられているため、カメラの正確な向きを取得することはできません。

このように、マーカーがポスターに印刷されていると認識した場合、3D モデルの正面をカメラ側に向けることで、キャラクターが正面を向くようにすることができます。

なお、キャラクターがマーカーに対して水平に表示されると、モデルの下側アングルからの表示が可能となってしまいます。これを抑止するために、ある程度下から見ようとした場合にはキャラクターをカード面に立たせるようにする処理を行います。

6 改訂履歴

版	改訂日	分類	改訂内容
1.3	2014-04-17	変更	・表 3-1 処理の段階分けと関連するクラス、3.5 マーカーの検出 に `MarkerDetectWorkStrict_cl` に関する記載を追加。
1.2	2013-01-30	追加	・5 アプリケーションでの応用例
		変更	・4.2 サンプルデモ共通の定義 `narDemoY2r.cpp` の変換係数を `nn::camera::GetSuitableY2rStandardCoefficient()` の返り値に変更。
1.1	2011-07-29	追加	・「3.8.2 マーカーまでの距離の計算方法」 ・「3.8.3 マーカー検出処理の負荷軽減」
		変更	・「3 ライブラリの使用方法」 図 3-1 の誤記を修正しました。 ・「3.3 マーカーテンプレートの準備」 ハテナカードのマーカーテンプレートで、実測値に関する記述を変更しました。 ・「3.4.2 デバッグ情報」 表 3-4 の誤記を修正しました。 ・「4.4 simple デモ」 「スルー画像」を「キャプチャ画像」に修正しました。
1.0	2011-05-18	–	・初版

記載されている会社名、製品名等は、各社の登録商標または商標です。

© 2014 Nintendo

任天堂株式会社の許諾を得ることなく、本書に記載されている内容の一部あるいは全部を無断で複製・複写・転写・頒布・貸与することを禁じます。