

UnityでのARアプリケーション開発

株式会社コンセプト 林建一

アウトライン



1. 会社概要とARについて
2. ARアプリケーション開発におけるUnity
 1. ARアプリケーションの仕組み
 2. Unityのメリット
3. Unityプラグインについて
4. UnityプラグインによるAR機能の実装
5. ARプラグインの紹介

主に対象となる方



- モバイルARに興味があるゲーム開発者の方
- Unityプラグインの開発に興味が有る方
- Unityでカメラデバイス制御/表示したい方

向けに資料をつくりました.

http://qconcept.jp/ar/seminar/unity_20110716.pdf

プロフィール



株式会社コンセプト 取締役/クリエイティブ・ディレクター

林 建一 博士(工学)



@hayashi

略歴

1981年10月19日生まれ

2004年3月 大阪大学基礎工学部 卒業

2006年3月 大阪大学大学院 基礎工学研究科 システム創成専攻 博士前期課程 修了

2007年4月～ 日本学術振興会 特別研究員

2008年7月 株式会社コンセプト設立、取締役/クリエイティブ・ディレクター

2009年3月 同大学院 博士後期課程 修了

専門

拡張現実感、ヒューマンインターフェース、コンピュータビジョン

主な業績

日本バーチャルリアリティ学会 2006年度 論文賞 (林建一, 加藤博一, 西田正吾)

日本バーチャルリアリティ学会 2008年度 論文賞 (林建一, 加藤博一, 西田正吾)

ICAT2005 Best paper award (Kenichi Hayashi, Hirokazu Kato, Shogo Nishida)



コンセプトが提供するAR技術/事例紹介

自然特徴点 AR技術

Keypoint recognition engine



e.g. NHK
(BIZ spo)



空間認識 AR技術

Space recognition engine



e.g. Dentsu
(TOKYO INNOVATION)



形状認識 AR技術

Shape recognition engine



e.g. GQ (Issued July 2010)
(BEAMS)
(GIRARD-PERREGAUX)



1. AR(拡張現実感) について

ARの実現方法



ビジョンベースとセンサベースAR

ビジョンベース

カメラ画像の認識結果からカメラ画像とCGの位置合わせを行う
画像内の特徴とシーンモデルのマッチングで
シーンに対する相対的な位置姿勢を推定



センサベース

厳密な位置合わせをしない
iPhone/Androidで使用可能なのは、
GPS、コンパス、加速度センサ、ジャイロセンサ



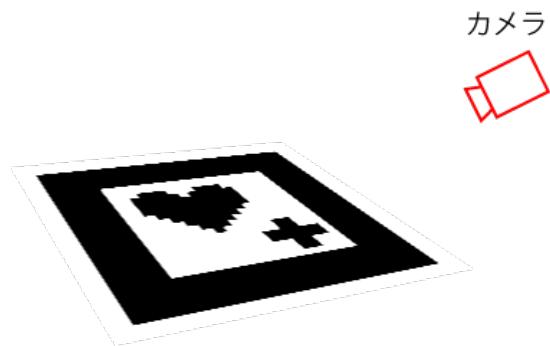
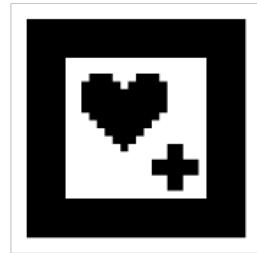
ビジョンベース型ARの種類



ビジョンベース型のARを現状使われているものは以下の種類に分類することができる

- マーカ型 AR
 - 正方マーカ
 - 特殊形状マーカ
- マーカレス型AR
 - 自然特徴点マッチングによるAR
 - 輪郭形状マッチングによるAR（ロゴ認識など）

マーク型AR



正方形マーカ

正方形の形状を画像中から検出して、その4頂点の画像中の位置をもとにマーカに対するカメラの（相対的な）位置姿勢を推定する



特殊形状マーカ

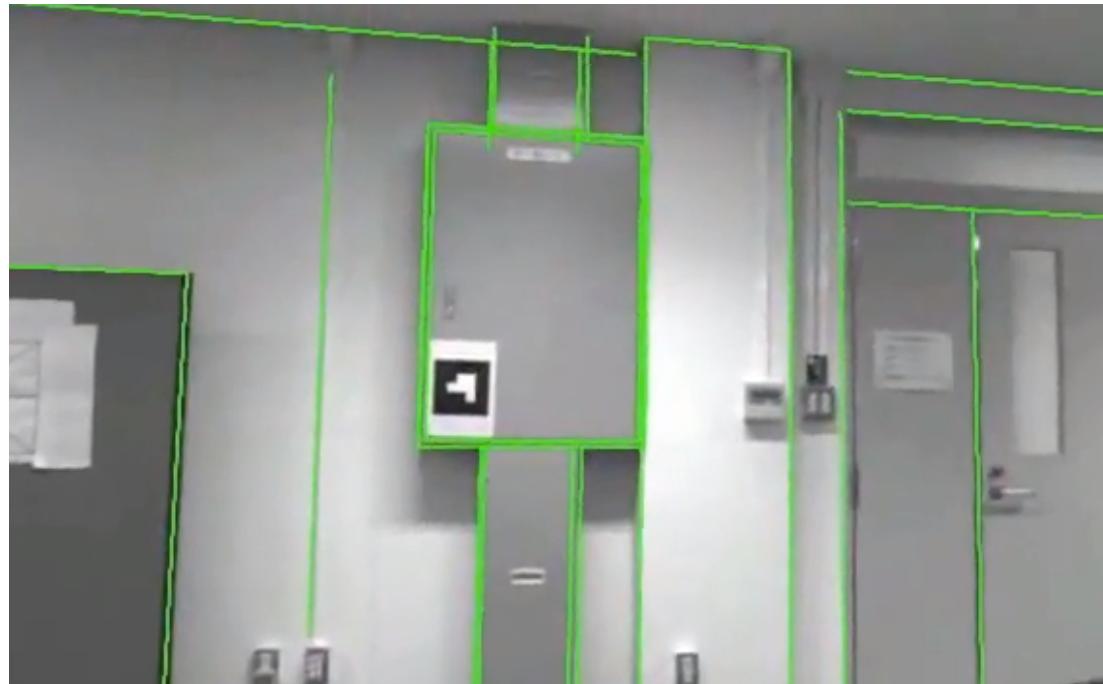
正方形以外の形状をもったマーカ
検出対象の形状が異なるだけで基本的な原理は
正方形マーカと同じ

特徴点ベース マーカレスAR



特徴点(キーポイント)ベース
シーン中の特徴的な点を検出&追跡することで、カメラの位置姿勢を推定

エッジベース マーカレスAR

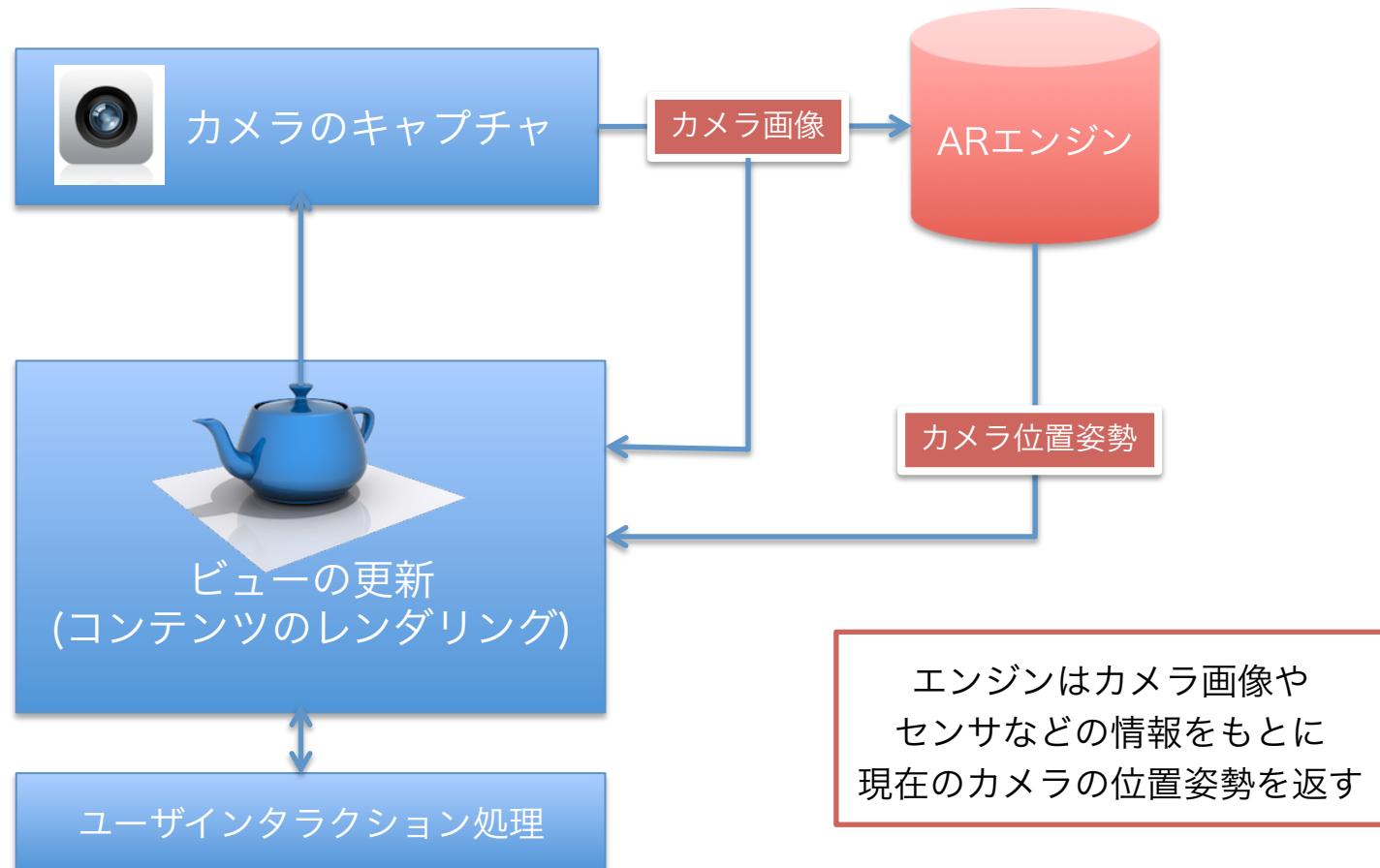


エッジベース

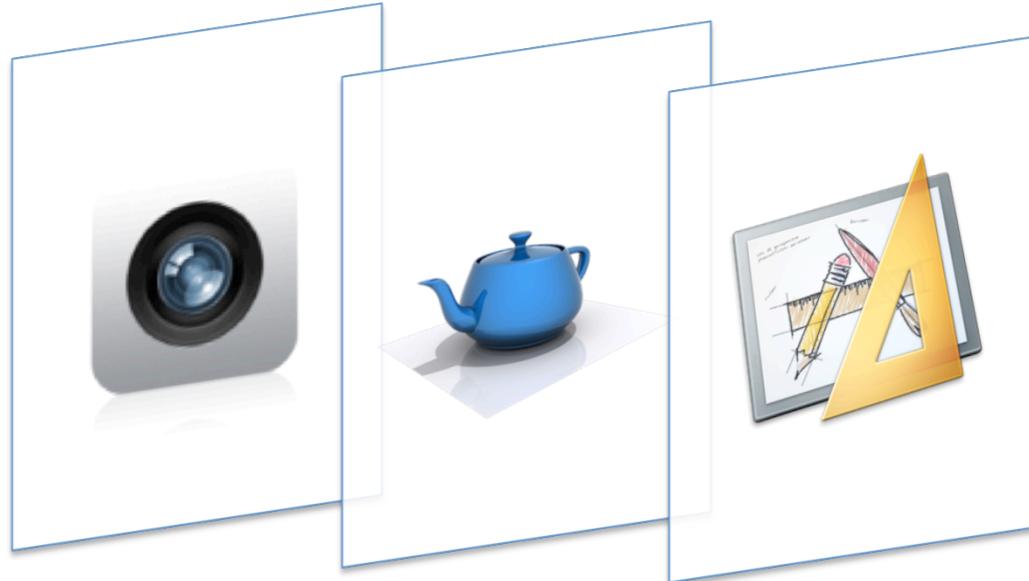
シーン中のエッジを検出&追跡することで、カメラの位置姿勢を推定

2. ARアプリケーション開発におけるUnity

典型的なARアプリケーションのフロー



典型的なARアプリケーションの構成



ARアプリの表示要素のレイヤー構成

1. カメラレイヤー（リアルタイムに更新される）
2. 3D/2D グラフィックレイヤー (合成されるCG)
3. アプリケーション固有のUIレイヤー (ボタンなど)

ARアプリケーション開発におけるUnityのメリット



いわずもがなのUnityの特徴ですが．．．

主に3Dコンテンツを扱うARアプリとゲームエンジンは相性がいい

ARアプリの表示系， インタラクション， 物理演算はすべてUnity側にまかせる

3Dコンテンツの表示はUnityにまかせる

3Dモデルのフォーマットはさまざま， デザイナーとのやりとりも多い

独自で3Dモデルのインポーター/レンダラを作るのもかなり大変

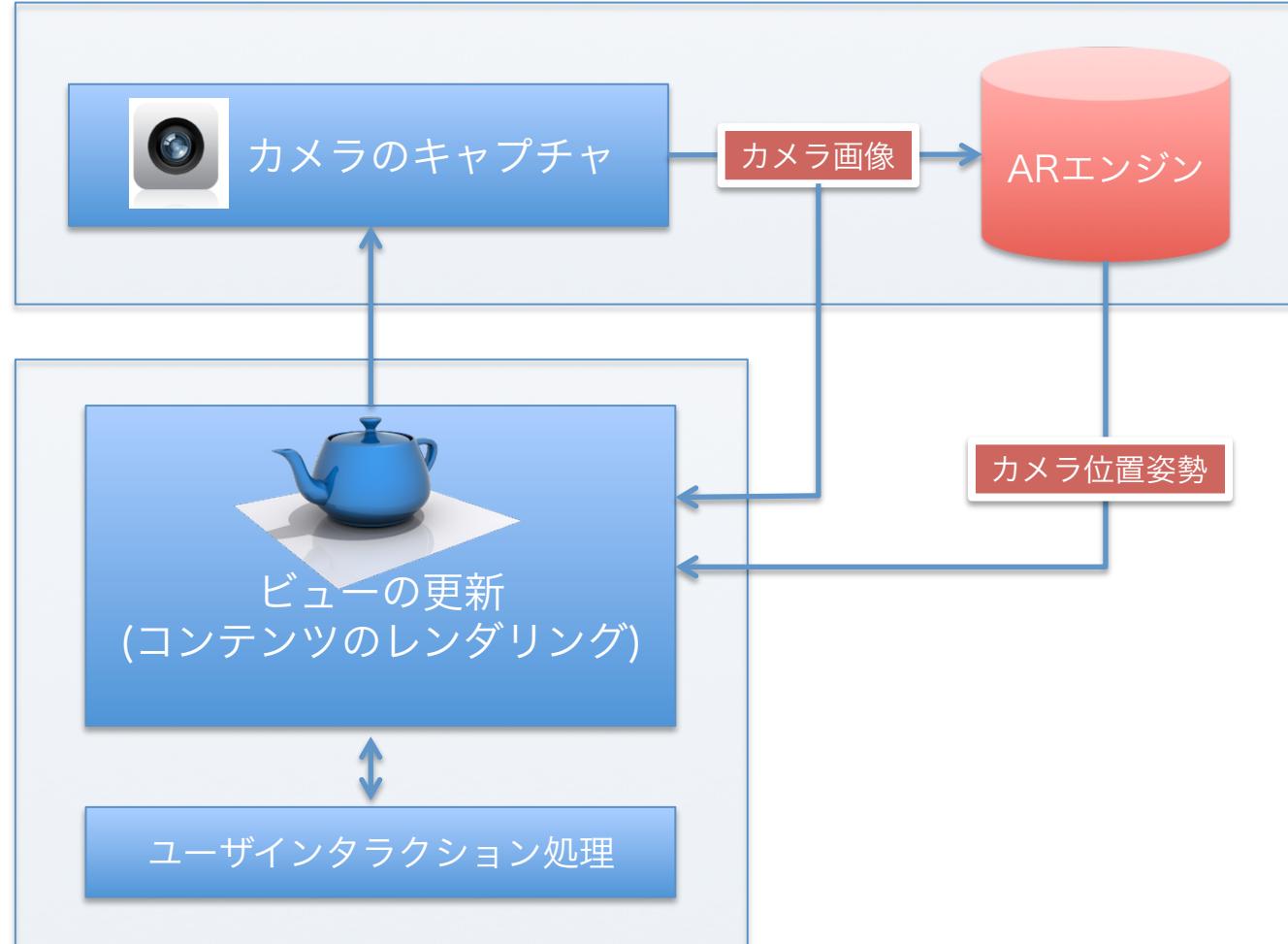
→ AR上でのモデルプレビューがすごく簡単に

マルチプラットフォーム

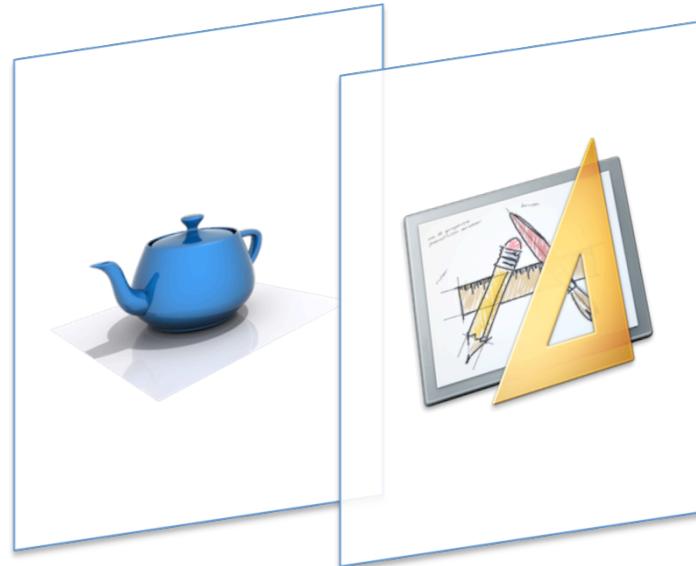
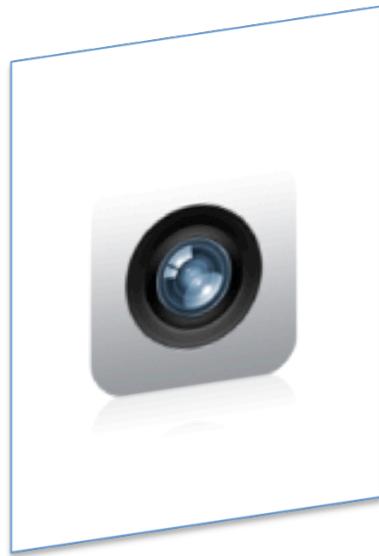
， iOS/Android向けへ同時にパブリッシュ

→ リアルタイムなカメラ画像の背景表示とカメラの位置姿勢更新さえできれば， Unityから書き出したアプリが一気にAR化できる

UnityとARプラグインの役割分担



UnityとARプラグインのレイヤー分担



AR プラグイン
カメラレイヤー



Unity
3D/2D グラフィックレイヤー
アプリケーション固有のUIレイヤー

3. Unityプラグインの開発方法

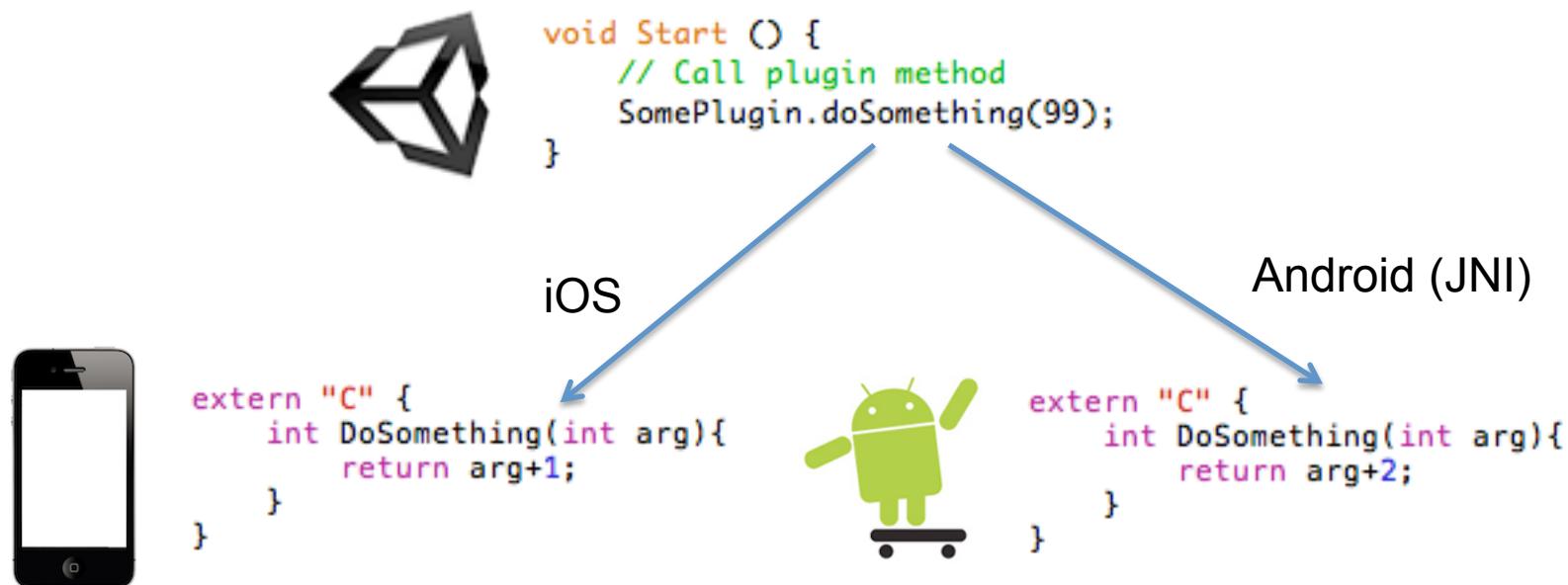
Unity プラグイン



Unity 上でのゲームスクリプト (javascript,C#) からネイティブのライブラリシンボルを呼び出す仕組み。

各プラットフォームごとに異なるバイナリを用意してプラグインメソッドをコール (後述, 少し工夫が必要)

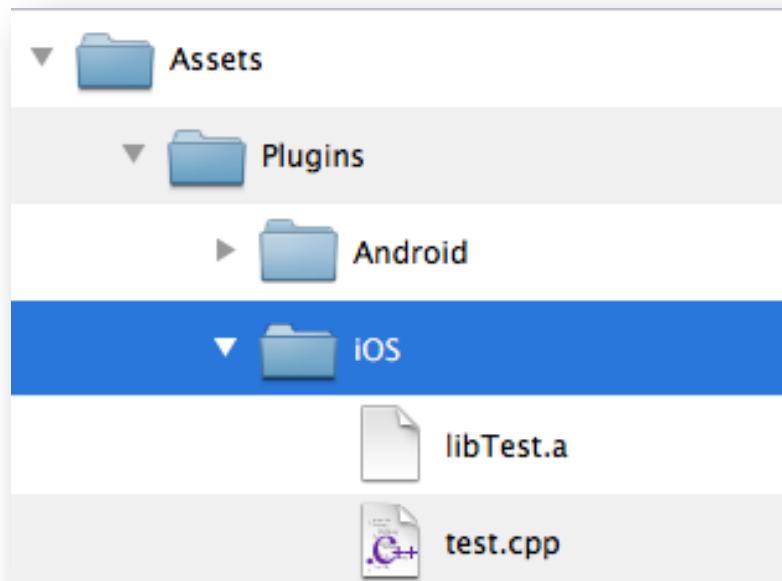
→ カメラのキャプチャなど端末固有各種デバイスとの連携や、高負荷の処理を端末ネイティブ側で実行できる





iOSの場合

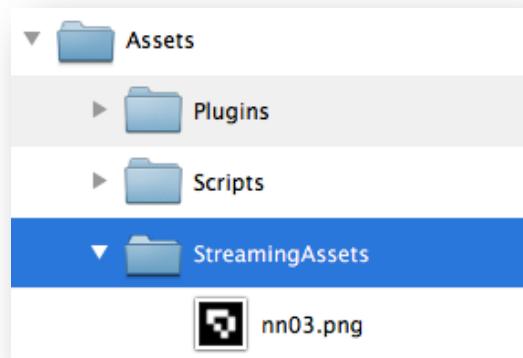
- プラグインとしてバンドルできるのは、
 - スタティックライブラリ (*.a)
 - C/C++ Objective-C/C++ソース (*.c, *.cpp, *.m, *.mm)
- [Assets/Plugins/iOS]以下に配置
- 書き出されるXcodeプロジェクトにこれらが追加されてビルドされる





Unity側とのファイル共有 (iOS)

- プラグイン側で使いたいファイルは、
[Assets/StreamingAssets] 以下においておくと. . .



- [アプリケーション.app/Data/Raw/] 以下に配置されるので、
ネイティブ側からNSBundleをつかってアクセスできる。

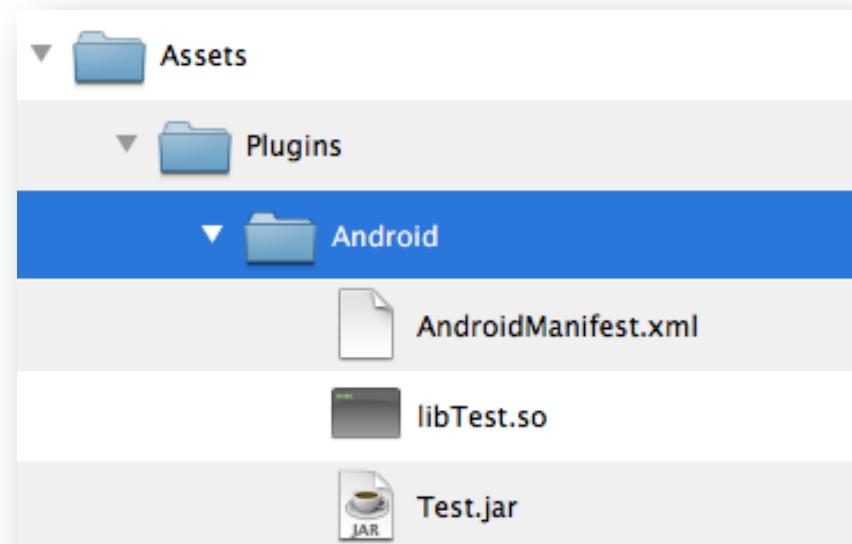
```
NSString *path = [NSString stringWithFormat:@"%@/Data/Raw/%s",
                  [[NSBundle mainBundle] resourcePath],filename];
FILE *fp = fopen([path UTF8String],"rb");
.
```

Unityプラグイン (Android)



Androidの場合

- プラグインとしてバンドルできるのは、
 - ダイナミックリンクライブラリ (*.so)
 - マニフェストファイル (AndroidManifest.xml)
 - jarパッケージ (*.jar)
- [Assets/Plugins/Android]以下に配置する



Unityプラグイン（Android）続き



- 書き出されると
 - JNIライブラリとして、*.soが使用可能になる
 - AndroidManifest.xmlがプラグインのものに置き換わる
 - 起動時のActivityが置き換えられる
 - アプリのパーミッション変更ができる
 - *.jarのクラスが使用可能に
 - 置き換えたActivityから使用するなどが可能

例えばカメラアクセスをする場合は
AndroidManifest.xmlの<manifest>タグに以下を加える

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-feature android:name="android.hardware.camera"/>
```



Unity側とのファイル共有 (Android)

- iOSと同様にプラグイン側で使いたいファイルは、
[Assets/StreamingAssets] 以下のファイルは、
[(アプリケーションapk)/assets/]以下に配置したファイルとして
Androidアプリ内からアクセスできる
(実際にはapk内にバンドルされる通常のファイルとしてはアクセスできない)
- assets以下のファイルは、Activity (Context)のgetAssets()から
AssetManagerを取得しInputStreamを開くことでアクセスする

```
AssetManager assetManager = getAssets();
InputStream stream = assetManager.open(name);
```

- JNI側からこれらのファイルにアクセスするときは、Activityのメソッドとして上記の処理を実装し、JNI側からそのメソッドをコールする

```
jclass cls = env->FindClass("jp/qconcept/q4u/MyActivity");
jmethodID mtd = env->GetMethodID(cls,"loadStreamingAsset","(Ljava/lang/String;) [B");
jstring filenamej = env->NewStringUTF(filename);
jbyteArray byteArray = env->CallObjectMethod(activity,mtd,filenamej);
```

iOSとAndroidプラグインのユニバーサル化



- 1つのプラグインでiOSとAndroid両方で動くものを作るには少し工夫が必要
- まず iOS, Androidそれぞれのプラグインメソッドをコールするクラスを用意
(それぞれは共通の抽象クラスを継承する)

```
// Plugin_iOS,Plugin_Android共通の抽象クラス (Plugin_abstract.cs)
public abstract class Plugin_abstract{
    public abstract void pluginMethod();
}

-----
// Plugin_iOSクラス (Plugin_iOS.cs)
public abstract class Plugin_iOS : Plugin_abstract{
    [DllImport ("__Internal")]
    private static extern void _PluginMethod();
    public void pluginMethod(){
        _PluginMethod();
    }
}

-----
// Plugin_Androidクラス (Plugin_Android.cs)
public abstract class Plugin_Android : Plugin_abstract{
    [DllImport ("PluginName")]
    private static extern void _PluginMethod();
    public void pluginMethod(){
        _PluginMethod();
    }
}
```

iOSとAndroidプラグインのユニバーサル化



- プラグインメソッドのラッパークラスを作成
- ラッパークラスから各プラットフォームに合わせたプラグインクラスのインスタンスを作成
- プラグインクラスのインスタンスから実際のプラグインメソッドをコール

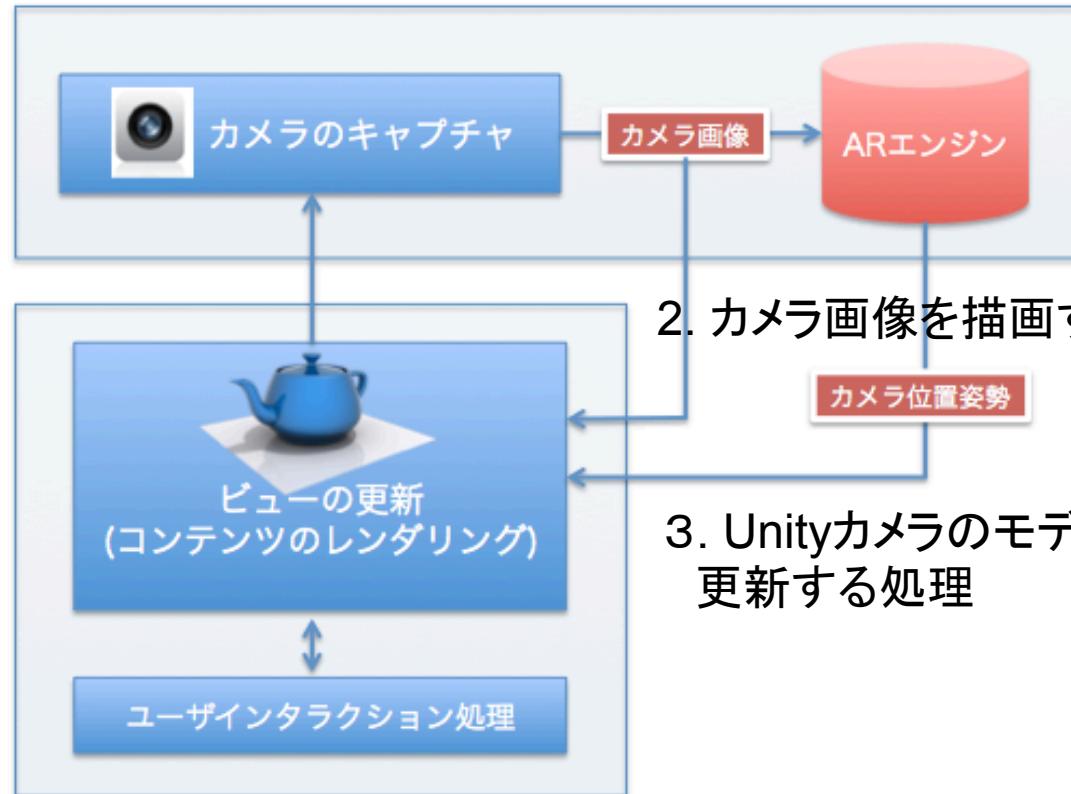
```
// Plugin_univクラス
public class Plugin_univ{
    private Plugin_abstract pluginInstance;
    public void init(){
        if( Application.platform == RuntimePlatform.IPhonePlayer ){
            pluginInstance = new Plugin_iOS();
        }else if( Application.platform == RuntimePlatform.Android ){
            pluginInstance = new Plugin_Android();
        }
    }
    public void pluginMethod(){
        pluginInstance.pluginMethod();
    }
}
```

4. UnityプラグインによるAR機能の実装

UnityとARプラグインの役割分担 (再)



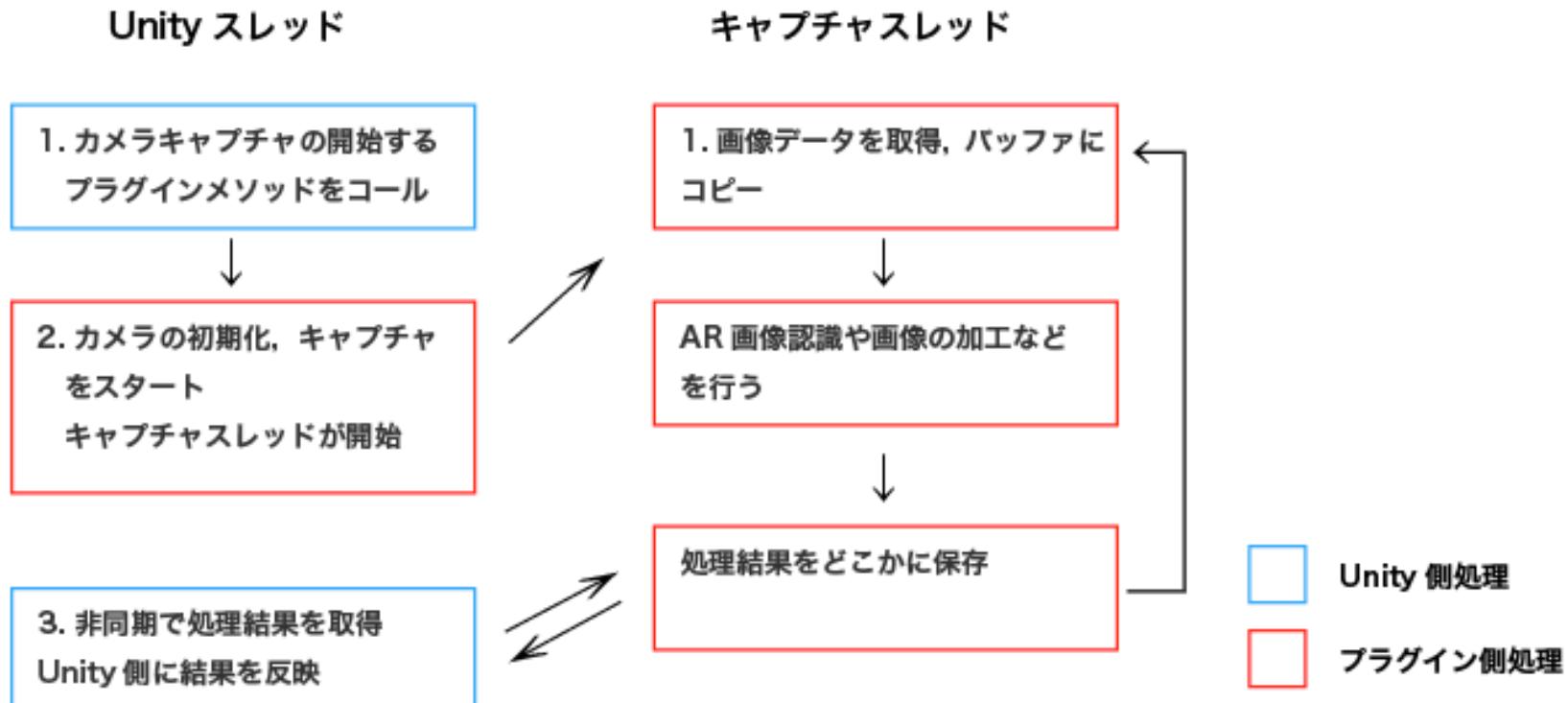
1. カメラ画像をキャプチャする処理





カメラ画像のキャプチャと表示（iOS）

- AVFoundationのAVCaptureSessionを使用
詳細は <http://goo.gl/PBilc> を参照
- キャプチャ毎にコールバックが呼ばれるので画像をバッファにコピー
→ ARの認識処理や画像の加工などに使用する
- Unity側からは非同期で結果を取得





カメラ画像のキャプチャと表示（iOS）

- 実際のキャプチャコールバック

```
- (void)captureOutput:(AVCaptureOutput *)output  
    didOutputSampleBuffer:(CMSampleBufferRef)buffer  
    fromConnection:(AVCaptureConnection *)connection  
{  
    CVImageBufferRef pixBuff = CMSampleBufferGetImageBuffer(buffer);  
    CVPixelBufferLockBaseAddress(pixBuff, 0);  
    void *buff = CVPixelBufferGetBaseAddress(pixBuff);  
  
    // なんらかの処理  
    [delegate doSomeshing:pixelBuffer];  
  
    CVPixelBufferUnlockBaseAddress(pixBuff, 0);  
}
```



カメラ画像のキャプチャと表示（iOS）

カメラ画像の表示方法は2通り

1. AVCaptureVideoPreviewLayerを使う

- ・実装が簡単
- ・GLのViewとカメラ画像のアスペクト比が異なることによる画像のクリッピングを自動的にやってくれる
- ・キャプチャした画像データと表示されている画像の間にディレイがある
→ ARでCGと合成したときにオブジェクトが遅れてついてきているよう見えてしまう

2. GLを使って自力で描画する

- ・実装がけっこう大変
- ・BGRAフォーマットならそのままテクスチャとしてつかえる
- ・YUV420フォーマットの場合、フラグメントシェーダでYUV->RGBへ変換をおこなう（Yプレーン、UVプレーンで別テクスチャとしてロード）
- ・キャプチャした画像データと表示されている画像の同期がきっちりとれる
- ・ピクセル値やテクスチャ座標を加工したカメラ画像を表示できる



AVCaptureVideoPreviewLayerを使った描画

- AVCaptureVideoPreviewLayerをAVCaptureSessionから取得し、
UIWindowの最下層のレイヤーに差し込む(もしくはEAGLLayerの直下)

```
AVCaptureSession *captureSession = [[AVCaptureSession alloc] init];
captureSession.sessionPreset = AVCaptureSessionPresetMedium;

NSError *error=nil;
AVCaptureDevice *captureDevice;
captureDevice = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
AVCaptureDeviceInput *input;
input = [AVCaptureDeviceInput deviceInputWithDevice:captureDevice error:&error];
[captureSession addInput:input];

UIWindow *win = [[UIApplication sharedApplication] keyWindow];
AVCaptureVideoPreviewLayer *previewLayer;
previewLayer = [AVCaptureVideoPreviewLayer layerWithSession:captureSession];
previewLayer.videoGravity = AVLayerVideoGravityResizeAspectFill;
previewLayer.frame = win.bounds;

[win.layer insertSublayer:previewLayer atIndex:0];

[captureSession startRunning];
```



AVCaptureVideoPreviewLayerを使った描画

- このときEAGLLayerは透過させるフォーマットにしなくてはいけない
(AVCaptureVideoPreviewLayerとEAGLLayerは別レイヤーのため)

```
UIWindow *win = [[UIApplication sharedApplication] keyWindow];
//glLayerを探す
NSArray *layers = win.layer.sublayers;
CAEAGLLayer *glLayer=nil;
for(int i=0;i<[layers count];i++){
    CALayer *layer = [layers objectAtIndex:i];
    if([layer isKindOfClass:[CAEAGLLayer class]]){
        glLayer = (CAEAGLLayer *)layer;
        NSLog(@"layer %d is GLLayer",i);
    }
}

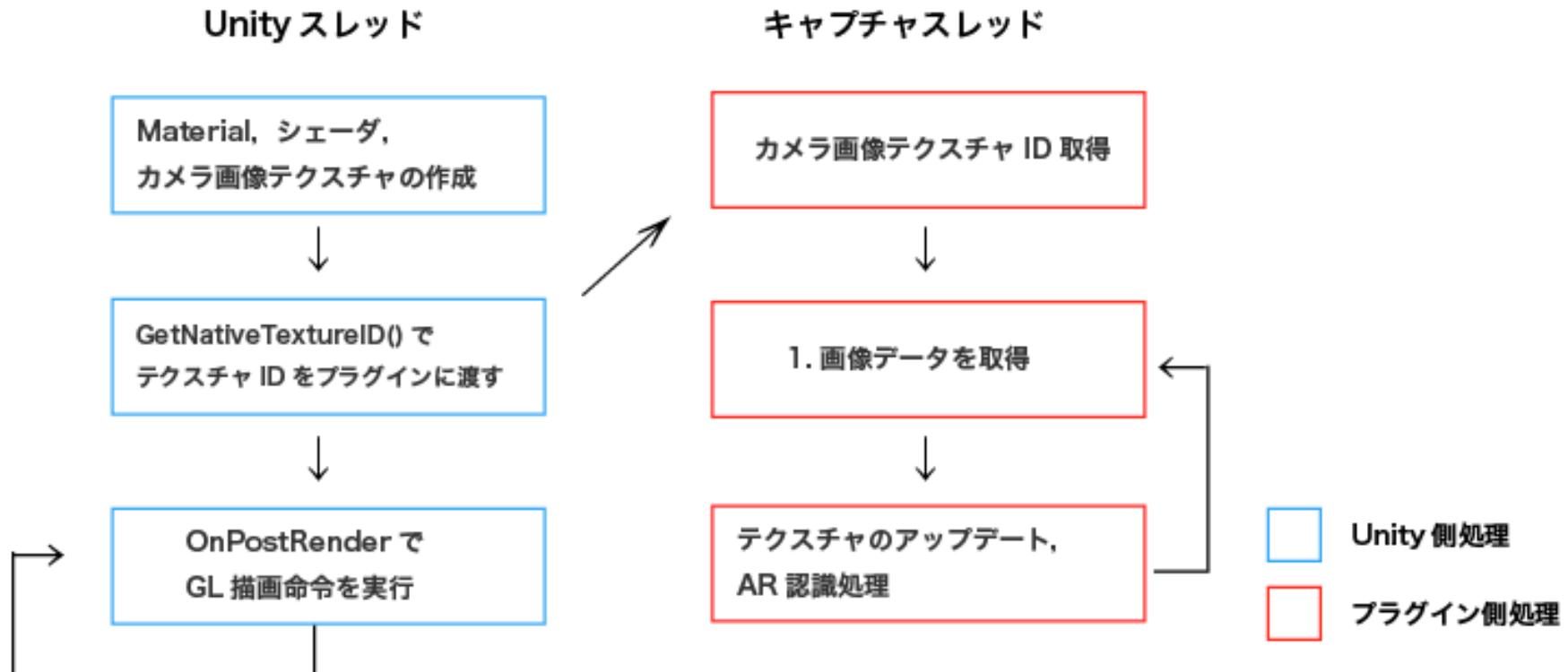
//背景透過
glLayer.opaque = NO;
glLayer.drawableProperties = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithBool:FALSE],
    kEAGLDrawablePropertyRetainedBacking,
    kEAGLColorFormatRGBA8, kEAGLDrawablePropertyColorFormat,
    nil];

EAGLContext *glContext = [EAGLContext currentContext];
[glContext renderbufferStorage:GL_RENDERBUFFER fromDrawable:glLayer];
```

GLを使ったダイレクト描画



- YUVはかなりテクってて説明が長くなるのでBGRAの時のみ
- 大まかなフローは、
キャプチャ → プラグイン側でGLテクスチャ更新 → Unity側で描画
- Texture.GetNativeTextureIDでGLのテクスチャIDがUnity側からとれる



GLを使ったダイレクト描画（プラグイン側）



- マルチスレッドでのGLステート更新は メインコンテキストと EAGLShareGroup を共有した別のサブEAGLContextを使って行う
これによってGLリソースのスレッド間共有を行う

```
EAGLContext *glMainContext, glShareContext;
glMainContext = [EAGLContext currentContext];
glShareContext = [[EAGLContext alloc] initWithAPI:glMainContext.API
                                         sharegroup:glMainContext.sharegroup];
```

- 別スレッドからはこのサブEAGLContextでテクスチャを更新

```
[EAGLContext setCurrentContext:glShareContext];
glBindTexture(GL_TEXTURE_2D, textureId);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_BGRA_EXT, GL_UNSIGNED_BYTE, rgbBuffer);
glBindTexture(GL_TEXTURE_2D, 0);
glFlush();
```

GLを使ったダイレクト描画 (Unity側)



- 実際の描画はOnPostRender() (or なんらかの描画コールバック) で、UnityのGLクラスを使って行う
- このときのマテリアルにはネイティブ側に渡したTextureIDをもつテクスチャをmainTextureとして設定
- GLクラス → <http://goo.gl/KULXt>

GLによるカメラ画像描画の例

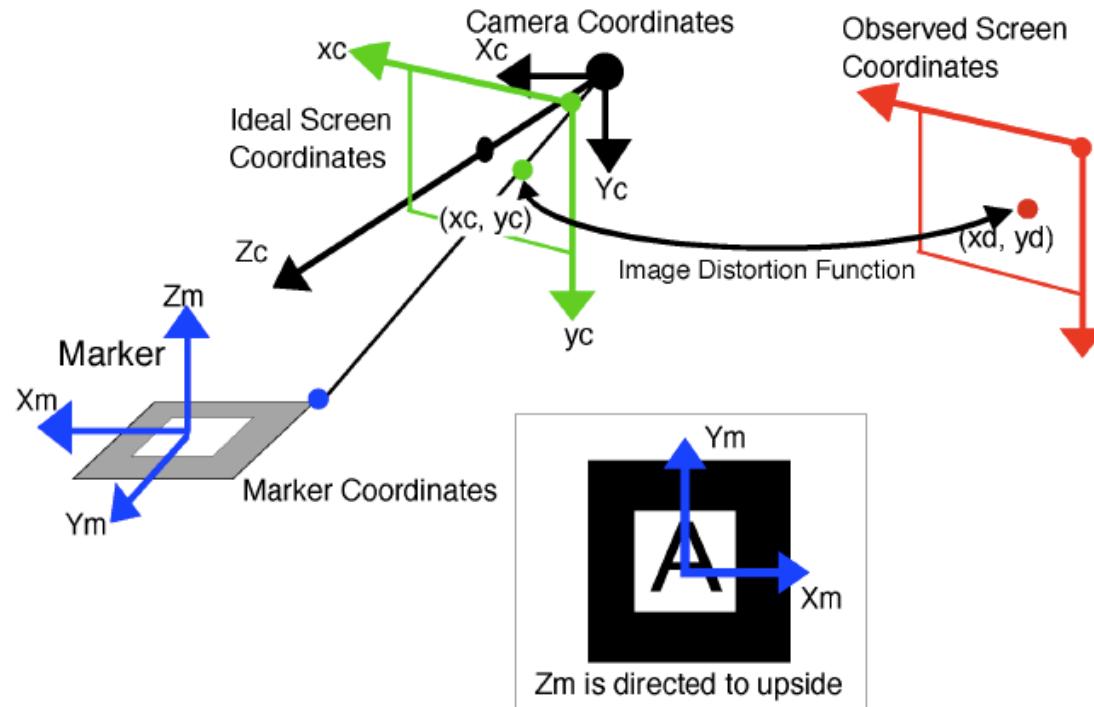
(実際にはカメラ画像とフレームバッファの縦横比の調整をテクスチャ座標値で調整する)

```
camImageShader.SetPass(0);
GL.LoadProjectionMatrix(Matrix4x4.identity);
GL.LoadIdentity();
GL.Begin(GL.QUADS);
float s = 1.0f;
float[,] vertex = {{-1,-1}, {-1,+1}, {+1,+1}, {+1,-1}};
float[,] texcoord = {{1,1}, {0,0}, {1,0}, {1,1}};
for(int i=0;i<4;i++){
    GL.Vertex3(vertex[i,0],vertex[i,1],+1);
    GL.TexCoord2(texcoord[i,0],texcoord[i,1]);
}
GL.End();
```



実世界のカメラとUnityカメラの位置合わせ

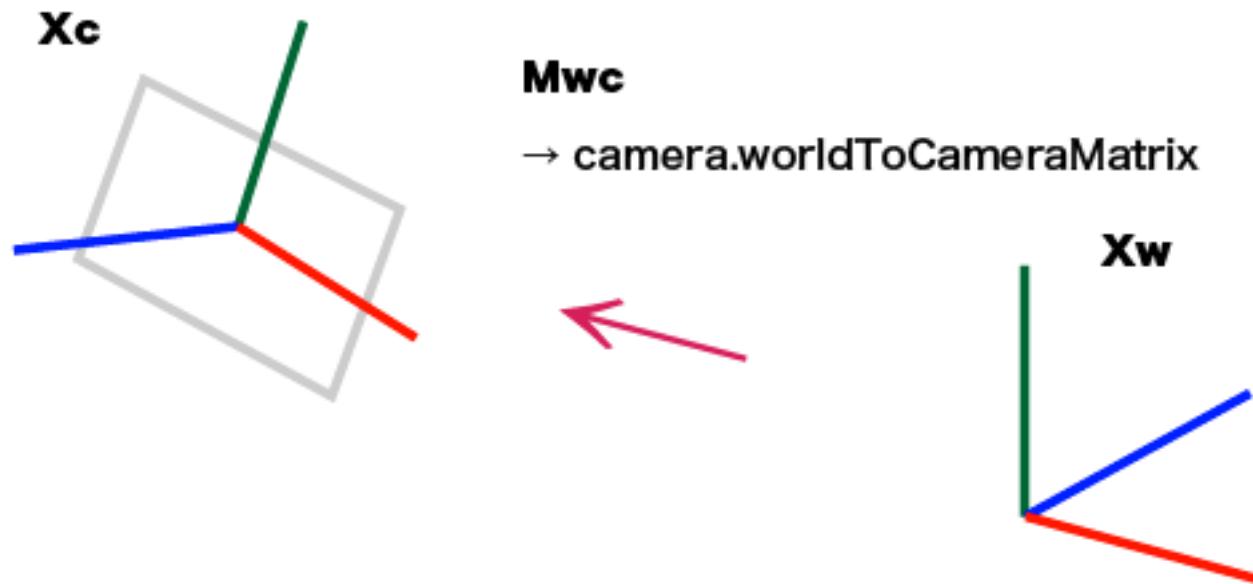
- ARにおけるCGの合成はモデルビュー行列（カメラの位置姿勢）とプロジェクション行列によって実現される
- マーカ座標系=Unityにおけるワールド座標系



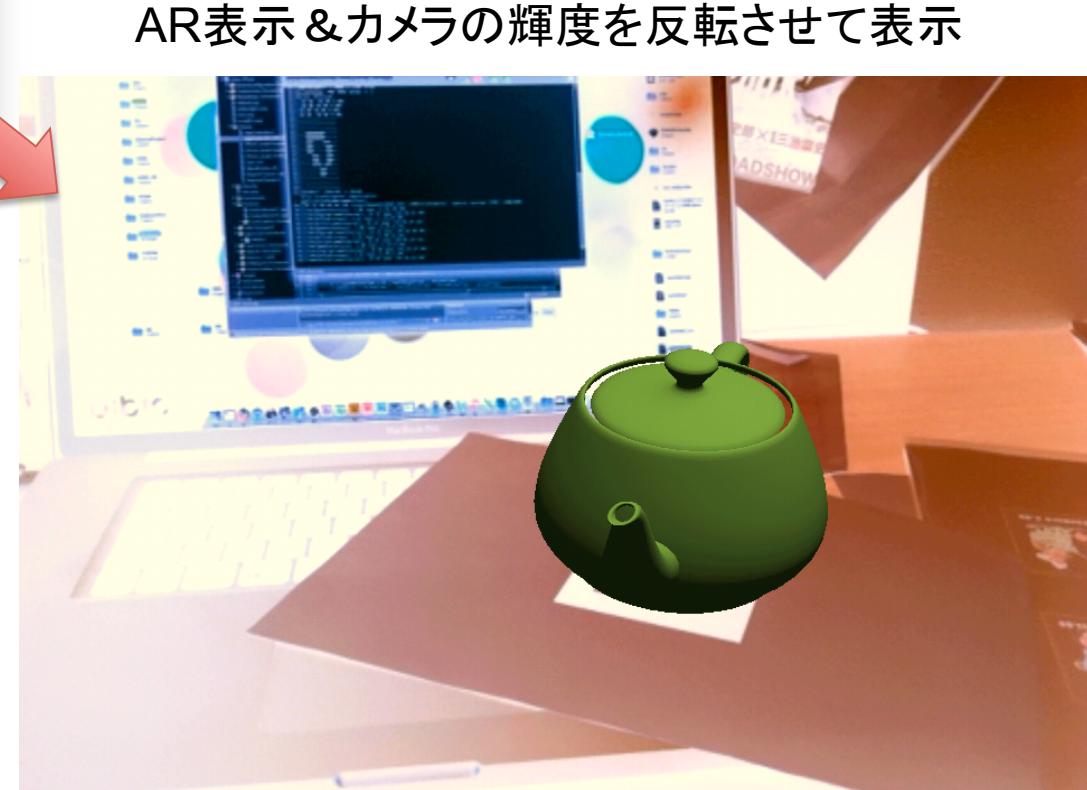


Unity側でのカメラ位置姿勢の更新

- Camera.worldToCameraMatrixに直接マーク座標系からカメラ座標系への変換行列を設定
- Camera.projectionMatrixにはデバイス側のカメラの画角などから、GLのプロジェクション行列を作成し設定
(iOSではGLフレームバッファの向きとカメラ画像の向きが90度回転しているのでCamera.projectionMatrixで向きを合わせる)



ARできあがりイメージ



AR表示 & カメラの輝度を反転させて表示

5. ARプラグインの紹介

最後に



以上の発表の内容だけでなく、それ以外も大量にノウハウがつまったARプラグインを近日発表いたします

- iOS/Androidのユニバーサルプラグイン
 - ARの各種設定（マーカ画像など）はUnity上で完結
 - ARMv7に特化した最新AR エンジン 搭載
(1フレーム処理時間: 平均3msec on iPhone4)
 - 高性能カメラキャプチャ/プレビュー (iOSのみ)
 - 複数マーカ同時認識対応
- などなど

資料URL



http://qconcept.jp/ar/seminar/unity_20110716.pdf