

开发原则

要使用 ARToolKit 开发应用有两个部分：编写应用程序，以及训练对增强现实应用中所用到的真实世界标志的图像处理例程。

使用 ARToolKit 编写应用是很简单的：新建一个 AR 应用需要一个简单的框架。我们在这个框架的基础上编写新的应用。同样地，因为应用这个简单的框架，训练模板的过程也被简化。

一个应用程序的主代码必须包含以下步骤：

初始化	1 初始化视频捕获，读取标识文件和相机参数
	2 抓取一帧输入视频的图像
	3 探测标识以及识别这帧输入视频中的模板
	4 计算摄像头相对于探测到的标识的转换矩阵
	5 在探测到的标识上叠加虚拟物体
关闭	6 关闭视频捕捉

第二步到第五步一直重复，直到应用程序退出。但是步骤一和步骤六只分别在应用程序的初始化时和关闭时才执行。除了这些步骤之外，一个应用程序还应该对鼠标、键盘或者其他特殊事件响应。

下一页将详细介绍各个步骤，紧接着是讲解如何训练标识（还有处理多个标识的情况）。

开发你的第一个程序：第一部分

Introduction

main

init

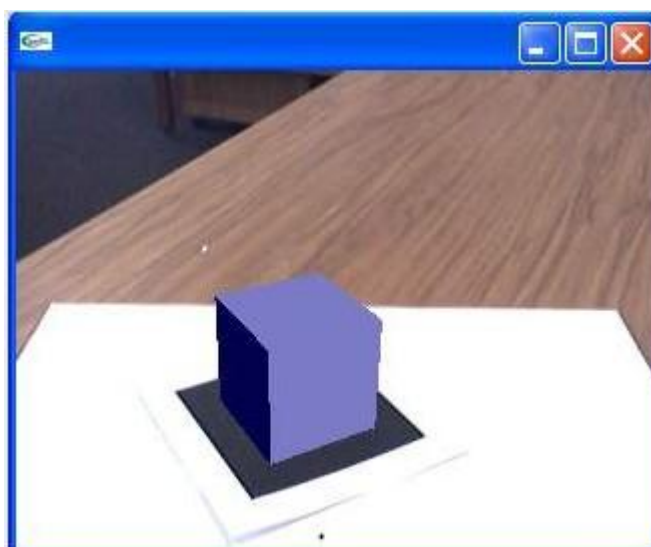
mainLoop

draw

cleanup

介绍

为了详细地示范怎么样开发一个 ARToolKit 的应用，我们将一步步地介绍一个现有的例程的源代码：simpleTest（或者在有的版本里是 simple）。可以在目录 examples/simple/里找到这个程序。



程序 simpleTest

我们要找的文件名字是 simpleTest.c (或者 simple.c)。这个程序仅仅包含了一个主函数和几个绘制图像的函数。

相应于上节介绍的六个应用步骤的函数列出在表 1 中。相应于步骤二到步骤五的函数在 mainLoop 函数（主循环）中被调用。

表格 1：相应于 ARToolKit 应用程序步骤的函数调用和代码

ARToolKit 步骤	函数
1、应用程序初始化	init
2、抓取一帧输入视频	arVideoGetImage (在主循环中调用)
3、探测标识卡	arDetectMarker(在主循环中调用)
4、计算摄像头的转移矩阵	arGetTransMat(在主循环中调用)

5、画上虚拟物体	draw(在主循环中调用)
6、关闭视频捕捉	cleanup

在这个程序中，最重要的函数是 `main` ,`init` , `mainloop` ,`draw` 和 `cleanup`。在本节的其他部分我们将详细地解释这些函数调用。

main

Simple 例程中 `main` 函数的流程如下所示：

```
main(int argc, char *argv[])
{
    init();
    arVideoCapStart();
    argMainLoop( NULL, keyEvent, mainLoop );
}
```

其中的初始化例程 `init` 包含的代码可以初始化视频捕捉，读取标识卡信息和摄像机参数信息，以及设置图像窗口。这相对于《开发原则》中的第一步。接下来，我们通过调用视频开始函数 `arVideoCapStart` 输入实时状态。再接着，函数 `argMainLoop` 被调用，这个函数启动了主要的程序循环，通过键盘事件与函数 `keyEvent` 结合使用，通过主要的图像显示循环与 `mainLoop` 结合使用。函数 `argMainLoop` 的定义在文件 `gsub.c` 中。

init

`init` 例程在 `main` 例程中被调用，它的作用是初始化视频捕捉以及读入 `ARToolKit` 应用的初始参数信息。

首先，视频通道被打开，确定视频图像大小：

```
/* open the video path */
if( arVideoOpen( vconf ) < 0 ) exit(0);

/* find the size of the window */
if( arVideoInqSize(&xsize, &ysize) < 0 ) exit(0);
printf("Image size (x,y) = (%d,%d)\n", xsize, ysize);
```

变量 `vconf` 包含了初始视频的配置，在 `simple.c` 的顶部被定义。但它的内容在你的平台的函数里可能很不一样：参照[视频配置链接](#)。对于每一个平台，都定义了一个默认的字符串，这个字符串一般都打开你的应用程序结构中第一个可用的视频流。

然后，我们需要初始化 `ARToolKit` 应用程序的参数。对于 `ARToolKit` 应用程序来说，关键的参数是：

- 可能被用来进行模板模式匹配的模板信息，以及这些模板锁对应的虚拟物体。

- 所用的视频摄像机的相机特性参数。

这些都是从文件里读取，这些文件的名称可以在命令行里被指定，或使用硬件编码的文件的默认名称。

因此，摄像机的参数信息通过默认的摄像机参数文件名 Data/camera_para.dat 被读入：

```
/* set the initial camera parameters */
if( arParamLoad(cparaname, 1, &wparam) < 0 ) {
    printf("Camera parameter load error !!\n");
    exit(0);
}
```

接下来，这些参数根据现有的图像大小被转换，因为摄像机的参数根据图像的大小而改变，甚至是使用相同的摄像机。

```
arParamChangeSize( &wparam, xsize, ysize, &cparam );
```

摄像机的参数被读入它的程序设置，摄像机的参数被输出显示到屏幕上：

```
arInitCparam( &cparam );
printf("*** Camera Parameter ***\n");
arParamDisp( &cparam );
```

这样之后我们通过默认的模板文件 Data/patt.hiro 读入模板的定义信息：

```
if( (patt_id=arLoadPatt(patt_name)) < 0 ) {
    printf("pattern load error !!\n");
    exit(0);
}
```

其中 patt_id 是一个已经被识别的模板的鉴定信息（告诉我们是哪一个模板，相当于人类的身份证）。

最终打开了图像窗口：

```
/* open the graphics window */
argInit( &cparam, 1.0, 0, 0, 0, 0 );
```

函数 arginit 的第二个参数定义了一个缩放函数，适应视频图像格式时的值设为 1.0，值设为 2.0 时是双倍大小（比如说，输入 320*240 图像，输出为 VGA AR 格式）。

mainloop

ARToolKit 应用程序的大部分调用都在这个例程里完成，这个例程包含了相对于《开发原则》中所要求的步骤二到步骤五。

首先通过函数 `arVideoGetImage` 来捕捉一个输入视频帧：

```
/* grab a video frame */
if( (dataPtr = (ARUint8 *)arVideoGetImage()) == NULL ) {
    arUtilSleep(2);
    return;
}
```

该视频图像立即被输出显示到屏幕上。这个图像可以是一幅没有被扭曲的图像，也可以是一幅根据摄像头的失真信息被扭曲修正。扭曲以修正图像可以生成更加正常的图像，但是可能会导致视频帧的速率明显降低。在下例中图像是已经被扭曲的：

```
argDrawMode2D();
argDispImage( dataPtr, 0,0 );
```

接着函数 `arDetectMarker` 被使用以搜索整个图像来寻找含有正确的标识模板的方块：

```
if( arDetectMarker(dataPtr, thresh, &marker_info, &marker_num) < 0 ) {
    cleanup();
    exit(0);
});
```

找到的标识卡的数量被存放在变量 `marker_num` 里，同时 `marker_info` 是一个指向一列标识结构体的指针，这个结构体包含了坐标信息，识别可信度，以及每个标识对应的鉴定信息和物体。`marker_info` 的详细信息在 [API documentation](#) 中。

此时，视频图像已经被显示和分析了。所以我们不需要再使用它：我们可以在使用新的函数的同时使用帧捕捉器来启动一个新的帧捕捉操作。完成这些工作，你只需要调用函数 `arVideoCapNext`：

```
arVideoCapNext();
```

备注：当你调用这个函数时，使用上一个视频图像缓冲会导致坏的结果（根据你的应用程序平台而定）。确保你已经处理好了视频图像缓冲。

接下来，所有的已经探测到的标识的可信度信息被加以比较，最终确定正确的标识鉴定信息为可信度最高的标识的鉴定信息：

```

/* check for object visibility */
k = -1;

for( j = 0; j < marker_num; j++ ) {
    if( patt_id == marker_info[j].id ) {
        if( k == -1 ) k = j;
        else if( marker_info[k].cf < marker_info[j].cf ) k = j;
    }
}

if( k == -1 ) {
    argSwapBuffers();
    return;
}

```

标识卡和摄像机之间的转移信息可以通过使用函数 `arGetTransMat` 来获取:

```

/* get the transformation between the marker and the real camera */
arGetTransMat(&marker_info[k], patt_center, patt_width, patt_trans);

```

相对于标识物体 *i* 的真实的摄像机的位置和姿态包含在一个 3*4 的矩阵 `patt_trans` 中。最后, 使用绘图函数, 虚拟物体可以被叠加在标识卡上:

```

draw();
argSwapBuffers();

```

备注: 如果没有标识被找到 (`k==-1`), 应用程序会做一个简单的优化步骤, 我们可以交换缓冲器而不需要调用函数 `draw`, 然后返回:

```

if( k == -1 ) {
    argSwapBuffers();
    return;
}

```

draw

函数 `draw` 分为显示环境初试化, 设置矩阵, 显示物体几个部分。你可以使用 `ARToolKit` 显示一个三维物体并设置最小的 OpenGL 状态来初始化一个 3d 显示:


```
argDrawMode3D();
argDraw3dCamera( 0, 0 );
glClearDepth( 1.0 );
glClear(GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
```

在这之后你需要这个把转移矩阵（3*4 的矩阵）转化成 OpenGL 适用的格式（16 个值的向量），可用函数 `argConvGlpara` 来完成此功能。这十六个值是真实世界的摄像机的位置和姿态信息，因此利用这些信息可以设置虚拟世界摄像机的位置，因此任何的图形物体都可以被准确地放置在相应的真实标识卡上。

```
/* load the camera transformation matrix */
argConvGlpara(patt_trans, gl_para);
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd( gl_para );
```

虚拟世界的摄像机的位置是用函数 `glLoadMatrixd(gl_para)` 来设置的。代码的最后是三维物体的显示。在这个例子中，显示的是白色光束下是一个蓝色立方体：

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambi);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_flash);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_flash_shiny);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMatrixMode(GL_MODELVIEW);
glTranslatef( 0.0, 0.0, 25.0 );
glutSolidCube(50.0);
```

在最后，你要重置某些 OpenGL 的参数为默认值：

```
glDisable( GL_LIGHTING );
glDisable( GL_DEPTH_TEST );
```

上述所讲到的步骤出现并贯穿了主要显示函数的始终，当这个程序在运行时，鼠标事件被鼠标事件函数控制，键盘事件被键盘函数控制。

cleanup

函数 `cleanup` 被调用的作用的停止视频处理以及关闭视频路径并释放它使其他的应用可以使用：

```
arVideoCapStop();  
arVideoClose();  
argCleanup();
```

这些工作可以使用函数 `arVideoCapStop`, `arVideoClose` 和 `argCleanup` 来完成。

你可以编译这个程序并运行它！

这个程序的一个限制的，它只使用模板 **Hiro**：使用其他多个模板是很有趣的！我们将在下一节介绍怎么样使用其他模板。

开发你的第一个程序：第二部分

使用其他的模板

程序 `simpletest` 使用模板匹配法来识别标识方框中的 `Hiro` 字样。输入视频流中的方块被系统与之前训练过的模板相比较。这些模板在运行时被加载，包含在 `bin` 目录下的名为 `data` 的目录下。这个目录下，我们找到了上次应用程序所用到的文件，比如说，名字为 `patt.hiro`。这个文件包含了模板的格式，仅仅是一个样本图案。

为了改变 `simpletest` 中识别的模板，你需要改动你的代码，创建一个新的模板文件。

你可以通过改变夹在文件夹名字来修改 `simpletest.c` 文件，将：

```
char *patt_name = "Data/patt.hiro";
```

改为：

```
char *patt_name = "Data/patt.yourpatt";
```

这段程序生成的新的模板文件名为 `mk_patt`，包含在 `bin` 目录下。`mk_patt` 的源代码在 `util` 目录下的文件 `mk_patt.c` 里。

要创建一个新的模板，首先应打印模板目录下的 `blankpatt.c` 文件。这只是一个黑方块，中间是空的白色方块。接着为需要的模板创建一个黑白或者彩色的、适合这个中心的方块的图像，并把它打印出来。好的模板应该是不对称，而且没有很细微的细节的模板。图 1 展示了一下样本模板。将做好的新模板粘在黑方块里。

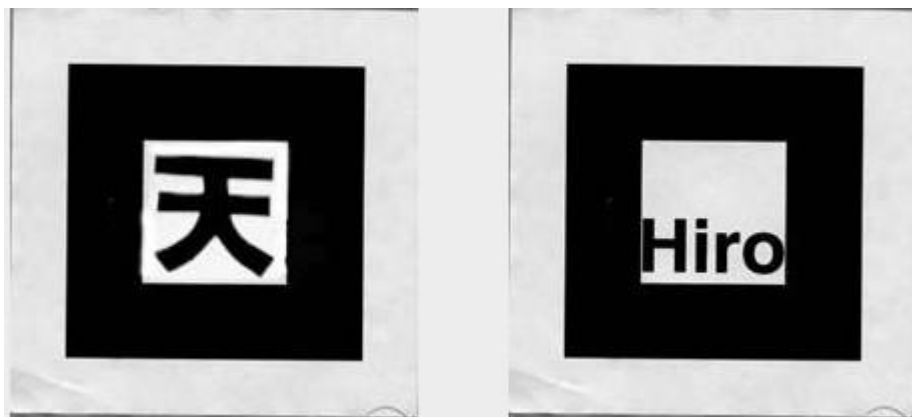


图 1 样本模板

一旦新的模板制作完毕，改变 `bin` 目录，运行 `mk_patt` 程序（仅在控制台模式下）。系统会提示你输入一个摄像机的参数文件夹名字。输入文件夹名：`camera_para.dat`。这是默认的摄像机的参数文件。

```
ARToolKit2.32/bin/mk_patt  
Enter camera parameter filename: camera_para.dat
```

这段程序接着会打开一个视频窗口，如图 2 所示：

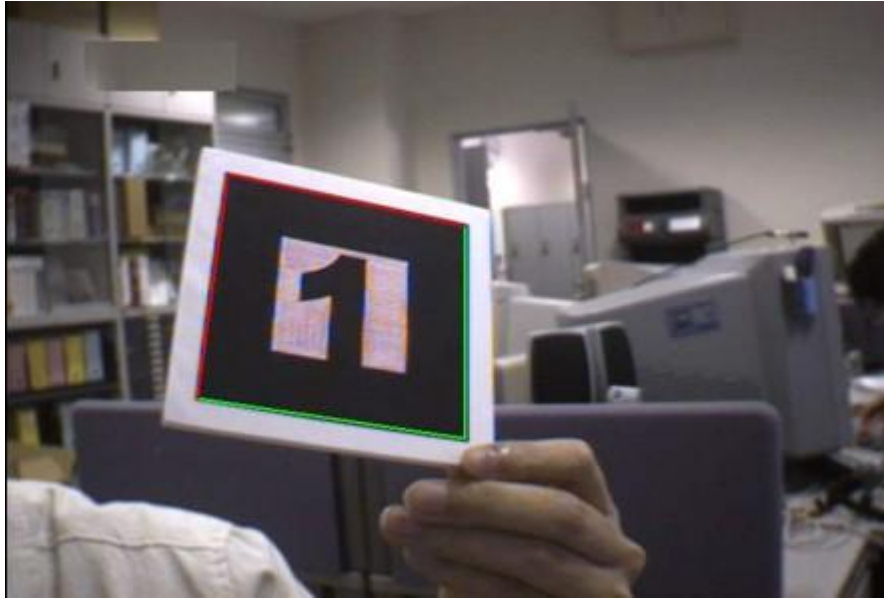


图 2 mk_patt 视频窗口

把要训练的模板放在一个平的表面上，光照条件应和运行识别应用程序时的光照条件相同。然后把视频摄像头拿起在标识的上面，向下直对着标识，转动它直到标识的周围出现一个红色和绿色的方框。这指示软件 `mk_patt` 已经找到了围绕在待测试的模板周围的方框。应该转动摄像头直到视频图像中的方块的左上方边角是高亮的方块的红色的边角，如图 2 中所示。一旦方块被找到且方位正确，单击鼠标左键。接着系统会提示你输入一个模板的文件名字。比如说，输入 `patt.yourpatt`。

一旦文件名字被输入，系统就生成了一个该模板的位图图像，位图图像被复制到以这个文件名命名的文件中。接下来这个将被用在 `ARToolKit` 的模板匹配中。为了使用这个新模板，这些数据要被拷贝到文件目录 `bin/Data` 下。重新编译 `simpletest` 后，现在，你就可以使用你自己的模板了！

训练了一个模板后，其他的模板也可以被训练，只需要用摄像头对着新模板并重复以上步骤，或者，单击鼠标右键可以退出应用程序。

使用多个模板

现在我们想要使用不止一个的模板，而不同的模板有各自不同的三维物体相对应。为达到此目的，我们将逐步分析目录 `examples/simplem/` 下的 `simplem` 文件的源代码。你会发现两个源文件，`simplemTest.c` 和 `object.c`。这个程序可以探测多个标识卡，并且在每个标

识上面显示不同形状的物体（锥体，立方体，球体）。

它和 `simple` 程序的主要区别是：

- 加载的文件中有多个模板的声明。
- 与模板相关联的结构不同，这意味着程序中检查代码以及转换调用不同。
- 语法重新定义，定义画图函数。

其他的代码则都是一样的！

系统建议使用一个特定的函数——`object.c` 中的 `read_ObjData` 来加载 `ARToolKit` 中的多个模板。利用此函数，可以用如下方法来加载标识：

```
if( (object=read_ObjData(model_name, &objectnum)) == NULL ) exit(0);  
printf("Objectfile num = %d\n", objectnum);
```

参量 `object` 是一个指向一个 `ObjectData_T` 的结构体的指针。参量 `model_name` 定义的不是一个模板定义文件名（在这里文件名是 `model_name`），而是一个特定的多个模板定义的文件名（警告：这个格式和多个模板跟踪文件名不同!!!）。文本文件 `object_data` 指定了哪些标识物体应被识别以及模板怎么样与各个物体相关联。文件 `object_data` 的开始处记录了要被指定的物体的数量，接着是每个物体的文本类型的数据结构。`object_data` 文件中每个标识都被以下结构体详细说明：

- 名字
- 模板识别文件名
- 跟踪模板的宽度
- 跟踪模板的中心

比如说，对应着与虚拟的立方体相关的标识的结构体如下：

```
#pattern 1  
cone  
Data/patt.hiro  
80.0  
0.0 0.0
```

请注意，以 `#character` 开始是代码是命令行，被文件读取器忽略。

`ARToolKit` 可以试着在 `arDetectMarker` 流程中识别多个模板了。因为我们现在是探测多个模板，我们需要保持每一个虚拟物体的可见性，同时修改对于以及探测到的模板的检查步骤。更进一步，我们还需要维持每个已探测模板的特定的转移。

```

/* check for object visibility */
for( i = 0; i < objectnum; i++ ) {
    k = -1;
    for( j = 0; j < marker_num; j++ ) {
        if( object[i].id == marker_info[j].id ) {
            if( k == -1 ) k = j;
            else if( marker_info[k].cf < marker_info[j].cf ) k = j;
        }
    }

    if( k == -1 ) {
        object[i].visible = 0;
        continue;
    }

    object[i].visible = 1;
    arGetTransMat(&marker_info[k],
object[i].marker_center, object[i].marker_width,
object[i].trans);
}

```

因此，如果标识被检测到，每一个标识都有一个视觉标志和一个新的转移矩阵。现在通过结构体 `ObjectData_T` 调用绘图函数来绘制虚拟物体。结构体 `ObjectData_T` 需要被赋予虚拟物体的参数以及虚拟物体的个数。

```

/* draw the AR graphics */
draw( object, objectnum );

```

绘图函数同样很容易理解：遍历物体的列表，如果物体可见，利用它的姿态按照相应的形状绘制物体。

现在可以编译 `simplem`，确保所有必须的文件已经被放在 `data` 文件目录下。结果如图 3 所示。

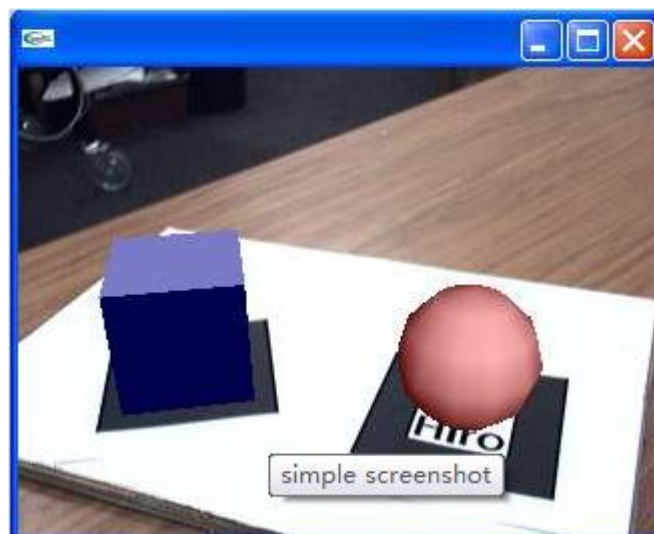


图 3 `simplem` 视频窗口

你可以修改文件 `object_data`，使用你自己的模板实验了！

教程 1：跟踪稳定性

介绍

在前面几节，我们已经看了怎样创建一个简单的 ARToolKit 程序。现在我们想要介绍 ARToolKit 的一个重要的特点：历史函数。在 bin 目录下找到 simpleTest2 程序，运行它。你可以看到与图 1 相似的屏幕抓图。

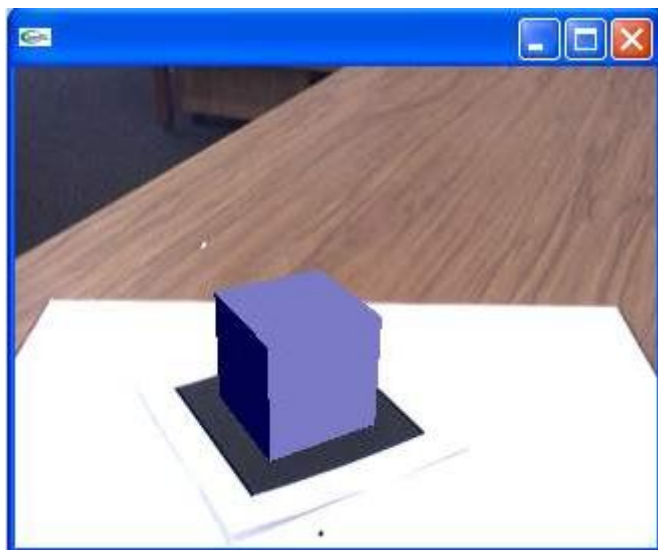


图 1 simpleTest2 程序

也许这个和 simpleTest 程序没有立即表现出不同。然而，如果移动模板靠近摄像头（如图 2 所示），按下键盘 ‘c’ 键，就出现了不同。在一种情况下，方块看起来很稳定；在另一种情况下，方块像是有点微微的抖动。前一种情况就是我们使用了历史函数，在后一种情况下没有使用。

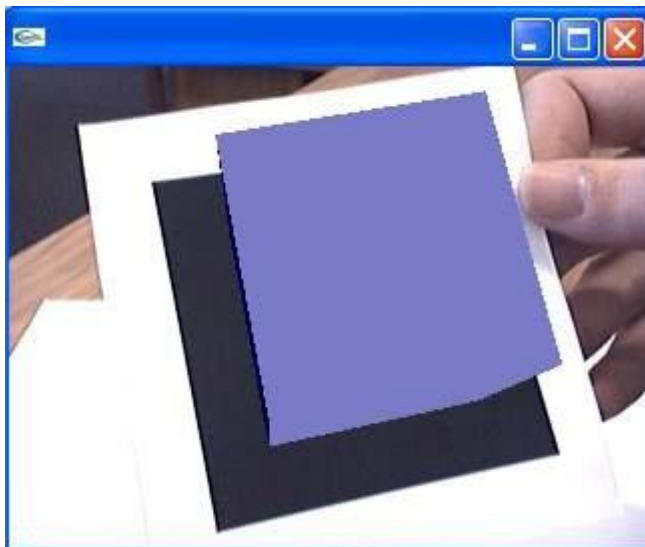


图 2 历史函数没有使用时更近距离的观察出现抖动

使用历史函数

打开 example/simple2 目录下的 simpleTest2.c 程序。在主循环 mainloop 函数中可以找到下列函数调用：

```
/* get the transformation between the marker and the real camera */
if( mode == 0 || contF == 0 ) {
    arGetTransMat(&marker_info[k], patt_center,
        patt_width, patt_trans);
} else {
    arGetTransMatCont(&marker_info[k], patt_trans, patt_center,
        patt_width, patt_trans);
}

contF = 1;
```

函数 arGetTransMatCont 使用以前的图像帧的信息来减小标识卡的抖动。利用函数 arGetTransMat，计算标识的位置时只用到了当前的图像帧的信息。使用历史函数时，结果的精确性会降低，因为历史信息的使用增加了精确性的损耗。

ARToolKit 提供了另外一个同样具有历史性的函数，但用在探测阶段。我们已经介绍过这个函数：arDetectMarker。相对应的、没有使用历史信息的函数是 arDetectMarkerLite。和前面一样，使用历史信息会降低精确度，但是提供了更为稳定的图像，而且速度稍快些。

用下列代码代替 arDetectMarker：

```
/* detect the markers in the video frame */
if( arDetectMarkerLite(dataPtr, thresh,
    &marker_info, &marker_num) < 0 ) {
    cleanup();
    exit(0);
}
```

重新编译 simpleTest2，放置标识卡使它面对着摄像头（如图 3）。你会发现方块确实在“跳”。

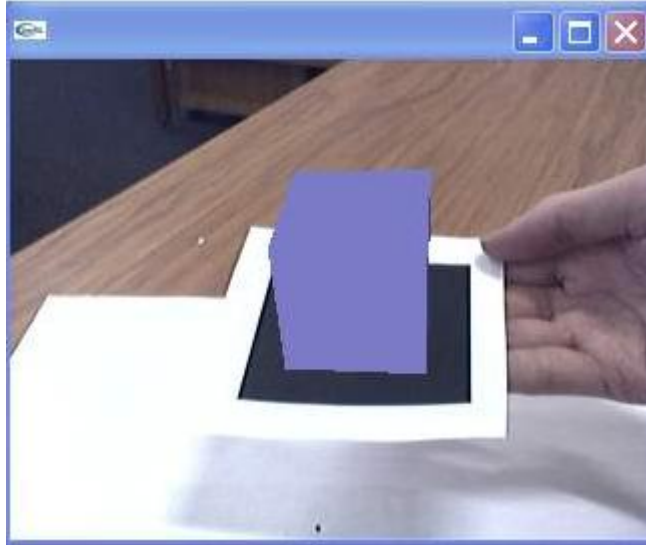


图 3 没有使用历史函数时近距离的观察出现抖动

跳动的效果的原因是没有上一帧的信息，标识的位置不够好，使探测时不够有效。如果有一个的大小、位置和上一帧的标识的几乎一样的标识，即使模板匹配不是很成功，这个标识仍被视为是和上一帧的标识一样。

教程 2：摄像头和标识关系

介绍

ARToolKit 提供了标识在摄像机的坐标系统中的位置，使用 opengl 矩阵系统计算出虚拟物体的位置。在本节教程中，我们会更详细地讲解这些不同的元素及其之间的关系。

坐标系统

我们先来介绍 simpleTest 程序。打开它，把下面这句代码加在 arGetTransMat 后：

```
printf("%f %f %f\n",patt_trans[0][3],patt_trans[1][3],patt_trans[2][3]);
```

重新编译，运行程序。注意输出值。如果把标识往左移动，第一个值就会增加，往上移动，第二个值就减小，往前移动最后一个值也会增加。下面是 ARToolKit 所使用的坐标系统（CS）。

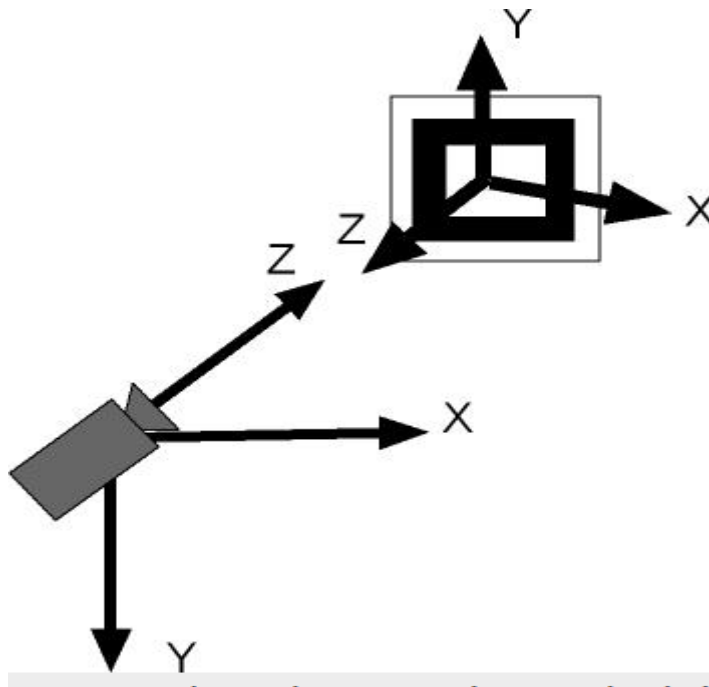


图 1 ARToolKit 坐标系（摄像机和标识）

输出的值与这些坐标系统相对应。标识坐标系统有着和 `opengl` 坐标系统一样的方位。因此任何应用于与标识关联的物体的转换都应遵循 `opengl` 的转换规则。比如说，如果不想把方块显示在标识坐标系统的中心，而是让它显示在顶部，可以把下面的代码：

```
glTranslatef(0.0, 0.0, 25.0);
```

代替为：

```
glTranslatef(0.0, 20.0, 25.0);
```

那么就会得到：

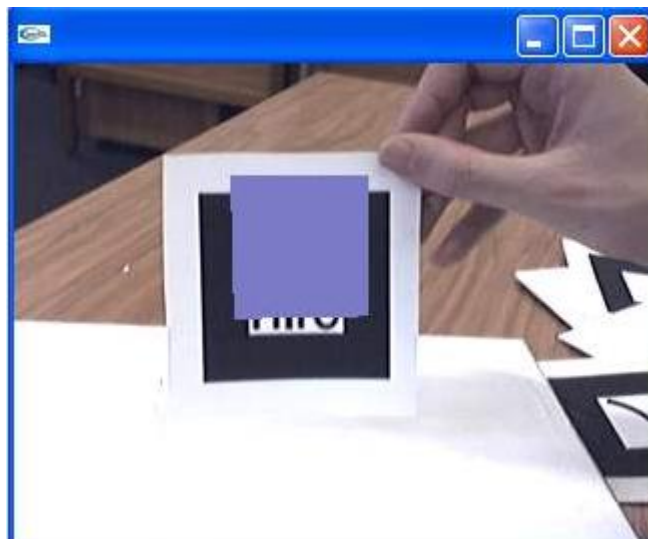


图 2 虚拟物体移动后的 simpleTest

更直观的是假设标识静止而真正的摄像头在移动。这就是说，我们想要得到摄像头在标识坐标系中的位置。在 `bin` 目录下运行 `exview`。这个程序的显示如图 3 所示，同时输出了摄像头在标识坐标系中的位置。另外一个视图显示的是摄像头和标识的三维模型。

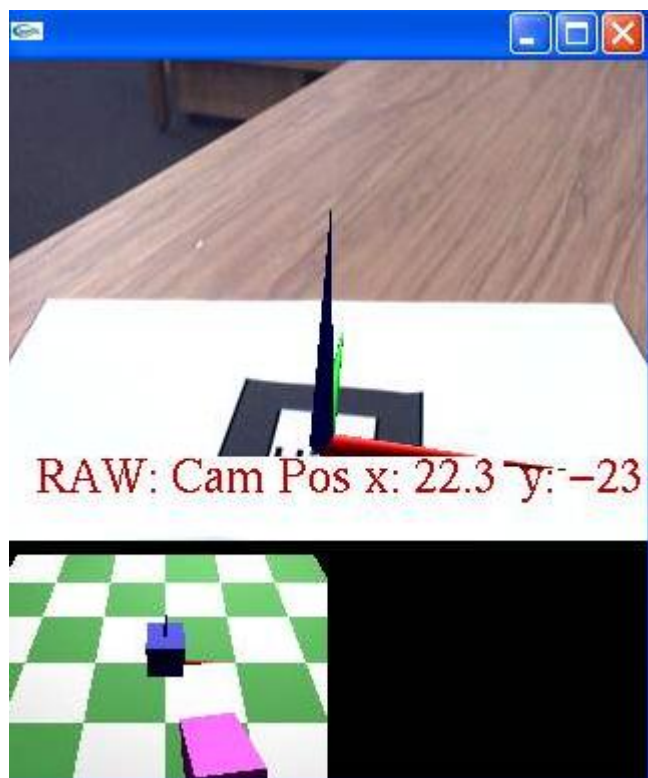


图 3 exview 视频截图。在上面的图像中，可以看到标识卡的坐标（蓝色 Z 轴，绿色 Y 轴，红色 X 轴）。下面的图像展示了摄像头在标识坐标系中的位置

你可以把鼠标移动到三维外部视图中，试着操作感受它们之间的关系（图 4 是另外的例子）。

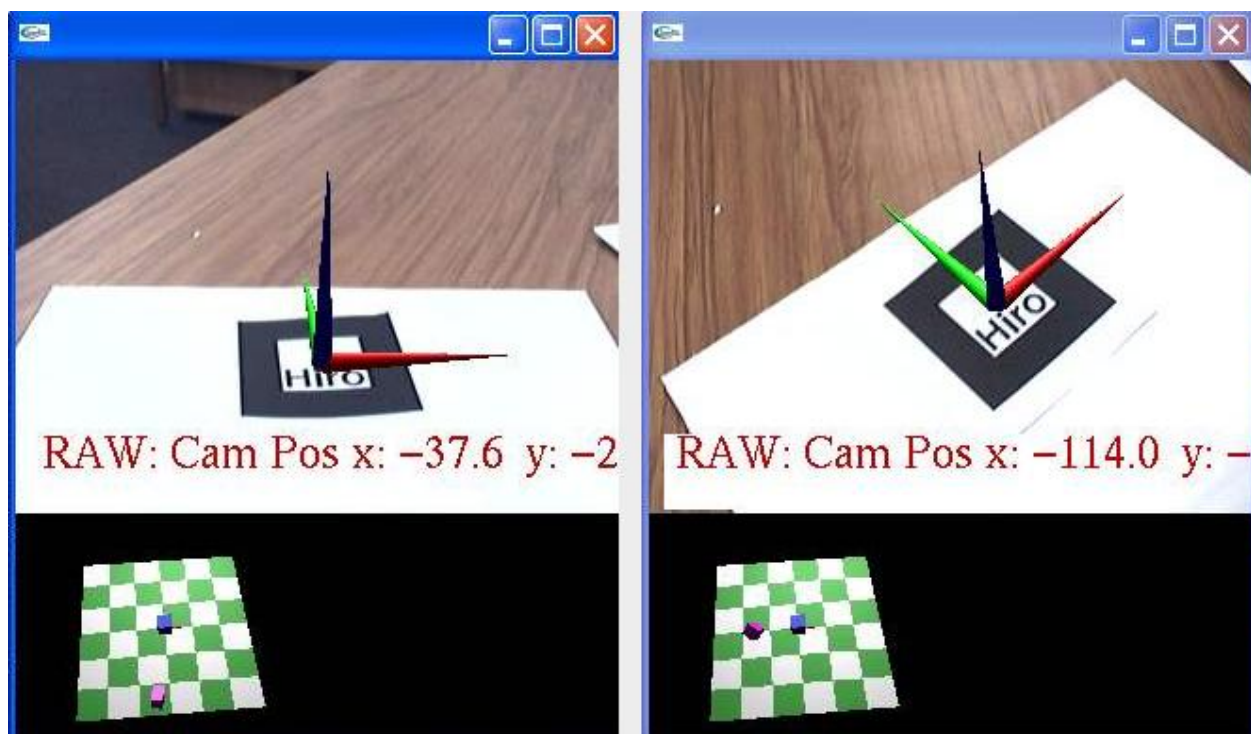


图 4 摄像头的不同位置

打开 examples/exview 下的 exview.c 文件。看程序 getResultRaw:

```
if( arGetTransMat(marker_info, target_center,
```

```

    target_width, target_trans) < 0 ) return;
if( arUtilMatInv(target_trans, cam_trans) < 0 ) return;
sprintf(string, " RAW: Cam Pos x: %3.1f y: %3.1f z: %3.1f",
    cam_trans[0][3], cam_trans[1][3], cam_trans[2][3]);

```

可以看到，同样调用了 `arGetTransMat`，又调用了 `arUtilMatInv`，得到转换后的位置。标识卡和摄像机之间的关系非常重要，它们之间的转换使多个坐标系统的工作成为可能。在这些坐标系统的基础上，同样可以得到任何两个不同的坐标系统之间的转换关系。打开 `examples/relation` 目录下的 `relationtest.c` 文件。可以得到和 `simplem` 一样的多个物体。主要的区别是增加了以下代码：

```

if( object[0].visible >= 0 && object[1].visible >= 0 ) {
    double wmat1[3][4], wmat2[3][4];
    arUtilMatInv(object[0].trans, wmat1);
    arUtilMatMul(wmat1, object[1].trans, wmat2);
    for( j = 0; j < 3; j++ ) {
        for( i = 0; i < 4; i++ ) printf("%8.4f ", wmat2[j][i]);
        printf("\n");
    }
    printf("\n\n");
}

```

这段代码计算了两个标识之间的相对转移矩阵。`object[0].trans` 是标识卡 1 在摄像机坐标系里面的转移矩阵，`object[1].trans` 是标识卡 2 在摄像机坐标系里面的转移矩阵。因此 `object[0].trans` 的转置乘以 `object[1].trans` 就得到了标识卡 2 在标识卡 1 坐标系里面的转移矩阵。运行程序，把标识卡放在如图所示位置上：

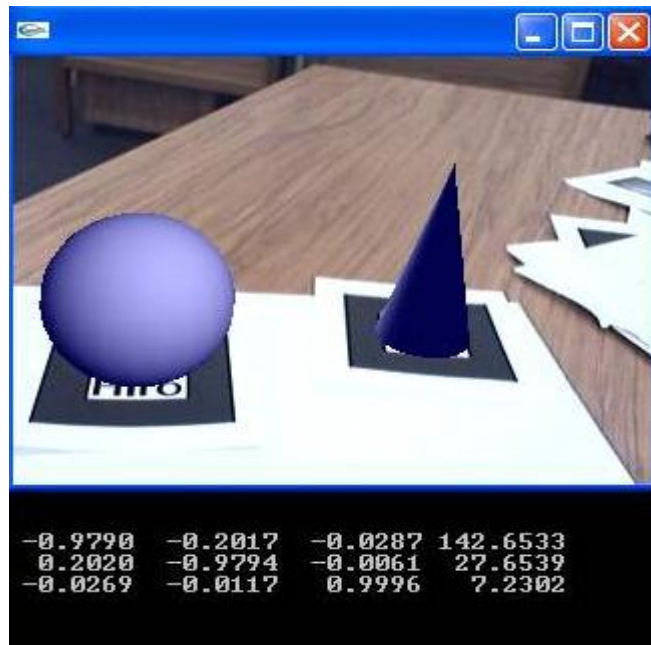


图 5 relationtest（渲染视图和控制台输出）

输出的最后一列是标识卡 2 在标识卡 1 坐标系里面的相对转移矩阵。锥体在球体的左侧（X 轴方向上 142mm 处），在标识卡 1 上（Y 轴方向上 27mm），它们几乎在同一平面上（Z 轴方向上 7mm）。移动标识卡 1 和标识卡 2，观察输出值，理解它们之间的关系。

尝试通过修改 `relationTest` 的代码做相反的操作（标识卡 1 在标识卡 2 坐标系里面的相对转移矩阵）。