Volume Number: 14
Issue Number: 8
Column Tag: Power Graphics

Desktop VR using QuickDraw 3D, Part II

by by Tom Djajadiningrat and Maarten Gribnau, Delft University of Technology
Edited by the MacTech Editorial Staff

Using the Pointing device Manager Implementation of a HeadTracked Display

Summary

Wouldn't it be cool to be able to look around three dimensional objects displayed on your monitor by moving your head, just as if the objects were standing there? Kind of like a hologram, but with the flexibility of 3D computer graphics. Futuristic and expensive? It could be easier and cheaper than you think. In a two part article we explain how to implement such a system, also known as a head-tracked display, on a PowerMacintosh. It provides the user with a sense of depth without the use of stereoscopy. To facilitate implementation we use QuickDraw 3D, Apple's 3D graphics library. In terms of hardware all you need is a PowerMac with QuickDraw 3D, an absolute position-measuring device with three degrees of freedom and, preferably, a QuickDraw 3D accelerator board. This month we discuss the hardware related aspects of the head-tracked display. The graphics related topics were covered in last month's issue.

Introduction

This article is the second and last part of our coverage of a head-tracked display system. The system has two parts: the viewer application, called MacVRoom and a driver. In last month's graphics issue we focused on the viewer application and explained how to control the camera by the user's head position and to show the corresponding perspective on the monitor. This month, we will explain how to write the driver and how to use the Pointing Device Manager to handle the communication between the driver and MacVRoom. We will also discuss a number of calibration methods that will achieve the maximum accuracy. Illustrated in **Figure 1** are the topics covered in both articles. In last month's issue we suggested which topics to read or skip depending on your goals. We also addressed the issue of which knowledge and hardware you require.

Road Map to This Article

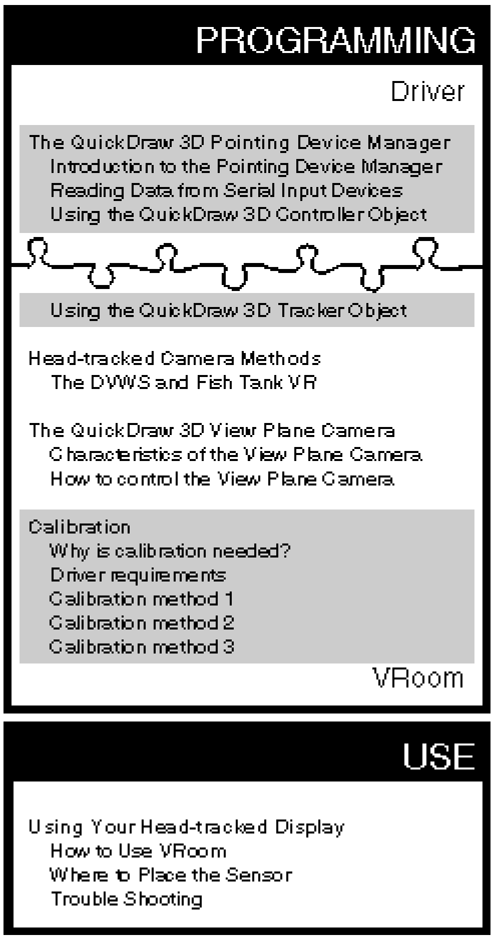
PROGRAMMING
Driver
The QuickDraw 3D Pointing Device Manager
Introduction to the Pointing Device Manager
Reading Data from Serial Input Devices
Using the QuickDraw 3D Controller Object
Using the QuickDraw 3D Tracker Object
Head-tracked Camera Methods
The DVWS and Fish Tank VR
The QuickDraw 3D View Plane Camera
Characteristics of the View Plane Camera
How to control the View Plane Camera
Calibration
Why is calibration needed?
Driver requirements
Calibration method 1
Calibration method 2
Calibration method 3
VRoom
USE
Using Your Head-tracked Display
How to Use VRoom
Where to Place the Sensor
Trouble Shooting

Figure 1. *Overview of the two-part article. The subjects which are covered this month are marked with a grey block.*

The QuickDraw 3D Pointing Device Manager

Introduction to the Pointing Device Manager

The MacVRoom application needs to read positions from a 3D input device to establish the position of the head of the user. We could have chosen to use one primary position sensing device and to include the code for reading the device into MacVRoom, but we would rather support any 3D input device. This is one of the reasons for using QuickDraw 3D's Pointing Device Manager (PDM). It enables you to separate the code for reading device positions from the application itself. Without the PDM driver code would have to be added to the application for each new device to be supported. Using the PDM, only the driver of the device needs to be written and every PDM-savvy application will be able to use it. The next few paragraphs will explain the basic concepts of the PDM.

The PDM was added to the QuickDraw 3D library to provide a means for 3D applications to support 3D pointing devices. Most computer systems come equipped with 2D pointing devices (like a mouse, a trackball or a graphics tablet) that measure positions in two dimensions. 3D pointing devices come in a lot of flavors but they are generally able to sense more than two independent variables. Most operating systems do not provide a way to transport data from 3D pointing devices to 3D applications. The PDM does this, and in addition, it supplies routines that you can use to determine what kind of devices are available and to assign them to tasks in your application.

The PDM defines two types of objects: controllers and trackers. Controller objects are the PDM's abstract representation of 3D pointing devices. They are intended to be created by device drivers. Trackers are created by a 3D application to track positions, orientations and button states of a pointing device. The purpose of this is that controllers and trackers can be in different applications. A driver creates a controller and updates it regularly with fresh device readings, while applications create a tracker, connect it to the controller and extract the device readings from it when needed.

Figure 2 shows the information streams that flow between connected controllers and trackers. The main objective of the controller object is that pointing devices can send their position and orientation and the state of their buttons to the tracker. The figure also shows two other data streams. The controller values are meant to report about optional device dependent information to the application, like the state of switches. Controller channels can be used to communicate data from the application back to the device. Some devices for instance, are equipped with alphanumeric displays that label the switches. The channels can be used to have the driver update these labels. Although controller channels and values may be useful in some applications, their use has the disadvantage that the application will have to contain device dependent code. This eliminates device independence of the application which is a motivation for using the PDM.

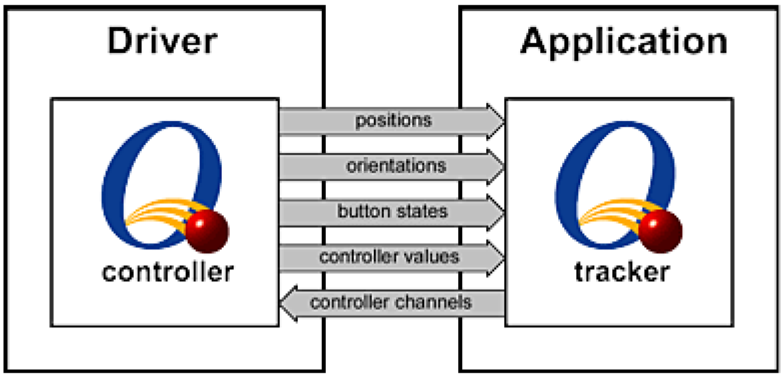

Figure 2. *Information streams between QuickDraw 3D Controller and Tracker objects.*

Positions and orientations can be reported to the tracker in two ways, absolute and relative. This is because there are two kinds of pointing devices. Absolute pointing devices measure positions and/or orientations, relative to the origin of their coordinate system. Relative pointing devices do not have an origin, they report only the changes in position and/or orientation since the last measurement. This is best illustrated by comparing two 2D input devices: a drawing tablet and a mouse. The tablet is an absolute device. When you pick up the stylus and move it to another location, the cursor will move to another location too because the tablet reports the new location to the system. A mouse however, does not report the new location to the system when it is picked up and moved to a new location. While the mouse is in the air the tracking is lost and it can not report changes in position to the system until it is put down again. In this project we need to establish the position of the head of the user relative to the monitor and therefore we need an absolute pointing device.

Using the PDM, you can use several input devices concurrently to control different application tasks. A useful application area of this feature is two-handed input where the simultaneous input of two hands is used (Gribnau and Hennessey, 1998). The PDM also enables you to connect multiple controllers to the same tracker. This means that several input devices can control one application task at the same time. The reverse situation is prohibited; it is not allowed to connect one controller to more than one tracker. In practice, connecting several devices to one task is useful only when, at most, one of those is absolute. When two or more absolute controllers are connected to one tracker they will compete for the current position and/or orientation of the tracker resulting in a jittery appearance. Relative devices do not behave dominantly. The connection of a few relative devices to one tracker will give predictable results. The changes in positions and/or orientations they report to the tracker are added to the current position and/or orientation of the tracker. The behavior of the combination of an absolute device and one or more relative devices is logical too. The offsets reported by the relative device(s) are overruled by the measurements of the absolute devices.

To illustrate the use of the PDM we will now describe how to create a QuickDraw 3D driver for a serial device and how to access it from the MacVRoom application. In the code presented we have removed all the device related parts and all the code concerning the user interface. Readers interested in the omitted code can download the source code of the drivers that we offer on the web site. The first section will deal with the reading of positions from a device. The second section will explain how to use the PDM controller object to report the positions to an application and in the last section you will learn how to use the QuickDraw 3D tracker object in an application to read positions from a QuickDraw 3D device driver.

Reading data from serial input devices

The main objective of the driver is to execute as many readings from the serial port as needed for a smooth operation. As the driver is running in the background we cannot poll the serial port periodically for new data because we are not sure we will get enough processor cycles. Instead of polling we will therefore use the Device Manager PBReadAsync routine to read the serial port. This routine can be used to request a number of bytes from the serial port and it will call a completion routine when they have arrived in the buffer. Here, we would like to extract the positions from the bytes in the buffer but regrettably we can't. Calling routines that move memory will crash the driver and it is not advisable to do lengthy operations that could block other operations. Hence, we have to find a way to process the data at a later time when it is safe to call lengthy routines that move memory around. This is exactly the functionality the Deferred Time Manager offers. You can use its services whenever you need to install a lengthy interrupt task capable of running with all interrupts enabled. In the completion routine we install a deferred task that will process the data and update the controller. At the end of the deferred task we request another reading from the serial port. This closes the circle of a read chain that will read data from the serial port and process it indefinitely. The read chain is illustrated in **Figure 3** and demonstrates the sequence of events.

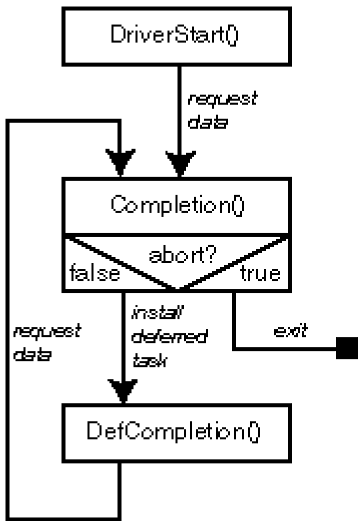

Figure 3. *The driver serial port read chain.*

Listing 1 contains constants, global variables and the code for the DriverStart routine. The routine prepares the global variables and the structures for both the completion routine (fIoParam) and the deferred task (fDefTask). The pointer to our completion routine is put in the ioCompletion field of the IOParam structure. The rest of the fields tell the Device Manager how much data is requested, from which device and were it should be stored. You may notice that the symbol PACK_LEN is not defined in the code. It is the number of bytes that the device needs to send a new reading and therefore different for each device.

To set up the DeferredTask structure we need to fill in only the qType, dtAddr, and dtReserved fields. The dtAddr field specifies the address of the routine whose execution you want to defer, we set it to our deferred completion routine. The dtParam field contains an optional parameter that is loaded into register A1 just before the routine specified by the dtAddr field is executed, we don't use it and clear it. The dtFlags and dtReserved fields of the deferred task record are reserved and the last one should be set to 0. At the end of the routine the serial port buffer is cleared, using SerRecvFlush (not listed), and a request is made for the first package of bytes.

Listing 1: Driver.c

```
DriverStart
#define BUF_LEN         256

// The state of the driver
static   Boolean        fRunning = false;
// The serial input port ID
static short            fSerInp = 0;
// The buffer that holds the bytes read from the port
static Byte             fBuf[BUF_LEN];
// The number of valid data bytes in the buffer
static long             fBufCount = 0;
// The completion routine device parameter block
static IOParam          fIoParam;
// The deferred task manager structure for our deferred completion routine
static DeferredTask     fDefTask;
// The current position of the device
TQ3Point3D              fPos;

OSErr DriverStart()
{
   OSErr err;

   // Set up the serial parameter block
   fIoParam.ioCompletion  = NewIOCompletionProc(Completion);
   fIoParam.ioRefNum      = fSerInp;
   fIoParam.ioPosMode     = fsAtMark;
   fIoParam.ioPosOffset   = 0;
   fIoParam.ioBuffer      = (Ptr) fBuf;
   fIoParam.ioReqCount    = PACK_LEN;

   // Set up the deferred task block
   fDefTask.qType         = dtQType;
   fDefTask.dtAddr        = NewDeferredTaskProc(DefCompletion);
   fDefTask.dtParam       = 0;
   fDefTask.dtReserved    = 0;

   // Flush the serial input port
   err = SerRecvFlush(fSerInp);
   if (err != noErr) goto exit;
   fBufCount = 0;

   // Ask for the first package
   err = PBReadAsync((ParmBlkPtr) &fSerParm);
   if (err != noErr) goto exit;
   fRunning = true;

exit:
   return err;
}
```

Listing 2 contains the Completion routine called when the Device Manager has read the number of bytes requested or when it has encountered an error. The error is passed through the ioResult field of the IOParam structure. If it is equal to abortErr there has been a request to cancel the io operations involving the serial port. In this case we will have to end the read chain by not installing the deferred completion task. On all other errors, we clear the buffer by setting the buffer size fBufCount to zero. It there was no error, fBufCount is updated. At the end of the routine the deferred completion task is installed using the DTInstall call.

Listing 2: Driver.c

```
Completion
```

```
static pascal void Completion(ParmBlkPtr /*paramBlock*/)
{
   if (fIOParm.ioResult == abortErr) {
      // KillIO was called, end the read chain
      return;
   }
   // Update the buffer byte count
   fBufCount += fIOParm.ioActCount;
   if (fIOParm.ioResult != noErr) {
      // Reset the buffer byte count on serial error
      fBufCount = 0;
   }
   // Install the deferred task to process the bytes in the buffer
   DTInstall(&fDefTask);
}
```

In Listing 3, you will find the deferred completion routine. This is the routine that actually processes the bytes read from the device and issues another read. The first thing the routine does is to save the current position of the device. Then it calls the DecodeBuffer routine (not listed) that is specific to the device the driver is created for. It is expected to decode the current position from the bytes in the buffer and to store it in the global variable fPos. During the decode the routine might notice that it is out of sync and needs more bytes to decode a package. The routine should then shift the remaining data bytes to the start of the buffer and reports their number by updating fBufferCount. After decoding the bytes, the controller is updated by calling UpdateController (discussed later), but only if the current position has changed.

The end of the routine establishes the number of bytes to read next and issues a request for them to the Device Manager. Normally the number of bytes requested will be equal to the number of bytes that the device uses to encode a position and/or orientation (defined by the symbol PACK_LEN). If there are bytes left in the buffer, we subtract their number from the number of bytes requested. This should get the decoding algorithm in sync with the device data. The IOParam structure is updated with the requested number of bytes and a pointer to the right position in the data buffer. In case the buffer is overrun, the buffer is reset by setting fBufCount to zero and a request for a new package is made to get in sync again.

Listing 3: Driver.c

```
DefCompletion
static pascal void DefCompletion(long /* dtParam */)
{
   // The number of bytes requested next
   long req;
   // Save the current position
   TQ3Point3D oldPos = fPos;

   // Decode the coordinates in the buffer to fPos, update the number of bytes
   DecodeBuffer(fBuf, &fBufCount);
   if ((fPos.x != oldPos.x) ||
     (fPos.y != oldPos.y) ||
     (fPos.z != oldPos.z)) {
      // The position has changed, update the controller
      UpdateController();
   }
   // Calculate the number of bytes that will be requested next
   req = PACK_LEN - fBufCount;
   if ((req > 0) && (fBufCount < (BUF_LEN - req))) {
      // The number of bytes requested fits in the buffer
      fIOParm.ioBuffer      = ((Ptr) fBuf) + fBufCount;
      fIOParm.ioReqCount    = req;
   }
   else {
      // Error situation, reset the buffer
      fIOParm.ioBuffer      = (Ptr) fBuf;
      fIOParm.ioReqCount    = PACK_LEN;
      fBufCount = 0;
   }
   // Request the next few bytes
   PBReadAsync((ParmBlkPtr) &fSerParm);
}
```

Using the QuickDraw 3D Controller Object

At this point the current position of the device is established. The next step is to create the controller object that can be polled by trackers. Listing 4 illustrates how to create the controller. The routine sets the fields of the PDM controller data structure and uses the Q3Controller_New routine to have QuickDraw 3D create the controller object. The signature field of the controller structure is a C-type string that applications can use to distinguish between controllers. As we choose not to support any controller values or channels the other fields are set to nil.

Listing 4: Driver.c

```
CreateController
static TQ3ControllerObject CreateController()
{
   TQ3ControllerData controllerData;

   controllerData.signature          = "Acme Driver";
   controllerData.valueCount         = 0;
   controllerData.channelCount       = 0;
   controllerData.channelGetMethod   = NULL ;
   controllerData.channelSetMethod   = NULL ;
   return Q3Controller_New(&controllerData);
}
```

The controller object is subsequently used by the UpdateController routine in Listing 5. The routine checks whether the controller is currently affecting the 2D system cursor. As you remember from the introduction to the PDM, trackers can be connected to controllers. If a controller is not connected to a tracker, the PDM will connect it to the 2D system cursor. This is nice for relative pointing devices, because you can use your 3D pointing device to operate your Mac. For absolute devices however, this is not so convenient. Depending on the coordinates the driver generates, it may park the cursor in a corner of the screen and make your Mac inoperable. The driver therefore uses Q3Controller_Track2DCursor to check whether the controller is connected to the system cursor. The PDM sets track2DCursor to kQ3False when the controller is not connected to the cursor, only then the current position is entered into the controller object. Devices that measure orientations and/or button states can pass their data in the same way.

Listing 5: Driver.c

```
UpdateController
static void UpdateController()
{
   TQ3Boolean track2DCursor;
   (* Check whether the controller is connected to the system cursor *)
   Q3Controller_Track2DCursor(fController, &track2DCursor);
   if (track2DCursor == kQ3False) {
      Q3Controller_SetTrackerPosition(fController, &fPos);
   }
}
```

Using the QuickDraw 3D Tracker Object

Our MacVRoom application needs to read the positions from a 3D pointing device to update the camera. It accomplishes this by creating a PDM tracker that can connect to controllers on the system. The question is, how does it find a controller and to which controller does it connect? The PDM provides a way to search for all the controllers present. Using this functionality the application's tracker can be connected to all the controllers active on the system. When connecting multiple controllers to one tracker, you might encounter the strange behavior that was discussed in the introduction to the PDM. Listing 6 presents the CreateTracker routine that implements this functionality.

First it creates a tracker. You may wonder why it passes NULL to the Q3Tracker_New routine. This is because there are two ways to find out if the position of a tracker has changed. If you provide a pointer to a notification routine in the Q3Tracker_New call, the routine will be executed by the PDM every time the position changes. The notification threshold can be changed, so you can set the distance the device will have to move before the notification routine is called. Our application does not use a notification routine and notifies the PDM of this fact by passing NULL. It polls the tracker instead to see whether the position of the tracker has changed.

Once the tracker is created, the CreateTracker routine connects it to the controllers it finds. The PDM offers the Q3Controller_Next call to loop through the controllers. The first parameter must be set to NULL to find the first controller. A reference to the controller is returned in the second parameter. This can be fed again into the first parameter to find the next controller. If the PDM can not find a controller the reference parameter returned is set to NULL. This is the sign that the end of the list is reached. If a controller is found, then it is connected to the tracker and the boolean variable foundController is updated. The variable is used at the end of the routine to check whether there were controllers present on the system. If no controllers were found, the routine disposes of the tracker and returns NULL. This way, MacVRoom knows there is no controller available and it will enable mouse control. If a controller was found, MacVRoom can use the tracker to extract the device coordinates. This will be discussed later when the adjustment of the camera parameters is covered.

Listing 6: (De)Init.c

```
CreateTracker()
TQ3TrackerObject CreateTracker()
{
   TQ3TrackerObject   tracker;
   TQ3Status            status;
   TQ3ControllerRef   ref;
   TQ3Boolean             foundController = kQ3False;

   tracker = Q3Tracker_New(NULL);
   status = Q3Controller_Next(NULL, &ref);
   while ((ref != NULL) && (status == kQ3Success)) {
      Q3Controller_SetTracker(ref, tracker);
      foundController = kQ3True;
      status = Q3Controller_Next(ref, &ref);
   }
   if (foundController == kQ3False) {
      Q3Object_Dispose(tracker);
      tracker = NULL;
   }
   return tracker;
}
```

Concluding our coverage of the PDM a warning is issued to readers that are planning to use our code in their own applications. In theory the PDM allows you to dynamically assign controllers to trackers. In practice however, you will probably encounter a bug when you try to connect a tracker to a controller a few times. If you are running with the QuickDraw 3D GM extensions, your application will crash. At the moment the only solution is to connect trackers to controllers once. Hopefully the QuickDraw 3D team will solve this problem in their next release.

Calibration

Why is calibration needed?

Calibration would be simple if the sensor could be put in the middle of the eye, the origin of the tracker base unit could be put in the middle of the pane, and the driver would report the position of the sensor in inches. All that would be required to find the position of the camera in world coordinates, would be to divide the sensor position by the width of the rendering pane in inches. Unfortunately, things are not that simple as the sensor cannot be mounted in the eye, and the tracker cannot be placed in the monitor.

For a movement parallax system based on the fishtank projection method calibration is quite important. If we get it wrong the virtual camera position will not correspond to the user's head position and as a result he will see a distorted image. One of the problems is that we do not know the dot pitch of your monitor and the current resolution. Even though we know the size of the rendering pane in pixels we do not know what its size will be in inches on your monitor.

Driver requirements

To simplify matters we will make three assumptions about our driver (the drivers which accompany this article conform to these assumptions. If you are writing a driver yourself, please make sure that it does so too). The first one is that it results in a right handed coordinate system with the Y-axis pointing upwards and the Z-axis pointing towards the user (see for example Figure 6). The second one is that the tracker is isotropic in X, Y and Z or that is has been made to appear isotropic to MacVRoom by appropriate scaling in the driver. For example, if the sensor is moved over a certain distance along the X-axis, the change in X-coordinate should be the same as the change in Y-coordinate when the sensor is moved over the same distance along the Y-axis. The third assumption is that the driver has scaled the tracker coordinates to inches. This means that if we move the sensor one inch along one of the axes the coordinate of that axis will change by one.

There are a number of methods to do the calibration. We will discuss three methods in order of increasing complexity and accuracy. In the first method, the set-up is measured with a ruler and calibration is performed once only at compilation time. In the second method, the user is required to hold the sensor in two locations, expressed in terms of the pane width, when the app is launched. In the third method, the user is required to hold the sensor in three locations of which the exact position relative to the pane is known. The three methods are summarized in Table 1. You may wish to refer to the decision chart (**Figure 4**) from time to time while you read the description of the three calibration methods. No matter what calibration method you choose, you should, of course, not burden the user with calibration every time MacVRoom is run. Although MacVRoom can neither save the calibration settings and nor recalibrate without quitting first, a real application should offer this functionality.

	source code supplied	*measurements*	*tracker & pane coord. sys. must share orientation?*	*accuracy*	*disadvantages*
method 1	no	• OtCp • pane width	no	high	tracker origin required
method 2	yes	none	no	• low without cardboard template • high with cardboard template	• low accuracy w/o template • high accuracy requires cardboard template

method 3	yes	PoCp	yes	high	cardboard template

Table 1. *Characteristics of the three calibration methods.*

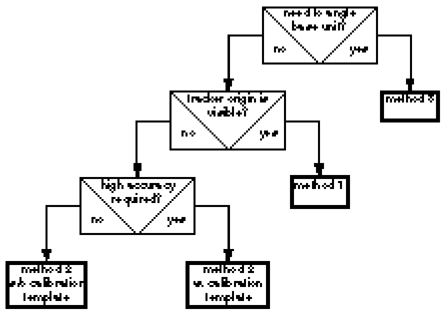
need to angle base unit?
no
yes
tracker origin is visible?
no
yes
method 3
high accuracy required?
no
yes
method 1
method 2 w/o calibration template
method 2 w. calibration template

Figure 4. *Decision flow chart for the three calibration methods.*

Calibration Method 1

In the first method we measure the distance in inches from the tracker origin to the pane center (OtCp) along the coordinate axes. We also measure the width of the pane in inches. From the driver we get the position of the sensor relative to the tracker origin in inches (OtP). The position of the sensor relative to the pane center (CpP) can now be found by CpP = OtP-OtCp (**Figure 5**). We have made life easier by making the image which the View Plane Camera takes out of the view plane one QuickDraw 3D unit wide. Since this image takes up the full width of the rendering pane, and the driver provides measurements in inches, we can find the position of the virtual camera in virtual world coordinates by scaling the position of the sensor relative to the pane center by dividing it by the pane width in inches. Thus the top-left corner of the pane in virtual world coordinates is {-0.5, 0.71, 0} and the bottom-right corner is {0.5, -0.71, 0}.

This calibration method is the easiest but has two disadvantages. The first is that the tracker origin needs to be known. With some trackers, for example the DynaSight, this is indeed the case as the origin is indicated on the base unit. However, with others, for example the FreeD, we do not know where the origin is located. The second disadvantage is that the tracker coordinate system needs to be aligned to the pane coordinate system. The base unit of the tracker thus needs to be accurately aligned with the monitor screen. Though theoretically it is possible to take the orientation of the tracker into account, in practice it is difficult to accurately measure its orientation with a protractor. For these reasons we have not implemented this method.

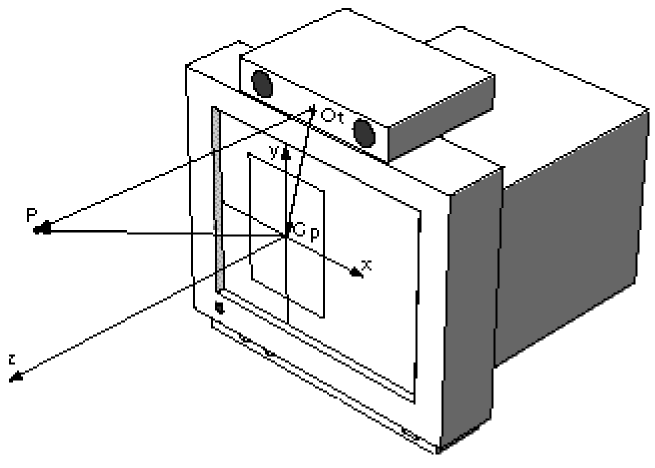

Figure 5. *In calibration method 1 the vector from the center of the pane (Cp) to the sensor (P) is found by CpP = OtP-OtCp, in which Ot is the origin of the tracker coordinate system. Note that the center of the pane need not coincide with the center of the screen.*

Calibration Method 2

The second method is the one which MacVRoom uses by default. The user is asked to keep the sensor twice the width of the pane in front of the top-left corner of the pane (**Figure 6**), press return and repeat the process for the bottom-right corner. Listing 7 shows how from these two 3D points the transformation matrix can be calculated. First we find the pane center relative to the tracker origin and the pane width in tracker coordinates. From these we can find the translation and scaling matrices, which are then combined into a composite calibration matrix. Note how we're not bothered anymore about the location of the tracker origin. Therefore you can simply hook up a different tracker, restart MacVRoom and calibrate the system for the new tracker. You can also switch screen resolution and work with a different size pane as this method takes pane width into account. This method also has some disadvantages. As with the first method the tracker and pane coordinate system need to have the same orientation. A second disadvantage is that it is difficult to keep the sensor perfectly in front of the window by twice the pane width, though this can be overcome with the help of a cardboard template. Still, we are providing you with this method because it gets you up and running quickly as - in its basic form - it does not require you to build any cardboard templates.

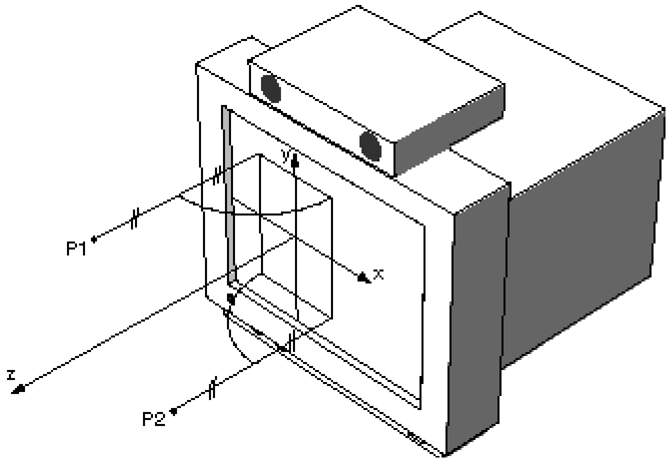

Figure 6. *The first calibration point (P1) is twice the pane width in front of the top left corner of the pane. The second calibration point (P2) is twice the pane width in front of the bottom right corner of the pane.*

Listing 7: Calibration.c

```
CalibrateTracker2
// From the two 3D points we can get the necessary info to do the calibration.

// Average the Z-coordinates of the two calibration points
calPoint1.z = calPoint2.z = (calPoint1.z+calPoint2.z)/2;

// Calculate width and height of window in raw tracker units
calWidth = calPoint2.x - calPoint1.x ;
calHeight = calPoint2.y - calPoint1.y ;

// Find matrix for sensor to eye center offset
Q3Matrix4x4_SetTranslate(&transSensorMatrix, 0, -kDy, 0);

// Calculate matrix to translate pane center to tracker origin
Q3Matrix4x4_SetTranslate(   &transCpOtMatrix,
                                    -(calPoint1.x + calWidth/2.0),
                                    -(calPoint1.y + calHeight/2.0),
                                    -calPoint1.z + 2*calWidth);

// Calculate scale matrix to scale to width = 1
Q3Matrix4x4_SetScale(    &scaleMatrix,
                              1/calWidth,
                              1/calWidth,
                              1/calWidth);

// Calculate the calibration matrix:
// Translate -OtCp, translate for sensor offset and scale.
Q3Matrix4x4_Multiply(   &transSensorMatrix,
                              &transCpOtMatrix,
                              &theDocument->fCalMatrix);

Q3Matrix4x4_Multiply(   &theDocument->fCalMatrix,
                              &scaleMatrix,
                              &theDocument->fCalMatrix);
```

Calibration Method 3

The main advantage of the third calibration method is that the orientation of the tracker coordinate system and the screen coordinate system need not be the same. This means that you have the freedom to place the tracker so that it looses track of the user as little as possible. For example, the volume wherein the Dynasight base unit can track its sensor is shaped like a cone, which emanates from the tracker base unit. To make sure the user stays within this cone, you may wish to reposition and reorient the tracker base unit. For example, in **Figure 7** the tracker base unit has been put under an angle.

To activate the third method change the conditional compiler statement "#define CALIBRATIONMETHOD3 0" to "#define CALIBRATIONMETHOD3 1" in MyDefines.h. This method does require some handiwork. You need to make a cardboard template on which to put the monitor. On the cardboard template mark out three points (P0, P1, P2) forming a right angle as shown in **Figure 7**. P0P1 must be parallel to the screen coordinate system's X-axis, and the P0P2 must be parallel to the screen coordinate system's Y-axis. To specify your set-up you will need to enter the vector from P0 to the center of the pane (P0Cp) and the width of the pane in inches in CalibrateTracker3. On start up you need to hold the tracker in the three positions and press return after each one. Now we have these three points in the tracker coordinate system. From the three points we can calculate unit vectors in the x and z directions of the screen coordinate system (Listing 8). A unit vector in the Y direction of the screen coordinate system is found by taking the cross-product of the unit vectors in the x and z direction. We can find the translation vector from the tracker origin to the center of the pane (OtCp) by adding the vector P0Cp expressed in the tracker coordinate system to P0. The orthonormal vectors (i.e. unit vectors which are perpendicular to each other) expressed in the tracker coordinate system need to rotate into the unit axes of the screen coordinate system. They therefore form the first three columns in the rotation matrix (If you are reading up on this, remember that many books on computer graphics, for example the highly recommended Foley et al. (1995), use post-multiplication by column vectors, while QuickDraw 3D uses pre-multiplication by row vectors. To go from one convention to the other you need to transpose the matrices and reverse their order: [AB]t = BtAt). Scaling can be done by dividing by the pane width in inches, which will give us a virtual world coordinate system in 'pane width units'.

As long as the pane is not moved and the monitor is not moved with respect to the cardboard template, there is no need to change the statements defining the vector P0Cp and the width of the pane. Re-enter the three calibration points and the set-up is calibrated again.

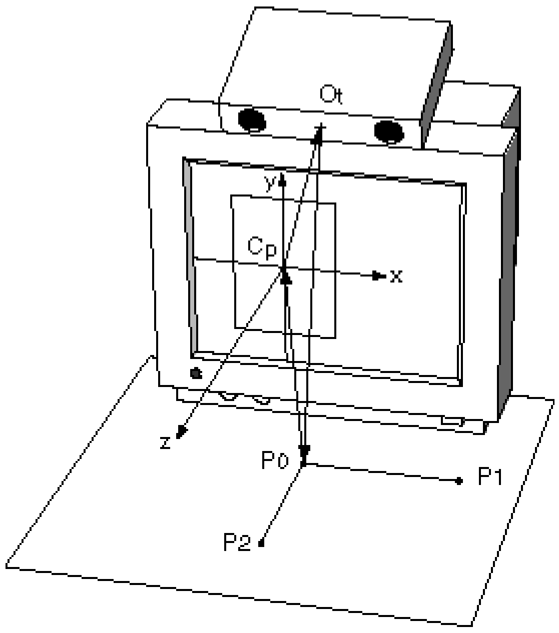

Figure 7. *For calibration method 3 a cardboard template with three calibration points (P0, P1, P2) is put underneath the monitor.*

Listing 8: Calibration.c

```
CalibrateTracker3
//parallel to X
Q3Point3D_Subtract(&P1, &P0, &Xraw) ;

//X normalized
Q3Vector3D_Normalize(&Xraw, &Xnorm) ;

//parallel to Z
Q3Point3D_Subtract(&P2, &P0, &Zraw) ;

//Z normalized
Q3Vector3D_Normalize(&Zraw, &Znorm) ;

//find the y vector which is perpendicular to x and z
Q3Vector3D_Cross(&Znorm, &Xnorm, &Ynorm) ;

// find OtCp in the tracker coordinate system.
Q3Vector3D_Scale(&Xnorm, P0Cp.x, &tempVector);
Q3Point3D_Vector3D_Add(   &P0,
                                 &tempVector,
                                 &tempPoint) ;

Q3Vector3D_Scale(&Ynorm, P0Cp.y, &tempVector);
Q3Point3D_Vector3D_Add(   &tempPoint,
                                 &tempVector,
                                 &tempPoint) ;

Q3Vector3D_Scale(&Znorm, P0Cp.z, &tempVector) ;
Q3Point3D_Vector3D_Add(   &tempPoint,
                                 &tempVector,
                                 &OtCp) ;

// find matrix for sensor to eye center offset
Q3Matrix4x4_SetTranslate(&transSensorMatrix, 0, -kDy, 0) ;

// find matrix for center pane to tracker origin vector
Q3Matrix4x4_SetTranslate(&transOtCpMatrix,
                                    -OtCp.x, -OtCp.y, -OtCp.z) ;

// Set up the rotation matrix
// Make sure the rotation matrix is set to the identity matrix
// before we start modifying it.
Q3Matrix4x4_SetIdentity(&rotMatrix) ;

// each column vector of the rotation matrix
// is rotated by the rotation matrix to lie
// on positive x, y and z axes.
rotMatrix.value[0][0] = Xnorm.x;
rotMatrix.value[0][1] = Ynorm.x;
rotMatrix.value[0][2] = Znorm.x;
rotMatrix.value[1][0] = Xnorm.y;
rotMatrix.value[1][1] = Ynorm.y;
rotMatrix.value[1][2] = Znorm.y;
rotMatrix.value[2][0] = Xnorm.z;
rotMatrix.value[2][1] = Ynorm.z;
rotMatrix.value[2][2] = Znorm.z;

// find scale matrix to scale inches to 'pane width units'
scale = 1/paneWidthInches;
Q3Matrix4x4_SetScale(&scaleMatrix, scale, scale, scale);

// translate -OtCp, rotate, translate for sensor offset, and finally scale
Q3Matrix4x4_Multiply(   &transOtCpMatrix,
                              &rotMatrix,
                              &theDocument->fCalMatrix);

Q3Matrix4x4_Multiply(   &theDocument->fCalMatrix,
                              &transSensorMatrix,
                              &theDocument->fCalMatrix);

Q3Matrix4x4_Multiply(&theDocument->fCalMatrix,
                              &scaleMatrix,
                              &theDocument->fCalMatrix);
```

Conclusions

In this month's Part II we covered the hardware related aspects of a head-tracked display on PowerMacintosh using QuickDraw 3D. We showed you how to use the Pointing Device Manager to handle communication between a viewer application and a three degrees of freedom position tracker. The Pointing Device Manager has the potential to enrich 3D interaction through input devices other than keyboard, mouse or trackball. Also, we covered some calibration methods that can be used for accurate operation of the head-tracked display system. In the two articles we hope to have shown you how the QuickDraw 3D API greatly facilitates the implementation of virtual reality systems "for the rest of us". What are you waiting for?

Acknowledgements

We gratefully acknowledge P.J. Stappers, J.M. Hennessey, C.J. Overbeeke and A. van der Helm, for their constructive criticism during the preparation of this article.

Bibliography and References

- Apple Computer, Inc. (1992). Inside Macintosh: Processes. Reading, MA: Addison-Wesley Publishing Company.
- Apple Computer, Inc. (1994). Inside Macintosh: Devices. Reading, MA: Addison-Wesley Publishing Company.
- Ascension Technology Corporation, P.O. Box 527, Burlington, VT 05402, USA, tel. (802)860-6440 or (800)321-6596 (inside the USA only), fax. (802)860-6439, email productsales@ascension-tech.com, http://www.ascension-tech.com.

- Foley, J.D., Van Dam, A., Feiner, S.K. and Hughes, J.F. (1996). Computer Graphics Principles and Practice, Second Edition in C. Reading, MA: Addison-Wesley Publishing Company.
- Gribnau, M.W. and Hennessey, J.M. (1998). Comparing one- and two-handed input in a 3D assembly task. Proceedings of CHI'98, accepted.
- Origin Instruments Corporation, 854 Greenview Drive, Grand Prairie, TX 75050, USA, tel. 972-606-8740, fax. 972-606-8741, email sales@orin.com, http://www.orin.com.
- Owl by Pegasus Technologies Ltd., Merkazim 2000, 5 Hazoref st., Holon 58856, Israel, tel.: +972-(3)-5500633, fax: +972-(3)-5500727, email pegasus@pegatech.com, http://www.pegatech.com.
- Quickdraw 3D home page, http://www.apple.com/quicktime/qd3d/.

Tom Djajadiningrat (J.P.Djajadiningrat@io.TUDelft.nl) is an industrial designer interested in products and computers which are intuitive in use. When not trying to convince others that he will now finish his PhD thesis on interfaces using head-tracked displays 'really soon', or hassling Maarten and the QuickDraw 3D mailing list with silly programming questions, he dreams of designing the 25th anniversary Macintosh.

Maarten Gribnau (M.W.Gribnau@io.TUDelft.nl) is an electrical engineer interested in Computer Graphics and Interaction Design. He has successfully delayed Tom's research project so they can now both convince others that they will complete their PhD theses 'really soon'. Apart from his research on two-handed interfaces for 3D modeling applications, he occasionally drinks a strong cup of Java when he is trying to program the ultimate internet golf game.