zSpace Developer

Native Programming Guide

Version 1.0

Before You Begin

This *zSpace Developer Native Programming Guide* describes how to program features using the zSpace™ software development kit (SDK).

Audience

This document is intended for developers who have built 3D applications and have a basic understanding of the principles associated with 3D programming for zSpace applications.

Scope

This document covers basic installation and then explores the SDK components and features.

Document Organization

This document is divided into five main sections:

- Buffer Allocation and Stereo Rendering
- Initializing and Integrating
- Frustum Attributes
- Coordinate Spaces
- Tracking

Related Documents

For more information about developing zSpace applications, refer to the following sources:

- zSpace developer resources at http://developer.zspace.com/
- zSpace developer documents at http://developer.zspace.com/docs/ specifically:
    - The *zSpace Developer Guide – SDK Introduction* provides an overview of how to build and port applications for the zSpace™ platform.
    - The *zSpace Developer Unity 3D Programming Guide* describes how to program features using the zSpace plugin for Unity 3D.

Contents

1: Introduction and Installation

Building or porting an application to the zSpace platform has two distinct parts. First, the application must be stereo enabled and use the zSpace stereo values to generate correct head tracked images. Second, the application needs to get stylus information into a coordinate system. After those two essentials are established, other aspects of the zSpace platform can be developed to create unique experiences, including viewer scaling, stylus vibration, and mouse emulation.

Setup Basics and Directories

When you download and install the zSpace SDK from http://developer.zspace.com/downloads, a number of directories are installed in **C:/zSpace/SDKs/[build#]/**:

- The **Inc** directory contains the header files used by the zSpace system.
- The **Lib** directory contains the libraries needed for the supported architecture.
- The **Media** directory contains the default zSpace icon.
- The **Samples** directory contains the sample programs that show how to use various features of the SDK. These samples are used throughout this document, as code references and complete samples, which demonstrate how to use zSpace features.

The zSpace SDK currently supports programming in C. Additional language bindings for the zSpace SDK are planned for the future, along with full documentation.

The zSpace **Samples** directory includes a text file called **CMakeLists.txt**. *Cmake* is the tool used to create Visual Studio project and solution files for the zSpace samples. Visual Studio 2008 or a newer version is required to build zSpace applications. For convenience, the **Samples/Prebuilt** directory contains executables for all the samples. Developers can execute these prebuilt samples rather than building them from Visual Studio.

2: Buffer Allocation and Stereo Rendering

Buffer Allocation

The zSpace stereo uses a quad buffer time sequential stereo mechanism. This mode of stereo requires coordination with the GPU to allocate and render into appropriate buffers. This section describes how to allocate the buffers for *OpenGL* and *DirectX*, and how to modify the normal rendering loop to render into stereo buffers. OpenGL is a cross-language, cross-platform API for rendering 2D and 3D graphics. DirectX is an API for creating and managing graphic images and multimedia effects.

OpenGL

The method for stereo buffer allocation in OpenGL is simple. On Windows, OpenGL applications allocate driver resources by using the *SetPixelFormat* Windows function. Existing OpenGL applications already call this function to allocate resources for the back buffer, stencil buffer, and depth buffer. To allocate the stereo buffers, simply add the **PFD_STEREO** flag to the **PIXELFORMATDESCRIPTOR** flags field.

For example, if the current **PIXELFORMATDESCRIPTOR** declaration looks like this.

```
PIXELFORMATDESCRIPTOR pfd =
{
    sizeof(PIXELFORMATDESCRIPTOR),  // Size Of This Pixel Format Descriptor
    1,                          // Version Number
    PFD_DRAW_TO_WINDOW |        // Format Must Support Windows
    PFD_SUPPORT_OPENGL |        // Format Must Support OpenGL
    PFD_DOUBLEBUFFER,           // Must Support Double Buffering
    PFD_TYPE_RGBA,              // Request an RGBA Format
    24,                         // 24-bit color depth
    0, 0, 0, 0, 0, 0,           // Color Bits Ignored
    0,                          // No Alpha Buffer
    0,                          // Shift Bit Ignored
    0,                          // No Accumulation Buffer
    0, 0, 0, 0,                 // Accumulation Bits Ignored
    32,                         // 32-bit Z-Buffer (Depth Buffer)
    0,                          // No Stencil Buffer
    0,                          // No Auxiliary Buffer
    PFD_MAIN_PLANE,             // Main Drawing Layer
    0,                          // Reserved
    0, 0, 0                     // Layer Masks Ignored
};
```

The new declaration to allocate quad-buffer stereo looks like this.

```
PIXELFORMATDESCRIPTOR pfd =
{
    sizeof(PIXELFORMATDESCRIPTOR),    // Size Of This Pixel Format Descriptor
    1,                         // Version Number
    PFD_DRAW_TO_WINDOW |       // Format Must Support Windows
    PFD_SUPPORT_OPENGL |       // Format Must Support OpenGL
    PFD_STEREO |               // Format Must Support Quad-buffer Stereo
    PFD_DOUBLEBUFFER,          // Must Support Double Buffering
    PFD_TYPE_RGBA,             // Request an RGBA Format
    24,                        // 24-bit color depth
    0, 0, 0, 0, 0, 0,          // Color Bits Ignored
    0,                         // No Alpha Buffer
    0,                         // Shift Bit Ignored
    0,                         // No Accumulation Buffer
    0, 0, 0, 0,                // Accumulation Bits Ignored
    32,                        // 32-bit Z-Buffer (Depth Buffer)
    0,                         // No Stencil Buffer
    0,                         // No Auxiliary Buffer
    PFD_MAIN_PLANE,            // Main Drawing Layer
    0,                         // Reserved
    0, 0, 0                    // Layer Masks Ignored
};
```

That is all you need to do to let the driver know that quad buffered stereo is requested.

DirectX

DirectX recently provided a vendor neutral API for quad buffered stereo support in version 11.1. It is possible to set up quad buffered stereo in DirectX 9, 10, and 11, but this requires functions specific to AMD and nVidia. Please refer to the **BasicStereoD3DSample** for details about setting up quad buffered stereo on AMD and nVidia for DirectX 9, 10, and 11.0. The following description applies to DirectX 11.1.

To initialize DirectX, you must create the device and swap chain. There are several steps in this process, and some changes are needed to ensure quad buffered stereo support. The **BasicStereoD3D11_1Sample** shows all of the steps outlined here.

First, make sure stereo is supported by checking the **IsWindowedStereoEnabled()** function on the factory interface.

```
// Create the IDXGIFactory2.
if (FAILED(CreateDXGIFactory1(__uuidof(IDXGIFactory2), (void**)&g_factory)))
{
    return false;
}

// Check if stereo is enabled.
if (!g_factory->IsWindowedStereoEnabled())
{
    return false;
}
```

The device created with the adapter from this factory supports quad buffered stereo.

Next, change the swap chain. In the swap chain descriptor, the Stereo flag needs to be set to **TRUE**. Also, the **SwapEffect** needs to be set to **DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL**.

```
// Set up the swap chain descriptor.
DXGI_SWAP_CHAIN_DESC1 swapChainDesc  = { 0 };
swapChainDesc.Width                  = 0;      // Use automatic sizing
swapChainDesc.Height                 = 0;      // Use automatic sizing
swapChainDesc.Format                 = DXGI_FORMAT_B8G8R8A8_UNORM;
swapChainDesc.Stereo                 = TRUE;
swapChainDesc.SampleDesc.Count       = 1;
swapChainDesc.SampleDesc.Quality     = 0;
swapChainDesc.BufferUsage            = DXGI_USAGE_RENDER_TARGET_OUTPUT;
swapChainDesc.BufferCount            = 2;
swapChainDesc.Scaling                = DXGI_SCALING_NONE;
swapChainDesc.SwapEffect             = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
swapChainDesc.Flags                  = 0;
```

That completes setup for quad buffered stereo allocation.

Stereo Rendering

This section uses OpenGL to show how to render in stereo. DirectX has very similar mechanisms, but uses different names with a slightly different syntax. For example, OpenGL uses draw buffers to indicate left or right buffer rendering. DirectX uses **RenderTargetView**.

A monoscopic application would use a single **RenderTargetView** for the back buffer. Quad buffered stereo rendering requires two **RenderTargetView** instances– one for each buffer. See the **BasicStereoD3D11_1Sample** to see how this is implemented in a DirectX application.

Render Loop

Most 3D applications start by setting up all of their rendering resources, and then move into a loop, which continuously renders the scene until the application exits. For monoscopic OpenGL applications, a simple render loop might look like this.

```
bool done = FALSE;
while (done == FALSE)
{
    // Update application state
    update();

    // Set the application window's rendering context as the current rendering context.
    wglMakeCurrent(g_hDC, g_hRC);

    // Clear the scene - color and depth buffers.
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Draw the scene.
    draw();

    // Wwap the back buffer to the front.
    SwapBuffers(g_hDC);
}
```

For each iteration of the loop the code updates the application state, makes the OpenGL context current, clears the back buffer, draws the scene, and swaps the back buffer to the front. This continues until the application exits.

Stereo Render Loop

For stereo rendering, the scene needs to be rendered twice, once for the left eye and once for the right eye. The following code shows how to modify the monoscopic rendering loop to support stereo.

There are a couple of things to note in this code. First, the scene is drawn twice, and the draw buffer is set to the appropriate value depending on which eye is being rendered. Second, the view and projection matrix values are different for each eye, so they need to be recalculated per frame for each eye.

```
bool done = FALSE;
while (done == FALSE)
{
    // Update application state
    update();

    // Set the application window's rendering context as the current rendering context.
    wglMakeCurrent(g_hDC, g_hRC);

    // Clear the scene - color and depth buffers.
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    // Draw the scene for each eye.
    drawSceneForEye(ZC_EYE_LEFT);
    drawSceneForEye(ZC_EYE_RIGHT);

    // Wwap the back buffer to the front.
    SwapBuffers(g_hDC);
}

void drawSceneForEye(ZCEye eye)
{
    // Set the view and projection matrices for the specified eye.
    computeViewMatrix(eye);
    setProjectionMatrix(eye);

    // Set the draw buffer based for the specified eye.
    if (eye == ZC_EYE_LEFT)
    {
        glDrawBuffer(GL_BACK_LEFT);
    }
    else
    {
        glDrawBuffer(GL_BACK_RIGHT);
    }

    // Clear the scene - color and depth buffers.
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Draw a cube.
    drawCube();
}
```

The following section presents the matrices' values and what these values represent.

3: Initializing and Integrating

Once the stereo rendering loop has been implemented, you need to integrate the zSpace SDK. This section discusses initializing and integrating zSpace SDK data into the stereo rendering loop.

zSpace Context

The zSpace runtime associates all of its internal data with a zSpace context. When you initialize the zSpace runtime system, a zSpace context is returned.

```
zcInitialize(ZCContext* context);
```

Many of the objects created and used by the zSpace runtime are associated with the zSpace context. These objects are represented with a **ZCHandle**, and the context may be retrieved from the handle with the following function.

```
zcGetContext(ZCHandle handle, ZCContext* context);
```

Version and Errors

The zSpace SDK has an API that gets the current version of the zSpace runtime. This API takes the zSpace context as a parameter and returns the major, minor, and patch numbers for the current runtime.

```
ZCError zcGetRuntimeVersion(ZCContext context, ZSInt32* major, ZSInt32* minor, ZSInt32* patch);
```

All zSpace SDK APIs return an error code value. For brevity, this error return value is omitted from descriptions in this document. There are many possible errors that can occur when using the SDK. The error returned by a function can vary based on the specific function. If there is no error, the returned value is **ZC_ERROR_OK**. If an error does occur, a string description of the error can be retrieved from the following function.

```
zcGetErrorString(ZCError error, char* buffer, ZSInt32 bufferSize);
```

The parameters for this function are the error, a pointer to a character buffer, and the size of the buffer.

Display

Once the context has been created, a zSpace application should find the correct zSpace Display object. There are several attributes that can be retrieved once you have the Display object. There may be several displays attached to the system, so there are several ways to get the information for each display. When zSpace runtime is initialized, the displays are discovered, but the application can rediscover this information using the following function.

```
zcRefreshDisplays(ZCContext context);
```

zSpace runtime can find three types of displays:

- **ZC_DISPLAY_TYPE_ZSPACE**
- **ZC_DISPLAY_TYPE_GENERIC**
- **ZC_DISPLAY_TYPE_UNKNOWN**

All display information can be retrieved for zSpace type displays, but other types of displays may not return all of the information defined in the zSpace SDK. Several types allow you to iterate over the discovered displays. The following functions return the number of displays discovered; the second version only returns the number of displays of the specified type.

```
zcGetNumDisplays(ZCContext context, ZSInt32* numDisplays);
zcGetNumDisplaysByType(ZCContext context, ZCDisplayType displayType, ZSInt32* numDisplays);
```

The following function gets the display at the position specified; this position is specified in pixel coordinates in the virtual desktop.

```
zcGetDisplay(ZCContext context, ZSInt32 x, ZSInt32 y, ZCHandle* displayHandle);
```

These functions get the display at the given index. Note that the index specified in the first function is relative to all displays. The index in the second version of the function is only relative to that type of display.

```
zcGetDisplayByIndex(ZCContext context, ZSInt32 index, ZCHandle* displayHandle);
zcGetDisplayByType(ZCContext context, ZCDisplayType displayType, ZSInt32 index, ZCHandle*
displayHandle);
```

Once you have a display handle, you can retrieve display attributes. The following function gets the display type.

```
zcGetDisplayType(ZCHandle displayHandle, ZCDisplayType* displayType);
```

The following function retrieves the number of the display as defined by the set resolution control panel window.

```
zcGetDisplayNumber(ZCHandle displayHandle, ZSInt32* number);
```

The adapter index is the index of the GPU connected to the display.

```
zcGetDisplayAdapterIndex(ZCHandle displayHandle, ZSInt32* adapterIndex);
```

The monitor index is the index of the display that is connected to a specific adapter—or GPU. This is useful if multiple displays are connected to a single GPU.

```
zcGetDisplayMonitorIndex(ZCHandle displayHandle, ZSInt32* monitorIndex);
```

Several string attributes, specified by Windows, are retrievable using the following functions.

```
zcGetDisplayAttributeStrSize(ZCHandle displayHandle, ZCDisplayAttribute attribute, ZSInt32*
size);
zcGetDisplayAttributeStr(ZCHandle displayHandle, ZCDisplayAttribute attribute, char* buffer,
ZSInt32 bufferSize);
```

The first API gets the size of the attributes string, and the second API gets the string value. Most of these attributes directly map to values defined by windows.

The **ZC_DISPLAY_ATTRIBUTE_MODEL** attribute is specific to zSpace. This specifies the model type for the zSpace display. It can have a value of **100**, **200**, **300**, or **Zvr**.

The following functions allow applications to get information specific to zSpace displays:

This function retrieves the physical size of the display in meters.

```
zcGetDisplaySize(ZCHandle displayHandle, ZSFloat* width, ZSFloat* height);
```

This function retrieves the upper left corner position of the zSpace display in the virtual desktop.

```
zcGetDisplayPosition(ZCHandle displayHandle, ZSInt32* x, ZSInt32* y);
```

This function retrieves the pixel resolution of the zSpace display.

```
zcGetDisplayNativeResolution(ZCHandle displayHandle, ZSInt32* x, ZSInt32* y);
```

The following function returns the current display angle of the zSpace display. The display angle is measured in degrees from a horizontal flat position. Note that except on the zSpace 100, this value is dynamic and can change. Applications that need this information must retrieve it for every rendered frame.

```
zcGetDisplayAngle(ZCHandle displayHandle, ZSFloat* x, ZSFloat* y, ZSFloat* z);
```

The following function returns the refresh rate of the display.

```
zcGetDisplayVerticalRefreshRate(ZCHandle displayHandle, ZSFloat* refreshRate);
```

The following function shows whether the display is currently connected to the system.

```
zcIsDisplayHardwarePresent(ZCHandle displayHandle, ZSBool* isHardwarePresent);
```

Another common feature that applications need is to determine is the intersection point where a virtual ray starting at the end of the stylus would intersect with the display. The following function takes all of the zSpace display attributes into account and calculates that intersection point.

```
zcIntersectDisplay(ZCHandle displayHandle, const ZCTrackerPose* pose, ZCDisplayIntersectionInfo*
intersectionInfo);
```

The pose information is in tracker space, and can be retrieved directly from the tracker system. More detail is provided in the documentation for **zcCreateStereoBuffer**, which is available online at http://developer.zspace.com/. The information returned shows whether or not the display was hit, the normalized and unnormalized pixel coordinates in the virtual desktop where the intersection occurred, and the distance (in meters) from the end of the stylus to the intersection point.

Stereo Objects

When you have a display handle, you can create the other objects needed for zSpace stereo calculations. Those objects include the stereo buffer, the viewport, and the frustum. zSpace does not do any stereo rendering. It simply provides data to the application to implement its own stereo rendering. These objects help manage the data calculated by the zSpace SDK.

Stereo Buffer

The stereo buffer represents the buffer where stereo rendering occurs. While zSpace does no scene rendering, there is one case in which zSpace needs to render information to synchronize the left and right eye buffers. The stereo buffer objects encapsulate this functionality. The following APIs are used to create and destroy the stereo buffer object.

```
zcCreateStereoBuffer(ZCContext context, ZCRenderer renderer, void* reserved, ZCHandle*
bufferHandle);
zcDestroyStereoBuffer(ZCHandle bufferHandle);
```

The **ZCRenderer** parameter specifies the rendering API to use. It can be one of: OpenGL, DirectX 9, 10, 11, or 11.1. The reserved parameter is used when the application would like zSpace to use its own window for left/right eye synchronization. See the documents bundled with the SDK for the acceptable values of the reserved parameter.

To run in fullscreen mode, the application must notify zSpace with the following function.

```
zcSetStereoBufferFullScreen(ZCHandle bufferHandle, ZSBool isFullScreen);
```

Use the next API to query whether or not the application is in fullscreen mode according to zSpace.

```
zcIsStereoBufferFullScreen(ZCHandle bufferHandle, ZSBool* isFullScreen);
```

At the beginning of every frame, before any rendering has occurred, the application needs to call the following function.

```
zcBeginStereoBufferFrame(ZCHandle bufferHandle);
```

The zSpace runtime system uses the previous function to continually ensure that left and right eye buffers are synchronized with the display.

An application can manually force synchronization with the following function.

```
zcSyncStereoBuffer(ZCHandle bufferHandle);
```

Once a stereo buffer object is created, a viewport may be defined.

Viewport

The viewport object defines a window on the zSpace display where stereo rendering occurs. As with other zSpace objects, the viewport does no rendering, it just serves as a data container to make sure calculations are correct. An application may define as many viewports as it likes, but it needs to make sure it uses the appropriate data when rendering to each viewport.

The following functions create and destroy the viewport.

```
zcCreateViewport(ZCContext context, ZCHandle* viewportHandle);
zcDestroyViewport(ZCHandle viewportHandle);
```

The only other functions needed for the viewport are to set and get the viewport position and size. They are as follows.

```
zcSetViewportPosition(ZCHandle viewportHandle, ZSInt32 x, ZSInt32 y);
zcGetViewportPosition(ZCHandle viewportHandle, ZSInt32* x, ZSInt32* y);
zcSetViewportSize(ZCHandle viewportHandle, ZSInt32 width, ZSInt32 height);
zcGetViewportSize(ZCHandle viewportHandle, ZSInt32* width, ZSInt32* height);
```

The application needs to make sure the rendering window and viewport position and size remain in sync. Otherwise, rendering is not correct.

Frustum

The frustum is an object which is associated with the viewport. It represents the stereo frustum defined by the current head position, the position and orientation of the zSpace display, and the position and size of the viewport. There are many attributes that can be modified in the stereo frustum. Those attributes are shown in the frustum adjustments in the following paragraphs. This section presents the basics of the frustum and how to integrate it into an application with all of the default attribute values.

The frustum object is automatically created and destroyed with the viewport object. To get the frustum object, the application uses the following function.

```
zcGetFrustum(ZCHandle viewportHandle, ZCHandle* frustumHandle);
```

By default, the zSpace runtime automatically reads the head pose and apply it to all known frustums. If tracking is disabled, the application can manually set and get the head pose used by the frustum to calculate stereo transforms and projections. The following functions are sometimes used to simulate head tracking while debugging.

```
zcSetFrustumHeadPose(ZCHandle frustumHandle, const ZCTrackerPose* headPose);
zcGetFrustumHeadPose(ZCHandle frustumHandle, ZCTrackerPose* headPose);
```

There are two transforms that zSpace calculates which need to be integrated into the rendering transforms. Both transforms are unique to each eye, so they need to be applied appropriately when rendering for an eye.

zSpace takes the head pose, which represents the position and orientation of the center of the glasses, and calculates the two transforms appropriate for the stereo frustum. The view matrix transform represents the relative transform, which combines the interpupillary distance, offset from the glasses to the eye, and the transform from camera space to display space. The view transform is retrieved with the following function.

```
zcGetFrustumViewMatrix(ZCHandle frustumHandle, ZCEye eye, ZSMatrix4* viewMatrix);
```

This matrix needs to be concatenated with the applications camera matrix when preparing to render the scene for the appropriate eye. There are many ways that applications represent their camera transformations, so applying this transform is unique to every application. For example, **StereoFrustumSample**, uses the OpenGL Mathematics (GLM) library for math calculations. In that case, the view matrix is concatenated with the camera matrix in the following code.

```
g_cameraTransform = glm::lookAt(eye, origin, up);
ZSError error = zcGetFrustumViewMatrix(g_frustumHandle, eye, &viewMatrix);

glm::mat4 zsViewMatrix = glm::make_mat4(viewMatrix.f);
g_viewMatrix = zsViewMatrix * g_cameraTransform;
```

With the view matrix processed, you need to process the second transform which is a projection matrix. Use the following function to get the projection matrix.

```
zcGetFrustumProjectionMatrix(ZCHandle frustumHandle, ZCEye eye, ZSMatrix4* projectionMatrix);
```

The projection matrix encodes an off axis projection into a matrix. The matrix in this function is an OpenGL style projection matrix. If you are using OpenGL, this projection matrix is compatible with the standard definition of an OpenGL projection, so it may be passed directly to OpenGL. For other rendering systems, the two following functions retrieve the off-axis projection information.

```
zcGetFrustumBounds(ZCHandle frustumHandle, ZCEye eye, ZCFrustumBounds* bounds);
zcGetFrustumEyePosition(ZCHandle frustumHandle, ZCEye eye, ZCCoordinateSpace coordinateSpace,
ZSVector3* eyePosition);
```

The first function returns the frustum bounds information as the standard six bounds values: left, right, top, bottom, near, and far. These values can be used with appropriate rendering APIs such as **glFrustum()** in OpenGL or **D3DXMatrixPerspectiveOffCenterLH()** in DirectX.

The second function enables the application to retrieve the left or right eye position in any coordinate system they like. A good example of using this function is for real time ray tracing applications. By getting the eye position in viewport space, and using the viewport position and size, the application can construct the appropriate starting ray for the rendering.

With these functions integrated into the application, the stereo should now be working correctly. Note that all values in zSpace are real world measurements where 1.0 is one meter. If the application has modeled objects at the real world scale, stereo rendering displays correctly.

If a different modeling scale was used, a scaling parameter may be changed. That is covered in Frustum Attributes – particularly the **ZC_FRUSTUM_ATTRIBUTE_VIEWER_SCALE** attribute on page 20. The next section describes tracking system integration.

Tracking Basics

When the zSpace runtime is initialized, the tracking system is also initialized. And, by default, the head pose is automatically read and applied to all known frustums. This is done when the application calls the following function.

```
zcUpdate(ZCContext context);
```

Call this function once for each rendering iteration. It caches the tracking data and applies head tracking.

For basic applications, processing the stylus is relatively simple. First, the application must get the tracking target associated with the stylus with the following function.

```
zcGetTargetByType(ZCContext context, ZCTargetType targetType, ZSInt32 index, ZCHandle*
targetHandle);
```

For the stylus, the target type is **ZC_TARGET_TYPE_PRIMARY** and the index is **0**. Once the stylus target is acquired, once per frame, the application needs to get the current stylus pose and transform it into a coordinate system appropriate for application use. To get the current stylus pose, use the following function.

```
zcGetTargetTransformedPose(ZCHandle targetHandle, ZCHandle viewportHandle, ZCCoordinateSpace
coordinateSpace, ZCTrackerPose* pose);
```

This returns the current stylus pose in the requested coordinate system. Note that since some coordinate systems are dependent on the viewport, a viewport handle is required. This also means that applications that wish to use the stylus in multiple viewports need to get the pose as appropriate for each viewport.

Camera space is the common coordinate system between zSpace and applications. Once the pose is retrieved in camera space, the application may want to transform it to world space.

Depending on how applications represent the camera to world space transforms, this may be different for each application. With **StereoFrustumSample**, again using GLM, our transform looks like this.

```
// Grab the stylus pose (position and orientation) in camera space.
ZCTrackerPose stylusPose;
error = zcGetTargetTransformedPose(g_stylusHandle, g_viewportHandle, ZC_COORDINATE_SPACE_CAMERA,
&stylusPose);

// Transform the pose to world space.
glm::mat4 stylusPoseCamera = glm::make_mat4(stylusPose.matrix.f);
g_stylusWorldPose = g_invCameraTransform * stylusPoseCamera;
```

That completes all the calls needed to start a zSpace application.

zSpace Stereo Loop

To structure a typical zSpace stereo loop, first initialize all of the zSpace objects when the application is initialized. This is taken directly from **StereoFrustumSample**.

```
ZCError error;
// Initialize the zSpace SDK. This MUST be called before
// calling any other zSpace API.
error = zcInitialize(&g_zSpaceContext);

// Create a stereo buffer to handle L/R detection.
error = zcCreateStereoBuffer(g_zSpaceContext, ZC_RENDERER_QUAD_BUFFER_GL, 0, &g_bufferHandle);

// Create a zSpace viewport object and grab its associated frustum.
error = zcCreateViewport(g_zSpaceContext, &g_viewportHandle);

error = zcGetFrustum(g_viewportHandle, &g_frustumHandle);

// Grab a handle to the stylus target.
error = zcGetTargetByType(g_zSpaceContext, ZC_TARGET_TYPE_PRIMARY, 0, &g_stylusHandle);

// Find the zSpace display and set the window's position
// to be the top left corner of the zSpace display.
error = zcGetDisplayByType(g_zSpaceContext, ZC_DISPLAY_TYPE_ZSPACE, 0, &g_displayHandle);

error = zcGetDisplayPosition(g_displayHandle, &g_windowX, &g_windowY);
```

This creates all of the necessary zSpace objects that are used in the render loop.

If you look back at the code presented in the Stereo Render Loop section, there were a number of functions that were undefined. These are defined here to show the zSpace calls required in the actual render loop. First is the **update()** function.

```
// Update the camera.
updateCamera();

// Update the zSpace viewport position and size based
// on the position and size of the application window.
error = zcSetViewportPosition(g_viewportHandle, g_windowX, g_windowY);

error = zcSetViewportSize(g_viewportHandle, g_windowWidth / 2, g_windowHeight);

// Update the OpenGL viewport size;
glViewport(0, 0, g_windowWidth / 2, g_windowHeight);

// Update the zSpace SDK. This updates both tracking information
// as well as the head poses for any frustums that have been created.
error = zcUpdate(g_zSpaceContext);

// Grab the stylus pose (position and orientation) in cameraspace.
ZCTrackerPose stylusPose;
error = zcGetTargetTransformedPose(g_stylusHandle, g_viewportHandle, ZC_COORDINATE_SPACE_CAMERA,
&stylusPose);

// Transform the pose to world space.
glm::mat4 stylusPoseCamera = glm::make_mat4(stylusPose.matrix.f);
g_stylusWorldPose = g_invCameraTransform * stylusPoseCamera;
```

This function starts by calling **updateCamera()**, which updates any camera transforms as defined by the application. Then, update the zSpace viewport object and OpenGL for any window size or position changes. Next, call **zcUpdate()** to enable zSpace runtime to update tracking information and frustums. Finally, calculate the stylus world pose as seen in the last section. The other two undefined functions related to zSpace are **computeViewMatrix()** and **setProjectionMatrix()**.

```
bool computeViewMatrix(ZCEye eye)
{
    // Get the view matrix from the zSpace StereoFrustum for the specified eye.
    ZCMatrix4 viewMatrix;
    ZCError error = zcGetFrustumViewMatrix(g_frustumHandle, eye, &viewMatrix);
    CHECK_ERROR(error);

    glm::mat4 zcViewMatrix = glm::make_mat4(viewMatrix.f);
    g_viewMatrix = zcViewMatrix * g_cameraTransform;
    return true;
}
```

This function gets the view matrix from zSpace and multiplies it by the existing camera transform. This composite transform can now be used to compute the model view matrix for each object to be rendered.

```
bool setProjectionMatrix(ZCEye eye)
{
   // Get the projection matrix from the zSpace StereoFrustum for a specified eye.
   ZSMatrix4 projectionMatrix;
   ZCError error = zcGetFrustumProjectionMatrix(g_frustumHandle, eye, &projectionMatrix);
   CHECK_ERROR(error);

   // Convert the projection matrix to glm, and pass down to the shader.
   glm::mat4 zcProjMatrix = glm::make_mat4(projectionMatrix.f);
   glUniformMatrix4fv(g_projectionUniform, 1, GL_FALSE, glm::value_ptr(zcProjMatrix));
   return true;
}
```

This function gets the projection matrix and passes it along to OpenGL.

Grabbing Things with the Stylus

One of the most common stylus features is picking up objects and manipulating them with the stylus. **StereoFrustumSample** shows an implementation of this feature, and its logic is presented here.

The first step is to determine the objects that intersect with the virtual ray in the scene. To construct this ray, use the following function.

```
glm::vec3 stylusPosition = glm::vec3(g_stylusWorldPose[3][0], g_stylusWorldPose[3][1],
g_stylusWorldPose[3][2]);
glm::vec3 stylusDirection = glm::normalize(glm::vec3(-g_stylusWorldPose[2][0], -
g_stylusWorldPose[2][1], -g_stylusWorldPose[2][2]));
```

The stylus position is the fourth column of the world pose. The ray direction is the normalized negative Z axis of the pose, which is the third column.

The determination of the object hit by this ray is application dependent. The sample checks the intersection of the ray with all of the cubes using the **checkCubeIntersections()** function. The first hit is returned. Once you find the object that intersects, and are going to drag it, compute information to process the drag correctly.

```
// This code figures out the beginning offset from the end
// of the virtual stylus to the center of the grabbed cube.
// It also computes the starting cumulative rotation of the stylus
// and the grabbed cube. These will be used to compute the correct
// modelview transform of the object while it is dragged.
glm::quat quat = glm::quat_cast(glm::mat3(g_cubes[g_currentCubeHit].modelView));
quat = glm::inverse(quat);
glm::mat4 matrix = glm::mat4_cast(quat);

glm::vec4 cubePosition = g_cubes[g_currentCubeHit].modelView * glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);
glm::vec4 stylusEnd = glm::vec4((stylusPosition + (stylusDirection * g_stylusLength)), 1.0f);

glm::vec4 offset = cubePosition - stylusEnd;
g_startOffset = matrix * offset;

glm::quat rotation = glm::quat_cast(glm::mat3(g_stylusWorldPose));
rotation = glm::inverse(rotation);
g_startRotation = rotation * glm::quat_cast(glm::mat3(g_cubes[g_currentCubeHit].modelView));
```

The first section computes the world rotation for the cube that was hit. Then compute the current offset from the end of the stylus to the end of the cube. This is the **g_startOffset** variable. Maintain this offset as the stylus moves. Also, cache the cumulative rotation of the stylus and the cube. This is the **g_startRotation** variable. This is the starting rotation to use when adding new stylus rotations. Once the object is grabbed, update its model view transform as it is moved and rotated.

Use the following code to set the new model view matrix to be the rotation.

```
glm::mat4 matrix = glm::mat4_cast(newRotation);
```

Then set the individual position elements directly using this code.

```
matrix[3][0] = newOffset.x;
matrix[3][1] = newOffset.y;
matrix[3][2] = newOffset.z;

glm::vec4 stylusEnd = glm::vec4((stylusPosition + (stylusDirection * g_stylusLength)), 1.0f);
glm::quat rotation = glm::quat_cast(glm::mat3(g_stylusWorldPose));
glm::quat newRotation = rotation * g_startRotation;

glm::mat4 matrix = glm::mat4_cast(newRotation);
glm::vec4 offset = matrix * g_startOffset;
```

Finally, set the individual position elements directly with the following code.

```
// Set the modelview matrix to be the new rotation and offset as calculated above.
glm::vec3 newOffset = glm::vec3(offset + stylusEnd);
matrix[3][0] = newOffset.x;
matrix[3][1] = newOffset.y;
matrix[3][2] = newOffset.z;
g_cubes[g_draggingCube].modelView = matrix;
```

The new rotation for the object is simply the rotation of the current stylus added onto the starting rotation. The new position is the starting offset transformed by the new rotation, and then added to the current end of the stylus. This code and all the logic associated are included in **StereoFrustumSample**. Each application may implement this slightly differently depending on the math library and transform representation, but the overall logic still applies.

Cleaning Up

To shut down the application, make the following call to clean up all zSpace objects.

```
zcShutDown(ZCContext context);
```

This shuts down the zSpace runtime and destroys any zSpace objects that were created.

4: Frustum Attributes

Frustum Adjustments

Most applications run well using the basic setup and logic explained previously. But there are many ways to adjust frustums to make the stereo experience better. This section describes all the attributes that can be adjusted. For most attributes, use the following get and set functions to set their values.

If the attribute is expecting a floating point value, use this function of the API.

```
zcSetFrustumAttributeF32(ZCHandle frustumHandle, ZCFrustumAttribute attribute, ZSFloat value);
zcGetFrustumAttributeF32(ZCHandle frustumHandle, ZCFrustumAttribute attribute, ZSFloat* value);
zcSetFrustumAttributeB(ZCHandle frustumHandle, ZCFrustumAttribute attribute, ZSBool value);
zcGetFrustumAttributeB(ZCHandle frustumHandle, ZCFrustumAttribute attribute, ZSBool* value);
```

If the attribute is expecting a Boolean value, use this function of the API.

```
zcSetFrustumAttributeB(ZCHandle frustumHandle, ZCFrustumAttribute attribute, ZSBool value);
zcGetFrustumAttributeB(ZCHandle frustumHandle, ZCFrustumAttribute attribute, ZSBool* value);
```

There is no automatic conversion between value types. If the floating point version of the API is used for a Boolean attribute, an error occurs. The same is true when using the Boolean API on a floating point attribute.

The set of attributes that may be retrieved are defined in the **ZCFrustumAttribute enum**. A description of each of these attributes follows.

Frustum Attributes

The following attributes affect the actual shape of the off axis frustums. A visual description of these attributes can be found in the *zSpace Developer Guide – SDK Introduction*.

ZC_FRUSTUM_ATTRIBUTE_IPD

The physical separation—or inter-pupillary distance—between the eyes, measured in meters. An IPD of zero (0) disables stereo since the eyes are at the same location.

ZC_FRUSTUM_ATTRIBUTE_VIEWER_SCALE

Viewer scale adjusts the display and head tracking for larger and smaller scenes. Use larger values for scenes with large models and smaller values for smaller models.

ZC_FRUSTUM_ATTRIBUTE_HEAD_SCALE

Uniform scale factor applied to the frustum's incoming head pose.

ZC_FRUSTUM_ATTRIBUTE_NEAR_CLIP

Near clipping plane for the frustum, in meters.

ZC_FRUSTUM_ATTRIBUTE_FAR_CLIP

Far clipping plane for the frustum, in meters.

ZC_FRUSTUM_ATTRIBUTE_GLASSES_OFFSET

Distance between the bridge of the glasses and the bridge of the nose, in meters.

The following code shows examples for setting these attributes.

```
float     g_ipd           = 0.056f;
float     g_viewerScale   = 10.0f;
float     g_headScale     = 1.0f;
zcSetFrustumAttributeF32(g_frustumHandle, ZC_FRUSTUM_ATTRIBUTE_IPD, g_ipd);
zcSetFrustumAttributeF32(g_frustumHandle, ZC_FRUSTUM_ATTRIBUTE_VIEWER_SCALE, g_viewerScale);
zcSetFrustumAttributeF32(g_frustumHandle, ZC_FRUSTUM_ATTRIBUTE_HEAD_SCALE, g_headScale);
```

Auto Stereo

The native SDK can track whether or not the glasses are visible. If the glasses are not visible, the runtime animates from a pair of stereo frustums to a mono frustum. The application does not need to change its rendering logic. When in mono mode, the left and right frustums are the same, and reflect a single frustum at the center eye point. When the glasses are again visible, the system animates back to two stereo frustums.

There are a few attributes that can change the behavior of this feature. The first attribute is the only Boolean attribute currently in the system.

ZC_FRUSTUM_ATTRIBUTE_AUTO_STEREO_ENABLED

Flag controlling whether the automatic transition from stereo to mono is enabled.

The next two attributes modify the delay and duration of the stereo to mono animation.

ZC_FRUSTUM_ATTRIBUTE_AUTO_STEREO_DELAY

The delay in seconds before the automatic transition from stereo to mono begins.

ZC_FRUSTUM_ATTRIBUTE_AUTO_STEREO_DURATION

The duration in seconds of the automatic transition from stereo to mono.

Portal Mode

As described in the *zSpace Developer Guide – SDK Introduction*, zSpace implements a fish tank VR style system. The viewport is the portal into the virtual world. When the viewport is moved on the display, or the display angle changes, there are two ways to react to these events. Either move the world with the viewport or display angle, or keep the virtual world static and move the viewport through the world. Applications control this behavior by adjusting the portal mode used by the frustum.

```
zcSetFrustumPortalMode(ZCHandle frustumHandle, ZSInt32 portalModeFlags);
zcGetFrustumPortalMode(ZCHandle frustumHandle, ZSInt32* portalModeFlags);
```

The following functions are used to adjust the portal mode values. The portal mode flags are a bitmask representing the portal modes to enable.

ZC_PORTAL_MODE_ANGLE

The scene's orientation is fixed relative to the physical desktop.

ZC_PORTAL_MODE_POSITION

The scene's position is fixed relative to the center of the display.

ZC_PORTAL_MODE_ALL

All portal modes except "none" are enabled.

ZC_PORTAL_MODE_NONE

The scene moves with the viewport and ignores the display angle.

If the portal mode is **ZC_PORTAL_MODE_NONE**, the application may want to simulate display angle changes. Set the following frustum attributes to simulate display angle changes:

ZC_FRUSTUM_ATTRIBUTE_DISPLAY_ANGLE_X
ZC_FRUSTUM_ATTRIBUTE_DISPLAY_ANGLE_Y
ZC_FRUSTUM_ATTRIBUTE_DISPLAY_ANGLE_Z

Display angle is in degrees about the X, Y, and Z axis.

The actual system display angle may still be retrieved with **zcGetDisplayAngle**. The above frustum display angle values are only used when portal mode is set to **ZC_PORTAL_MODE_NONE**.

StereoFrustumSample shows how to set different portal modes and the effect on the scene.

Stereo Comfort

Many factors contribute to a comfortable stereo viewing experience. The document *Understanding zSpace Aesthetics* at http://developer.zspace.com/docs/aesthetics/ describes these aspects. It is important to understand these aspects because they affect the user experience. The document defines a coupled zone (where stereo is comfortable), a crossed (negative parallax) zone, and an uncrossed (positive parallax) zone. The limits of these zones, as measured in pixel disparity, can be retrieved as frustum attributes.

ZC_FRUSTUM_ATTRIBUTE_CC_LIMIT

Maximum pixel disparity for crossed images (negative parallax) in the coupled zone.

ZC_FRUSTUM_ATTRIBUTE_UC_LIMIT

Maximum pixel disparity for uncrossed images (positive parallax) in the coupled zone.

ZC_FRUSTUM_ATTRIBUTE_CU_LIMIT

Maximum pixel disparity for crossed images (negative parallax) in the uncoupled zone.

ZC_FRUSTUM_ATTRIBUTE_UU_LIMIT

Maximum pixel disparity for uncrossed images (positive parallax) in the uncoupled zone.

Applications can also get the physical depth limits for the edges of the coupled zone with these attributes.

ZC_FRUSTUM_ATTRIBUTE_CC_DEPTH

Maximum depth in meters for negative parallax in the coupled zone.

ZC_FRUSTUM_ATTRIBUTE_UC_DEPTH

Maximum depth in meters for positive parallax in the coupled zone.

There are also two utility functions that allow applications to guide some stereo comfort scenarios.

```
zcGetFrustumCoupledBoundingBox(ZCHandle frustumHandle, ZCBoundingBox* boundingBox);
```

This function returns the **boundingBox** of the coupled zone in a camera space. It can tell the application where to focus on the scene.

```
zcCalculateFrustumFit(ZCHandle frustumHandle, const ZCBoundingBox* boundingBox, ZSFloat*
viewerScale, ZSMatrix4* lookAtMatrix);
```

This function takes a **boundingBox** in world space, then calculates an appropriate **viewerScale** and **lookAtMatrix** to make the stereo viewing experience of that area comfortable.

```
zcCalculateFrustumDisparity(ZCHandle frustumHandle, const ZSVector3* point, ZSFloat* disparity);
```

The last utility function takes a point in camera space and calculates its disparity. Positive disparity means that it is uncrossed (positive parallax), while negative disparity is crossed (negative parallax). The application may use this along with the frustum attributes to determine if a point would be in the coupled zone.

StereoFrustumSample has some debug visualization features that show the extent of the coupled zone, and how to use the **ZC_FRUSTUM_ATTRIBUTE_CC_DEPTH** and **ZC_FRUSTUM_ATTRIBUTE_UC_DEPTH** attributes.

Focal Point and Zero Parallax

A commonly performed operation for applications is to place an object at the zero parallax plane. The zero parallax plane is a fixed distance away from the virtual camera defined by the application. This can be thought of as the focal point. This distance is represented in the zSpace SDK by the camera offset vector. This vector can be set with the following functions.

```
zcSetFrustumCameraOffset(ZCHandle frustumHandle, const ZSVector3* cameraOffset);
zcGetFrustumCameraOffset(ZCHandle frustumHandle, ZSVector3* cameraOffset);
```

The length of the camera offset vector is the distance from the applications virtual camera to the center of the display/viewport. Since zSpace uses a right-handed coordinate system, the display is in the direction of the negative Z axis in camera space. Applications can use this value to know how far to place the virtual camera relative to the center of the display/viewport.

It is possible for the application to modify the camera offset vector, but it is not recommended.

It is also necessary to have objects be coplanar with the display. The easiest way to do this is to define the coplanar object in the viewport space, and have each frame transform it into the world space before rendering. The following code shows the process.

```
viewportLowerLeft = glm::vec4(-0.01, -0.01, 0.0f, 1.0f);
viewportUpperLeft = glm::vec4(-0.01, 0.01, 0.0f, 1.0f);
viewportUpperRight = glm::vec4(0.01, 0.01, 0.0f, 1.0f);
viewportLowerRight = glm::vec4(0.01, -0.01, 0.0f, 1.0f);

ZSMatrix4 vToC;
zcGetCoordinateSpaceTransform(g_viewportHandle, ZC_COORDINATE_SPACE_VIEWPORT,
ZC_COORDINATE_SPACE_CAMERA, &vToC);
glm::mat4 viewportToCamera = glm::make_mat4(vToC.f);
glm::mat4 viewportToWorld = g_invCameraTransform * viewportToCamera;

worldLowerLeft = viewportToWorld * viewportLowerLeft;
worldUpperLeft = viewportToWorld * viewportUpperLeft;
worldLowerRight = viewportToWorld * viewportLowerRight;
worldUpperRight = viewportToWorld * viewportUpperRight;
```

StereoFrustumSample uses similar code to place the UI boxes at the zero parallax plane, and keeps them there, no matter where the virtual camera is placed.

Head Pose

The head pose is one of the most important data points feeding the frustum. But, by default, the application needs to only call **zcUpdate()** to have the head pose processed. It is possible to query the head pose and set a custom head pose for the frustum.

```
zcSetFrustumHeadPose(ZCHandle frustumHandle, const ZCTrackerPose* headPose);
zcGetFrustumHeadPose(ZCHandle frustumHandle, ZCTrackerPose* headPose);
```

Getting the head pose can enhance the immersive experience. Processing the head pose is the same as processing the stylus when using the data for application processing.

The application may also set a custom head pose if desired. For the system to use the applications head pose, the application must call **zcSetFrustumHeadPose** after **zcUpdate**, but before any frustum data is used for rendering. This overwrites the one set in **zcUpdate**.

5: Coordinate Spaces

The coordinate spaces used by zSpace are described in the *zSpace Developer Guide – SDK Introduction*. It is important to understand these coordinate spaces when building experiences for zSpace. There are two functions in the SDK used to coordinate space processing: **zcGetCoordinateSpaceTransform** and **zcTransformMatrix**.

Use **zcGetCoordinateSpaceTransform** to get the coordinate system transform from one space to another.

```
zcGetCoordinateSpaceTransform(ZCHandle viewportHandle, ZCCoordinateSpace a, ZCCoordinateSpace b,
ZSMatrix4* transform);
```

Here is an example of its usage and converting that result for use in GLM—the math library used in **StereoFrustumSample**.

```
zcGetCoordinateSpaceTransform(g_viewportHandle, ZC_COORDINATE_SPACE_VIEWPORT,
ZC_COORDINATE_SPACE_CAMERA, &vToC);
glm::mat4 viewportToCamera = glm::make_mat4(vToC.f);
```

The **zcTransformMatrix** uses a right-handed column major format, so it is directly usable with GLM or OpenGL directly.

```
zcTransformMatrix(ZCHandle viewportHandle, ZCCoordinateSpace a, ZCCoordinateSpace b, ZSMatrix4*
matrix);
```

This function can be used to do an *in place* transform of the given matrix from coordinate space “a” to coordinate space “b” as shown here.

```
zcTransformMatrix(g_viewportHandle, ZC_COORDINATE_SPACE_TRACKER, ZC_COORDINATE_SPACE_CAMERA,
&stylusPose.matrix);
```

This transforms the stylus pose read from the tracking system into the camera space.

6: Tracking

Previous sections described how to get data from the head and stylus tracking targets for basic zSpace integration. The rest of this document describes more tracking features and APIs that may be used by the application. This includes more detailed tracker device and target information and exposing all features available with the stylus.

Tracker Devices

The tracking system defines a number of tracker devices. Each tracker device known by the system can be iterated by the application using the tracker device functions.

The following functions allow the application to iterate through all tracker devices, or get a particular named tracker device.

```
zcGetNumTrackerDevices(ZCContext context, ZSInt32* numDevices);
zcGetTrackerDevice(ZCContext context, ZSInt32 index, ZCHandle* deviceHandle);
zcGetTrackerDeviceByName(ZCContext context, const char* deviceName, ZCHandle* deviceHandle);
```

The following functions allow an application to enable or disable a tracker device and check if a tracker device is enabled.

```
zcSetTrackerDeviceEnabled(ZCHandle deviceHandle, ZSBool isEnabled);
zcIsTrackerDeviceEnabled(ZCHandle deviceHandle, ZSBool* isEnabled);
```

The following functions allow an application to get the name of a tracker device.

```
zcGetTrackerDeviceName(ZCHandle deviceHandle, char* buffer, ZSInt32 bufferSize);
zcGetTrackerDeviceNameSize(ZCHandle deviceHandle, ZSInt32* size);
```

Tracker Targets

Tracker devices have a number of tracker targets associated with them. A tracker target has a type associated with it as well. The three tracker target types include **ZC_TARGET_TYPE_HEAD**, **ZC_TARGET_TYPE_PRIMARY**, and **ZC_TARGET_TYPE_SECONDARY**. The head target is the glasses, the primary target is the stylus, and the secondary target is a secondary six degree of freedom input device.

There are a number of APIs that query or set attributes of tracker targets. If a given tracker target does not support that attribute or feature, the error **ZC_ERROR_CAPABILITY_NOT_FOUND** is returned. An example of this is checking the button state of a head tracker target.

To iterate over tracker targets, the following functions are available.

These get the number of known targets, and the target at the specified index, respectively.

```
zcGetNumTargets(ZCHandle deviceHandle, ZSInt32* numTargets);
zcGetTarget(ZCHandle deviceHandle, ZSInt32 index, ZCHandle* targetHandle);
```

These get the number of known targets of the given type, as well as the target of that type at the specified index.

```
zcGetNumTargetsByType(ZCContext context, ZCTargetType targetType, ZSInt32* numTargets);
zcGetTargetByType(ZCContext context, ZCTargetType targetType, ZSInt32 index, ZCHandle*
targetHandle);
```

The following function retrieves the target which has the specified name. There are a few attributes of the tracker target that may be retrieved.

```
zcGetTargetByName(ZCHandle deviceHandle, const char* targetName, ZCHandle* targetHandle);
```

The following function gets the name of the tracker target.

```
zcGetTargetName(ZCHandle targetHandle, char* buffer, ZSInt32 bufferSize);
zcGetTargetNameSize(ZCHandle targetHandle, ZSInt32* size);
```

The following functions allow an application to enable or disable a tacker target. The application can also determine whether or not the target is currently enabled. The third function allows the application to know if the target is currently visible by the tracking system. Applications may want to do something unique if a tracker target is not visible. The SDK uses this feature to implement auto stereo with the head tracker target.

```
zcSetTargetEnabled(ZCHandle targetHandle, ZSBool isEnabled);
zcIsTargetEnabled(ZCHandle targetHandle, ZSBool* isEnabled);
zcIsTargetVisible(ZCHandle targetHandle, ZSBool* isVisible);
```

Tracker targets generate poses. A pose includes both the orientation and position of the tracker target. An application may ask for the current pose of a tracker target at any point in time. Note that **zcUpdate** caches the current pose information for all tracker targets, so multiple calls to retrieve the pose all return the same value.

The application needs to call **zcUpdate** to get another cached pose.

```
zcGetTargetPose(ZCHandle targetHandle, ZCTrackerPose* pose);
zcGetTargetTransformedPose(ZCHandle targetHandle, ZCHandle viewportHandle, ZCCoordinateSpace
coordinateSpace, ZCTrackerPose* pose);
```

The first function to get the pose returns the pose in a tracker space. As a convenience, the second function may be used to return the pose in the coordinate space requested. Since, transforming poses into coordinate systems is viewport dependent, the second function requires a viewport. The following two code snippets are equivalent.

```
// Using zcGetTargetPose
ZCTrackerPose stylusPose;
zcGetTargetPose(g_stylusHandle, &stylusPose);

// Transform the stylus pose from tracker to camera space.
zcTransformMatrix(g_viewportHandle, ZC_COORDINATE_SPACE_TRACKER, ZC_COORDINATE_SPACE_CAMERA,
&stylusPose.matrix);
```

```

// Using zcGetTargetTransformedPose
ZCTrackerPose stylusPose;
zcGetTargetTransformedPose(g_stylusHandle, g_viewportHandle, ZC_COORDINATE_SPACE_CAMERA,
&stylusPose);
```

Certain operations with the stylus can cause the stylus to change its pose as an operation is occurring; for example, when you press the button on the stylus. The act of pressing the button slightly changes the direction of the stylus from where it was when the press started. The zSpace SDK has a feature called pose buffering which can help alleviate this problem.

When pose buffering is enabled, the runtime system keeps the last *N* poses for that target, where N is defined by the capacity of the pose buffer. The application can then get this pose buffer for processing. The following functions enable or disable pose buffering and check to see if it is enabled.

```
zcSetTargetPoseBufferingEnabled(ZCHandle targetHandle, ZSBool isPoseBufferingEnabled);
zcIsTargetPoseBufferingEnabled(ZCHandle targetHandle, ZSBool* isPoseBufferingEnabled);
```

The following functions query the capacity of the pose buffer and allow the capacity to be resized. The capacity is the number of poses to keep. The buffer contains the last N poses.

```
zcResizeTargetPoseBuffer(ZCHandle targetHandle, ZSInt32 capacity);
zcGetTargetPoseBufferCapacity(ZCHandle targetHandle, ZSInt32* capacity);
```

This following function retrieves the pose buffer.

```
zcGetTargetPoseBuffer(ZCHandle targetHandle, ZSFloat minDelta, ZSFloat maxDelta, ZCTrackerPose*
buffer, ZSInt32* bufferSize);
```

Buttons

Primary and secondary tracker targets can have *N* number of buttons. The number of buttons and state of each button can be queried with the following functions.

```
zcGetNumTargetButtons(ZCHandle targetHandle, ZSInt32* numButtons);
zcIsTargetButtonPressed(ZCHandle targetHandle, ZSInt32 buttonId, ZSBool* isButtonPressed);
```

StylusButtonPolledSample shows how to track the button state using tracker target polling. Doing this with events is discussed in the "Event Handlers" section on page 32.

LED

The stylus has an LED in the middle of it, and the state of the LED and its color may be changed with the zSpace SDK. The following functions perform change the state of the LED.

```
zcSetTargetLedEnabled(ZCHandle targetHandle, ZSBool isLedEnabled);
zcIsTargetLedEnabled(ZCHandle targetHandle, ZSBool* isLedEnabled);
```

The following functions set and get the color of the LED. Note that the LED cannot represent all possible specified RGB values, so the system does a closest match to the color specified. See **StylusLedSample** for an example of how to set the LED color.

```
zcSetTargetLedColor(ZCHandle targetHandle, ZSFloat r, ZSFloat g, ZSFloat b);
zcGetTargetLedColor(ZCHandle targetHandle, ZSFloat* r, ZSFloat* g, ZSFloat* b);
```

Vibration

The zSpace stylus has built-in vibration capabilities. Developers can program the vibration pattern and control when it starts and stops. The following functions enable and disable vibration and check whether or not vibration is currently enabled.

```
zcSetTargetVibrationEnabled(ZCHandle targetHandle, ZSBool isVibrationEnabled);
zcIsTargetVibrationEnabled(ZCHandle targetHandle, ZSBool* isVibrationEnabled);
```

The vibration is defined as a repeating set of on and off periods. The intensity during the *on* period may also be specified; however, some older stylus devices may ignore the intensity parameter and vibrate at full intensity. The following function is used to start a vibration pattern.

```
zcStartTargetVibration(ZCHandle targetHandle, ZSFloat onPeriod, ZSFloat offPeriod, ZSInt32
numTimes, ZSFloat intensity);
```

The *on* and *off* periods are specified in seconds.

The intensity is specified as a percentage between 0.0 and 1.0. For example, the following function is set at an intensity of 0.5 is 50% intensity.

```
zcIsTargetVibrating(ZCHandle targetHandle, ZSBool* isVibrating);
zcStopTargetVibration(ZCHandle targetHandle);
```

These last two vibration functions allow the developer to check whether the stylus is currently vibrating and stops the stylus vibration. To see the vibration functions in use, see **StylusVibrateSample**.

Tap

Another stylus input feature is a *tap* event. The stylus can detect when it has been tapped on the surface of the display. The following function checks if the stylus is currently tapped on the display.

```
zcIsTargetTapPressed(ZCHandle targetHandle, ZSBool* isTapPressed);
```

The stylus can also detect a *tap hold* event, which occurs when the stylus is pressed against the display for a certain amount of time. This time is called the hold threshold, and it can be adjusted using the following functions.

```
zcSetTargetTapHoldThreshold(ZCHandle targetHandle, ZSFloat seconds);
zcGetTargetTapHoldThreshold(ZCHandle targetHandle, ZSFloat* seconds);
```

The hold threshold is specified in seconds. Example usage of these functions can be found in the **StylusTapPolledSample** and **StylusTapEventSample**.

Event Handlers

All of the stylus features so far have been presented in a polling fashion. The application needs to continually check the system to detect a change in the state. There are a few stylus features that can generate asynchronous events. Applications may listen for these events by adding the following event handlers.

```
zcAddTrackerEventHandler(ZCHandle targetHandle, ZCTrackerEventType trackerEventType,
ZCTrackerEventHandler trackerEventHandler, const void* userData);
zcRemoveTrackerEventHandler(ZCHandle targetHandle, ZCTrackerEventType trackerEventType,
ZCTrackerEventHandler trackerEventHandler, const void* userData);
```

These APIs add and remove an event handler to be called when the specified event is generated.

The events with possible event handlers include.

```
// Stylus Move Event
ZC_TRACKER_EVENT_MOVE                // Stylus has moved

// Button Events
ZC_TRACKER_EVENT_BUTTON_PRESS        // Stylus button has been pressed
ZC_TRACKER_EVENT_BUTTON_RELEASE      // Stylus button has been released

// Tap Events
ZC_TRACKER_EVENT_TAP_PRESS           // The stylus was pressed on the display
ZC_TRACKER_EVENT_TAP_RELEASE         // The stylus was released from the display
ZC_TRACKER_EVENT_TAP_HOLD            // The stylus was pressed for a period of time
ZC_TRACKER_EVENT_TAP_SINGLE          // The stylus was pressed/released once
ZC_TRACKER_EVENT_TAP_DOUBLE          // The stylus was pressed/released twice
```

Use the following function signature for the event handler.

```
typedef void (*ZCTrackerEventHandler)(ZCHandle targetHandle, const ZCTrackerEventData* eventData,
const void* userData);
```

The tracker event data includes the type of the event, the timestamp, the stylus pose, and the button ID for button events. The user data is the same as when the event handler was added to the system.

Since the stylus is practically never at rest, the stylus move event has a threshold attribute that can adjust how much movement in the stylus is needed to generate a stylus move event. Use the following functions to set and get that threshold.

```
zcSetTargetMoveEventThresholds(ZCHandle targetHandle, ZSFloat time, ZSFloat distance, ZSFloat
angle);
zcGetTargetMoveEventThresholds(ZCHandle targetHandle, ZSFloat* time, ZSFloat* distance, ZSFloat*
angle);
```

The time parameter controls how much time, in seconds, needs to pass before a new event is generated. The distance parameter defines the physical distance movement, in meters, that needs to occur before a new event is generated. The angle, in degrees, is the amount of rotation on any axis that needs to change before a move event is generated. Examples of using event handlers for these actions can be found in **StylusButtonEventSample**, **StylusMoveEventSample**, and **StylusTapEventSample**.

Mouse Emulation

Many applications have existing user interface elements that can interact with the mouse. If you imagine the virtual ray emanating from the end of the stylus and follow it to where it intersects the display, this is a natural point for a 2D mouse to exist. Combine that with the three buttons on the stylus, and you can emulate a mouse with the stylus.

Use the following functions to enable, disable, or check if mouse emulation is currently enabled.

```
zcSetMouseEmulationEnabled(ZCContext context, ZSBool isEnabled);
zcIsMouseEmulationEnabled(ZCContext context, ZSBool* isEnabled);
```

Mouse emulation can be generic to any tracking target that generates six degree of freedom poses. The following functions allow the application to set and get the tracking target used to generate the poses.

```
zcSetMouseEmulationTarget(ZCContext context, ZCHandle targetHandle);
zcGetMouseEmulationTarget(ZCContext context, ZCHandle* targetHandle);
```

When the system calculates the point on the display where the virtual ray intersects, it can apply this data to the mouse in two different ways.

The following functions set and get the current emulation mode.

```
zcSetMouseEmulationMovementMode(ZCContext context, ZCMouseMovementMode movementMode);
zcGetMouseEmulationMovementMode(ZCContext context, ZCMouseMovementMode* movementMode);
```

The mouse emulation mode can be either **ZC_MOUSE_MOVEMENT_MODE_ABSOLUTE** or **ZC_MOUSE_MOVEMENT_MODE_RELATIVE**. In absolute mode, the system moves the mouse position to the exact location of the intersection.

This is very natural, but if the mouse is moving at the same time, there can be cursor interference. If the mode is relative, the difference between the current position and last position is applied to the cursor position. This causes no interference with the mouse, but is somewhat less intuitive to the user.

To turn on the mouse only when the stylus is close to the display, use the following functions.

```
zcSetMouseEmulationMaxDistance(ZCContext context, ZSFloat maxDistance);
zcGetMouseEmulationMaxDistance(ZCContext context, ZSFloat* maxDistance);
```

The mouse emulation max distance is the maximum distance, perpendicular to the display, where mouse emulation begins to occur. If the stylus is further than the specified distance, the mouse emulation does not occur. The distance is in meters.

The application may want the buttons on the stylus to map to mouse buttons in a particular way. The following functions let the application assign and retrieve which stylus buttons map to which mouse button.

```
zcSetMouseEmulationButtonMapping(ZCContext context, ZSInt32 buttonId, ZCMouseButton mouseButton);
zcGetMouseEmulationButtonMapping(ZCContext context, ZSInt32 buttonId, ZCMouseButton*
mouseButton);
```

The button ID is the stylus button number, and the mouse button may be **ZC_MOUSE_BUTTON_LEFT**, **ZC_MOUSE_BUTTON_RIGHT**, or **ZC_MOUSE_BUTTON_CENTER**.

For examples of mouse emulation API usage, see **MouseEmulationSample**.