zSpace Developer

Unity 3D Programming Guide

Guide Version 2.0
Plugin Version 5.0

Before You Begin

This *zSpace Developer Unity 3D Programming Guide* describes how to set up and use the zSpace™ plugin for Unity 3D. Unity is a flexible and powerful development platform for creating multiplatform 3D and 2D games and interactive experiences. The zSpace plugin for Unity 3D simplifies zSpace application development.

Audience

This document is intended for experienced Unity application developers who understand the fundamentals of 3D development.

Scope

This document describes how to access zSpace functions from the Unity system, **zSpace Plugin versions 5.0 and later.**

Document Organization

This document is divided into four sections:

- System Requirements and Setup
- zSpace Plugin Architecture
- Editor UI Properties and Debug Information
- Core API – zSpace functions in the **zSpace.Core.ZCore** APIs.

Related Documents

For more information about developing zSpace applications, see:

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

Contents

1: System Requirements and Setup

The zSpace plugin supports Unity 5.4 and above. Additionally, the plugin supports the following:

- **32-bit** and **64-bit** Unity editors and native standalone players
- **D3D11** and **OpenGL** rendering pipelines

In order to use the zSpace plugin, Unity can either be run using D3D11 or OpenGL. D3D11 is Unity's default and recommended rendering pipeline.

Optionally, to enable OpenGL mode, use the following command line argument and player setting:

- Argument: **-force-glcore**
- Player Setting: Disable **Auto Graphics API for Windows** located via ***Edit > Project Settings > Player > Other Settings > Auto Graphics API for Windows*** and manually add **OpenGLCore** to the list

The zSpace plugin also needs the Unity standalone player applications to allocate resources for stereo. The following command line argument and player setting are required to enable stereo:

- Argument: **-vrmode stereo**
- Player Setting: Enable **Virtual Reality Supported** located via ***Edit > Project Settings > Player > Other Settings > Virtual Reality Supported*** and manually add **Stereo Display (non-head mounted)** to the list

The above command line arguments are only required for Unity standalone player applications. They are not necessary for the Unity editor since it does not support stereo rendering out-of-the-box. As a result, the zSpace plugin provides stereo preview support for Unity's default D3D11 rendering pipeline. The stereo preview window can be accessed via the following ways once the zSpace plugin has been imported into your project:

- Select the **Open Preview Window** menu item from the **zSpace** dropdown menu
- Click the **Open Preview Window** button in the **Debug** section of the **ZCore** inspector window

Note: For the preview window to function properly, please make sure to enable the **Run In Background** player setting located via ***Edit > Project Settings > Player > Resolution and Presentation > Run In Background***

zSpace Plugin Setup

The zSpace plugin is distributed as a Unity package. When you are ready to import the package, use the **Assets\Import Package\Custom Package…** menu item. The included files appear in the following figure.

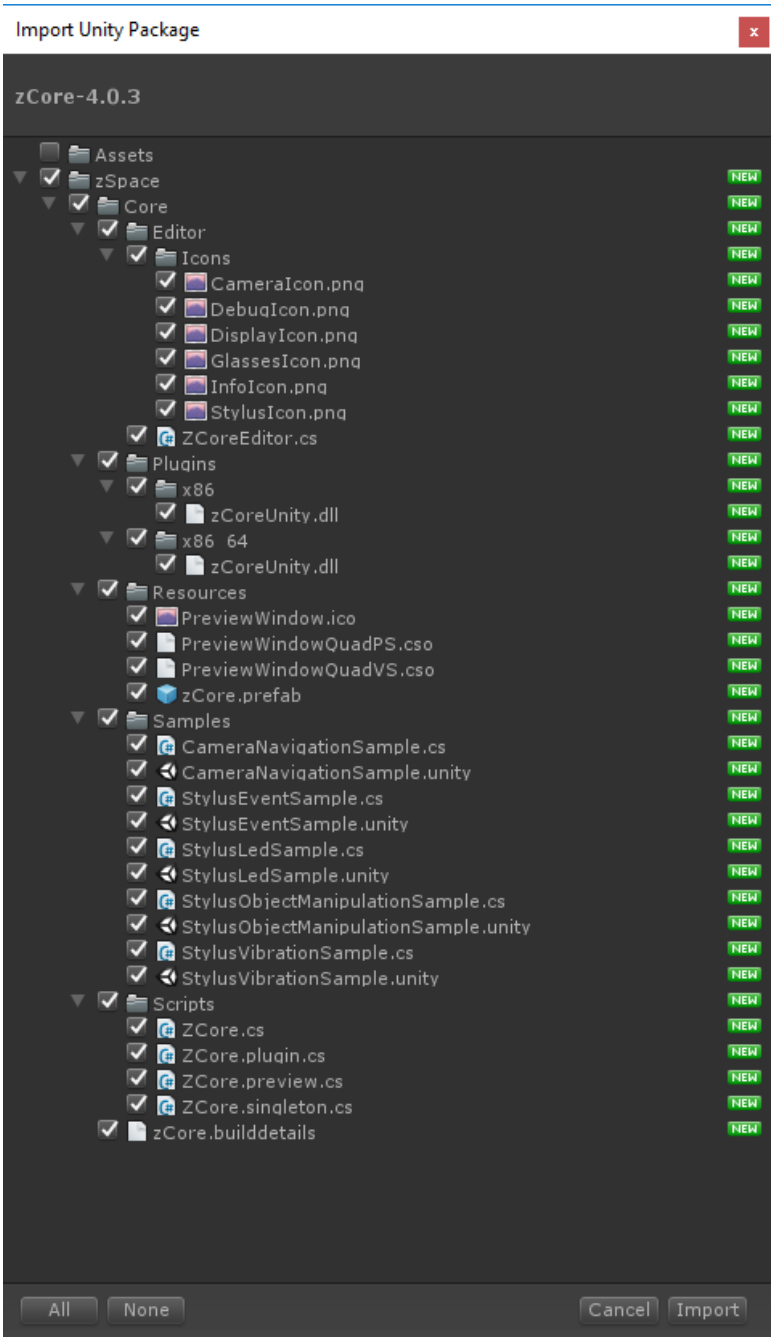

Figure 1: Importing Package

Click **Import** to import the zSpace plugin into the project.

The following directories are part of the import package:

- The **Editor** directory contains all scripts and icons for the zCore inspector window and debug visualizations.
- The **Plugins** directory contains the 32-bit dll (**Plugins\x86**) and 64-bit dll (**Plugins\x86_64**) for communicating with the zSpace runtime system.
- The **Resources** directory contains the zCore prefab to be instantiated in the Unity project.
- The **Samples** directory contains a number of sample Unity scenes to be referenced as examples of zSpace features throughout this document.
- The **Scripts** directory contains all of the internal scripts used by the plugin and the external APIs in the zSpace.Core.ZCore script.

Once the assets are imported, try a test project with simple settings. Create a new project with a cube at the origin and a scale of (0.1, 0.1, 0.1) as described here.

1. From the **Project** window open the **zSpace\Core\Resources** directory.
2. Drag the **zCore** prefab into the **Hierarchy** window or add the **ZCore** MonoBehaviour script to any GameObject in the scene.

   Important Note: Please make sure that only one instance of the ZCore MonoBehaviour has been added to the scene.
3. Drag your **Main Camera** object from the **Hierarchy** window onto the **Current Camera** property in the **Stereo Camera** section of the zCore object.
4. To look at the world origin in the scene, set the **Viewport Center** property to (0, 0, 0) in the **Stereo Camera** section.
5. These values assume real world scale, and a camera looking at the origin. Managing camera placement and non-real world scales is described later in this document.
6. Press play to test the scene.

   You should see a head tracked monoscopic view from the center eye's perspective in the GameView window. To see a stereoscopic view, please follow the instructions specified in page 1 to open the zSpace Preview Window.

Continue to the next section for a description of how the plugin implements this function.

2: zSpace Plugin Architecture

The Unity developer does not need to modify anything in the zSpace plugin to access all of its functionality. But it is helpful to understand the architecture and implementation when building applications. This section explores the plugin architecture.

ZCore MonoBehaviour

The zSpace plugin consists of a single MonoBehaviour called ZCore. For backward compatibility, the zCore.prefab is still available, but has been simplified to a single GameObject due Unity supporting stereo rendering via the UnityEngine.Camera API. Alternatively, in lieu of using the zCore.prefab, the ZCore MonoBehaviour script can be added to any arbitrary GameObject in the scene.

Current Camera Object

Leveraging the following Unity stereo rendering Camera APIs:

```
Camera.SetStereoViewMatrix(Camera.StereoscopicEye eye, Matrix4x4 matrix);
Camera.SetStereoProjectionMatrix(Camera.StereoscopicEye eye, Matrix4x4 matrix);
```

the ZCore MonoBehaviour is responsible for updating the Camera assigned to the ZCore.CurrentCameraObject in order to render stereo correctly for zSpace. Below is an example for how to apply the stereo view and projection matrices from the ZCore MonoBehaviour to a Unity Camera:

```
// NOTE: zCore and camera should be private members that are cached upon initialization for
//       more optimal performance. The following is written for brevity.
ZCore zCore = GameObject.FindObjectOfType<ZCore>();
Camera camera = this.GetComponent<Camera>();

// Update the left and right view matrices.
Matrix4x4 leftView = zCore.GetFrustumViewMatrix(ZCore.Eye.Left) * camera.worldToCameraMatrix;
Matrix4x4 rightView = zCore.GetFrustumViewMatrix(ZCore.Eye.Right) * camera.worldToCameraMatrix;

camera.SetStereoViewMatrix(Camera.StereoscopicEye.Left, leftView);
camera.SetStereoViewMatrix(Camera.StereoscopicEye.Left, leftView);

// Update the left and right projection matrices.
Matrix4x4 leftProjection = zCore.GetFrustumProjectionMatrix(ZCore.Eye.Left);
Matrix4x4 rightProjection = zCore.GetFrustumProjectionMatrix(ZCore.Eye.Right);

camera.SetStereoProjectionMatrix(Camera.StereoscopicEye.Left, leftProjection);
camera.SetStereoProjectionMatrix(Camera.StereoscopicEye.Right, rightProjection);
```

The following operations occur for every frame in the plugin.

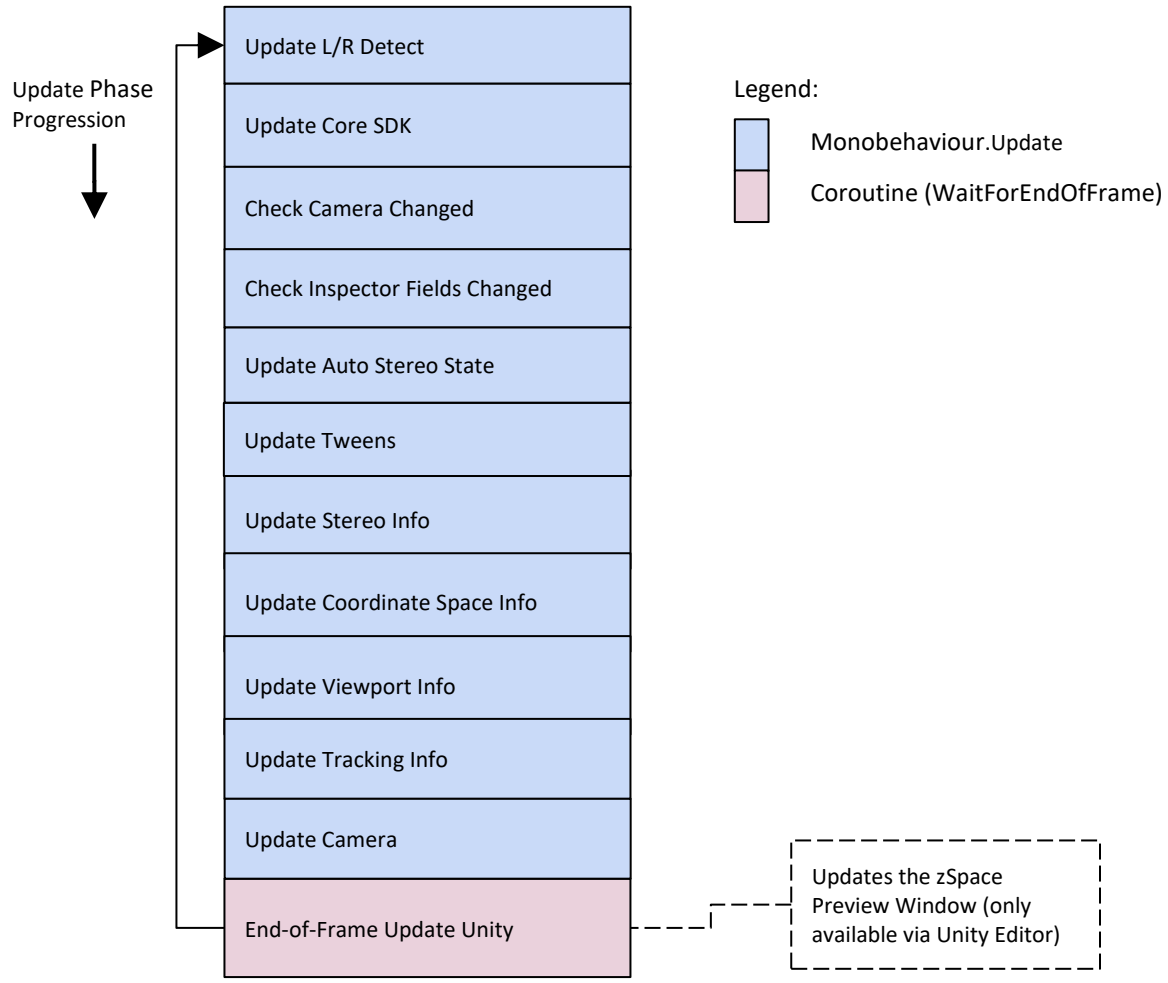

Figure 2: Plugin Operations – This Happens for Every Frame in the Plugin

Plug-In Definitions

Update L/R Detect

Handles any pending L/R sync requests.

Update Core SDK

Updates all underlying Core SDK tracking and stereo related information.

Check Camera Changed

Checks to see if the current camera has changed.

Check Inspector Fields Changed

Checks if any inspector fields have changed—such as IPD, viewer scale, and so on—and pushes updates down to the Core SDK accordingly.

Update Tweens

Updates any internal tweens that have been queued, such as auto stereo tweens, camera navigation tweens, and so on.

Update Auto Stereo State

Updates the internal auto stereo state depending on whether or not the head target has been seen by the tracking cameras.

Update Display Info

Updates any cached display info, such as angle of the zSpace display.

Update Stereo Info

Updates any cached stereo info, such as viewport size, viewport position, view/projection matrices for each eye, and so on.

Update Coordinate Space Info

Updates all cached coordinate space transformations for the stereo rig's corresponding stereo viewport.

Update Tracking Info

Updates all cached tracking info, such as poses for each target in each coordinate space.

Update Camera

Updates the CurrentCameraObject's Camera component with the appropriate left and right view/projection matrices.

End-of-Frame Update

Updates the zSpace Preview Window (only available in the Unity Editor).

3: Editor UI Properties and Debug Information

The zSpace plugin has a number of features that are accessible from the Unity editor including settable properties, debug visualizations, and real time data from the tracking system.

zCore Inspector

The primary tool for getting data about zSpace execution is the inspector pane for the zCore object. This inspector pane has a number of sections, and each of those sections is described in this chapter.

General Info

The General Info section contains the version information for both the zSpace plugin and the runtime version of the zSpace system.

Debug

The Debug section contains a number of checkboxes which control the debug visualizations that are visible in the Scene window of the Unity editor. These visualizations update dynamically, and are present in both edit mode and play mode. Figure 3 shows an example of these visualizations.

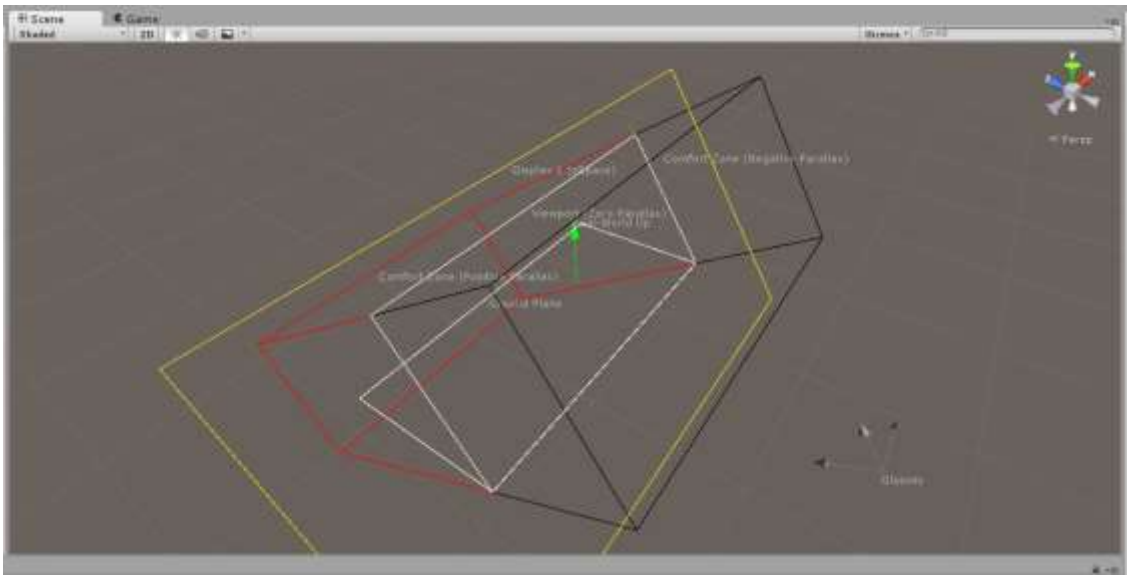

Figure 3: Debug Visualizations

The checkboxes are defined as follows:

- **Show Labels** – Controls whether the text labels of all of the zSpace visualizations are shown.
- **Show Viewport (Zero Parallax)** – Controls showing the viewport being rendered into. The viewport is white and is coplanar with the display.
- **Show Comfort Zone** (**Negative Parallax**) – Controls showing the stereo comfort zone in front of the screen. This box is outlined in black.
- **Show Comfort Zone** (**Positive Parallax**) –Controls showing the stereo comfort zone behind the screen. This box is outlined in red.
- **Show Display** – Controls showing the position and orientation of the physical display. It is outlined in yellow.
- **Show Real-World Up** – This green arrow shows the up direction of the real world. It is the opposite of the gravity vector.
- **Show Ground Plane** – This outline shows the physical ground plane. The real world up vector is the normal for this plane. It is outlined in white.
- **Show Glasses** – Shows the position and orientation of the coordinate system that represents the center position of the glasses.
- **Show Stylus** – Shows the position and orientation of the coordinate system that represents the end position of the stylus.

Additionally the Debug section contains the following button:

- **Open/Close Preview Window** – Toggles between opening and closing the zSpace Preview Window.

Stereo Camera

The Stereo Camera section of the inspector controls properties of zSpace stereo viewing. The properties and functions are described here.

Current Camera

This property contains a reference to the current application camera which will be updated with the appropriate left and right view/projection matrices to enable accurate head-tracked stereo rendering for zSpace.

Viewport Center

The center position of the primary viewport in world space. Modifying the viewport center will update the current camera's transform.

Viewport Rotation

The rotation of the primary viewport in world space. Modifying the viewport rotation will update the current camera's transform.

Enable Auto-Transition to Mono

The zSpace plugin can track whether or not the glasses are currently visible. As the glasses transition from visible to not visible, the plugin can animate from a stereo view to a mono view and back. This property enables that stereo to mono and back transition. Note that when rendering in this mono mode, both eyes are still being rendered. They are just rendered with the same frustum.

IPD

This is the interpupillary distance that is used by the system for separating the left and right eyes. Applications should use as close to an accurate value as possible to maximize stereo comfort.

Viewer Scale

By default, zSpace uses a scale of 1.0 to one meter. All head and stylus tracking information is defined and processed at this scale. It is suggested that applications also model everything in real world scales. However, it is sometimes not possible for applications to do this. The viewer scale property allows applications to model in a different scale. This is a direct multiplier to the frustum calculations. Values greater that 1.0 grow the frustum and distance from the Current Camera to the display, while values less that one shrink the frustum and distance from the Current Camera to the display.

The scale factor directly maps to real world measurements. For example, by default the display may be 0.521 meters wide. The frustums match these real world measurements. If the viewer scale is set to 10.0, then the effective screen width would be 5.21 meters. Note that this also multiplies the distance from the Current Camera to the display. If it is usually 0.41 meters away, then a viewer scale of 10.0

would place it 4.1 meters away. Applications can adjust the viewer scale and position of the Current Camera to match the modeling scale of their application.

Note: Modifying viewer scale via the inspector will update the current camera's transform in order to maintain the viewport's current center and rotation (as specified via the inspector).

Auto Stereo Delay and Duration

These control the delay and duration of the auto stereo to mono transition.

Glasses

The glasses section of the inspector provides real time data about the glasses. Currently that information includes the pose data for the glasses.

Tracker-Space Pose

The tracker space pose is the current position and orientation of the glasses in tracker space. The position is at the center of the glasses. The orientation is the X, Y, and Z axis rotations in degrees.

World-Space Pose

The world space pose is the current position and orientation of the glasses in world space. The position is at the center of the glasses. The orientation is the X, Y, and Z axis rotations in degrees.

Stylus

The stylus section of the inspector contains real time data about the stylus, as well as checkboxes to enable some stylus features. The stylus features are mouse emulation and mouse auto hide.

Enable Mouse Emulation

This checkbox enables and disables mouse emulation. Many applications have existing user interface elements that can be interacted with using the mouse. If you imaging the virtual ray emanating from the end of the stylus and follow it to where it intersects with the display, this is a very natural point for the 2D mouse to exist. Combine that with the three buttons on the stylus, and it is easy to emulate a mouse with the stylus. That is the function of the mouse emulation feature.

Enable Mouse Auto-Hide

This checkbox enables or disables this feature. By default, the mouse cursor is visible in the zSpace window. This is sometimes needed when interacting with 2D user interface elements. It can also be distracting in a stereoscopic environment. The mouse auto hide feature turns off the cursor visibility if a certain amount of time has passed since the mouse has moved or a mouse button was pressed.

Tracker-Space Pose

The tracker space pose is the current position and orientation of the stylus in the tracker space. The position is at the end of the stylus. The orientation is the X, Y, and Z axis rotations in degrees.

World-Space Pose

The world space pose is the current position and orientation of the stylus in world space. The position is at the end of the stylus. The orientation is the X, Y, and Z axis rotations in degrees.

Display

The display section of the inspector shows real time data related to the properties of the displays attached to the system. Each display shows the following information.

Position

This is the upper left corner position of the display relative to the overall windows virtual desktop. The position is given in pixels.

Size

The physical size of the display in meters.

Resolution

The current resolution of the display in pixels.

Angle

The current angle of the display. The angles are rotations about the appropriate axis from the horizontal position measured in degrees. Currently, only the rotation about the X axis is tracked.

Simple Stylus Access

With the editor properties, most Unity applications can get head tracked stereo working. This section presents some very simple code to get access to the stylus. See “Core API” on page 15 for details about stylus functions.

Stylus Events

The plugin provides a way to get stylus move and button events. Using the ZCore script, you can add a listener to stylus move and button press and release events. The following code shows how to do this.

```
_core = GameObject.FindObjectOfType<ZCore>();

// Register event handlers.
_core.TargetMove          += HandleMove;
_core.TargetButtonPress   += HandleButtonPress;
_core.TargetButtonRelease += HandleButtonRelease;
```

And then, each event callback has the following forms.

```
private void HandleMove(ZCore sender, ZCore.TrackerEventInfo info)
{
}

private void HandleButtonPress(ZCore sender, ZCore.TrackerButtonEventInfo info)
{
}

private void HandleButtonRelease(ZCore sender, ZCore.TrackerButtonEventInfo info)
{
}
```

The TrackerEventInfo contains the tracker target that generated the event, the type for the target, and the world space pose at the time of the event. For button events, the button ID is also included. For more information, see the “Tracker Events” section on page 32.

Stylus Game Object Manipulation

A very common operation in zSpace applications is to pick up and directly manipulate objects with the stylus. The **StylusObjectManipulationSample** scene and script show how to do this, but the high level details are presented here.

The sample implements a scheme where the object closest to the stylus may be grabbed when the primary button is pressed. The object must be in the direction the stylus is pointing. While the button is pressed, the object is attached to the point where the virtual stylus ray intersected the object.

In each frame, check whether or not the stylus is currently grabbing an object. If not, check to see if the ray from the stylus intersects anything. Note that only objects with a collider are reported as intersecting the ray.

```
Core.Pose pose = _core.GetTargetPose(ZCore.TargetType.Primary,ZCore.CoordinateSpace.World);
RaycastHit hit;
if (Physics.Raycast(pose.Position, pose.Direction, out hit))
{
    // Update the stylus beam length.
    _stylusBeamLength = hit.distance;

    // If the front stylus button was pressed, initiate a grab.
    if (isButtonPressed && !_wasButtonPressed)
    {
        // Begin the grab.
        this.BeginGrab(hit.collider.gameObject, hit.distance, pose.Position, pose.Rotation);

        _stylusState = StylusState.Grab;
    }
}
```

If something is detected when the button is pressed, start the grab. When the grab is started, be sure to cache the state.

```
private void BeginGrab(GameObject hitObject, float hitDistance,Vector3 inputPosition, Quaternion
inputRotation)
{
    Vector3 inputEndPosition = inputPosition + (inputRotation * (Vector3.forward * hitDistance));

    // Cache the initial grab state.
    _initialGrabOffset  = Quaternion.Inverse(hitObject.transform.localRotation) *
    (hitObject.transform.localPosition - inputEndPosition);
    _initialGrabRotation = Quaternion.Inverse(inputRotation) * hitObject.transform.localRotation;
}
```

First, compute the world rotation for the object. Then compute the current offset from the end of the stylus to the end of the object. This is the **_initialGrabOffset** variable. Maintain this offset as the stylus moves. Also, be sure to cache the cumulative rotation of the stylus and the object, which is the **_initialGrabRotation** variable. This is the starting rotation to use when adding new stylus rotations.

When grabbing in the current frame, update the position and orientation of the grabbed object based upon the current stylus pose.

```
case StylusState.Grab:
{
    // Update the grab.
    this.UpdateGrab(pose.Position, pose.Rotation);

    // End the grab if the front stylus button was released.
    if (!isButtonPressed && _wasButtonPressed)
    {
        _stylusState = StylusState.Idle;
    }
}
break;

private void UpdateGrab(Vector3 inputPosition, Quaternion inputRotation)
{
    Vector3 inputEndPosition = inputPosition + (inputRotation * (Vector3.forward *
    _initialGrabDistance));

    // Update the grab object's rotation.
    Quaternion objectRotation = inputRotation * _initialGrabRotation;
    _grabObject.transform.rotation = objectRotation;

    // Update the grab object's position.
    Vector3 objectPosition = inputEndPosition + (objectRotation * _initialGrabOffset);
    _grabObject.transform.position = objectPosition;
}
```

The new rotation of object is the rotation of the current stylus added to the starting rotation. The new position is the starting offset transformed by the new rotation added to the current end of the stylus.

See the **StylusObjectManipulationSample** scene and script for the detailed code and logic of this feature.

4: Core API

All of the features presented so far can be accessed both programmatically and through the editor. These features in this section are accessible through the zSpace.Core.ZCore script, which is attached to the zCore game object. There are many more features available in the script than are available in the editor. The section describes all the features available in the zSpace.Core.ZCore script.

Public Properties

There are several features that are available as public properties on the ZCore script.

Debug Visualization

The following properties control the debug visualizations.

```
bool ShowLabels
bool ShowViewport
bool ShowCCZone
bool ShowUCZone
bool ShowDisplay
bool ShowRealWorldUp
bool ShowGroundPlane
bool ShowGlasses
bool ShowStylus
```

Plugin Operation

Applications can dynamically change the camera for which stereoscopic rendering is enabled. To do this, set the following property.

```
GameObject CurrentCameraObject
```

Stereo Control

Several properties control aspects of stereo rendering.

The following property changes the interpupillary distance used by the system.

```
float Ipd
```

The following property changes the viewer scale. Unlike the Viewer Scale property in the ZCore inspector, modifying the ViewerScale property via the ZCore API does not apply any changes to the CurrentCameraObject's transform.

```
float ViewerScale
```

The following properties control enabling of auto stereo and adjust the delay before auto stereo animation starts and the duration of the animation from stereo to mono.

```
bool EnableAutoStereo
float AutoStereoDelay
float AutoStereoDuration
```

Mouse Features

These properties control mouse emulation and mouse auto hide.

To enable or disable either of these features, use the following two properties.

```
bool EnableMouseEmulation
bool EnableMouseAutoHide
```

The following property controls how long to wait for no mouse activity before hiding the cursor.

```
float MouseAutoHideDelay
```

zSpace Plugin

The rest of this document covers the methods available in the ZCore script and the basic operation of the plugin. The first few methods allow applications to see if the plugin is initialized and get the version information for the plugin as well as the zSpace runtime system currently on the system.

```
bool IsInitialized()
string GetPluginVersion()
string GetRuntimeVersion()
```

The next methods allow applications to enable or disable, and check the enabled state for the tracking system. If tracking is disabled, the glasses and the stylus are not tracked.

```
void SetTrackingEnabled(bool isEnabled)
bool IsTrackingEnabled()
```

Stereo

Most of the remaining methods in the ZCore script directly map to the native zSpace SDK. Because of that, a couple of points should be explained. The methods presented here are meant to serve two purposes. First, they give low level access to all the parameters and functions available in the native SDK to the Unity application. They are also an initial definition of a C# language binding to the zSpace native SDK.

In the native SDK, a number of zSpace objects are referenced by a **ZCHandle** object. In the C# bindings, this is represented as an **IntPtr**. This makes sense when using these methods from a generic C# application. For Unity applications, many of these zSpace low-level objects are created and initialized for the system. Rather than force the application to carry around these handles, the SDK exposes versions of the methods that do not take any handle when it would usually require one. In these cases, the method acts upon the low-level zSpace object that is appropriate for the Unity application. For example, these two methods are functionally the same:

```
void SetFrustumAttribute(FrustumAttribute attribute, bool value)
void SetFrustumAttribute(IntPtr frustumHandle, FrustumAttribute attribute, bool value)
```

The Unity application programmer would use the first version, since the method applies to the frustum object appropriate for the Unity application. A generic C# application would use the second one. This document lists both versions for completeness, but Unity applications always use the version of the method that does not take a handle.

Display

This section describes all of the methods related to the display and its attributes.

The displays are discovered when the zSpace runtime is initialized, but the application can rediscover this information by using the following function.

```
void RefreshDisplays()
```

There are three types of displays that the zSpace runtime can find. They are zSpace, Generic, and Unknown. All display information can be retrieved for zSpace type displays. Other types of displays may not return all of the information defined in the zSpace SDK. There are several functions that allow you to iterate over the discovered displays.

These functions return the number of displays discovered. The second version only returns the number of displays of the specified type.

```
int GetNumDisplays()
int GetNumDisplays(DisplayType displayType)
```

The following function allows you to get the display at the specific position specified. This position is specified in pixel coordinates in the virtual desktop.

```
IntPtr GetDisplay(int x, int y)
```

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

```
IntPtr GetDisplay(int index)
IntPtr GetDisplay(DisplayType displayType, int index)
```

Once you have a display handle, there are many attributes of the display that may be retrieved. The following function gets the type of the display.

```
DisplayType GetDisplayType()
DisplayType GetDisplayType(IntPtr displayHandle)
```

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

```
int GetDisplayNumber()
int GetDisplayNumber(IntPtr displayHandle)
```

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

```
int GetDisplayAdapterIndex()
int GetDisplayAdapterIndex(IntPtr displayHandle)
```

The monitor index is the index of the display that is connected to a specific adapter - or GPU. This is useful in the case where multiple displays are connected to a single GPU.

```
int GetDisplayMonitorIndex()
int GetDisplayMonitorIndex(IntPtr displayHandle)
```

A number of string attributes specified by windows are retrievable with the following functions. The following functions get the attribute string value. Most of these attributes directly map to values defined by windows, with one that is specific to zSpace is the Model attribute. 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 attributes specific to zSpace displays.

```
string GetDisplayAttributeString(DisplayAttribute attribute)
string GetDisplayAttributeString(IntPtr displayHandle, DisplayAttribute attribute)
```

The following functions retrieve the physical size of the display in meters.

```
Vector2 GetDisplaySize()
Vector2 GetDisplaySize(IntPtr displayHandle)
```

The following functions retrieve the upper left corner position of the zSpace display in the virtual desktop.

```
Vector2 GetDisplayPosition()
Vector2 GetDisplayPosition(IntPtr displayHandle)
```

The following functions retrieve the pixel resolution of the zSpace display.

```
Vector2 GetDisplayNativeResolution()
Vector2 GetDisplayNativeResolution(IntPtr displayHandle)
```

The following functions return the current display angle of the zSpace display.

```
Vector3 GetDisplayAngle()
Vector3 GetDisplayAngle(IntPtr displayHandle)
```

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 constantly change. Applications that need this information must retrieve it for every rendered frame. These return the refresh rate of the display.

```
float GetDisplayVerticalRefreshRate()
float GetDisplayVerticalRefreshRate(IntPtr displayHandle)
```

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

```
bool IsDisplayHardwarePresent()
bool IsDisplayHardwarePresent(IntPtr displayHandle)
```

Another common feature that applications need to determine is where a virtual ray starting at the end of the stylus, pointing in the direction of the stylus, would intersect the display. The following functions take all of the zSpace display attributes into account and calculate that intersection point:

```
DisplayIntersectionInfo IntersectDisplay(Pose pose)
DisplayIntersectionInfo IntersectDisplay(IntPtr displayHandle, Pose pose)
```

The pose information is in tracker space, and retrieved directly from the tracker system. The information returned shows whether or not the display was hit, 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.

Viewport

The viewport object defines a window on the zSpace display where stereo rendering occurs. The viewport does no rendering; it serves as a data container ensuring correct calculations. An application may define as many viewports as it likes, but it needs to use the appropriate data when rendering to each viewport. The following functions create and destroy the viewport. Unity applications generally do not use these methods because the plugin handles all of this processing.

```
IntPtr CreateViewport()
void DestroyViewport(IntPtr viewportHandle)
```

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

```
void SetViewportPosition(IntPtr viewportHandle, int x, int y)
Vector2 GetViewportPosition()
Vector2 GetViewportPosition(IntPtr viewportHandle)
void SetViewportSize(IntPtr viewportHandle, int width, int height)
Vector2 GetViewportSize()
Vector2 GetViewportSize(IntPtr viewportHandle)
```

The application needs to ensure that the rendering window and viewport position and size remain in sync, or the rendering is not correct. The plugin handles this for Unity applications with no added work.

Frustum

The frustum is the 3D region visible on the screen. It represents the stereo frustum defined by the current head position and the position and orientation of the zSpace display. There are many attributes that can be modified in the stereo frustum. The frustum object is automatically created and destroyed with the viewport object. To get the frustum object, the application uses the following function.

```
IntPtr GetFrustum(IntPtr viewportHandle)
```

Most applications run well using the basic setup and logic. Developers can adjust many attributes to customize the stereo experience for their application. For most attributes, the following get and set functions may be used to set their values.

```
void SetFrustumAttribute(FrustumAttribute attribute, float value)
void SetFrustumAttribute(IntPtr frustumHandle, FrustumAttribute attribute, float value)
float GetFrustumAttributeFloat(FrustumAttribute attribute)
float GetFrustumAttributeFloat(IntPtr frustumHandle, FrustumAttribute attribute)
void SetFrustumAttribute(FrustumAttribute attribute, bool value)
void SetFrustumAttribute(IntPtr frustumHandle, FrustumAttribute attribute, bool value)
bool GetFrustumAttributeBool(FrustumAttribute attribute)
bool GetFrustumAttributeBool(IntPtr frustumHandle, FrustumAttribute attribute)
```

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

Frustum Attributes & Definitions

The set of attributes that may be retrieved are defined in the FrustumAttribute enum. A description of each of these attributes follows. The following attributes affect the actual shape of the off axis frustums. A high level visual description of these attributes can be found in the *zSpace Developer Guide – SDK Introduction*.

- **Ipd** – The physical separation, or inter-pupillary distance, between the eyes in meters. An IPD of zero (0) effectively disables the stereo since the eyes are assumed to be at the same location.
- **ViewerScale** – This is a direct multiplier to the frustum calculations. Values greater that 1.0 grow the frustum and distance from the Current Camera to the display, while values less that one shrink the frustum and distance from the Current Camera to the display.
- **HeadScale** – Uniform scale factor applied to the frustum's incoming head pose.
- **NearClip** – Near clipping plane for the frustum in meters.
- **FarClip** – Far clipping plane for the frustum in meters.
- **GlassesOffset** – Distance between the bridge of the glasses and the first nodal point of the eye in meters.

Stereo Comfort

Many factors combine to contribute to a comfortable stereo viewing experience. Understanding zSpace Aesthetics describes these aspects. The aesthetics document defines a coupled zone (where stereo is comfortable), as well as crossed (negative parallax) and uncrossed (positive parallax) zones.

The limits of these zones, as measured in pixel disparity, can be retrieved as frustum attributes, which are defined here.

- **CCLimit** – Maximum pixel disparity for crossed images (negative parallax) in the coupled zone.
- **UCLimit** – Maximum pixel disparity for uncrossed images (positive parallax) in the coupled zone.
- **CULimit** – Maximum pixel disparity for crossed images (negative parallax) in the uncoupled zone.
- **UULimit** – Maximum pixel disparity for uncrossed images (positive parallax) in the uncoupled zone. As a convenience, applications can also get the physical depth limits for the edges of the coupled zone with these attributes.
- **CCDepth** – Maximum depth in meters for negative parallax in the coupled zone.
- **UCDepth** – Maximum depth in meters for positive parallax in the coupled zone. Also, as a convenience, there are two utility functions that allow applications to guide some stereo comfort scenarios.

```
Bounds GetFrustumCoupledBoundingBox()
Bounds GetFrustumCoupledBoundingBox(IntPtr frustumHandle)
```

This function returns the bounding box of the coupled zone in camera space. It tells the application where to put the main focus area of the scene.

Portal Mode

As described in *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. You can either move the world with the viewport or display angle, or you can keep the virtual world static and move the viewport through the world. Applications can control this behavior by adjusting the portal mode used by the frustum. Use the following functions to adjust the portal mode values.

```
void SetFrustumPortalMode(int portalModeFlags)
void SetFrustumPortalMode(IntPtr frustumHandle, int portalModeFlags)
int GetFrustumPortalMode()
int GetFrustumPortalMode(IntPtr frustumHandle)
```

The portal mode flags are a bitmask representing the portal modes to enable. The flags are as follows:

- **None** –The scene moves with the viewport and ignores the display angle.
- **Angle** – The scene's orientation is fixed relative to the physical desktop.
- **Position** – The scene's position is fixed relative to the center of the display.
- **All** – All portal modes except "none" are enabled.

  If the portal mode is None, the application may still want to simulate display angle changes. By setting the following frustum attributes, this may be achieved.

  - **DisplayAngleX**
  - **DisplayAngleY**
  - **DisplayAngleZ**

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

  The actual systems display angle may still be retrieved with GetDisplayAngle. The above frustum display angle values are only used when portal mode is set to *None*.

Focal Point and Zero Parallax

A very common operation for applications is to place an object at the zero parallax plane. The zero parallax plane is a fixed distance away from the Current Camera defined by the application. This can be thought of as the focal point. This distance is represented in the zSpace plugin by the camera offset vector. This vector can be gotten or set with the following functions.

```
void SetFrustumCameraOffset(Vector3 cameraOffset)
void SetFrustumCameraOffset(IntPtr frustumHandle, Vector3 cameraOffset)
Vector3 GetFrustumCameraOffset()
Vector3 GetFrustumCameraOffset(IntPtr frustumHandle)
```

The length of the camera offset vector is the distance from the applications Current Camera to the center of the display/viewport. 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.

Camera Control

It is very common for applications to position and orient the Current Camera so that an object is at the center of the zSpace display. The ZCore script provides two utility functions to help with this operation.

```
void SetViewportWorldTransform(Vector3 center, Quaternion rotation, float viewerScale)
```

This function updates the CurrentCameraObject's transform and the ViewerScale property such that the primary viewport will be centered and oriented in world space based on the specified center, rotation and viewerScale parameters. This is the utility function used by the Stereo Camera property pane in the Unity Editor.

```
Vector3 ComputeCameraPosition(Vector3 focalPoint, Quaternion cameraRotation, float viewerScale)
```

This function returns the world space position where the application should place the Current Camera to have the given focalPoint at the center of the viewport/display. The given cameraRotation represents from which direction the Current Camera is oriented towards the focal point object.

For more advanced camera control operations, see the **CameraNavigationSample** script and scene.

Head Pose

By default, the zSpace plugin automatically reads the head pose and applies it to all known frustums. If tracking is disabled, the application can be used to manually set and get the head pose used by the frustum to calculate stereo transforms and projections.

```
void SetFrustumHeadPose(IntPtr frustumHandle, Pose headPose)
Pose GetFrustumHeadPose()
Pose GetFrustumHeadPose(IntPtr frustumHandle)
```

This is sometimes used to simulate head tracking while debugging.

Transforms

There are two transforms that zSpace calculates that get integrated into the rendering transforms of the Unity system. Both transforms are unique to each eye and need to be applied appropriately when rendering 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:

```
Matrix4x4 GetFrustumViewMatrix(Eye eye)
Matrix4x4 GetFrustumViewMatrix(IntPtr frustumHandle, Eye eye)
```

While the zSpace plugin automatically applies the view matrix to the camera, it is sometimes useful for the application to get the view matrix for other purposes. The view matrices are returned in right-handed space (instead of Unity's default left-handed space) since the Unity Camera.SetStereoViewMatrix() API call expects view matrices to be in right-handed space.

The second transform is a projection matrix. The following function is used to get the projection matrix.

```
Matrix4x4 GetFrustumProjectionMatrix(Eye eye)
Matrix4x4 GetFrustumProjectionMatrix(IntPtr frustumHandle, Eye eye)
```

The projection matrix simply encodes an off axis projection into a matrix. The matrix in the following function is an OpenGL style projection matrix. There are two other ways to get the off axis projection information. The first set of these functions return the frustum bounds information as the standard six bounds values: left, right, top, bottom, near, and far.

```
FrustumBounds GetFrustumBounds(Eye eye)
FrustumBounds GetFrustumBounds(IntPtr frustumHandle, Eye eye)
Vector3 GetFrustumEyePosition(Eye eye, CoordinateSpace coordinateSpace)
Vector3 GetFrustumEyePosition(IntPtr frustumHandle, Eye eye, CoordinateSpace coordinateSpace)
```

The second set of functions allows the application to retrieve the eye position in any coordinate system they like. You can use this function 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.

Coordinate Systems

The coordinate spaces used by zSpace are defined in the *zSpace Developer Guide – SDK Introduction*. It is important to understand these coordinate spaces for building experiences for zSpace. The Unity plugin provides support for the World coordinate space in addition to the standard SDK coordinate systems. It can do this because it has access to the Current Camera's camera to world transform. There are two functions in Core for coordinate space processing.

```
Matrix4x4 GetCoordinateSpaceTransform(CoordinateSpace a, CoordinateSpace b)
Matrix4x4 GetCoordinateSpaceTransform(IntPtr viewportHandle, CoordinateSpace a, CoordinateSpace
b)
```

The following function is used to get the coordinate system transform from coordinate space "a" to coordinate space "b".

```
Matrix4x4 TransformMatrix(CoordinateSpace a, CoordinateSpace b, Matrix4x4 matrix)
Matrix4x4 TransformMatrix(IntPtr viewportHandle, CoordinateSpace a, CoordinateSpace b, Matrix4x4
matrix)
```

Tracking

The tracking system is responsible for supporting all tracking devices, and their tracking targets, connected to the zSpace system. This section describes tracking devices, tracking targets, and all of the features that a tracking target may support. The Unity sample scenes show the usage for a number of these features, and they are identified when appropriate.

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.

```
int GetNumTrackerDevices()
IntPtr GetTrackerDevice(int index)
IntPtr GetTrackerDevice(string deviceName)
```

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

```
void SetTrackerDeviceEnabled(IntPtr deviceHandle, bool isEnabled)
bool IsTrackerDeviceEnabled(IntPtr deviceHandle)
```

The following function allows an application to get the name of a tracker device.

```
string GetTrackerDeviceName(IntPtr deviceHandle)
```

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 two tracker target types include Head and Primary. The head target is the glasses and the primary target is the stylus.

There are several functions that query or set attributes of tracker targets. If a given tracker target does not support that attribute or feature, the **CapabilityNotFoundException** is thrown. An example of this is checking the button state of a head tracker target. To iterate over tracker targets, the following functions are available.

The following functions get the number of known targets, and the target at the specified index respectively.

```
int GetNumTargets(IntPtr deviceHandle)
IntPtr GetTarget(IntPtr deviceHandle, int index)
```

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

```
int GetNumTargets(TargetType targetType)
IntPtr GetTarget(TargetType targetType, int index)
```

This retrieves the target which has the specified name.

```
IntPtr GetTarget(IntPtr deviceHandle, string targetName)
```

There are a few attributes of the tracker target that may be retrieved. The following functions get the name of the tracker target.

```
string GetTargetName(TargetType targetType)
string GetTargetName(IntPtr targetHandle)
```

These allow an application to enable or disable a tacker target, as well as get whether or not the target is currently enabled.

```
void SetTargetEnabled(TargetType targetType, bool isEnabled)
void SetTargetEnabled(IntPtr targetHandle, bool isEnabled)
bool IsTargetEnabled(TargetType targetType)
bool IsTargetEnabled(IntPtr targetHandle)
```

The following functions allow the application to know if the target is currently visible by the tracking system.

```
bool IsTargetVisible(TargetType targetType)
bool IsTargetVisible(IntPtr targetHandle)
```

Applications may want to do something unique if a tracker target is not visible. The plugin uses this feature to implement auto stereo with the head tracker target.

Tracker Target Poses

Tracker targets generate poses. A pose includes a matrix which encodes the position and orientation of the pose, the position, the rotation, the forward direction, a timestamp when the pose occurred, and the coordinate space for the pose. An application may ask for the current pose of a tracker target at any point in time. Note that the plugin **Update()** method caches the current pose information for all tracker targets, so multiple calls to retrieve the pose return the same value.

The following functions get the pose in the specified coordinate space.

```
Pose GetTargetPose(TargetType targetType, CoordinateSpace coordinateSpace)
Pose GetTargetPose(IntPtr targetHandle, CoordinateSpace coordinateSpace)
```

Certain operations with the stylus can cause the stylus to change its pose as the operation is occurring. An example of this is 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 plugin 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 and process it any way that it wishes.

The following functions enable or disable pose buffering and check to see if it is enabled.

```
void SetTargetPoseBufferingEnabled(TargetType targetType, bool isPoseBufferingEnabled)
void SetTargetPoseBufferingEnabled(IntPtr targetHandle, bool isPoseBufferingEnabled)
bool IsTargetPoseBufferingEnabled(TargetType targetType)
bool IsTargetPoseBufferingEnabled(IntPtr targetHandle)
```

The following functions retrieve the pose buffer.

```
IList<Pose> GetTargetPoseBuffer(TargetType targetType, float minDelta, float maxDelta, int
maxNumPoses)
IList<Pose> GetTargetPoseBuffer(IntPtr targetHandle, float minDelta, float maxDelta, int
maxNumPoses)
```

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.

```
void ResizeTargetPoseBuffer(TargetType targetType, int capacity)
void ResizeTargetPoseBuffer(IntPtr targetHandle, int capacity)
int GetTargetPoseBufferCapacity(TargetType targetType)
int GetTargetPoseBufferCapacity(IntPtr targetHandle)
```

Buttons

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

```
int GetNumTargetButtons(TargetType targetType)
int GetNumTargetButtons(IntPtr targetHandle)
bool IsTargetButtonPressed(TargetType targetType, int buttonId)
bool IsTargetButtonPressed(IntPtr targetHandle, int buttonId)
```

These can be used to track button states in a polling fashion. To do this, keep track of the last button state and compare it to the current button state. The **StylusObjectManipulationSample** scene and script implements this logic. Excerpts from that script are shown here.

```
private bool     _wasButtonPressed = false;
private StylusState _stylusState   = StylusState.Idle;

void Update()
{
    bool isButtonPressed = _core.IsTargetButtonPressed(ZCore.TargetType.Primary, 0);

    switch (_stylusState)
    {
        case StylusState.Idle:
        {
            // If the front stylus button was pressed, initiate a grab.
            // if (isButtonPressed && !_wasButtonPressed)
            {
                _stylusState = StylusState.Grab;
            }
        }
    }
    break;

    case StylusState.Grab:
    {
        // End the grab if the front stylus button was released.
        if (!isButtonPressed && _wasButtonPressed)
        {
            _stylusState = StylusState.Idle;
        }
    }
    break;

    default:
    break;
    }
        // Cache state for next frame.
        _wasButtonPressed = isButtonPressed;
    }
}
```

Each time **Update()** is called, the code checks the state of the button. If it was not pressed, and it is now pressed, a button press has happened. If the button was pressed and now it is not, the button release has occurred. Button states can also be monitored with events. This is described in “Tracker Events.”

LED

The stylus has an LED in the middle of it, and the state of the LED and the color may be changed using the zSpace ZCore script. The following functions enable or disable the LED as well as check if it is currently enabled.

```
void SetTargetLedEnabled(TargetType targetType, bool isLedEnabled)
void SetTargetLedEnabled(IntPtr targetHandle, bool isLedEnabled)
bool IsTargetLedEnabled(TargetType targetType)
bool IsTargetLedEnabled(IntPtr targetHandle)
```

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 the **StylusLedSample** scene and script for an example of how to set the LED color.

```
void SetTargetLedColor(TargetType targetType, Color ledColor)
void SetTargetLedColor(IntPtr targetHandle, Color ledColor)
Color GetTargetLedColor(TargetType targetType)
Color GetTargetLedColor(IntPtr targetHandle)
```

Vibration

The zSpace stylus has vibration capabilities built into it. Developers can program the vibration to be a pattern as well as control when it starts and stops.

The following functions enable and disable vibration and check whether or not vibration is currently enabled.

```
void SetTargetVibrationEnabled(TargetType targetType, bool isVibrationEnabled)
void SetTargetVibrationEnabled(IntPtr targetHandle, bool isVibrationEnabled)
bool IsTargetVibrationEnabled(TargetType targetType)
bool IsTargetVibrationEnabled(IntPtr targetHandle)
```

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 functions are used to start a vibration pattern. The on and off period are specified in seconds. The intensity is specified as a percentage between 0.0 and 1.0. For example, an intensity of 0.5 is 50% intensity.

```
void StartTargetVibration(TargetType targetType, float onPeriod, float offPeriod, int numTimes,
float intensity)
void StartTargetVibration(IntPtr targetHandle, float onPeriod, float offPeriod, int numTimes,
float intensity)
```

These vibration functions allow the developer to check whether the stylus is currently vibrating and stops the stylus vibration. To see the vibration APIs in use, see the **StylusVibrationSample** scene and script.

```
bool IsTargetVibrating(TargetType targetType)
bool IsTargetVibrating(IntPtr targetHandle)
void StopTargetVibration(TargetType targetType)
void StopTargetVibration(IntPtr targetHandle)
```

TAP

The stylus has another input feature available to developers. The stylus can detect when it has been tapped on the surface of the display. The following function is used to check if the stylus is currently tapped on the display.

```
bool IsTargetTapPressed(TargetType targetType)
bool IsTargetTapPressed(IntPtr targetHandle)
```

Mouse Emulation

Many applications have existing user interface elements that can be interacted with the mouse. If you imaging the virtual ray emanating from the end of the stylus and follow it to where it intersects the display, this is a very natural point for the 2D mouse to exist. Combine that with the three buttons on the stylus, and it becomes pretty easy to emulate a mouse with the stylus. That is the function of the mouse emulation feature. The following two functions are used enable, disable, or check if mouse emulation is currently enabled.

```
void SetMouseEmulationEnabled(bool isEnabled)
bool IsMouseEmulationEnabled()
```

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 which tracking target is being used to generate the poses. The first method allows the application to set the mouse emulation target to be Head or Primary.

```
void SetMouseEmulationTarget(TargetType targetType)
void SetMouseEmulationTarget(IntPtr targetHandle)
IntPtr GetMouseEmulationTarget()
```

When the system calculates the point on the display where the virtual ray intersects, it can apply this data to the mouse in a couple of different ways. The next functions set and get the current emulation mode.

```
void SetMouseEmulationMovementMode(MovementMode movementMode)
MovementMode GetMouseEmulationMovementMode()
```

The mouse emulation mode can be either **Absolute** or **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 fighting. If the mode is relative, the difference between the current position and last position is applied to the cursor position. This causes no fighting with the mouse, but is somewhat less intuitive to the user.

It is also useful to turn on mouse emulation only when the stylus is close to the display. The following function can control that feature.

```
void SetMouseEmulationMaxDistance(float maxDistance)
float GetMouseEmulationMaxDistance()
```

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

Finally, the application may want the buttons on the stylus to map to mouse buttons in a particular way. These last functions let the application assign and retrieve, which stylus buttons map to which mouse button.

```
void SetMouseEmulationButtonMapping(int buttonId, MouseButton mouseButton)
MouseButton GetMouseEmulationButtonMapping(int buttonId)
```

The button id is the stylus button number, and the mouse button may be Left, Right, or Center.

Tracker Events

All of the tracker target features have been presented in a polling fashion. The application needs to continually check the state of something to detect a change in the state. There are a few tracking target features that can also generate asynchronous events. Applications may listen for these events by adding event handlers. The following method signatures are defined for the listeners of these events.

```
delegate void EventHandler(ZCore sender);
delegate void TrackerEventHandler(ZCore sender, TrackerEventInfo info);
delegate void TrackerButtonEventHandler(ZCore sender, TrackerButtonEventInfo info);
```

The following events may be known. The **PreUpdate()** gets called before the plugin starts its per frame **Update()** processing. The **PostUpdate()** gets called after the plugin finishes its per frame **Update()** processing.

```
event EventHandler PreUpdate;
event EventHandler PostUpdate;
```

If the tracker target has moved, the TrackerEventInfo contains the tracker target that generated the event, the type for the target, and the world space pose at the time of the event.

```
event TrackerEventHandler TargetMove;
```

To detect if the tracker target has been pressed onto the display, use the following function.

```
event TrackerEventHandler TargetTapPress;
```

To detect if the tracker target has been released from the display, use the following function.

```
event TrackerEventHandler TargetTapRelease;
```

To detect if the tracker target button has been pressed, use the following function. The **TrackerButtonEventInfo** has all the information in the **TrackerEventInfo** and the button ID of the button.

```
event TrackerButtonEventHandler TargetButtonPress;
```

To detect if the tracker target button has been released, use the following function.

```
event TrackerButtonEventHandler TargetButtonRelease;
```

The **StylusEventSample** script shows examples of these event handlers. In a **Start()** callback, the code sets up the event handlers.

```
void Start()
{
    _core = GameObject.FindObjectOfType<ZCore>();
    if (_core == null)
    {
        Debug.LogError("Unable to find reference to zSpace.Core.ZCore Monobehaviour.");
        this.enabled = false;
        return;
    }

    // Register event handlers.
    _core.TargetMove           += HandleMove;
    _core.TargetButtonPress    += HandleButtonPress;
    _core.TargetButtonRelease  += HandleButtonRelease;
    _core.TargetTapPress       += HandleTapPress;
    _core.TargetTapRelease     += HandleTapRelease;
}
```

And then, each event callback has the following forms.

```
private void HandleMove(ZCore sender, ZCore.TrackerEventInfo info)
{
}

private void HandleButtonPress(ZCore sender, ZCore.TrackerButtonEventInfo info)
{
}

private void HandleButtonRelease(ZCore sender, ZCore.TrackerButtonEventInfo info)
{
}

private void HandleTapPress(ZCore sender, ZCore.TrackerEventInfo info)
{
}

private void HandleTapRelease(ZCore sender, ZCore.TrackerEventInfo info)
{
}
```

This allows applications to use event notification instead of polling.