

zSpace Developer

SDK Guide - Introduction

Version 1.0



© zSpace, Inc. 2015. zSpace is a registered trademark of zSpace, Inc. All other trademarks are the property of their respective owners.

Table of Contents

- 1: zSpace System Overview and Setup..... 1**
 - zSpace Features 1
 - Stereo Display 1
 - Angle Awareness 1
 - Glasses 2
 - Stylus..... 3
 - System Architecture..... 4
 - zSpace Systems 5
 - zSpace 100 5
 - zSpace 200 6
 - zSpace 300 7
 - HP Zvr..... 8
 - zSpace System Setup..... 8
 - System and GPU Requirements..... 8
 - System Check and Control Panel 9
- 2: Stereo & Stereo Rendering 10**
 - Mono Frustums..... 10
 - Binocular Vision..... 11
 - Stereo Frustums..... 11
 - Parallax/Disparity..... 12
 - Stereo Comfort..... 13
 - Head Tracking..... 14
 - Head Mounted Display (HMD) versus Fish Tank VR 14
 - Rendering Stereo Overview 16
 - Rendering Buffers 16
 - HMD Buffers 16
 - Side-by-Side Stereo..... 17
 - Quad Buffered Stereo..... 18
 - Rendering Camera Attributes 18
 - Head Position..... 19
 - Interpupillary Distance and Glasses Offset..... 19
 - View Transform and Projection..... 20
- 3: Coordinating Space & Scale 21**
 - Camera Space..... 21

Viewport Space	22
World Space	22
Display Space	23
Tracker Space	24
Viewer Scale	24

1: zSpace System Overview and Setup

This document describes the fundamentals of building and porting applications to the zSpace platform. It is designed for developers who have built 3D applications, but who have not developed stereo or virtual reality (VR) applications. It provides introductory and conceptual information and explores stereo rendering and coordinating spaces. To learn about developing for zSpace using native engines or existing engines such as Unity3D, there are two advanced guides, which are companions to this document. Only the essential fundamentals are covered here.

zSpace Features

zSpace systems are all different, but they have several features in common. They all have stereo displays, glasses, stylus devices and tracking systems. The following section explains these features.

Stereo Display

The zSpace display is a quad buffered stereo display. It has two back buffers and two front buffers, instead of the usual single back buffer and single front buffer. In the zSpace system, there is a back buffer and front buffer for each eye, which are referred to as the left and right buffers, respectively. The display sequentially presents the left and right buffers. The refresh rate for the display is 120 Hz, and the buffers have a resolution of 1920x1080. This allows for a full 1080p resolution for each eye, with an effective refresh rate of 60 Hz for the application. The full hardware specifications for each of the zSpace systems can be found [here](#).

Angle Awareness

The zSpace system acts as a window into a virtual world, where certain objects can come through the window. In some zSpace systems, the angle of the display system can be dynamically changed, which changes the view of the virtual world. The system tracks this information, incorporates it into the stereo calculations and makes it available to the application.

Glasses

The zSpace system has an integrated system that tracks certain objects. One of the tracked objects is the stereo glasses, which enable users to see into the virtual world. The 3D position and orientation of each object is calculated, and the objects are tracked asynchronously at a rate of at least 100 Hz. This is commonly referred to as an object with six degrees of freedom (6DOF). The data that encodes this information is called a pose. (Pose data is described in the “SDK Native Programming Guide” and the “SDK Unity 3D Programming Guide.”) This guarantees an accurate low latency pose for each rendered frame. The pose position is located at the center of the glasses, near the bridge of the nose. The glasses are oriented to a right-handed coordinate system, with the X axis projecting along the right of the glasses, the Y axis projecting up from the glasses, and the Z axis projecting into the head of the viewer. This is shown in Figure 1.

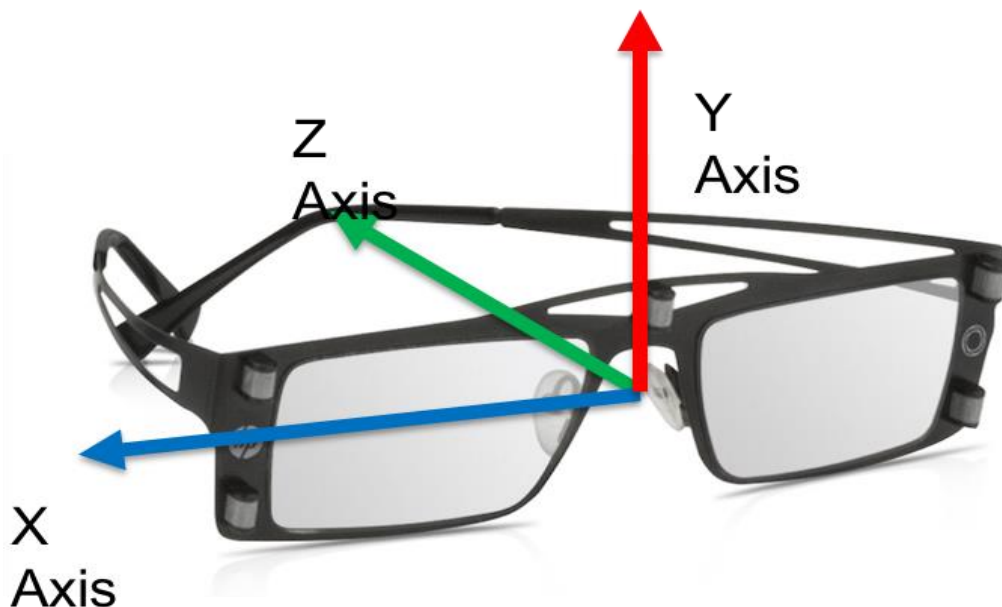


Figure 1: zSpace Glasses

This pose position can be transformed into a number of coordinate spaces. The Software Development Kit (SDK) uses the pose position to provide stereo information to the application.

Stylus

The stylus is also a tracked object. The pose position for the stylus is located at the front end of the stylus. Like the glasses, the stylus is oriented to a right-handed coordinate system, with the X axis projecting from the right of the end, the Y axis projecting up from the end, and the Z axis following along the stylus. This is shown in Figure 2.

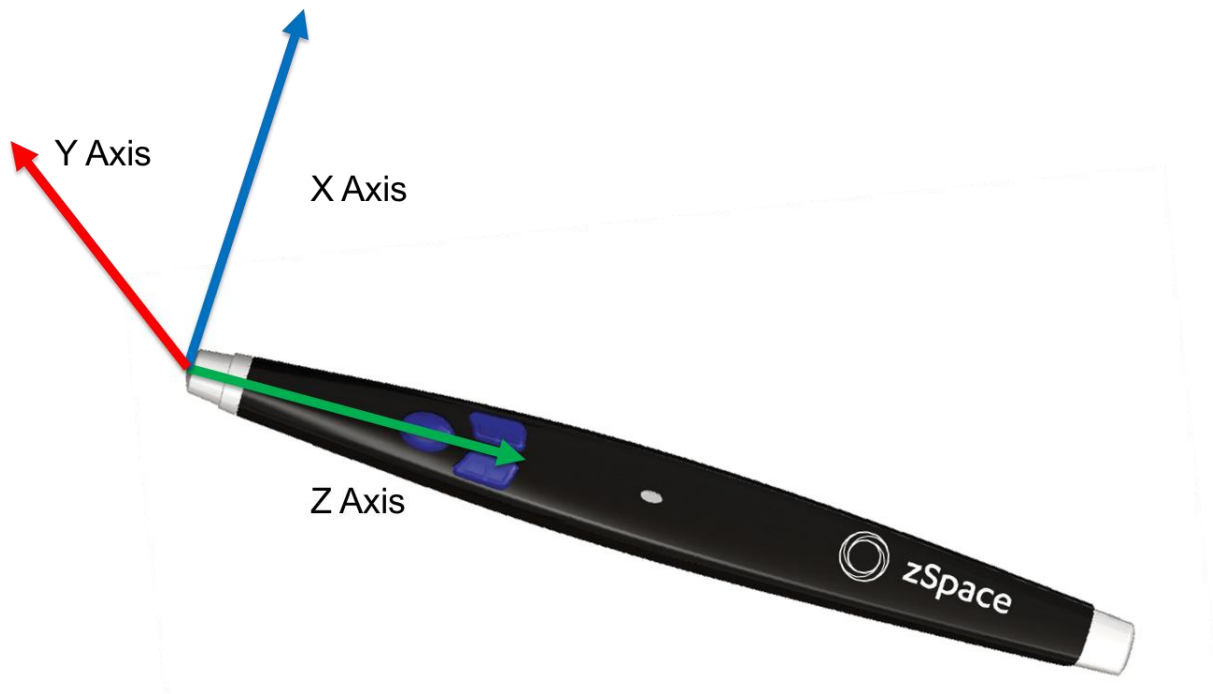


Figure 2: zSpace Stylus

This stylus pose can be transformed into a number of coordinate spaces.

Buttons

The stylus has buttons. The number of buttons and the state of each button can be queried by the application.

LED

At the center of the stylus is an LED light. This light can be controlled with the SDK. It can be turned on and off, and set to an arbitrary red, green, blue combination.

Vibration

The stylus has a built in vibration capability that can provide some simple haptic feedback to the user. The vibration can be turned on and off, and the vibration pattern can be programmed.

Tap

The stylus can also detect a tap event, where the stylus is tapped on the screen.

System Architecture

The overall architecture is shown in Figure 3.

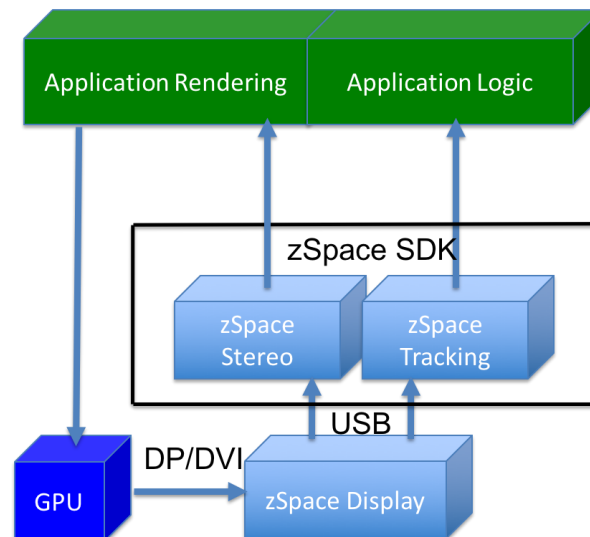


Figure 3: System Architecture

The zSpace system presents a standard quad buffered stereo display, which interacts with the operating system and graphics APIs. The stereo transforms are provided to the application through the SDK, but all rendering is performed by the application. That is why the display connection from the computer to the display is a standard DisplayPort or DVI connection from the GPU. More details on data generation are provided in the “SDK Native Programming Guide” and the “SDK Unity 3D Programming Guide.”

For tracking, most of the computation happens in the zSpace display. Those results are passed to the zSpace system software through a single USB connection to the computer. Some computation occurs on the computer (through the SDK), but the system load for this processing is negligible.

While this high level overview represents all zSpace systems and applications are presented with the same APIs, there are differences between the systems, which are articulated in the next section.

zSpace Systems

There are currently four different zSpace systems; zSpace 100, zSpace 200, zSpace 300 and HP Zvr. While the implementation in each system may be very different, this section focuses on the differences that matter to an application developer.

zSpace 100



Figure 4: zSpace 100

The zSpace 100 is the first zSpace system, and it is a display only system. An external computer is needed to drive the system. It has a fixed display angle of 30 degrees, and this cannot be changed. The stylus has three buttons, an LED light and vibration capabilities.

zSpace 200



Figure 5: zSpace 200

The zSpace 200 system is also a display only system. An external computer is needed to drive the system. Unlike the zSpace 100, this system continuously tracks the display angle of the system to provide angle awareness. However, the legs of the system lock it into a few predefined angles. The stylus has three buttons, an LED light and both vibration and tap capabilities.

zSpace 300



Figure 6: zSpace 300

The zSpace 300 is an all in one zSpace system. It continuously tracks the display angle of the system to provide angle awareness, and the system can be positioned at any angle. The stylus has three buttons, an LED light, both vibration and tap capabilities, and a built-in touchscreen.

The zSpace 300 also contains the display and tracking systems as well as an embedded computer. The computer specifications are as follows:

- Processor: Intel i3-4370
- Memory: 8 GB RAM
- Graphics: AMD FirePro W5170M GPU
- Storage: 500 GB HDD

Full specifications for this system can be found [here](#).

HP Zvr

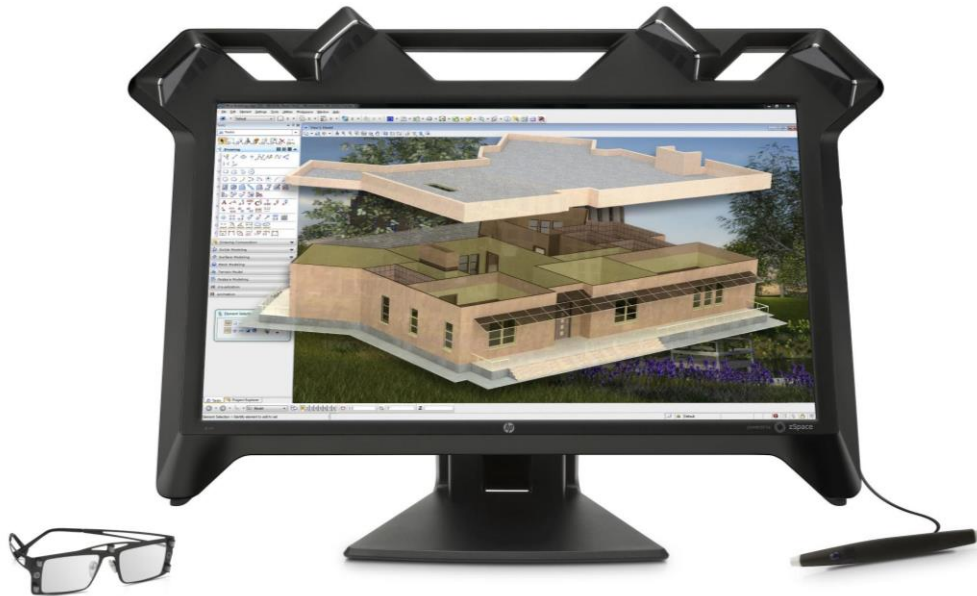


Figure 7: HP Zvr

The HP Zvr system is also a display only system. An external computer is needed to drive the system. This system continuously tracks the display angle of the system to provide angle awareness. The stylus has three buttons, an LED light, and both vibration and tap capabilities.

zSpace System Setup

This section covers zSpace system setup, specifically for development. Depending on the zSpace model, some steps may differ, and this is noted. Also, some steps can be performed before receiving a zSpace display system. For the zSpace 100, 200, and HP Zvr, the system and GPU requirements can be setup before the display arrives. You can also setup Visual Studio and Unity depending on the application. This can't be done on the zSpace 300, because it's an all in one system.

System and GPU Requirements

When developing on the zSpace 100, 200, or HP Zvr systems, certain system level and GPU requirements must be met. The details of these requirements are described in [this requirements document](#). Since the zSpace 300 is an all in one system, it meets all the operating system and GPU requirements for developing zSpace applications.

The most unique part of the requirements relates to the operating system and GPU selection. zSpace stereo currently uses quad buffered stereo, which means that the GPU must support it. Notably, some GPUs only support quad buffered stereo when the application is running in full screen mode. [This article](#) goes into the details of which GPUs can be used with the zSpace display system.

Note that the requirements for Unity3d development will be covered in the developing with Unity3d document.

System Check and Control Panel

Once you have set up your computer and connected the zSpace display, you can run the system check program. For the zSpace 100 and 200, follow the steps in [this article](#). For the zSpace 300 and HP Zvr, follow the steps in [this article](#).

Note that for the zSpace 100, 200 and HP Zvr, if you do not have GPU that supports windowed stereo, the stereo test in the system check may appear to fail. This does not mean the system isn't working properly. The system check does not run in full screen mode. Simply run one of the demo programs listed below to check if the system is running properly. If it's still not working, contact [customer support](#).

Once the system check is done, you can run demo applications on the zSpace system. These applications can be downloaded [here](#). Developers can also use the new zSpace App Manager to download and try applications. It can be found [here](#). The zSpace Experience has a demo, and that can be downloaded with App Manager. You can also try the following demos:

- [Angry Bots Demo](#)
- [Avogadro for zSpace 1.0 BETA](#)
- [CMU - Zimension Demo](#)
- [Hydrospace 1.0 Demo](#)
- [Siemens Teamcenter Visualization Mockup 10.1 for zSpace Installation Guide](#)
- [zSpace Cubemaze Releases](#)
- [zSpace Haptics Demo](#)
- [zSpace NASA Demos](#)
- [zSpace Rubik's Cube Demo](#)

The zSpace SDK includes additional sample programs. These will be referenced often in the "SDK Native Programming Guide." Now that you have a zSpace system setup for development, the next chapter will introduce Stereo and Stereo Rendering.

2: Stereo & Stereo Rendering

This document describes the basics of stereo and stereo rendering, and it provides the background for understanding the data that the zSpace SDK provides to applications.

Mono Frustums

In standard 3D graphics applications, the rendering camera uses a monoscopic frustum model as shown in Figure 8. A frustum is the 3D region visible on the screen.

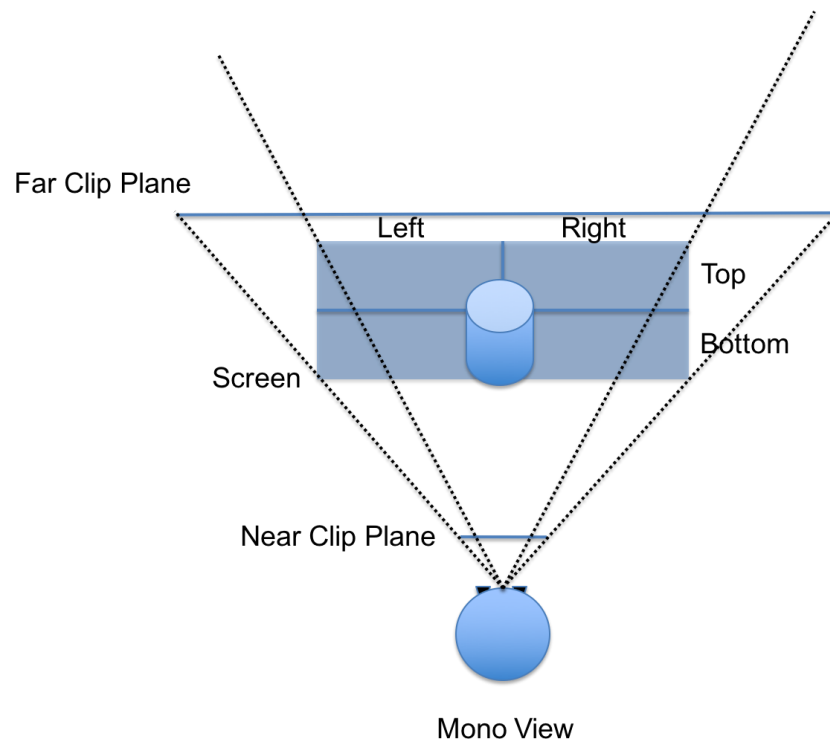


Figure 8: Mono Frustums

The frustum is composed of six values. Those values define the left, right, top and bottom bounds of the frustum, i.e., the screen viewing area. They also define the near and far clip planes. Notice that the left, right, top and bottom values are symmetrical to the centerlines of the screen. Sometimes the bounds can be specified by a field of view and aspect ratio. The tip of the frustum is located at the position of the virtual camera, as defined by the application.

Binocular Vision

The human visual system is not composed of a single camera. Humans see using two cameras (eyes), which generate two images that are fused into a single image by the brain. This is how we perceive depth. As a developer, we will use this processing in the brain to simulate the perception of depth in 3D applications. This is the basis for all stereoscopic 3D applications.

Stereo Frustums

To simulate what happens in the human visual system, we use two frustums instead of one. The diagram in Figure 9 shows this configuration in 2D.

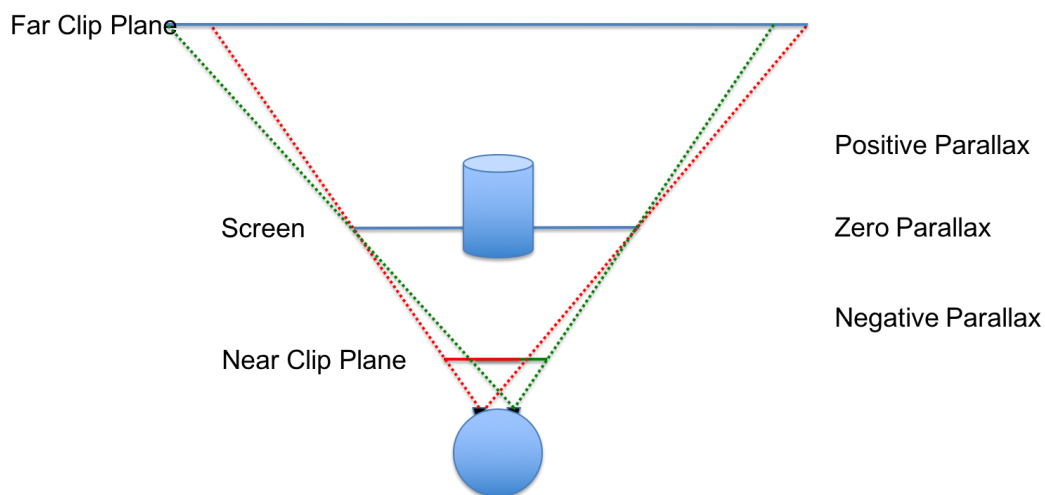


Figure 9: Stereo Frustums

The two frustums originate near the center of each eye and intersect at the edges of the screen. This configuration creates asymmetrical bounds at the screen. If you project the eye point on to the screen, you might see bounds like the one shown in Figure 10.

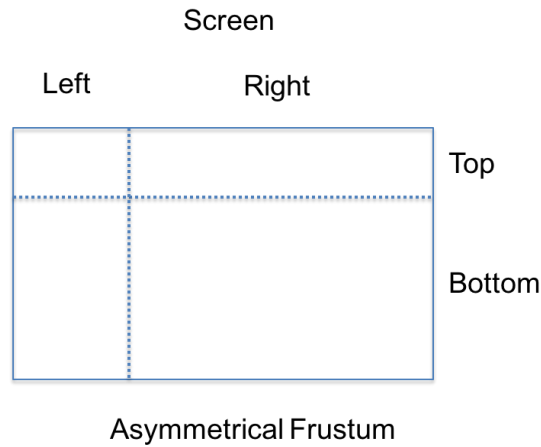


Figure 10: Asymmetrical Frustum

By rendering the scene twice using these frustums, we generate two images to display to the user. Objects in front of the screen are in negative parallax. Objects behind the screen are in positive parallax. Objects that are coplanar with the screen are at zero parallax. We will define these terms more concretely in the next section.

Parallax/Disparity

Two common terms often used in stereoscopic processing are parallax and disparity. They refer to the same phenomenon in the human visual system. Figure 11 helps show these terms.

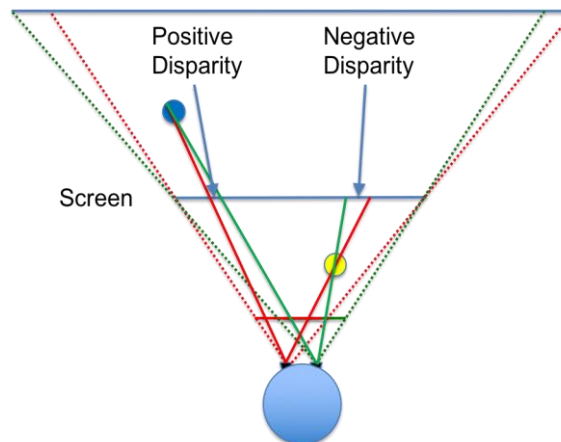


Figure 11: Positive & Negative Disparity

By using stereo frustums, the points of an object may project onto the screen in more than one location. If the object originates behind the screen, the left eye projects to the left of the right eye point. Their projected rays do not cross. This is referred to as uncrossed disparity. The distance between the two projected points on the screen measures the disparity. Since the projected rays do not cross, this distance is positive. This is why it is called positive disparity or parallax. The dark blue sphere in the figure depicts this case. When the brain processes images with positive disparity, it perceives that the objects are behind the screen.

If the object originates in front of the screen, the left eye projects to the right of the right eye point. Their projected rays do cross. This is referred to as crossed disparity. The measure of disparity is still the distance between the projected points on the screen. But, since the projected rays cross, the distance is negative. Hence, this is called negative disparity or parallax. The yellow sphere in the figure depicts this case. When the brain processes images with negative disparity, it perceives the objects as being in front of the screen.

If the object projects to the exact same point on the screen for both the left and right eyes, this distance is zero. This is referred to as zero disparity or parallax. These objects will have no stereo effect, and therefore will look like they are coplanar with the screen. In other words, they won't be perceived by the brain to have any depth.

Stereo Comfort

We can measure the disparity of an object given its location in the world, the position and orientation of the screen, and the position and orientation of the viewer. When the human visual system encounters disparity in the images that it captures in the eyes, it uses pattern matching to calculate the disparity and therefore estimate depth based on these disparities. We use this trick in stereo rendering to get the human visual system to perceive depth when in reality there is no depth. Both images are displayed on a flat plane at relatively the same distance from the eyes. While this can be a very compelling illusion, it uses the eyes in a non-standard way, which can cause discomfort.

When we see objects in the real world, there are two systems at work. One part of the visual system focuses on the object to bring more clarity (and data) for processing. This is similar to focusing a camera lens on an object. This is called accommodation. Another system uses the muscles in the eyes to converge or orient the eyes towards the object of interest. This is called vergence. In the real world, these two systems work together, each influencing the other to get the best possible result.

In stereo rendering, we sometimes strain these systems. We are simulating objects at various depths. The vergence system wants to orient the eyes towards the perceived depth of the objects. The accommodation system, however, is continually focused at the screen. This discrepancy is a well-known and researched problem, but not completely solved. It is called the vergence accommodation conflict. There are ways to set up your scene to minimize this, and we have written a document that describes

various approaches. It can be found [here](#). We have also included features in our SDK, for both native and Unity 3D, to help applications compute disparity and locate objects at an appropriate distance from the viewer to minimize this issue.

Head Tracking

While stereo rendering helps us present virtual objects as if they exist in the real world, it is not fully immersive. For example, 3D movies are an enjoyable experience, but they are not completely immersive. The feature that makes stereo truly immersive is head tracking.

Head tracking is the process of monitoring and providing the position and orientation of the viewer's head to the stereo system. With this information, the system can do a much better job of simulating reality. It does this by dynamically adjusting the stereo frustums based on the viewer's head position.

Head Mounted Display (HMD) versus Fish Tank VR

There are two main classifications of Virtual Reality (VR) systems. The basic difference between the two is whether or not the screens are attached to the viewer's head. There are many new systems emerging, and most of them are ones where the screen is attached to the head. These are called Head Mounted Displays (HMD). The Oculus Rift and Sony Morpheus are examples of these.

In other types of VR systems, the screens are fixed in the room, and the viewer's head moves around. Essentially, the screens are windows into a virtual world. This includes systems like zSpace and immersive CAVE systems, where there are multiple large screens connected together. These systems are often called Fish Tank VR systems, because looking through the screen is like looking through a fish tank.

If we look at the frustums in these two scenarios, we can see that the platform differences change how the applications adapt to them. Figure 12 shows a diagram of a head mounted case.

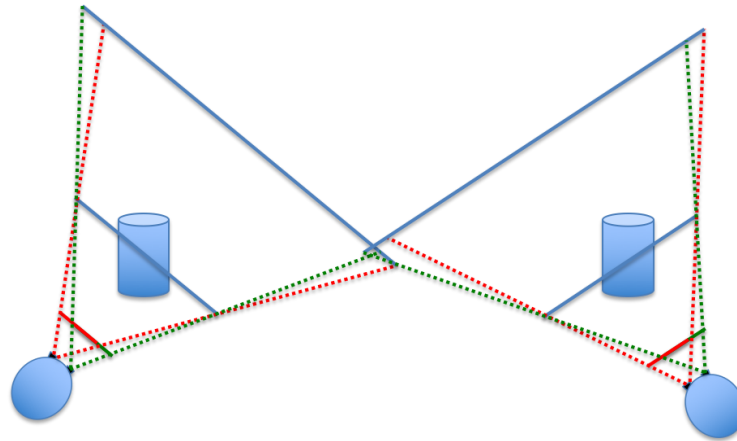


Figure 12: Head Mountain Display Example

In the case of HMD systems, the frustums are actually static when the system starts up. Head tracking simply changes the position and orientation of the virtual camera.

Figure 13 shows the frustums in the case of Fish Tank VR systems.

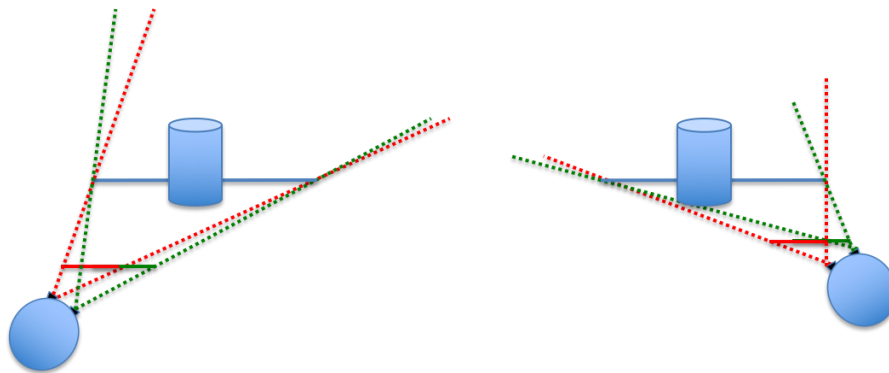


Figure 13: Frustums in Fish Tank VR

In this Fish Tank VR system, the screens are static, and the head tracking information changes both the virtual camera position/orientation as well as the frustum bounds. This is how the zSpace system works. The “SDK Native Programming Guide” describes how this information is presented to the application.

In sum, HMD display systems have a fixed frustum, and the head tracking information modifies only the virtual camera position and orientation. In Fish Tank VR systems, head tracking information modifies both the virtual camera position/orientation and the frustum. zSpace primarily uses fish tank technology. From a developer’s standpoint, this means that the frustum changes with every frame.

Rendering Stereo Overview

This section describes the architectural considerations for converting an application from monoscopic rendering to stereoscopic rendering. First, we will look at the buffer configurations that are used in zSpace and other common stereo configurations. Then, we will look at how virtual camera transforms and projections need to be modified for zSpace support.

Rendering Buffers

In standard 3D applications, there is a single back buffer and single front buffer. In stereo rendering, there are two back buffers and two front buffers - one for each eye. There are three basic buffer configurations used by stereo systems.

HMD Buffers

The most common buffer configuration used by HMD systems is shown in Figure 14.

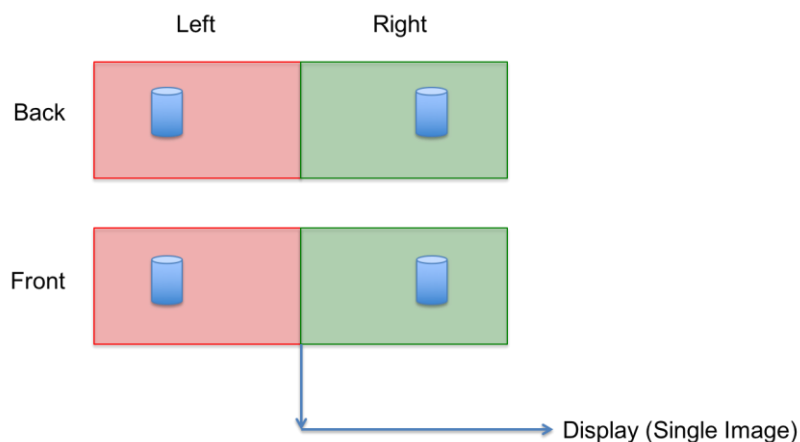


Figure 14: Common Buffer Configuration

HMD stereo rendering configures one large back buffer and virtually divides it in half during rendering. The left half is the left eye image, and the right half is the right eye image. It is configured this way because HMDs actually display one continuous buffer simultaneously. The system then uses lenses to focus on the buffer assigned to each eye. One of the benefits of this model is that the Graphics Processing Unit (GPU) doesn't have to know anything about left or right eye rendering. It is all handled by the rendering system used by the application. However, this system can only render in full screen mode. Windowed stereo is not possible.

Side-by-Side Stereo

Another buffer configuration, similar to HMD, is called side-by-side stereo. This is primarily used by TV stereo systems. The diagram of this configuration is shown in Figure 15.

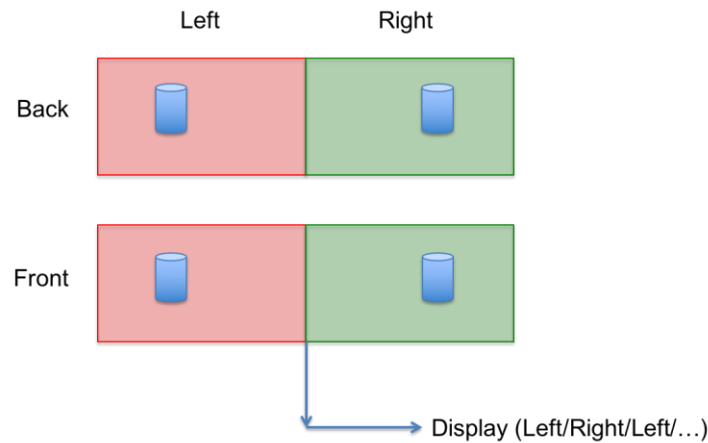


Figure 15: Side-by-Side Stereo

Similar to HMD buffers, there is one continuous buffer that contains both the left and right stereo images. One difference is that the resolution of these buffers is tightly defined by the resolution of the display device. The overall resolution of the device is divided exactly by two, and that defines the horizontal resolution supported for each eye. In the case of HMDs, they often scale up the resolution for the best possible post-lens corrected resolution. So, you get the correct aspect ratio resolution for HMD devices. Another difference between side-by-side and HMD is that side-by-side stereo usually alternates between displaying left and right images sequentially to the viewer. This is somewhat dependent on the eyewear being used, but most systems use a time sequential display technology.

Side-by-side stereo means that the GPU does not have to be stereo aware, which is a benefit. The downside is that each eye only gets half of the full resolution of the display. Side-by-side stereo is also a full screen only stereo mode. Note that there are other formats that act like side-by-side, but use a different configuration. Top/bottom stereo is like side-by-side, except it cuts the vertical resolution in half.

Quad Buffered Stereo

Quad buffered stereo is the buffer configuration used by zSpace. Its configuration is shown in Figure 16.

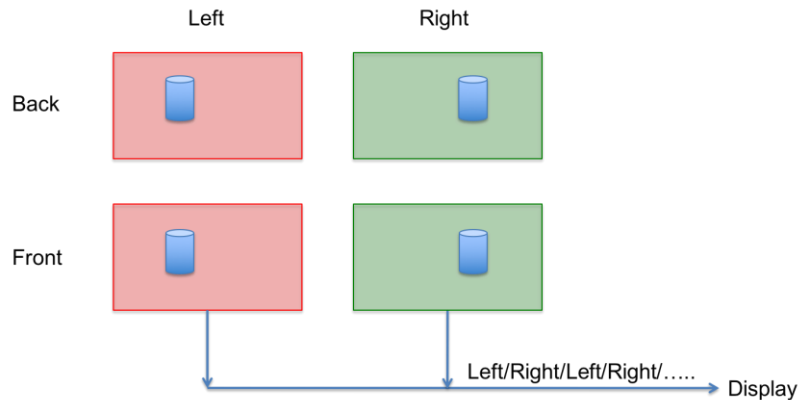


Figure 16: Quad Buffered Stereo

In this configuration, the left and right buffers are logically separated, and the GPU knows that the buffers exist and are separate. This means that the buffers need to be allocated and managed by the GPU. The application renders into the buffers, synchronizing with the GPU, and then presents the buffers for display. The GPU then sends the buffers to the display in a time sequential fashion, presenting each image to the user. With this configuration, windowed stereo is possible and each eye will get full 1080p resolution. But the GPU needs to know that this configuration is being used and as such, it has to be configured correctly.

Rendering Camera Attributes

Once you have set up the rendering buffers correctly for stereo rendering, you can modify the camera attributes. As demonstrated, monoscopic rendering places the origin of the frustum at the applications virtual camera point. When processing head tracking for rendering, the origin of the frustums are near the center of each eye. There are several attributes that we need to process to correctly configure the system. We need to take into account the position and orientation of the head, the interpupillary distance (the distance between the center of the pupils of the two eyes), the offset from the glasses to the eyes, and the position of the screen. Each of these is described in the following section.

Head Position

The zSpace system tracks the center of the glasses in real world units. So the current head position and the zSpace display define the off axis frustums used for rendering. Figure 17 shows the frustums.

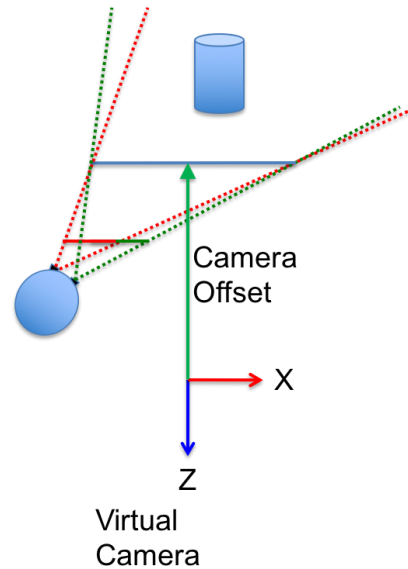


Figure 17: Frustum & Head Position

It is important to note the relationship between the applications virtual camera and the zSpace display. The zSpace system defines a camera offset vector at the position that the virtual camera is oriented, which is in the Z axis direction. This vector is pointing at the center of the applications rendering window. Most of the zSpace systems can be rotated at an arbitrary angle, so the screen is also rotated by a display angle, which can be queried from the system. By using this vector and display angle, we can dynamically compute the exact position and orientation of any geometry that we want to be coplanar with the display. This is useful for things like 2D user interface elements. Both native and Unity 3D samples explain how to achieve this.

Interpupillary Distance and Glasses Offset

The head position and orientation is not enough information to construct the left and right frustums for rendering. You need to define attributes that describe the glasses and head position. These are shown in Figure 18.

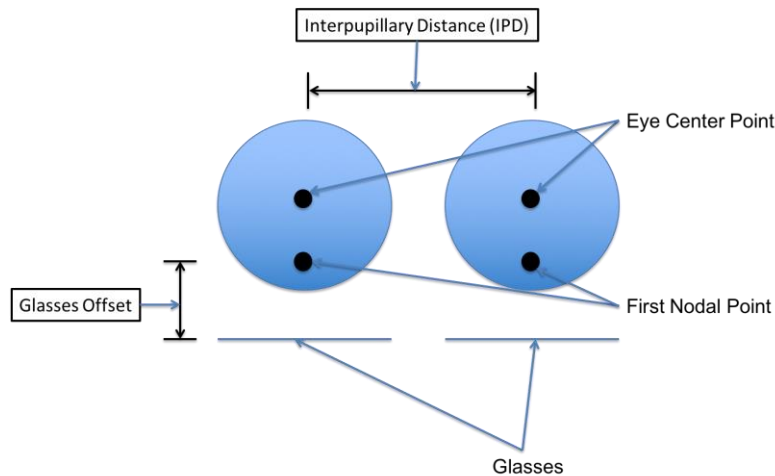


Figure 18: Interpupillary Distance and Glasses Offset

To compute the correct values for the left and right frustums, you need to account for two attributes. First, use the interpupillary distance to offset the center point of the glasses. Next, move the origin point of the frustum to the first nodal point of the eyes. This moves the frustum away from the glasses lenses. This is called the glasses offset. Both the interpupillary distance and the glasses offset can be queried, and modified, with the zSpace SDK. With the head position and orientation and these two attributes, we can correctly construct the left and right frustums.

View Transform and Projection

The zSpace system does all the calculations described in the previous section and makes it available to the application in a number of different formats. The information is split into two parts: a view transform and a projection. The view transform represents the transform needed to orient the virtual camera so that it lines up correctly with the viewer's physical head position and orientation. It is currently represented as a 4x4 matrix. This transform should be concatenated with the transform that the application uses to position and orient the virtual camera.

The projection represents the off axis projection needed to render the stereo image correctly. The projection can be retrieved in two different ways. It can be retrieved as a 4x4 matrix. This allows OpenGL applications to use it directly as their projection matrix. It can also be retrieved as top, bottom, right, left, near and far bounds values. This is for applications that customize the format of their projections.

The system supports the left or right eye position in different coordinate spaces. This allows alternate rendering technologies, like real time ray tracing, to get the eye positions to construct their own appropriate frustums. The “SDK Native Programming Guide” and the “SDK Unity 3D Programming Guide” explain how to get the view transform and projection information from the zSpace system.

3: Coordinating Space & Scale

When developing for the zSpace platform, you should be familiar with coordinate systems so you can render objects correctly. The zSpace system manages these coordinate systems, and in the case of head tracking, it is transparent to you. But for interacting with the stylus, you need to understand the coordinate spaces used to ensure proper visualization and use of the stylus.

Scale is related to coordinate systems, and all 3D applications choose a scale. zSpace does all of its processing in the real world physical scale. In order to implement the desired experience, you should understand the relationships between the zSpace scale and the application scale, and how they behave in Fish Tank VR systems.

Camera Space

Camera space is the coordinate system defined by the virtual camera in the application. Figure 19 shows how camera space relates to the zSpace display.

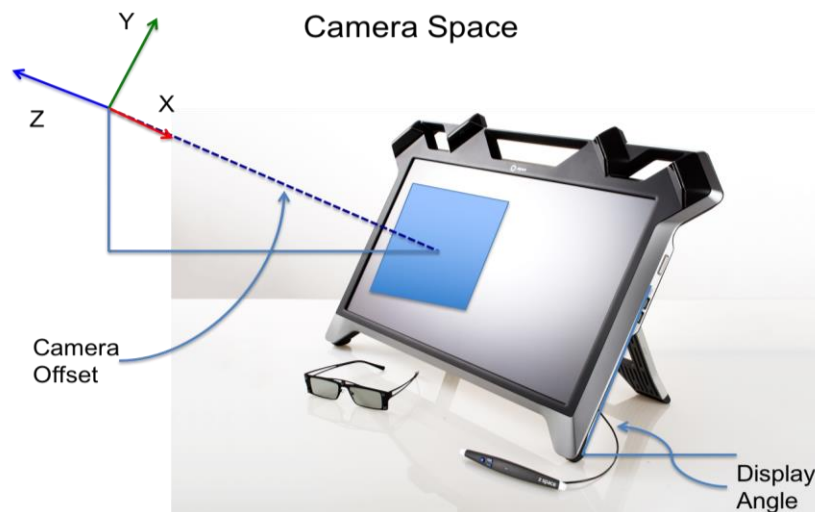


Figure 19: Camera Space

The window that the application is rendering into is called the viewport. The origin of camera space is at the position of the application's virtual camera. The viewport on the zSpace screen is a specific distance

away from the virtual camera, and positioned in the direction that the camera is oriented. The virtual camera points to the center of the viewport. If the application is a full screen application, then the virtual camera points to the center of the screen. The distance from the virtual camera to the screen can be calculated by using the zSpace camera offset and display angle. That calculation is presented in the “SDK Native Programming Guide” and the “SDK Unity 3D Programming Guide.” It is important to know this distance when trying to position an object at or near the zero parallax plane.

Viewport Space

Viewport space is a coordinate system with its origin on the zSpace display. The X axis is parallel to the bottom of the screen and extends to the right. The Y axis is parallel to the edge of the screen and extends up. The Z axis is perpendicular to the screen and extends out of the screen.

The viewport space is shown in Figure 20.

Viewport Space



Figure 20: Viewport Space

World Space

World space is a coordinate system that is only known to the application. zSpace does not know how the application has defined the transform from the virtual camera space to application world space. When applications want to use the stylus, they often want to use the stylus in world space. To get any pose in

world space, the application must get the pose from zSpace in the camera space, and then transform it into world space using its own internal camera. The native SDK and Unity 3D documents show samples of how to calculate this transform.

Display Space

Display space is similar to viewport space, but it is centered at the center of the applications rendering window. (The display space is centered at the center of the display. If the center of the rendering window is at the center of the display, then they are identical.) Its orientation is the same as viewport space. When the application is running in full screen mode, display space and viewport space are also identical. Most applications should use viewport space and will not need to use display space.

Figure 21 illustrates display space.

Display Space

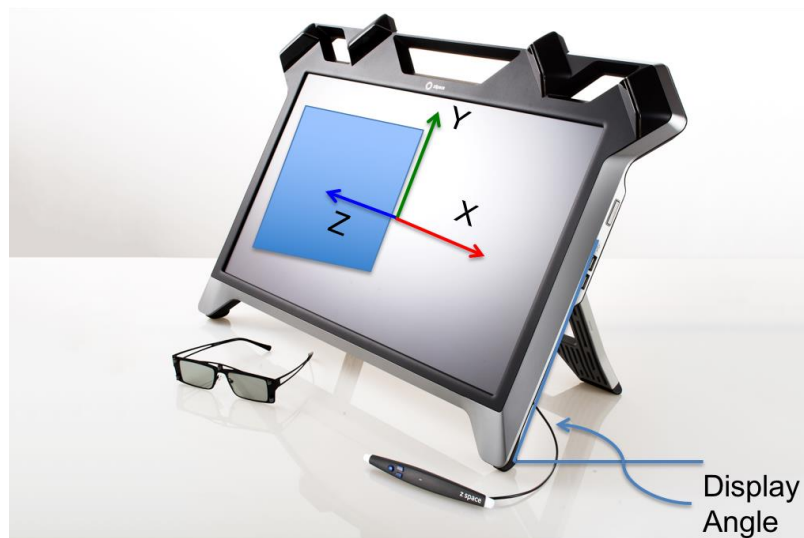


Figure 21: Display Space

Tracker Space

Tracker space is the coordinate system in which the raw tracker data is reported. Its origin is the center of the screen. The X axis is parallel to the physical ground and extends along the right of the screen. The Y axis is perpendicular to the physical ground and extends up towards the sky. The Z axis is parallel to the physical ground and extends out of the screen. This coordinate space is used internally, and most applications will not need to use the tracker space.

Figure 22 illustrates tracker space.

Tracker Space



Figure 22: Tracker Space

Viewer Scale

Scale is the last general topic that we need to cover before diving into the details of programming for zSpace. All of zSpace applications process data in a real world scale where 1.0 is one meter. All attributes, transforms, and projections are assumed to be in this scale. This is necessary to properly compute frustums and all the appropriate data as it occurs in the real world. This ensures that frustums line up with the viewer's head, and any virtual objects tracking the end of the stylus match correctly as well. Many 3D applications have been created with a scale that is most convenient for the application, and does not necessarily match the real world or the scale of zSpace. To make it easier for these applications to port to zSpace, viewer scale can be used.

The viewer scale attribute is a multiplier to the frustum. As an example, the default width of the zSpace screen is 0.521 meters. By setting the viewer scale to 10.0, the new effective width of the display is 5.21 meters. The viewer scale also effectively moves the head position back by this scale factor. This is important to note when placing geometry at the zSpace display. The distance will vary based on the viewer scale.

When Fish Tank VR simulates reality in an undesirable way, the viewer scale can help.

If you are looking at an object that is far away from the screen, as you move your head, it appears to move across the screen. This is actually physically correct. You can look out an office window at a faraway object and move your head, and you will see this phenomenon. By increasing the viewer scale, this effect can be minimized.

Figure 23 shows one example of this.

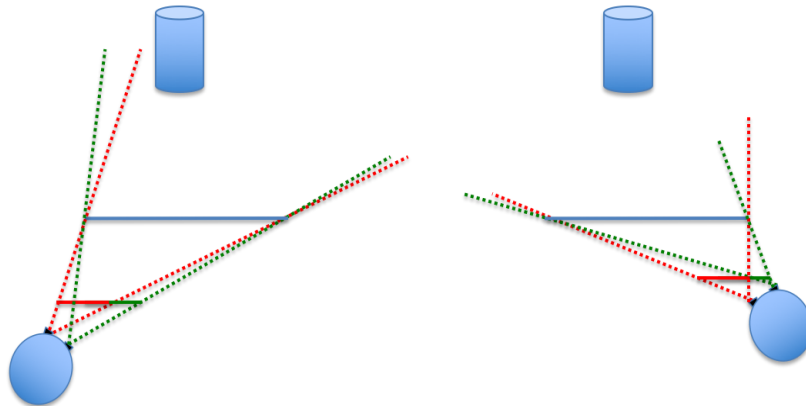


Figure 23: Viewer Scale

The “SDK Native Programming Guide” and the “SDK Unity 3D Programming Guide” present viewer scale effects on the scene being rendered. They also present examples of the effects of modifying the viewer scale. Those documents provide the developer’s next steps.