

# zSpace Addendum:

# Unity Curricular Framework

Created for Teachers

---

## Table of Contents

### [Where To Integrate Zspace Units](#)

#### [Zspace Unit 1: Zspace Introduction](#)

##### [1.A: Unit Overview](#)

###### [1.A.1: Unit Description](#)

###### [1.A.2: Major Topics](#)

###### [1.A.3: Learning Objectives](#)

###### [1.A.4: Materials](#)

##### [1.B: Course Outline](#)

##### [1.C: Learning Activities Guide](#)

###### [1.C.1: How To Zspace](#)

###### [1.C.2 Unity Curriculum 1: Zspace Introduction](#)

##### [1.D: Standards Alignment Guide](#)

###### [1.D.1: Professional Standards For Interactive Application And Video Game Creation](#)

###### [1.D.2: Common Core State Standards \(Ccsc\)](#)

###### [1.D.3: Stem Career Clusters \(Sc\)](#)

###### [1.D.5: Next Generation Science Standards \(Ngss\)](#)

##### [1.E: Assessment Reference Guide](#)

###### [1.F: Suggested Resources](#)

#### [Zspace Unit 2: Zspace Stereo Mechanics](#)

##### [2.A: Unit Overview](#)

###### [2.A.1: Unit Description](#)

###### [2.A.2: Major Topics](#)

###### [2.A.3: Learning Objectives](#)

[2.A.4: Materials](#)

[2.B: Course Outline](#)

[2.C: Learning Activities Guide](#)

[2.C.1 Unity Curriculum 2: Zspace Stereo Mechanics](#)

[2.D: Standards Alignment Guide](#)

[2.D.1: Professional Standards For Interactive Application And Video Game Creation](#)

[2.D.2: Common Core State Standards \(Ccss\)](#)

[2.D.3: Stem Career Clusters \(Scc\)](#)

[2.D.5: Next Generation Science Standards \(Ngss\)](#)

[2.E: Assessment Reference Guide](#)

[2.E.1: Assessment Rubric](#)

[2.E.2: Assessment Of Learning Objectives](#)

[2.F: Suggested Resources](#)

## [Zspace Unit 3: Zspace User Interface And Interaction](#)

[3.A: Unit Overview](#)

[3.A.1: Unit Description](#)

[3.A.2: Major Topics](#)

[3.A.3: Learning Objectives](#)

[3.A.4: Materials](#)

[3.B: Course Outline](#)

[3.C: Learning Activities Guide](#)

[3.C.1 Unity Curriculum 3: Zspace User Interface And Interaction](#)

[3.D: Standards Alignment Guide](#)

[3.D.1: Professional Standards For Interactive Application And Video Game Creation](#)

[3.D.2: Common Core State Standards \(Ccss\)](#)

[3.D.3: Stem Career Clusters \(Scc\)](#)

[3.D.5: Next Generation Science Standards \(Ngss\)](#)

[3.E: Assessment Reference Guide](#)

[3.E.1: Assessment Rubric](#)

[3.E.2: Assessment Of Learning Objectives](#)

[3.F: Suggested Resources](#)


















## Welcome to the zSpace addition to the Unity Certification Program!

We are excited you've chosen to add learning about programming in zSpace to your Unity curriculum. This guide will support you as you integrate key components of development for zSpace into Unity's Curricular Framework.

The chapters your students will utilize during the course are also included in this guide for your reference.

If you have questions about this document or as you are supporting your students in development, please reach us at [profdev@zspace.com](mailto:profdev@zspace.com) or through our support team at [877-977-2231](tel:877-977-2231)

## Overview: Where to Integrate zSpace Units

	Unit 1: Introduction to Game Design
	Unit 2: Critical Thinking in Game Design
	Unit 3: Game Design Theory
	Unit 4: Story and Game Creation
	Unit 5: System Dynamics and Scripting Fundamentals
	Unit 6: Game Development Tools, Functions, and Properties
	Unit 7: Interfaces, Environments, Asset Management, and Animation
	Unit 8: Physics and the Build Process
	<b>zSpace Unit 1: zSpace Introduction</b>
	Unit 9: Constructs of Game Design
	Unit 10: Principles of Cameras and Lighting in Game Environments
	<b>zSpace Unit 2: zSpace Stereo Mechanics</b>
	Unit 11: Principles of Sound and Audio for Gamers
	Unit 12: Strategic Game Development Techniques and Concepts
	<b>zSpace Unit 3: zSpace User Interface and Interaction</b>
	Unit 13: Principles of Quality and Functionality Assurance in Game Development
	Unit 14: Principles of Versioning and Game Release

# zSpace Unit 1: zSpace Introduction

Begin this zSpace unit after completing Unit 8 of Unity's Curricular Framework.

## 1.A UNIT OVERVIEW

### 1.A.1 UNIT DESCRIPTION

In this unit, students will be introduced to the zSpace components, terminology and core concepts. Students will identify all the hardware components of a zSpace and begin to understand how pose positions allow for tracking a user's position in a 3D environment. Students will learn about zSpace interactions and identify how a user can manipulate objects in stereoscopic 3D. This is a knowledge-building unit that does not include any programming work.



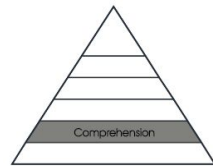

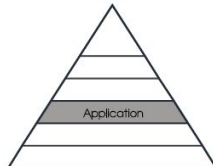

### 1.A.2 MAJOR TOPICS

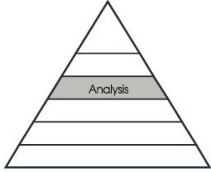

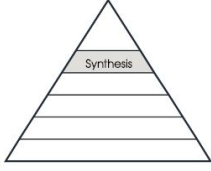

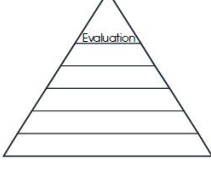

In this unit, learners will explore the following topics:

- Introduction to zSpace components
- Introduction to stereo concepts and aesthetics
- Introduction to user interface concepts
- Introduction to zView (optional)

### 1.A.3 LEARNING OBJECTIVES

By the end of this unit, learners should be able to perform the following tasks:

	Blooms Domain	Learning Objective	Level of Difficulty
1			
2		Identify each component of a zSpace system Identify and explain the coordinate systems for the zSpace eyewear and stylus Describe the importance of stereo display and angle awareness	
3			

4		Understand the significance of zSpace-specific user interface concepts and distinguish between application-level controls and the application scene	
5			
6			

#### 1.A.4 MATERIALS

- Full hardware specifications for each of the zSpace systems:  
<https://support.zspace.com/hc/en-us/categories/200498489-zSpace-Hardware>

#### 1.B UNIT OUTLINE

- Components
  - Stereoscopic 3D Display
  - Eyewear
  - Stylus
  - Mouse
  - Keyboard
  - Other Peripheral Devices
- Stereo Concepts and Aesthetics
  - Stereo Display
  - Angle Awareness
- User Interface Concepts
  - zSpace Interaction
  - User Interface
    - Application-Level compared to Scene
    - Application-Level Controls
    - Left and Right-Handed Layouts

## 1.C LEARNING ACTIVITIES GUIDE

This section provides a guide for delivering the unit content. When reviewing content in this unit, important questions to consider may include:

- What learning experiences can your learners engage in during this unit?
- How can you integrate formative assessments into these learning experiences?
- How can you integrate formative assessments into the tangible deliverables (e.g. documents, projects, tests applications, game builds) that your learners produce?
- How can you integrate summative assessments towards the end of this unit?

As these can be challenging questions, this section will provide resources and recommendations to help you determine the appropriate answers.

### 1.C.1 How to zSpace

This activity will allow learners to get acquainted with the zSpace system. This exercise will also introduce them to the components, stereo concepts, aesthetics and user interface concepts of zSpace. While this is a brief exploration, it will set the stage for the learners on how to use the zSpace system.

Introduce learners to this topic by having them watch the following tutorials:

- Getting Started: How to: zSpace300  
<https://edu.zspace.com/info/getting-started>
- Getting Started with zSpace Course  
<https://edu.zspace.com/learn>

### 1.C.2 Unity Curriculum 1: zSpace Introduction

#### Introduction to zSpace

zSpace creates a mixed reality computing experience that is immersive, interactive, and lifelike. The zSpace platform allows developers and users to interact with computer-generated objects in a three dimensional holographic-like environment.

Seeing objects and scenes in 3D is the most natural and efficient way to understand the complex spatial relationships of the world around us. zSpace presents computer-generated imagery in the same dimensional way that we see the world. This creates visual excitement and an instant understanding of complex structures that would otherwise be unfathomable if viewed on a traditional 2D screen.

## Components

zSpace hardware technology is currently available in two forms:

1. All-in-one Systems, combining zSpace technology with integrated CPU, GPU, and operating system.
2. Stand-alone Displays, including zSpace technology, connected to a user supplied system.

## Stereoscopic 3D Display

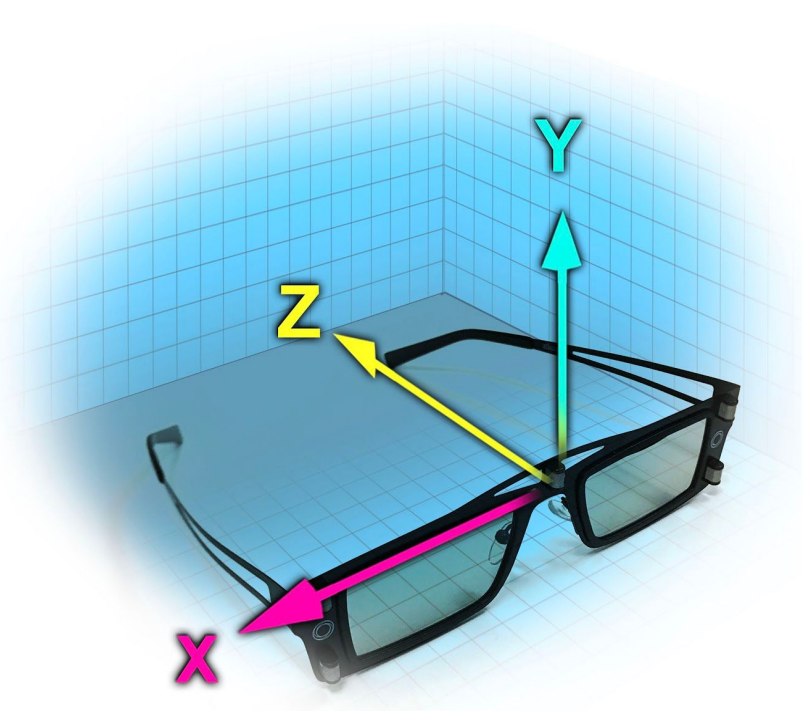
zSpace features a high-definition (1080p HD) 3D/2D computer display, featuring advanced VR/AR tracking systems and angle-awareness sensor.

## Eyewear

The zSpace platforms have 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 system calculates the 3D position and orientation of each object, and tracks the objects 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. 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 back toward the viewer's head. See the following figure.

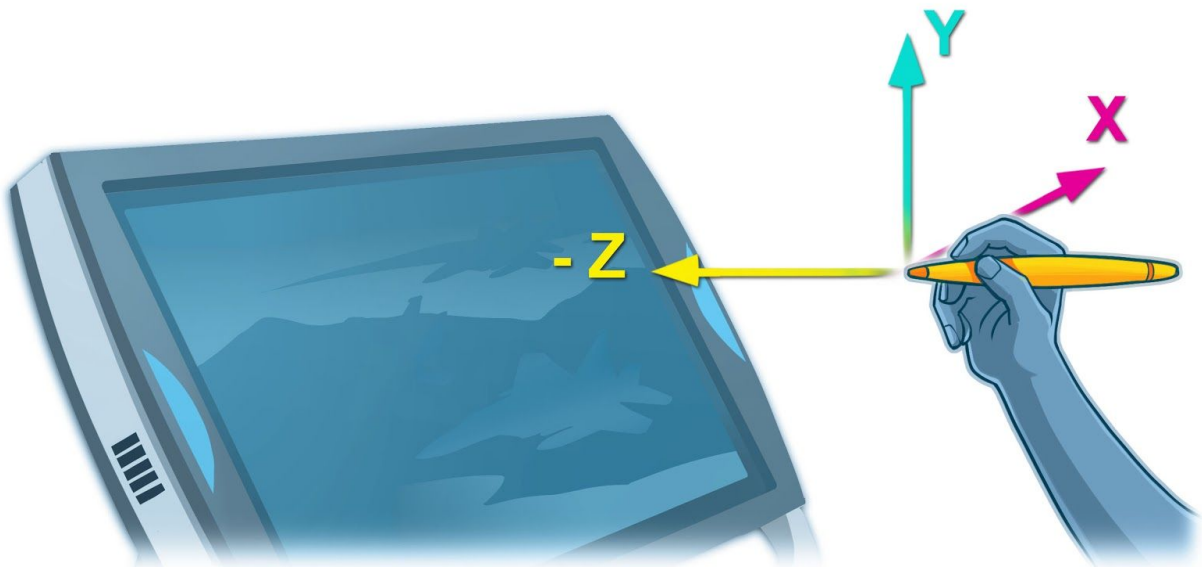
The system can transform this pose position into a number of coordinate spaces. The SDK uses the pose position to provide stereo information to the application.





## Stylus

The system also tracks the stylus. The pose position is located at the front tip of the stylus. Like the glasses, the stylus is oriented to a right-handed coordinate system. The X axis projects from the right of the tip, the Y axis projects up from the tip, and the Z axis runs back along the stylus. The system can transform the stylus pose into a number of coordinate spaces.



The stylus features include three buttons, LED, vibration, and tap sensitivity. All input and haptics can be controlled using the SDK.

## Mouse

Standard 3-button mouse with scroll wheel.

## Keyboard

A full-sized 78 key, space-efficient keyboard. The keyboard includes function keys and supports configuring shortcuts.

## Other Peripheral Devices

You can attach peripherals to complement zSpace applications such as trackballs, inertial-sensor gamepads, and so on. Your application must be programmed to support them.

## Stereo Concepts and Aesthetics

zSpace applications use stereoscopic display to deliver augmented reality. In order to provide the best viewing experience, applications need to support display requirements and follow certain design guidelines.

## 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 at <https://support.zspace.com/hc/en-us/categories/200498489-zSpace-Hardware>.

## Angle Awareness

Angle awareness is a feature you can turn on or off. When on, the display is aware of the direction of real-world gravity, which can help you to align virtual gravity with it. This adds to realism in games which use physics or otherwise should feel aligned to the user's desk. When this feature is off, the world alignment is not based on real-world gravity, making the content in some regard attached to the screen.

## User Interface Concepts

zSpace applications are more engaging if they build on user experience and knowledge. Objects rendered in VR/AR should mimic the real world, though this need not limit creative or magical experiences.

A computer-generated stereoscopic 3D environment is richer and more complex than a traditional 2D computer display. Design with this space in mind. Keep the user's focus on your primary visual entity. Minimize background and peripheral distractions. The design should be only as complex as necessary for the user to accomplish the task at hand.

## zSpace Interaction

Everyone knows how to handle objects in the real world, and most people know how to use computers. When implementing a zSpace application, take advantage of both types of acquired knowledge to make VR/AR easier and more intuitive.

zSpace lets people directly manipulate objects in stereoscopic 3D. For example, you can rotate, reposition, resize, or zoom an object. Using zSpace, computer interaction becomes intuitive because we already know how to move objects around in the 3D world.

Use the zSpace stylus to directly manipulate objects whenever possible. The stylus supports a wider range of movement than the mouse.

You can add constraints to make it easier for the user to complete specific tasks. In some cases, you may constrain input to fewer degrees of movement, such as limiting input to the x and y axes. Another constraint could be a snap-to-grid setting. You can let the user enable and disable constraints.

You can give users a sense of stability by providing a user interface that meets their expectations. To achieve this:

- Provide a consistent set of menu items and UI controls. If a menu item is not in use at a particular time, change it to appear disabled (grayed out) rather than hiding it completely. You can also use context menus. This matches user expectations based on 2D UI experiences.
- Save the user application settings from one session to the next. Users are more comfortable and efficient if the application looks the same each time it opens.

### User Interface

User interface controls are easier to read and use when rendered in 2D rather than in stereoscopic 3D. Users have more experience with controls in 2D and the physical interaction is less complex.

Most application scenes and objects need to appear in stereoscopic 3D. It is more powerful to model the real world and leverage the user's real world experience with 3D.

### Application-Level compared to Scene

To implement a zSpace user interface, you must distinguish between application-level controls and the application scene or content.

Application-level controls are top-level functions such as exit, file management, and navigation. Place application-level controls at zero parallax to benefit from the advantages of 2D:

- Text is easier to read in 2D.
- Viewing comfort is maximized at zero parallax.
- UI controls are easier to interact with at zero parallax.

Disadvantages are:

- 3D content will not render properly and comfortably with some 2D UI frameworks, since these frameworks lack correct depth positioning relative to 3D content.

The application-level controls are the equivalent of a HUD (head-up display). Like a HUD in a computer game, the controls may either be permanently displayed or temporarily displayed as needed.

The scene is the core of your application, containing your 3D content. For your scene, your application should take advantage of the strengths of the zSpace system: a realistic stereoscopic 3D display, complete with head tracking and direct manipulation via a stylus.

## Application-Level Controls

The zSpace design guidelines suggest three types of application-level controls: an application control bar, palette, and inspector.

The application control bar is the application's main control and contains all of the application-level commands. It should contain commands for file operations, preferences, and exiting the application. The application control bar appears at the bottom of the screen.

The palette contains stylus tools, or modes, such as selection, scale, and the camera path tool. The palette appears on the right side of the screen.

The inspector shows the details of an object. The inspector updates as the user changes an object in the scene. The inspector also appears on the right side of the screen.

All applications require the application control bar. Most applications also need a palette, but only some applications use an inspector.

## Left and Right-Handed Layouts

The suggested layout of the palette and inspector are best for right-handed users. With the controls on the right, the user can reach them more easily and the controls are not blocked by the position of the user's hand and arm. Make the location configurable for left-handed users via a *Preferences* option on the application control bar. When the user switches the layout, provide an option to switch the stylus settings as well.

# 1.D STANDARDS ALIGNMENT GUIDE

## 1.D.1 PROFESSIONAL STANDARDS FOR INTERACTIVE APPLICATION AND VIDEO GAME CREATION

### 1. INTERACTIVE APPLICATION AND VIDEO GAME DESIGN

#### 1.1 OVERVIEW OF KEY ASPECTS IN THE DESIGN PROCESS

- 1.1.9. Explain the relevance of ambiance and environment in game design
- 1.1.14. Explain the relevance of platform capabilities and constraints (CPU, GPU, Memory, Storage, etc.) to design
- 1.1.15. Describe common hardware interface devices and their usage (keyboards, controllers, etc.)
- 1.1.16. Describe available target platforms, their capabilities and constraints.

#### 1.7. TOOLS AND TECHNOLOGY

- 1.7.9. Demonstrate a working knowledge of game development tools
- 1.7.18.1 Determine necessary technical capabilities
- 1.7.18.3. Specify computers and hardware to effectively run the necessary tools

## 2. INTERACTIVE APPLICATION AND VIDEO GAME DEVELOPMENT

### 2.8. HUMAN COMPUTER INTERFACE/GRAPHICAL USER INTERFACE

- 2.8.1. Explain and demonstrate principles of visual communication
- 2.8.3. Using examples, describe key principles behind graphical user interfaces (UIs)
- 2.8.4. Define usability as an objective for user interfaces (UIs)

### 1.D.2 COMMON CORE STATE STANDARDS (CCSS)

- CCSS.ELA-Literacy.RST.11-12.2 • Determine the central ideas or conclusions of a text; summarize complex concepts, processes, or information presented in a text by paraphrasing them in simpler but still accurate terms.
- CCSS.ELA-Literacy.RST.11-12.4 • Determine the meaning of symbols, key terms, and other domain-specific words and phrases as they are used in a specific scientific or technical context relevant to grades 11-12 texts and topics.

### 1.D.3 STEM CAREER CLUSTERS (SCC)

- SCC01 ACADEMIC FOUNDATIONS: Achieve additional academic knowledge and skills required to pursue the full range of career and postsecondary education opportunities within a career cluster.

### 1.D.5 NEXT GENERATION SCIENCE STANDARDS (NGSS)

- Science and Engineering Practices:
  - NGSS4: Analyzing and interpreting data
  - NGSS8: Obtaining, evaluating, and communicating information

## 1.E SUGGESTED RESOURCES

Listed below is a recommendation of resources to consider for this unit:

- Getting Started: How to: zSpace300  
<https://edu.zspace.com/info/getting-started>
- Getting Started with zSpace Course  
<https://edu.zspace.com/learn>
- Image of the stylus showing the button layout:



## zSpace Unit 2: zSpace Stereo Mechanics

Begin this zSpace unit after completing Unit 10 of Unity's Curricular Framework.

### 2.A UNIT OVERVIEW

#### 2.A.1 UNIT DESCRIPTION

In this first half of the unit, students will learn about the factors involved with rendering objects in stereoscopic 3D. Students should demonstrate understanding of stereo comfort and how to use different coordinate spaces to achieve a variety of effects, without causing user discomfort. The first half of the unit builds the student's knowledge base on coordinate systems, head tracking, viewer scale, and angle awareness and does not involve programming.

In the second half of the unit, students will import the zSpace Unity Plugin and become familiar with the plugin's architecture and inspector pane functions. The zCore object is a vital part of creating a stereoscopic 3D application on a zSpace system and allows students to modify the camera and stylus, as well as use debug tools. Finally, students will begin to modify the Survival Shooter tutorial from Unity's curriculum to work on a zSpace system using the knowledge they have gained.



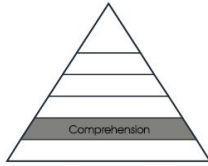

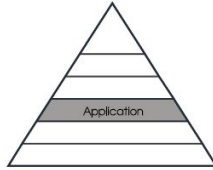

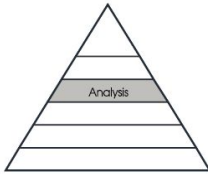

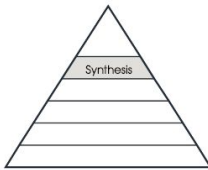

#### 2.A.2 MAJOR TOPICS

In this unit, learners will explore the following topics:

- Documentation Additions
  - Quad buffer stereo
  - Coordinate systems
  - Head tracking
  - Viewer scale
  - Angle awareness
  - Unity plugin deep dive
- Modification of Survival Shooter Tutorial
  - Import zSpace plugin
  - Modify viewer scale and camera location

### 2.A.3 LEARNING OBJECTIVES

By the end of this unit, learners should be able to perform the following tasks:

	Blooms Domain	Learning Objective	Level of Difficulty
1			
2		<p>Describe the stereoscopic 3D mechanics</p> <p>Identify the 5 coordinate spaces that a zSpace system uses</p> <p>Explain the importance of viewer comfort and viewer scale</p> <p>Understand the architecture of the zSpace Unity Plugin</p> <p>Learn the features of the inspector pane for the zCore object (debug visualizations, properties of stereo viewing, glasses data, stylus features)</p>	
3		Import the zSpace Unity Plugin	
4		<p>Understand the difference between head-mounted VR and fish tank VR.</p> <p>Analyze the pros and cons of enabling angle awareness for a zSpace application</p>	
5		<p>Enable stylus functions in Unity</p> <p>Adapt the Unity 3D Survival Shooter tutorial to set up the stereoscopic cameras</p>	

6			
---	---	--	---

## 2.A.4 MATERIALS

- Information about stereo comfort: <http://developer.zspace.com/docs/aesthetics/>
- [zSpace Developer Unity 3D Programming Guide](#)
- [Unity 3D Manual](#)
- [zSpace Unity Plugin Releases](#)
- Plugin Order of Events: [Execution Order of Event Functions](#)
- Script Lifecycle Flowchart: <http://imgur.com/gallery/lpuKrMt>
- [Unity 3D Survival Shooter](#) tutorial

## 2.B UNIT OUTLINE

1. Stereo Mechanics
  - a. Stereo Comfort
  - b. Head Tracking
    - i. Head Mounted Display (HMD) versus Fish Tank VR
  - c. Quad Buffered Stereo
  - d. Rendering Camera Attributes
    - i. Head Position
    - ii. Interpupillary Distance and Glasses Offset
    - iii. View Transform and Projection
  - e. Coordinate Systems
    - i. Camera Space
    - ii. Viewport Space
    - iii. World Space
    - iv. Display Space
    - v. Tracker Space
  - f. Viewer Scale
  - g. Angle Awareness
    - i. Avoiding Spatial Conflicts
2. Unity Plugin
  - a. zSpace Plugin Setup
  - b. zSpace Plugin Architecture
    - i. Plugin Objects and Hierarchy
    - ii. Plugins
    - iii. Camera Callback Events



- iv. Camera Scripts
    - c. Editor UI Properties and Debug Information
      - i. zCore Inspector
      - ii. Simple Stylus Access
      - iii. Stylus Game Object Manipulation
  3. Prepare to Modify Survival Shooter Tutorial
    - a. Import the zSpace Plugin
    - b. Set up the zCore Camera Rig
    - c. Add Postprocessing to the Stereoscopic Camera

## 2.C LEARNING ACTIVITIES GUIDE

This section provides a guide for delivering the unit content. When reviewing content in this unit, important questions to consider may include:

- What learning experiences can your learners engage in during this unit?
- How can you integrate formative assessments into these learning experiences?
- How can you integrate formative assessments into the tangible deliverables (e.g. documents, projects, tests applications, game builds) that your learners produce?
- How can you integrate summative assessments towards the end of this unit?

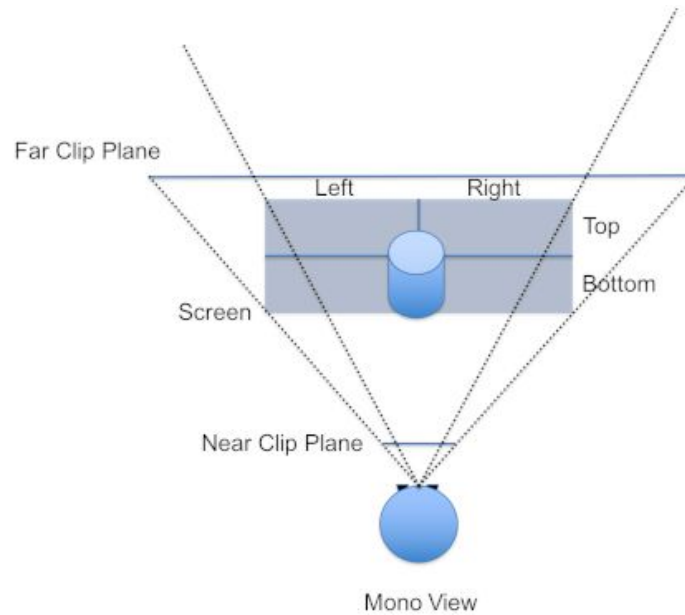
As these can be challenging questions, this section will provide resources and recommendations to help you determine the appropriate answers.

### 2.C.1 zSpace Stereo Mechanics

#### **zSpace Stereo Mechanics**

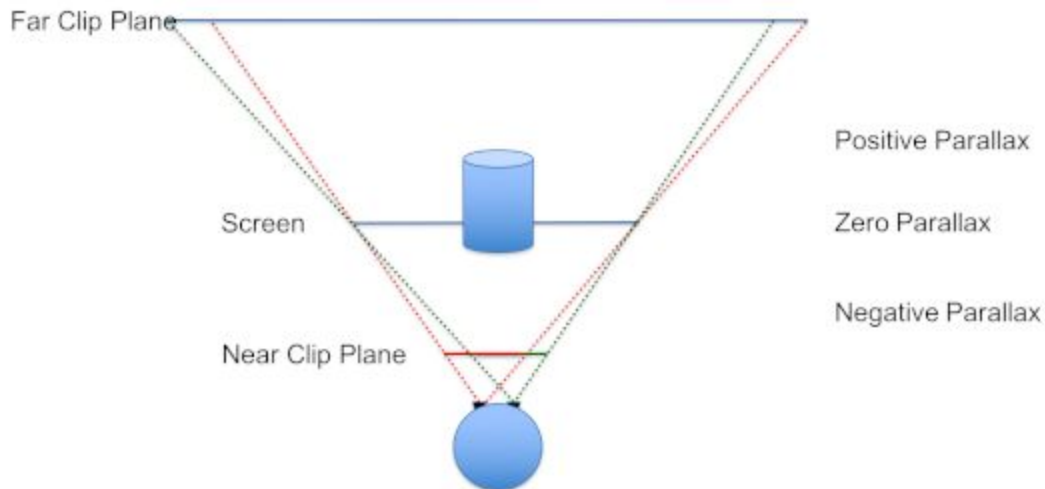
zSpace renders VR/AR objects in stereoscopic 3D. To achieve realistic rendering, the system uses quad buffered stereo. This requires developers to consider a number of attributes including coordinate systems, head tracking, viewer scale, and angle awareness.

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



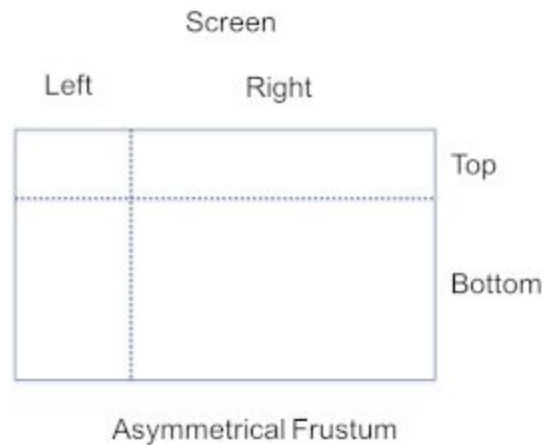
Humans see using two cameras (eyes), which generate two images that are fused into a single image by the brain. This is how we perceive depth. As a developer, we use this processing in the brain to simulate the perception of depth in 3D applications. This is the basis for all stereoscopic 3D applications.

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

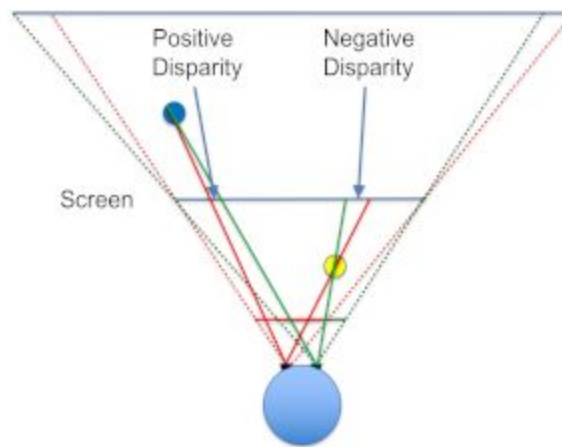


The two frustums originate near the center of each eye and intersect at the edges of the screen. This configuration creates asymmetrical bounds at the screen. If you project the eye point onto the screen, you might see bounds like those depicted in the next figure.

By rendering the scene twice using these frustums, we generate two images to display to the viewer.



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.



Stereo frustums can project the points of an object 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 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, shown by the blue circle in the preceding figure. When the brain processes images with positive disparity, the objects appear 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. This is called negative disparity or parallax, as shown by the yellow circle in the preceding figure. When the brain processes images with negative disparity, the objects appear 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 have no stereo effect, and appear coplanar with the screen.

## Stereo Comfort

zSpace uses disparity in stereo rendering to create the illusion of 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 can also cause discomfort for some viewers.

For more information about stereo comfort, see <http://developer.zspace.com/docs/aesthetics/>.

zSpace includes features in the SDK to help applications compute disparity in order to locate objects at the best distance from the viewer to maximize comfort.

## Head Tracking

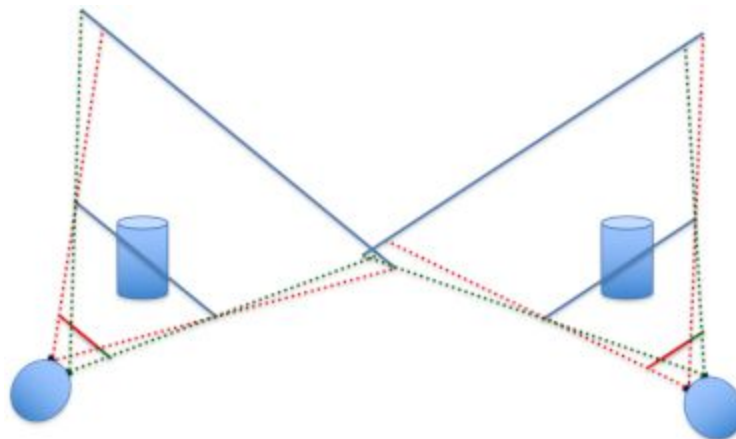
Stereo rendering helps us present virtual objects as if they exist in the real world, but it is not fully immersive. For example, 3D movies are an enjoyable experience, but they are not completely immersive. The feature that makes stereo 3D truly immersive is head tracking.

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 better simulate reality by dynamically adjusting the stereo frustums based on the viewer's head position.

## Head Mounted Display (HMD) versus Fish Tank VR

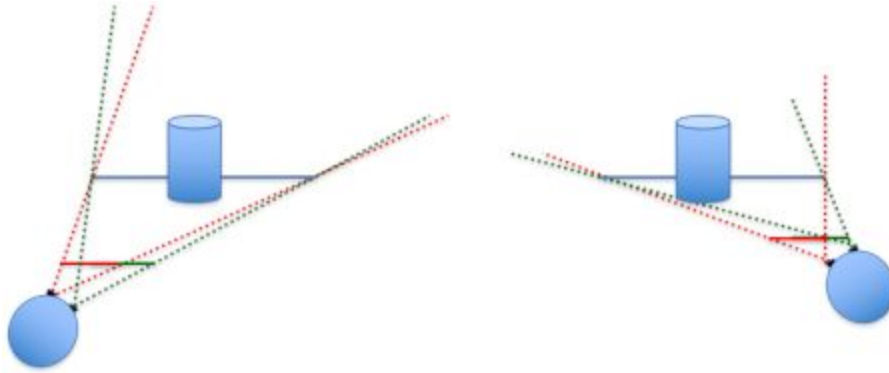
There are two main classifications of Virtual Reality (VR) systems: Head Mounted Displays (HMD) and Fish Tank VR systems. zSpace is considered a Fish Tank VR system because looking at the scene is like looking through a fish tank.

If we look at the frustums in these two scenarios, we can see that the platform differences change how the applications adapt to them. The following diagram shows a head mounted display example.



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

The following figure shows the frustums of a fish tank VR system.



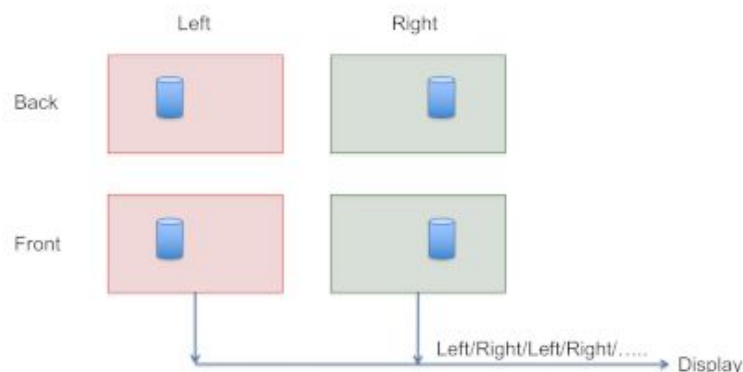
In the zSpace VR/AR system, the screens are static and the head tracking information changes both the virtual camera position/orientation and the frustum bounds. Head tracking information modifies both the virtual camera position/orientation and the frustum. From a developer's standpoint, this means that the frustum changes with every frame.

## Quad Buffered Stereo

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 stereo buffers – Is the most common. It uses one large back buffer and virtually divides it in two during rendering.
- Side-by-side stereo buffers – Primarily used for 3D TV. The system simply displays one continuous buffer and one resolution.
- Quad buffered stereo – The stereo buffering configuration used by zSpace.

In this configuration, the left and right buffers are logically separated, and the graphics processing unit (GPU) knows that the buffers exist and are separate. This means that the buffers need to be allocated and managed by the GPU.



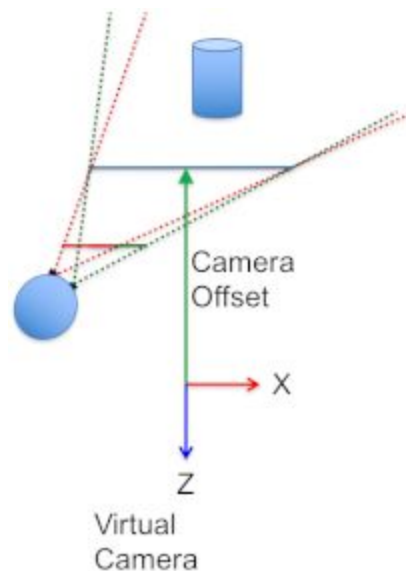
The application renders into the buffers, synchronizing with the GPU, and then presents the buffers for display. The GPU sends the buffers to the display in a time sequential fashion, presenting each image to the viewer. With this configuration, windowed stereo is possible and each eye receives full 1080p resolution. The GPU needs to know that this configuration is being used, and as such, it must be configured correctly.

## Rendering Camera Attributes

Once you have set up the rendering buffers for stereo rendering, you can modify the camera attributes. Monoscopic rendering places the origin of the frustum at the virtual camera point of the application. When processing head tracking for rendering, the frustum origins are near the center of each eye. There are several attributes that must be processed to correctly configure the system. You 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.

### Head Position

The zSpace system tracks the center of the glasses in real world units. The current head position and the zSpace display define the off-axis frustums used for rendering, as depicted in the following figure.



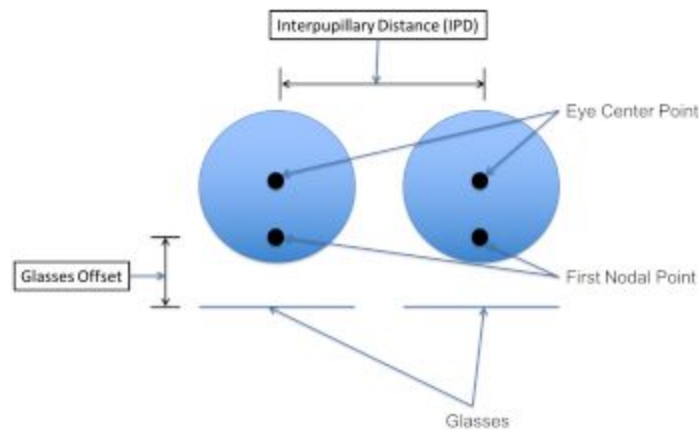
Note the relationship between the virtual camera of the application and the zSpace display. The zSpace system defines a camera offset vector at the position that the virtual camera is oriented, in the Z axis direction. This vector points at the center of the application rendering window.

Most 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. Use this vector and display angle to dynamically

compute the exact position and orientation of any geometry you want to be coplanar with the display. This is useful for 2D user interface elements.

### Interpupillary Distance and Glasses Offset

The head position and orientation alone do not provide enough information to render the left and right frustums. You need to define attributes for the glasses and head position using the 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 lenses. This is called the glasses offset. Both the interpupillary distance and the glasses offset can be queried and modified using the zSpace SDK.

### View Transform and Projection

The zSpace system does all the preceding calculations and makes the results 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 is a 4x4 matrix that represents the transform needed to orient the virtual camera so that it lines up correctly with the viewer's physical head position and orientation. 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 [zSpace Developer Unity 3D Programming Guide](#) explains how to get the view transform and projection information from the zSpace system.

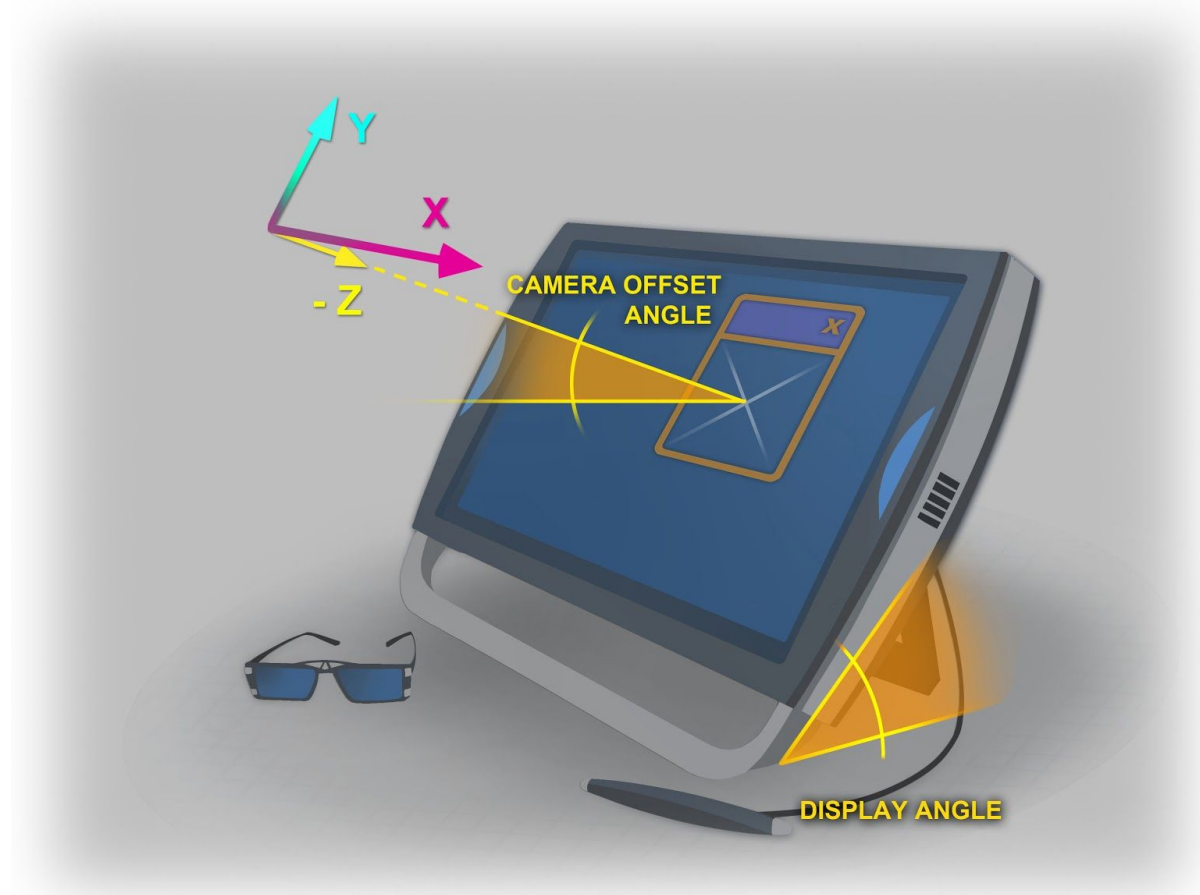
## Coordinate Systems

The zSpace system uses coordinate systems for head tracking and stylus tracking. Head tracking is transparent to developers. Stylus tracking requires an understanding of coordinate spaces to enable visualization and use.

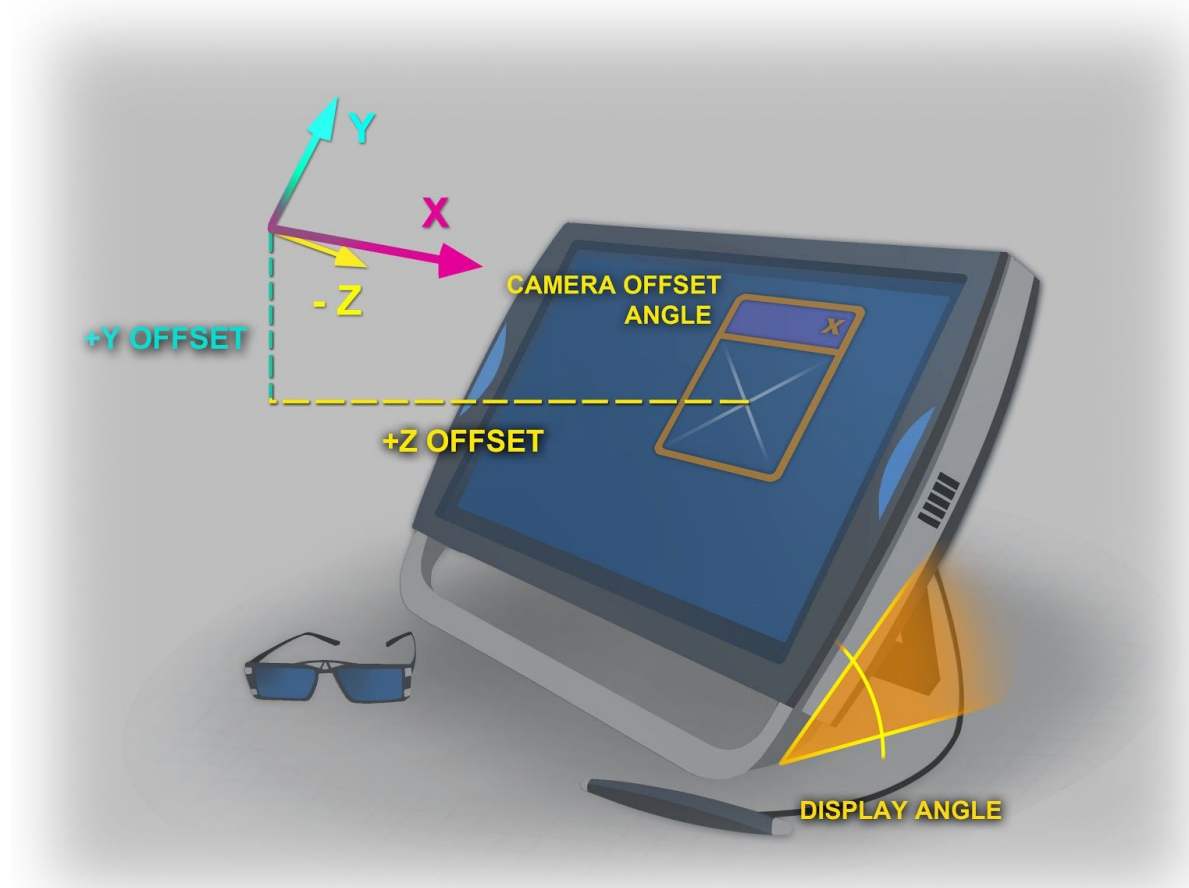
Scale is related to a coordinate system. All 3D applications use a scale. zSpace does all processing in the real world physical scale. To achieve the desired experience, you need to be aware of the relationships between the zSpace scale and the application scale.

### Camera Space

Camera space is the coordinate system defined by the virtual camera in the application. The following figures show how camera space relates to the zSpace display.



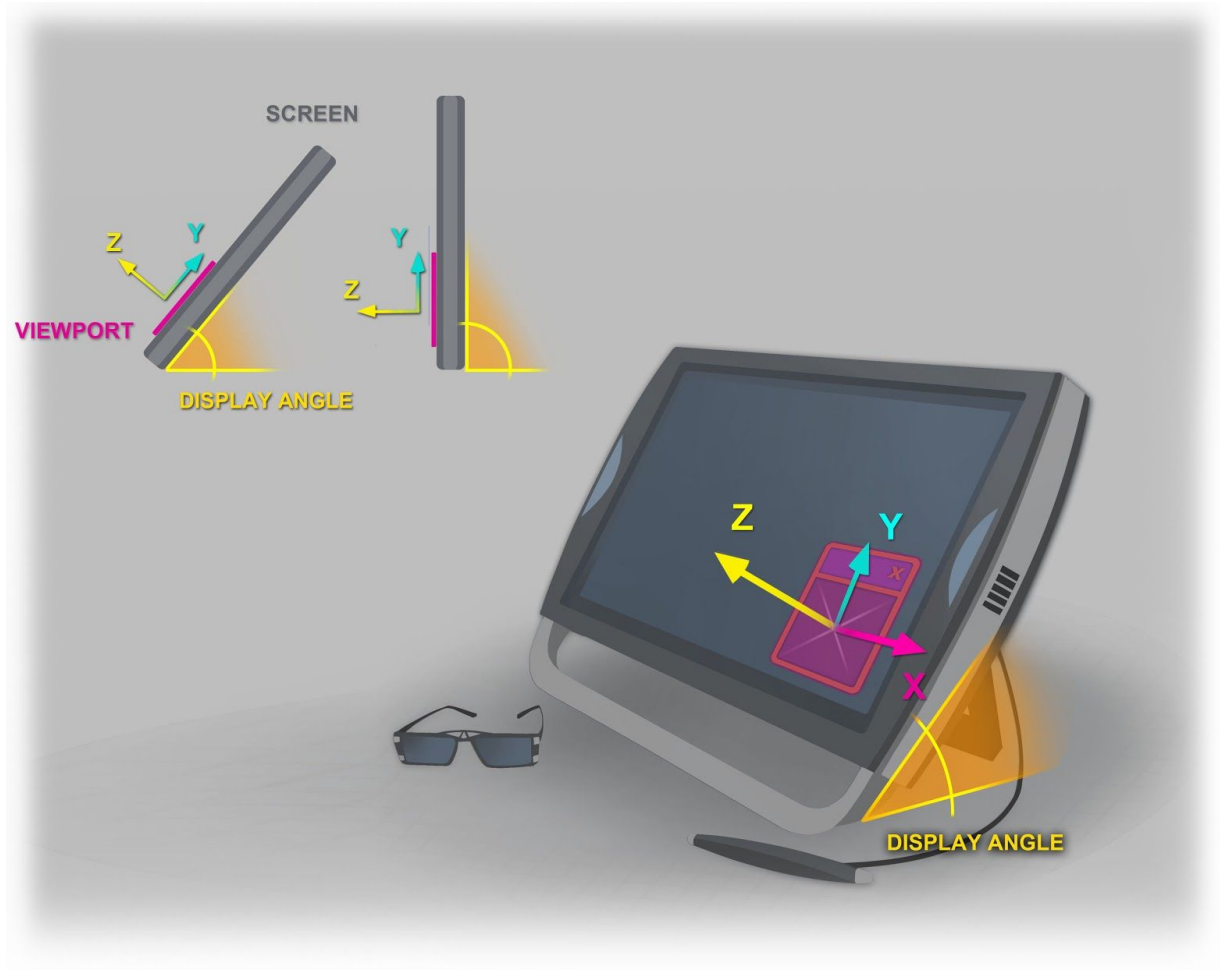




The window that the application renders into is called the viewport. The origin of camera space is at the position of the application's virtual camera. The viewport on the zSpace screen is a specific distance away from the virtual camera, and positioned in the direction that the camera is oriented. The virtual camera points to the center of the viewport. If the application is a full-screen application, 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 [zSpace Developer Unity 3D Programming Guide](#). You need to know this distance in order to position an object at or near zero parallax.

## 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.

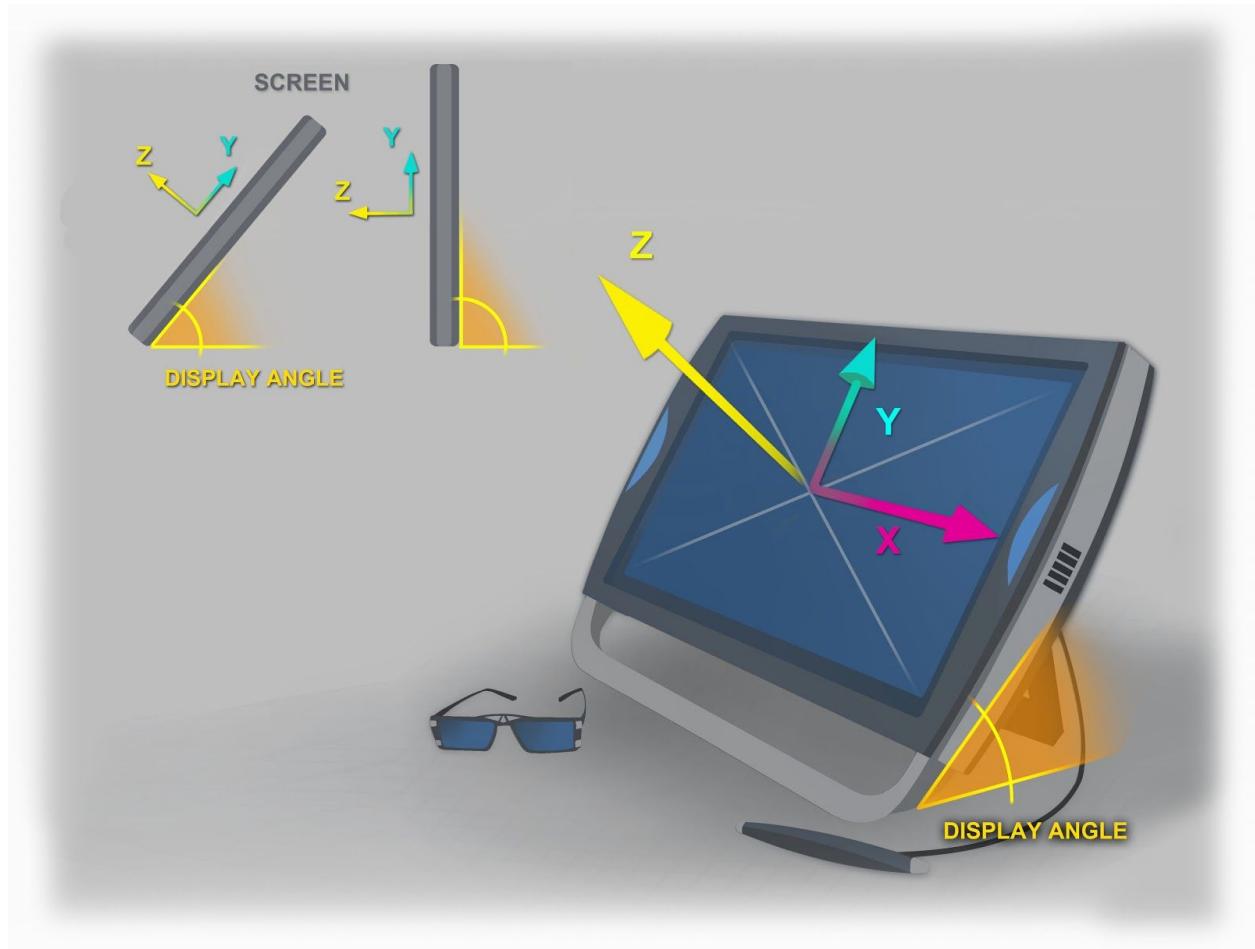


## World Space

World space is a coordinate system that is only known to the application. zSpace does not need to 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 zSpace Developer Unity 3D Programming Guide include samples of how to calculate this transform.

## Display Space

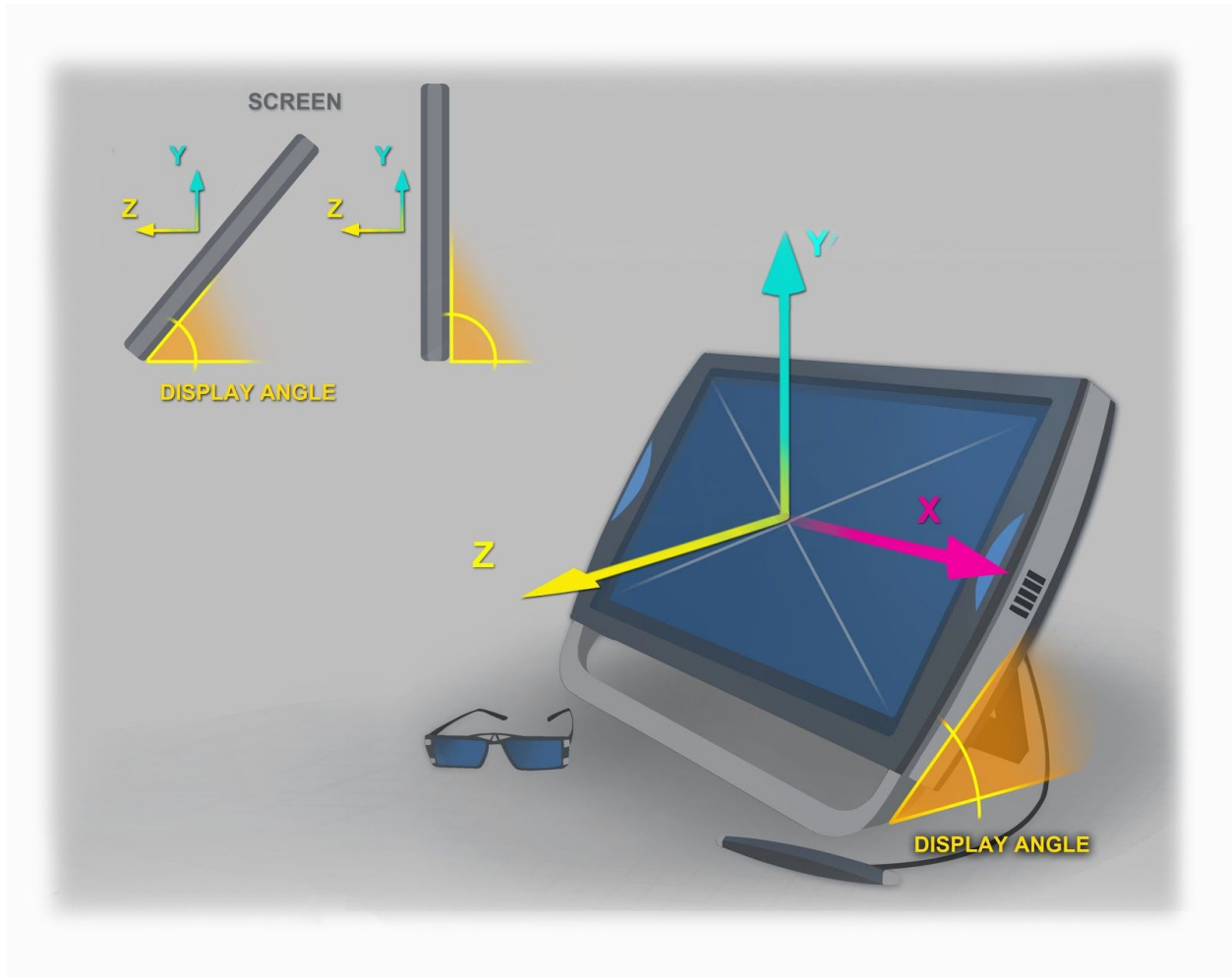
Display space is similar to viewport space, but it is located at the center of the display. If the center of the rendering window is at the center of the display, then they are identical.



The display space 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 do not need to use display space.

## Tracker Space

Tracker space is the coordinate system in which raw tracker data is reported. The tracker space 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 do not need to use the tracker space.



## Viewer Scale

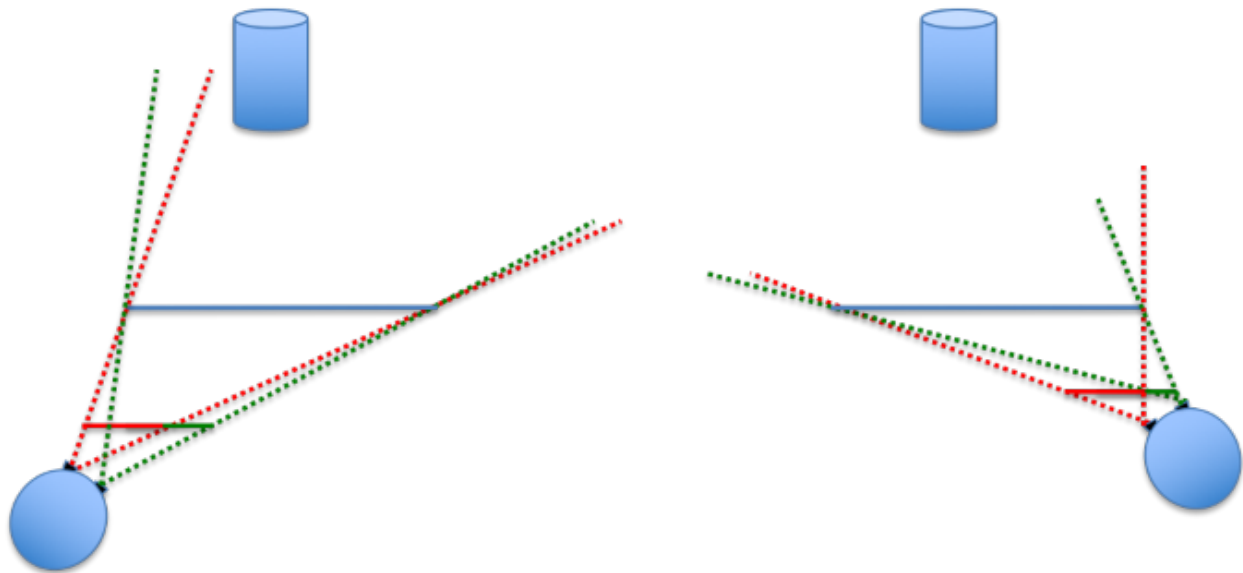
All zSpace applications process data in a real world scale where 1.0 is one meter. All attributes, transforms, and projections, are assumed to be in this scale. This is necessary to properly compute frustums and all the appropriate data as it occurs in the real world. This also ensures that frustums line up with the viewer's head, and any virtual objects tracking the end of the stylus match correctly.

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, you can use viewer scale.

The viewer scale attribute is a multiplier to the frustum. For 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.

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

The viewer scale also effectively moves the head position back by this scale factor. This is important to remember when placing geometry at the zSpace display. The distance varies 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 to see this phenomenon. By increasing the viewer scale, you can minimize this effect.

## Angle Awareness

The zSpace system acts as a window into a virtual world, where certain objects can come through the window. The system tracks this information, incorporates it into the stereo calculations, and makes it available to the application. In the zSpace systems, viewers can dynamically adjust the display angle to increase viewing comfort. We recommend angles between 30 and 60 degrees.

In some zSpace systems, the angle of the display is dynamically tracked and this information can be incorporated into the experience.

- **Angle Awareness Disabled** – If you want 3D application to stay locked to the display, adjust the display angle to rotate the world in tandem.
- **Angle Awareness Enabled** – If you want the virtual world to be locked to the real world, move the display (as a window) independently from the 3D experience.

Designing your experience for disabled angle awareness is simpler, as the framing of your experience does not change. This however can feel odd for applications which use physics and gravity, as the user's display angle may mean virtual gravity no longer points in the same direction as real-world gravity.

Enabling angle awareness means your virtual gravity always aligns to real gravity, however your framing of the experience is now dynamic and may require more design effort. This is a bit like designing web experiences for different form factor devices or display orientations such as portrait versus landscape.

### Avoiding Spatial Conflicts

It is important to prevent application controls — the control bar, palette, and inspector — from interfering with your in-scene objects. Similarly, objects in the scene should not prevent the user from interacting with the application controls.

Three possible solutions:

- **Temporarily make the in-scene objects translucent** – When displaying a control, gradually fade any interfering objects to become translucent. The gradual change prevents a blinking effect as the stylus moves around the scene. This approach maximizes available 3D space.
- **Reserve space for your application-level controls** – Reserve part of the display for the application controls. This solution provides the user with a sense of stability as the scene is never affected. This approach does cut into some of the available 3D space.
- **Temporarily move the viewpoint** – Gradually move the viewpoint so that all objects are in positive parallax (behind the monitor), leaving room for the user interface. Scale objects as needed to display the entire scene. When the UI controls are no longer required, gradually return the viewpoint to its previous setting. This approach maximizes 3D space.

In all three approaches, notify the user that any changes are temporary.

## Unity Plugin

The zSpace plugin supports both Unity 4 and Unity 5. The plugin also supports both the 32 and 64 bit Unity editor and players. The plugin currently only supports OpenGL rendering, and needs to allocate stereo resources, so additional command line options need to be specified for the editor and all applications created with Unity.

For implementation details see the [zSpace Developer Unity 3D Programming Guide](#).

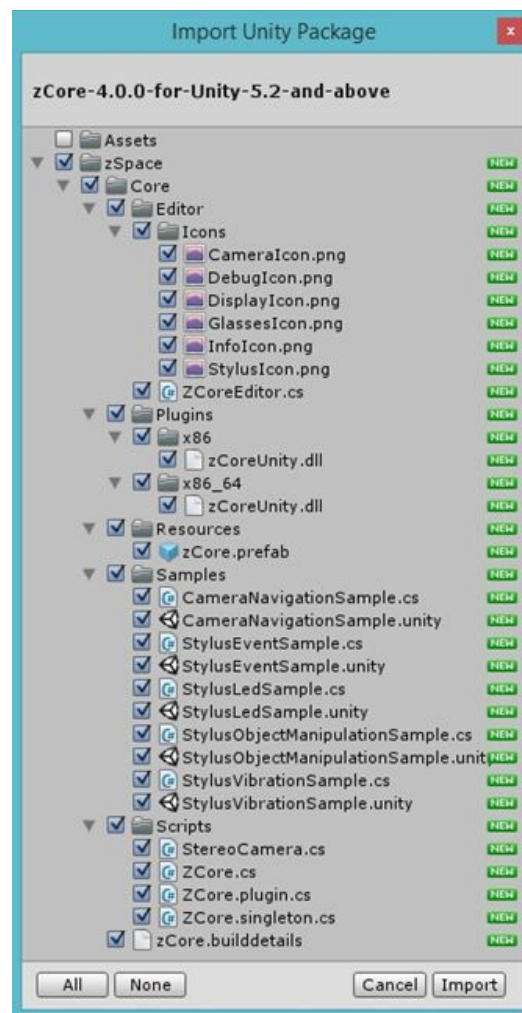
For more information about Unity development see the [Unity 3D Manual](#).

In order to use the zSpace plugin, Unity needs to run using OpenGL. To enable Unity 4 or 5 OpenGL mode and allocate resources for stereo, you need to enter commands in the Unity editor, or to the command line used to run the resulting application. These specific command lines and options are listed in the [zSpace Developer Unity 3D Programming Guide](#).

**Important Note:** A patch is required to support Unity versions before 4.5. Download a patch for Unity 4.3.0 or 4.1.3 from [zSpace Unity Plugin Releases](#).

## zSpace Plugin Setup

The zSpace plugin is distributed as a Unity package, importing the files listed in the following figure.



Import Unity Package

## zSpace Plugin Architecture

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

### Plugin Objects and Hierarchy

The zSpace plugin consists of a hierarchy of game objects, cameras, and scripts. At the top of the hierarchy is the zCore game object. This game object contains all of the editor properties, the ZCore script that contains all of the main APIs, and the stereo rig hierarchy. This section describes the stereo rig hierarchy and how it implements zSpace head tracked stereo.

The public functions and editor properties are covered in [Editor UI Properties and Debug Information](#).

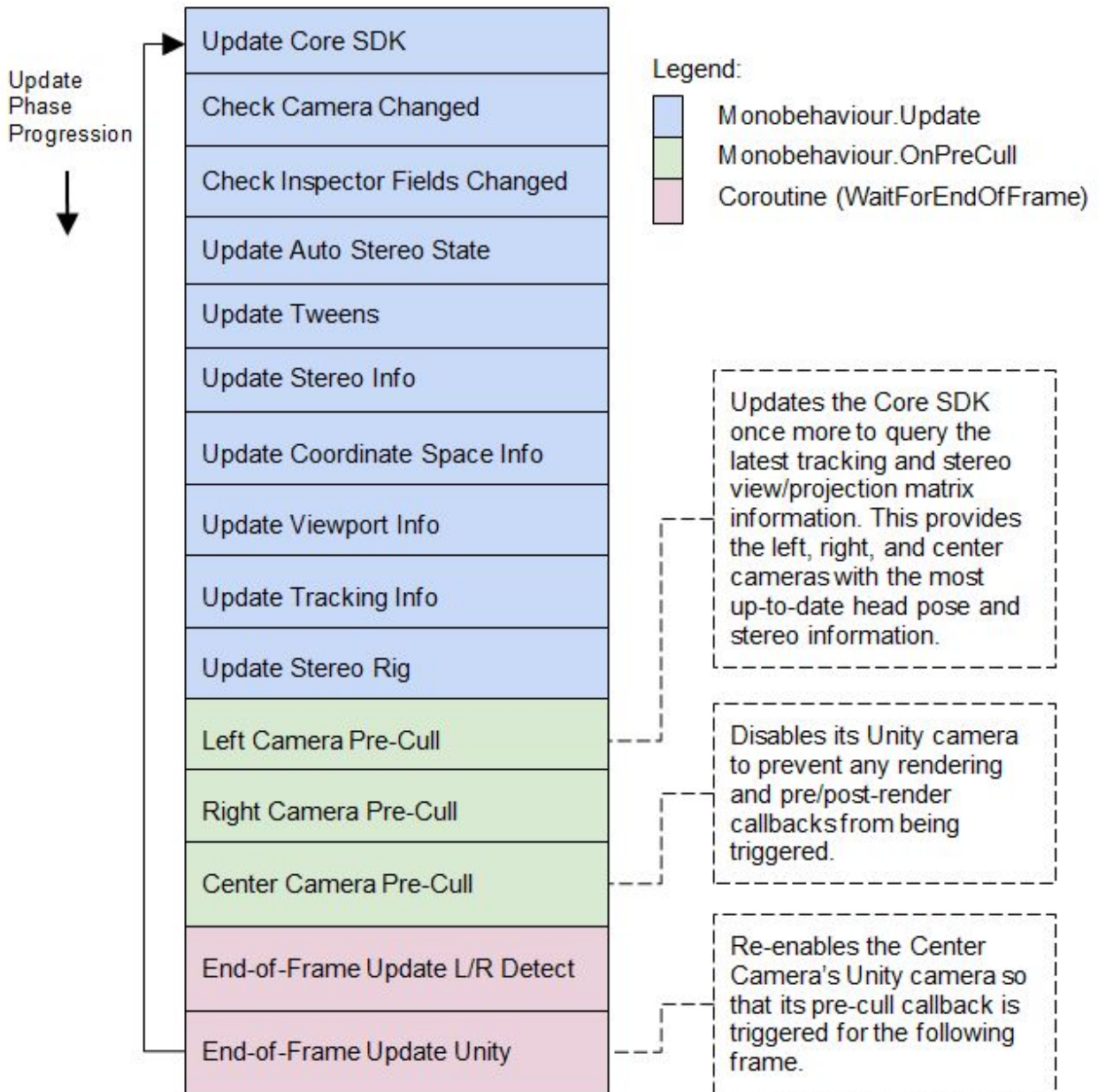
### StereoRig Game Object

The root of the stereo rig hierarchy is the StereoRig game object. StereoRig gets the position, orientation, and scale transform from the current camera property of zCore. StereoRig also includes three objects: LeftCamera, RightCamera, and CenterCamera. Each of these objects include a Unity Camera object and a StereoCamera script.

Each Camera object renders the scene for its respective eye. The StereoCamera script sets the appropriate buffer state for rendering. The Cameras also get the appropriate position, orientation, and scale for their respective eye. The CenterCamera does no rendering. It defines the post stereo rendering state, and serves as camera location for any raycast operations.



The following operations occur for every frame in the plugin.



### Plugin Operations – This Happens for Every Frame in the Plugin

For more information about the order of events, see the Unity Manual [Execution Order of Event Functions](#) and the [Script Lifecycle Flowchart](#).

## Plugins

The zSpace SDK includes a number of plugins to handle common events. For more information see the [zSpace Developer Unity 3D Programming Guide](#).

### Camera Callback Events

The zSpace plugin relies on Unity camera callbacks to put the rendering buffers into the proper state for stereo rendering. Applications may also want to be notified when these events occur. To be notified of these events after zSpace has done its processing, there are three events available for listening: PreCull, PreRender, and PostRender.

Applications can use these to be notified of those events after zSpace has done its processing for the events. These are defined in the StereoCamera script, so applications may listen to any of the three cameras.

### Camera Scripts

The zSpace plugin disables the Current Camera in the zCore object when rendering, so any custom scripts on that camera are not executed. Applications should add any of these custom scripts (like Effects) to both the LeftCamera and RightCamera for the scripts to execute correctly in a zSpace stereo environment.

## 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

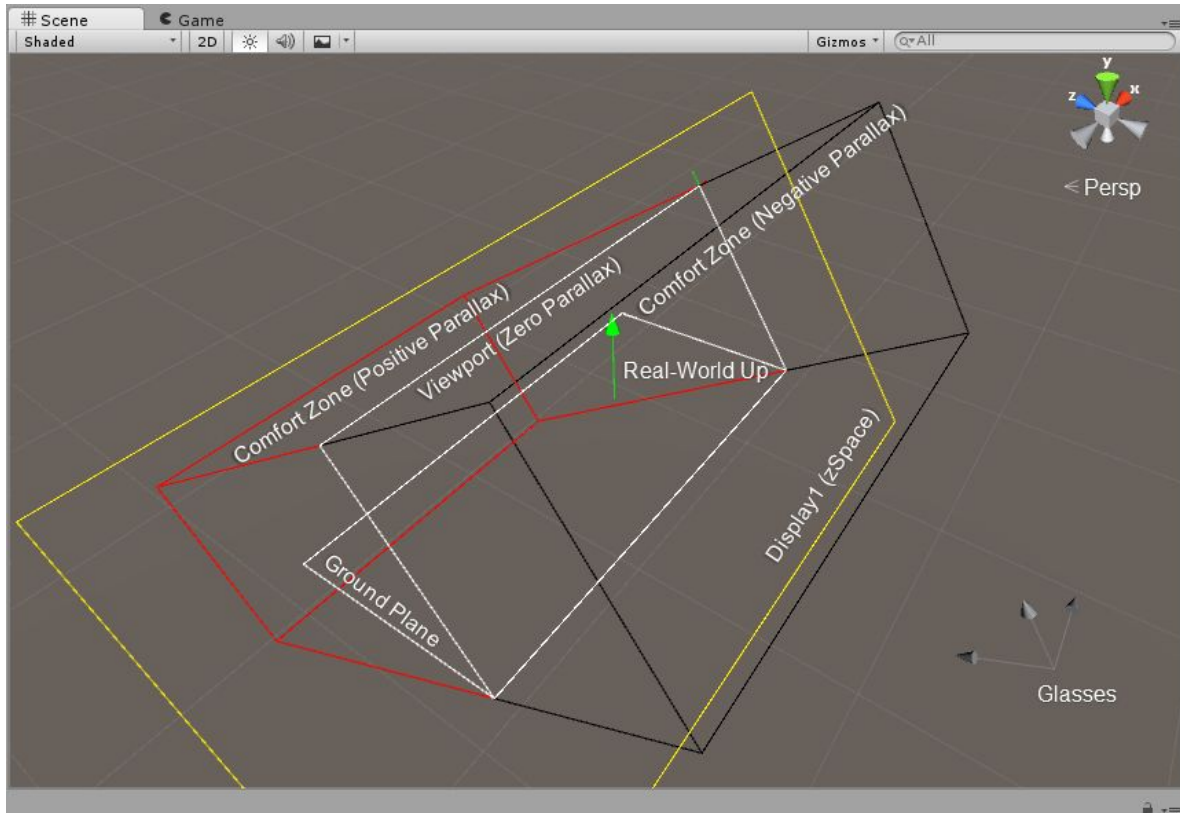
The primary tool for getting data about zSpace execution is the inspector pane for the zCore object. This inspector pane has a number of sections, and each of those sections is described in this chapter.

### 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. The following figure shows an example of these visualizations.



### Debug Visualizations

The checkbox definitions are:

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

## Stereo Rig

The Stereo Rig section of the inspector controls properties of zSpace stereo viewing. The properties and functions are:

- **Current Camera** – This property contains a reference to the current application camera defining the position, orientation, and scale of the stereo rendering.
- **Update Current Camera Transform** – A button enabling developers to adjust the Current Camera position and orientation.
- **Enable Stereo** – This property enables or disables stereo rendering.
- **Enable Auto-Transition to Mono** – This plugin tracks whether or not the glasses are currently visible. Based on that visibility, the plugin can animate from a stereo view to a mono view and back.
- **Copy Current Camera Attributes** – Controls whether camera attributes are copied from the current camera to the stereo left and right cameras.
- **Minimize Latency** – This plugin updates all of its data in a Unity **Update()** method and updates the tracking data. To minimize latency, the plugin can also do an update in the **PreCull()** callback for the LeftCamera.
- **Interpupillary Distance (IPD)** – The distance between left and right eyes.
- **Viewer Scale** – By default, zSpace uses a scale of 1.0 to one meter. Values greater than 1.0 grow the frustum and distance from the Current Camera to the display. Values less than 1.0 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. Applications can adjust the viewer scale and position of the Current Camera to match the modeling scale of their application.
- **Auto Stereo Delay and Duration** – Control the delay and duration of auto stereo to mono transition.

## Glasses

The glasses section of the inspector provides real time data about the glasses. Data includes:

- **Tracker-Space Pose** – The position and orientation of the glasses in tracker space.
- **World-Space Pose** – The position and orientation of the glasses in world space.

## Stylus

The stylus section of the inspector contains real time data about the stylus, and checkboxes to enable some stylus features.

### Enable Mouse Emulation

This checkbox enables and disables mouse emulation. Many applications have user interface elements that can be interacted with using the mouse. If you imagine 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

Using the editor properties, most Unity applications can enable head tracked stereo. This section presents some very simple code to get access to the stylus. See [Stylus](#) for details about enabling specific 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.

**TrackerEventInfo** contains the tracker target that generated the event, the target type, and the world space pose at the time of the event. For button events, the button ID is also included.

## 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.

The sample describes how to 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 and cache the cumulative rotation of the stylus and the object in the **\_initialGrabRotation** variable. This is the starting rotation to use when adding new stylus rotations.

The **StylusObjectManipulationSample** scene and script are pre-built and included with the zSpace Core SDK.

## Prepare to Modify Survival Shooter Tutorial

The Unity 3D Survival Shooter tutorial guides you through creating an isometric 3D game. Adapting this game for zSpace is a good way to learn the process, then apply these principles to your own projects.

This section describes how to import the zSpace plugin asset package for Unity 3D, set up the stereo camera rig, and add postprocessing to the stereoscopic camera.

Before continuing, you should complete the primary [Unity 3D Survival Shooter](#) tutorial. In this section and makes a second pass to adapt the Survival Shooter project to work with zSpace.

## Import the zSpace Plugin

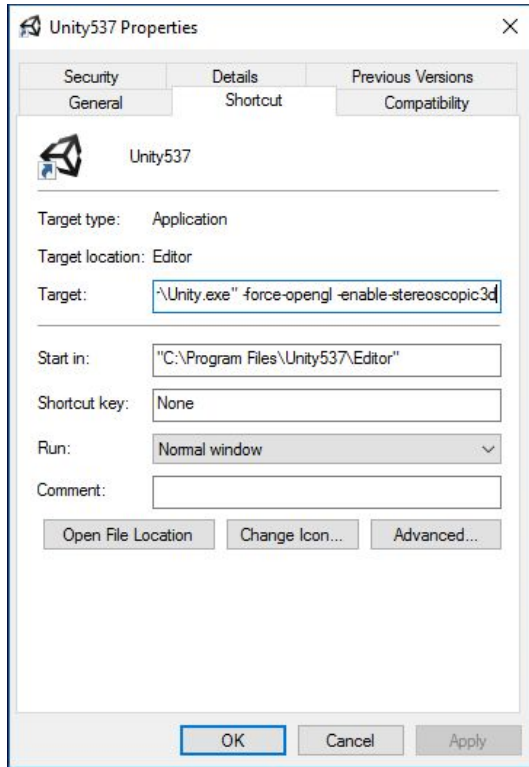
1. Launch unity with **-force-opengl** and **-enable-stereoscopic3d**.

**Unity.exe -force-opengl -enablestereoscopic3d**

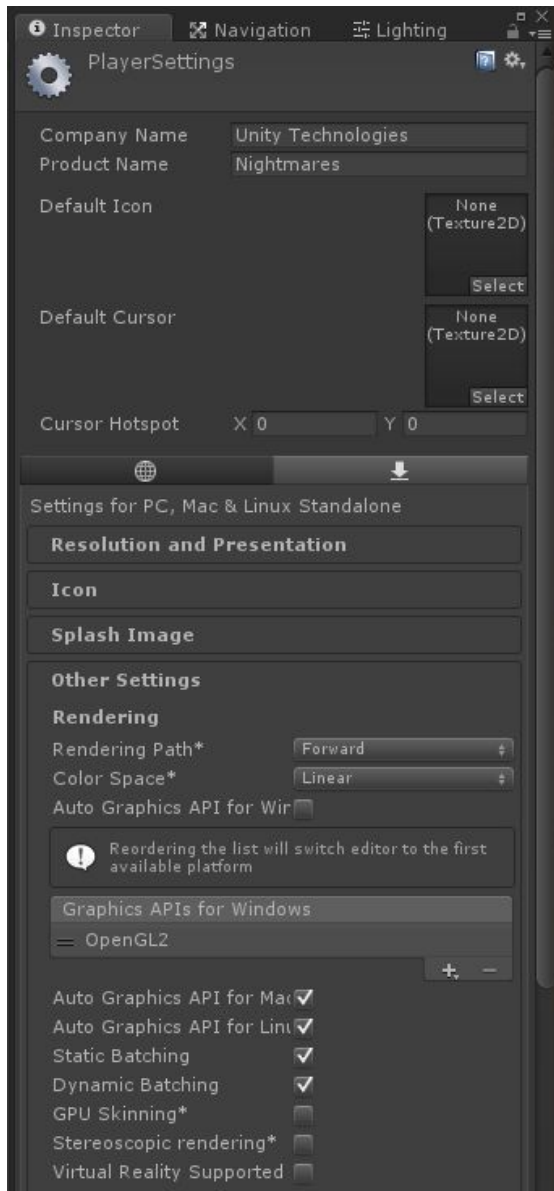
or

**Unity.exe -force-opengl -enablestereoscopic**

The *enable stereoscopic* parameter can vary depending on the Unity version. You can write these parameters into a Unity Editor shortcut as shown in the following figure.

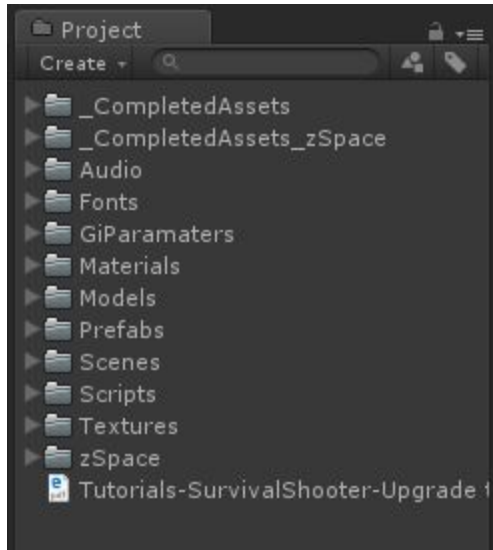


2. Open the **Inspector** dialog box.
3. Uncheck **Auto Graphics API for Windows**.
4. Set the graphics API to exclusively use **OpenGL2**. This makes sure that builds run *opengl* just as you are running it in the editor.



5. Open the **Project** browser dialog.
  
6. Make sure that the zSpace plugin asset package is imported into the base directory of your Survival Shooter project.



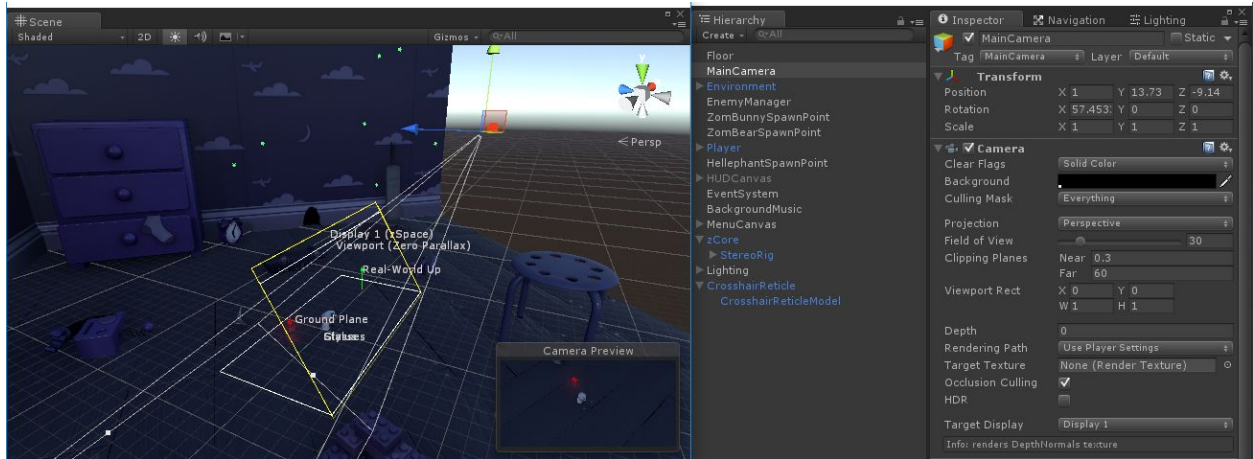


### Set up the zCore Camera Rig

1. Drag the zCore prefab into the scene.
2. Change the **Main Camera** from **Orthographic** to **Perspective** with **Field of View: 30**.
3. In the **zCore Stereo Rig** options, set the main camera as the **Current Camera**.
4. Set **Viewer Scale** to **30**.
5. In **Debug options**, turn on **Show Ground Plane**.

6. Reposition and rotate the Main Camera until the stereo rig's ground plane matches the scene's ground and the player avatar resides in the center of the camera's preview.

Setting X-axis degrees to **57.5** approximately levels the rig's ground plane.



7. Save your changes.
8. In the zCore **Stylus** options, toggle **Enable Mouse Emulation On**.
9. Click **Play**.

Game play should work with zSpace. The stylus mimics the mouse aiming and firing. If game play fails, double-check the steps described previously in this section.

### Add Postprocessing to the Stereoscopic Camera

To add postprocessing to the stereoscopic rig, the components must be applied to both the left and right cameras. If they are already applied to the main camera according to the primary course, then for each postprocess component you need to right click and **Copy Component**. Then **Paste Component as New** on the left and right cameras in the zCore rig hierarchy.

## 2.D STANDARDS ALIGNMENT GUIDE

### 2.D.1 PROFESSIONAL STANDARDS FOR INTERACTIVE APPLICATION AND VIDEO GAME CREATION

#### 1. INTERACTIVE APPLICATION AND VIDEO GAME DESIGN

##### 1.7. TOOLS AND TECHNOLOGY

- 1.7.9. Demonstrate a working knowledge of game development tools
- 1.7.17. Determine appropriate programming and scripting languages to create desired game mechanics, control the environment, UI and gameplay
- 1.7.18.1 Determine necessary technical capabilities

## 2. INTERACTIVE APPLICATION AND VIDEO GAME DEVELOPMENT

### 2.1. INTERACTIVE / REAL-TIME EDITING

- 2.1.1. Explain and demonstrate the use of Cartesian coordinate systems
- 2.1.2. Convert between local and world coordinate systems
- 2.1.3. Demonstrate successful navigation of 2D and 3D scenes using pan, zoom, orbit, walk-thru modes, etc.
- 2.1.7. Demonstrate the use of object preferences and inspector tools
- 2.1.11. Create or import extensions to enhance component functionality

### 2.2. ASSET MANAGEMENT AND RESOURCES

- 2.2.1. Organize assets and components
- 2.2.2. Understand and use hierarchical organization structures

### 2.5. LIGHTS, CAMERAS AND RENDERING

- 2.5.3. Demonstrate the creation, transformation, modification and use of cameras
- 2.5.9.4. Find, show examples and explain OpenGL

### 2.8. HUMAN COMPUTER INTERFACE/GRAPHICAL USER INTERFACE

- 2.8.1. Explain and demonstrate principles of visual communication
- 2.8.8. Explain how target platform capabilities and constraints affect the choice of user interfaces

## 3. INTERACTIVE APPLICATION AND VIDEO GAME DEVELOPMENT

### 3.3. TESTING, DEBUGGING, PROFILING AND OPTIMIZING

- 3.3.7. Demonstrate the use of a debugger to inspect code at runtime

### 2.D.2 COMMON CORE STATE STANDARDS (CCSS)

- CCSS.ELA-Literacy.RST.11-12.2 • Determine the central ideas or conclusions of a text; summarize complex concepts, processes, or information presented in a text by paraphrasing them in simpler but still accurate terms.
- CCSS.ELA-Literacy.RST.11-12.3 • Follow precisely a complex multistep procedure when carrying out experiments, taking measurements, or performing technical tasks; analyze the specific results based on explanations in the text.
- CCSS.ELA-Literacy.RST.11-12.4 • Determine the meaning of symbols, key terms, and other domain-specific words and phrases as they are used in a specific scientific or technical context relevant to grades 11-12 texts and topics.

### 2.D.3 STEM CAREER CLUSTERS (SCC)

- SCC01 Academic foundations: Achieve additional academic knowledge and skills required to pursue the full range of career and postsecondary education opportunities within a career cluster.

- SCC10 Technical skills: Use the technical knowledge and skills required to pursue the targeted careers for all pathways in the career cluster, including knowledge of design, operation, and maintenance of technological systems critical to the career cluster.

#### **2.D.5 NEXT GENERATION SCIENCE STANDARDS (NGSS)**

- Science and Engineering Practices:
  - NGSS4: Analyzing and interpreting data
  - NGSS5: Using mathematics and computational thinking
  - NGSS8: Obtaining, evaluating, and communicating information

### **2.E SUGGESTED RESOURCES**

Listed below is a recommendation of resources to consider for this unit:

- zSpace Online Developer Documents: <https://developer.zspace.com/docs/>

## zSpace Unit 3: zSpace User Interface and Interaction

Begin this zSpace unit after completing Unit 12 of Unity's Curricular Framework.

### 3.A UNIT OVERVIEW

#### 3.A.1 UNIT DESCRIPTION

In the first half of the unit, students will complete their knowledge base of zSpace by learning about how to present the stylus pointer and beam in an application, as well as how to emulate the stylus using the mouse (and how to emulate the mouse using the stylus). Students will also determine how best to present a 2D user interface in a stereoscopic environment. The first half of the unit builds the student's knowledge base and does not involve programming.

In the second half of the unit, students will continue to modify the Survival Shooter tutorial from Unity's curriculum. Their modifications are continued from the work they completed in zSpace Unit 2. The implementation in this unit focuses on the zSpace stylus. Students will learn how to integrate the stylus and will program haptic and LED feedback. Students will also program the stylus to grab in-game objects.



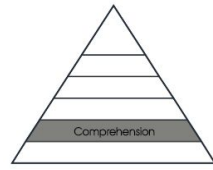

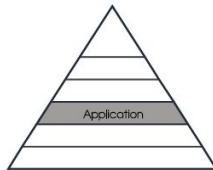

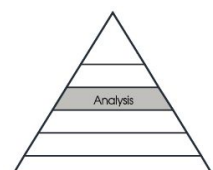

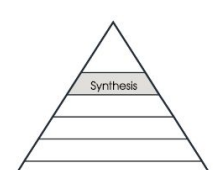

#### 3.A.2 MAJOR TOPICS

In this unit, learners will explore the following topics:

- Documentation Additions
  - Stylus interaction
  - Mouse emulation
  - Haptics
  - LED
  - 2D user interface content
  - Trackball (optional)
  - Stylus emulation (optional)
  - TAP events (optional)
  - Audio (optional)
- Modification of Survival Shooter Tutorial
  - Mouse emulation
  - Stylus integration
  - Haptics
  - LED
  - Sample 2D UI
- zView Integration (optional)

### 3.A.3 LEARNING OBJECTIVES

By the end of this unit, learners should be able to perform the following tasks:

	Blooms Domain	Learning Objective	Level of Difficulty
1			
2		Understand different stylus beam methods Understand the meaning of mouse emulation for the mouse and stylus	
3			
4			
5		Modify the Unity 3D Survival Shooter tutorial to work in a stereoscopic 3D environment. Implement mouse emulation Integrate the stylus so it can be used in place of the mouse Program haptic and LED feedback for the stylus Add prefabs (health pickups) that can be grabbed with the stylus when in the game Configure your new build of the game so it opens in stereoscopic 3D	

6			
---	---	--	---

### 3.A.4 MATERIALS

- [zSpace Developer Unity 3D Programming Guide](#)
- [Unity 3D Survival Shooter](#) tutorial

## 3.B UNIT OUTLINE

1. Stylus Interaction
  - a. Stylus Pointer
2. Mouse Emulation
3. 2D User Interface Content
  - a. Application-Level Compared to Scene
  - b. Stability
  - c. Application-Level Controls
  - d. Left and Right-Handed Layouts
  - e. Text and Legibility
  - f. Context Menu
4. Trackball (optional)
5. Keyboard
6. Modify Survival Shooter Tutorial
7. Mouse Emulation
8. Stylus Integration
  - a. Haptics
  - b. LED
  - c. Enable Mouse Emulation from Stylus
  - d. Toggle Stylus Feedback
9. Camera Distance Control
10. User Manipulated Objects
  - a. Enable and Test Grabbable Objects
11. Running a Compiled build
12. Health Pickups

## 3.C LEARNING ACTIVITIES GUIDE

This section provides a guide for delivering the unit content. When reviewing content in this unit, important questions to consider may include:

- What learning experiences can your learners engage in during this unit?
- How can you integrate formative assessments into these learning experiences?
- How can you integrate formative assessments into the tangible deliverables (e.g. documents, projects, tests applications, game builds) that your learners produce?
- How can you integrate summative assessments towards the end of this unit?

As these can be challenging questions, this section will provide resources and recommendations to help you determine the appropriate answers.

### 3.C.1 Unity Curriculum 3: zSpace User Interface and Interaction

#### zSpace User Interface and Interaction

zSpace lets you directly manipulate objects in stereoscopic 3D. For example, you can rotate, reposition, resize, or zoom an object. Using zSpace, computer interaction becomes intuitive because we already know how to move objects around in the 3D world.

This chapter provides an overview of zSpace user interface and interaction. For Unity 3D implementation details, see the [zSpace Developer Unity 3D Programming Guide](#).

#### Stylus Interaction

The zSpace stylus is a natural extension of pointing at objects using a pen, pencil, or laser pointer. People quickly build on that experience by grabbing things in a 2D and 3D environment.

The stylus provides six degrees of freedom. The stylus can provide the application with x, y, and z coordinates about its location, as well as its angle (yaw, pitch, and roll).

Sometimes you should add constraints to make it easier for the user to complete specific tasks. In some cases, you may constrain input to fewer degrees of freedom, such as limiting input to the x and y axes. Another constraint could be a snap-to-grid or snap-to-orientations (up, 90 degrees, and so on). You can let the user enable and disable constraints, or decide whether specific tasks should always assist the user with constraints.

#### Stylus Pointer

Users are familiar with the motion and function of a mouse pointer. The stylus is more complicated, but needs to display similar visual cues.

Objects in the zSpace display can appear at different depths, so your application needs to display both a stylus beam and a tip:

- The beam extends from the physical stylus along the stylus' orientation. The beam can be a fixed length or an adaptive beam that truncates when it intersects an object.



- The tip indicates the point of interaction with objects.

An important decision is whether to use a fixed-length beam or an adaptive beam. Both have their advantages and disadvantages.

- **Fixed Beam:** A fixed beam's length remains constant. If it touches an object closer than the end of the beam, it does not interact with that object. This allows the user to pass the beam through one object and select one that is partly hidden. Use this visual representation when the user requires precise control and the ability to select small objects in a crowded environment. You can expand the power of this beam by giving the user explicit control over the beam length and size of the stylus tip. This stylus beam has a longer learning curve. If you choose a fixed-length beam, your application can use volumetric selection. Volumetric selection lets the user click & drag to select multiple objects in a 3D area.
- **Adaptive Beam:** The beam's length is infinite until it intersects an object, then it adapts and truncates to the distance to the object. This allows easier object selection, as long as precise control is not required. It is also easier to select objects that are deep in the scene. Use the adaptive beam for new users or to provide a short learning curve.

You can include both a fixed beam and adaptive beam virtual stylus in your application. Some tools work best with one type. For example, the camera path tool requires a fixed-length beam because the user must place points at a specific location. The selection tool can work with either an adaptive or fixed-length beam. You can also let the user switch between the fixed beam and adaptive beam.

For either stylus beam, maintain alignment between the physical stylus and its visual representation in the display, so that they appear connected. Also maintain a consistent scale of movement between the physical stylus and the visual representation. Given a 1:1 scale between the stylus and its representation, if the user moves the physical stylus 10 cm in the x, y, and z axes, the visual representation should move 10 cm along the same axes. The same principle holds true for rotation.

## Mouse Interaction

Although the stylus is a natural tool for zSpace, the mouse may be better in some situations. For example, if you are porting an existing application into the zSpace system and users have a lot of experience with the mouse, it makes sense for the mouse to emulate the stylus.

A zSpace mouse plugin displays content in a stereoscopic 3D viewport. The mouse pointer is displayed in stereoscopic 3D within the viewport and appears as a standard 2D mouse over the rest of the user interface. There is no stylus beam, so a Z coordinate helps the mouse pointer move at different depths.

## Stylus-based Mouse Emulation

Stylus-based mouse emulation is the ability to imitate the behavior of a mouse using the zSpace stylus. The stylus can be used to drive the operating system mouse cursor. It is almost always better to design your application to use input devices as originally intended—zSpace stylus for manipulating

objects in 3D space, mouse for user interface screen input, and keyboard for typing—rather than using mouse emulation. However, if you have legacy applications or must rely on an operating system-provided UI, enabling mouse emulation can be convenient for the user to avoid frequently switching between the stylus and mouse.

If you use mouse emulation in a 3D application, a key challenge is avoiding depth cue conflicts between the pointer and objects in the scene. The mouse pointer should always be on top of objects, which can appear at any depth. That said, using mouse emulation need not be global.

For instance, you can enable / disable mouse emulation dynamically depending on user task:

- If the user is pointing the stylus at the scene or non-legacy UI, disable mouse emulation.
- If the user is pointing the stylus anywhere else, or the legacy UI is visible, enable mouse emulation.
- You can also use a combination of the previous guidelines with a distance-rule, such as the stylus must be within a certain distance of the display to enable mouse emulation.

For mouse emulation, map the stylus buttons to the native OS settings for the mouse.

**NOTE:** If your application supports both the stylus and mouse as input devices, it should display only one pointer at a time. For example, when the user switches from the mouse to the stylus, remove the mouse pointer from the display so that only the stylus beam appears.

## Modify Survival Shooter Tutorial

In Chapter 2, the Prepare to Modify the Survival Shooter Tutorial describes how to import the zSpace Unity plugin and prepare the cameras.

This section describes how to finish adapting the [Unity 3D Survival Shooter](#) tutorial for zSpace. The tutorial refers to additional assets not present in the original tutorial. To successfully follow along, please download and use the [extended .unitypackage at this url](#). This extended package also includes a subfolder containing namespaced and completed versions of the scripts constructed throughout the tutorial for easy reference and demonstration.

## Mouse Emulation

The Survival Shooter tutorial casts a ray from the mouse cursor to the floor. For zSpace you need to cast the ray from the end of the stylus to the floor.

1. Open the **zCore** prefab.
2. In **Stylus** options, clear the **Enable Mouse Emulation** checkbox.
3. Save the changes and close **Stylus** options.

When using a stylus there is no mouse no cursor to provide visual feedback. This adaptation uses a GameObject to mark where the player is aiming.

1. Find the included **CrosshairReticle** prefab and drag it into the scene.
2. Open **PlayerMovement.cs** and add the line:

```
using zSpace.Core;
```

3. Add new variables:

```
ZCore _core;  
GameObject reticle;
```

4. In **Awake()**, create references to the core and reticle:

```
_core = FindObjectOfType<ZCore> ();  
reticle = GameObject.Find ("CrosshairReticle");
```

5. Save the changes.

In **Turning()** change the raycasting logic to use the stylus pose.

1. Comment out the following line:

```
// Ray camRay = Camera.main.ScreenPointToRay (Input.mousePosition);
```

2. Add the following lines to set the raycast:

```
ZCore.Pose pose = _core.GetTargetPose (ZCore.TargetType.Primary,  
ZCore.CoordinateSpace.World);  
  
RaycastHit floorHit;  
  
if (Physics.Raycast (pose.Position, pose.Direction, out floorHit,  
camRayLength, floorMask))  
{  
    Vector3 playerToMouse = floorHit.point - transform.position;  
    playerToMouse.y = 0f;  
    Quaternion newRotation = Quaternion.LookRotation (playerToMouse);  
    playerRigidbody.MoveRotation (newRotation);  
  
    reticle.transform.position = floorHit.point;  
    reticle.transform.rotation = newRotation;  
}
```

The last two lines position the reticle where the stylus ray intersects with the floor and rotates it in the same direction as the player's avatar.

3. Save the changes.

## Stylus Integration

In this section you integrate the stylus so that the center button fires the player's gun..

1. Open **PlayerShooting.cs** and add a reference to zCore:

```
using zSpace.Core;
```

2. In the script class add:

```
ZCore _core;
```

3. In **Awake()** add:

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

4. In **Update()** replace:

```
if (Input.GetButton ("Fire1")) ...
```

with the following line, to check for the state of the stylus center button:

```
if (_core.IsTargetButtonPressed (ZCore.TargetType.Primary, 0) ...
```

5. Save the changes.

## Haptics

Program the stylus to vibrate in response to game events.

1. Open **PlayerHealth.cs** and add a reference to zCore:

```
using zSpace.Core;
```

2. In **Update()** add the following code to briefly vibrate the stylus:

```
if (damaged)
{
    damageImage.color = flashColour;
    _core.StartTargetVibration (ZCore.TargetType.Primary, 0.02f, 0.02f,
10, 1f);
}
```

3. Save the changes.

## LED

Program the Stylus LED to light up with the same color as the enemy currently being aimed at.

1. Open **PlayerShooting.cs** and add the following lines to set raycast and color variables. There is already a reference to **zCore** here from setting up the fire button.

```
Ray aimRay;
RaycastHit aimHit;
Color bunnyColor      = new Color(0f,0f,1f);
Color bearColor       = new Color(1f,0f,1f);
Color hellephantColor = new Color(1f,1f,0f);
```

2. Write a new **setLedColor** function to detect the color of the enemy being targeted, and set the LED to the corresponding color. Call the function in **Update()**.

```
void setLedColor ()
{
    aimRay.origin = transform.position;
    aimRay.direction = transform.forward;

    if (Physics.Raycast(aimRay, out aimHit, range, shootableMask))
    {
        _core.SetTargetLedEnabled(ZCore.TargetType.Primary, true);

        if (aimHit.transform.gameObject.name.Contains("Bunny"))
            _core.SetTargetLedColor(ZCore.TargetType.Primary,
bunnyColor);

        else if (aimHit.transform.gameObject.name.Contains("Bear"))
            _core.SetTargetLedColor(ZCore.TargetType.Primary,
bearColor);

        else if
(aimHit.transform.gameObject.name.Contains("Hellephant"))
            _core.SetTargetLedColor(ZCore.TargetType.Primary,
hellephantColor);

        else
            _core.SetTargetLedEnabled(ZCore.TargetType.Primary,
false);
    }
}
```

3. To make sure the LED turns off when the game stops, add the following lines to the **PlayerShooting** class.

```
void OnApplicationQuit()
{
    _core.SetTargetLedEnabled(ZCore.TargetType.Primary, false);
}
```

4. Save the changes.

## Enable Mouse Emulation from Stylus

The stylus does not natively interact with UI elements, so you need to switch over to mouse emulation for control buttons, toggles, sliders, and so on.

This section sets the stylus to emulate the mouse when three inches from the screen or closer.

1. Open the **zCore** prefab.
2. In Stylus options, check **Enable Mouse Emulation**.
3. Check **Enable Mouse Auto-Hide** and set it to **2**. This hides the mouse cursor when the stylus is not close enough for mouse emulation.
4. To enable mouse emulation when the stylus is within about three inches from the screen, Add this line to **Awake()** in **PlayerMovement.cs**:  
  

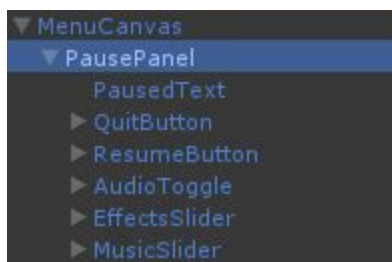
```
_core.SetMouseEmulationMaxDistance (0.1f);
```
5. Save the changes.
6. To test in the game, press **ESC** to access the pause menu with sound adjustment sliders and on/off toggle.

## Toggle Stylus Feedback

This section describes how to let the user enable and disable stylus feedback.

Add radio buttons to the **Pause** menu to enable/disable vibration and LED.

1. Open the **MenuCanvas** prefab.



2. Resize the **PausePanel** height.
3. Make two copies of the **Sound on/off** toggle entry.
4. Move the copies to the bottom of the list.
5. Rename one **VIBRATION ON/OFF** and the other **LED ON/OFF**.



Create a vibration function that can be enabled/disabled.

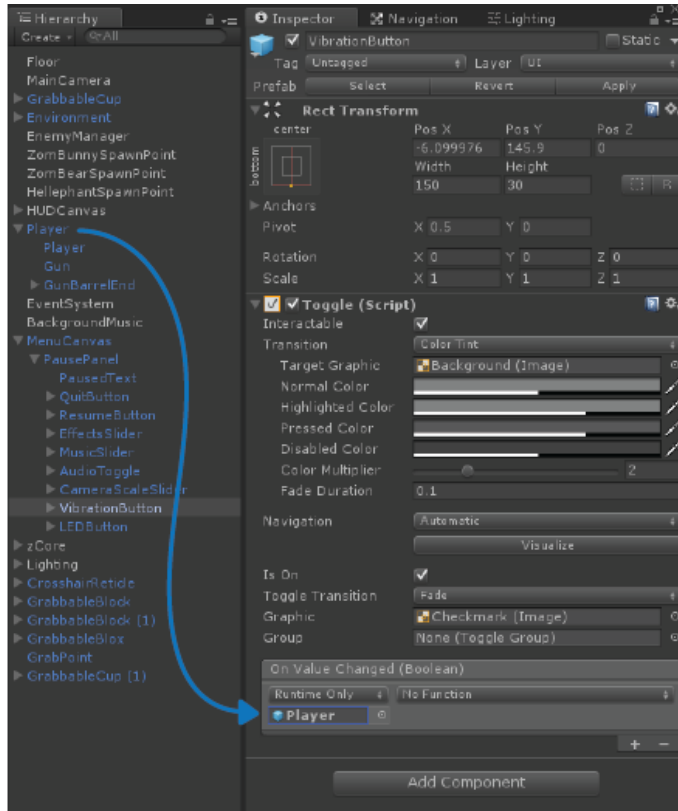
1. Open **PlayerHealth.cs**.
  2. Add a new Boolean:
- ```
bool vibrationFeedbackActive = true;
```
3. Add a public function that accepts the Boolean as an argument:

```
public void setVibrationFeedbackActive(bool b)
{
    vibrationFeedbackActive = b;
}
```

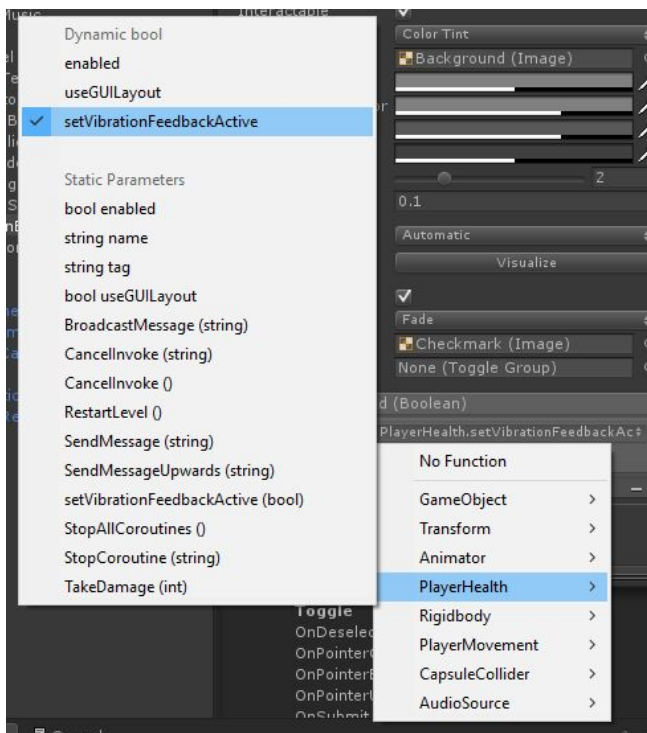
4. Make **StartTargetVibration** conditional to this variable in **Update()**:

```
if (vibrationFeedbackActive)
    _core.StartTargetVibration(ZCore.TargetType.Primary, 0.02f, 0.02f, 10,
1f);
```

5. Drag the **Player** prefab into the **Toggle** script of the vibration button.



6. Select **setVibrationFeedbackActive** as the called function.





7. Save the changes.

Create an LED function that can be enabled/disabled.

1. Open **PlayerShooting.cs**.

2. Add a new Boolean.

```
bool ledFeedbackActive = true;
```

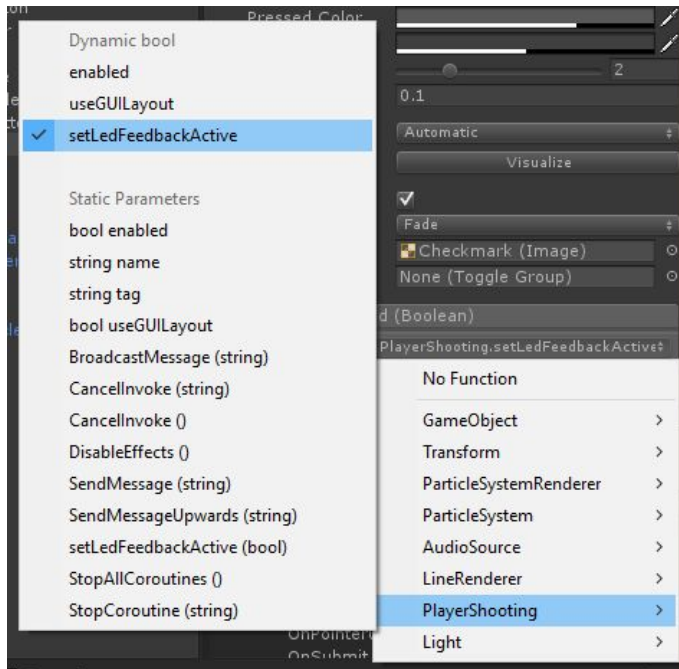
3. Add a public function that accepts the Boolean as an argument and turn off the LED in case it is already on:

```
public void setLedFeedbackActive(bool b)
{
    ledFeedbackActive = b;
    _core.SetTargetLedEnabled(ZCore.TargetType.Primary, false);
}
```

5. At the beginning of **setLedColor()** add the line:

```
if (!ledFeedbackActive)
    return;
```

6. Point the LED toggle **OnValueChanged** entry to the **GunBarrel** object, where the **PlayerShooting.cs** script is attached.
7. Select the **setLedFeedbackActive** function from the list.

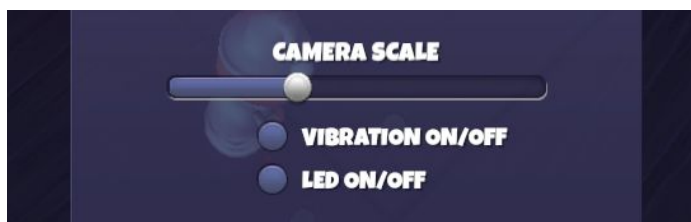


8. Save the changes and test the **Vibration** and **LED** toggles in the game.

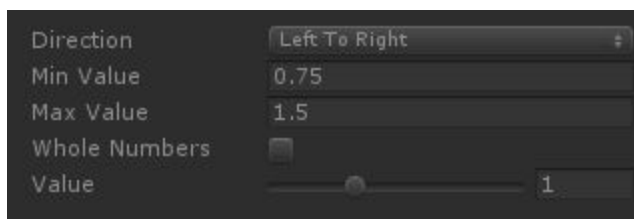
## Camera Distance Control

Add a slider to the menu to control the distance between the camera and the player's avatar.

1. Copy the **Music Slider** and rename it **CAMERA SCALE**.



2. Change the slider values to allow adjustments between **75%** and **150%** of the default camera distance.



3. Adjust the zCore **Viewer Scale** so that the **Zero Parallax** maintains a comfortable position within the scene.

4. Save the changes.

Reference zCore from the **CameraFollow** script.

1. Open **CameraFollow.cs** and add a reference to zCore:

```
Using zSpace.Core;
```

2. Add the following variables:

```
float CameraScale = 1f;  
ZCore _core;  
float base_ViewerScale;
```

3. In **Start()**, set the zCore reference and store the default **Viewer Scale**:

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

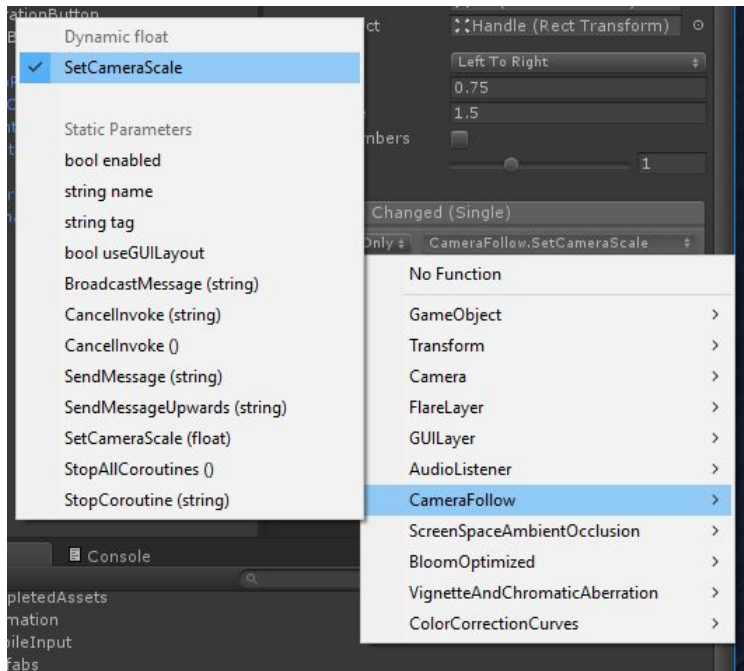
4. In **FixedUpdate()** multiply the offset by the scaling value:

```
Vector3 targetCamPos = target.position + offset * CameraScale;
```

5. Add a slider function to the control from and adjust **Viewer Scale**:

```
public void SetCameraScale(float n)  
{  
    CameraScale = n;  
    _core.ViewerScale = base_ViewerScale * CameraScale;  
}
```

6. Point the slider's **OnValueChanged** entry to the **Main Camera** and find the **SetCameraScale** function.



7. Save the changes.

## User Manipulated Objects

One of the most compelling aspects of zSpace is the ability to grab and manipulate the position of objects in 3D space.

In the prefabs folder are three grabbable prefabs that provide health pickups when grabbed.

- **GrabbableBlock**
- **GrabbableBlox**
- **GrabbableCup**

Add the grabbable prefabs to the game.

1. Create a new layer named **Grabbable** and assign it to these prefabs before adding them to the scene. If prompted to change children in this layer, choose **No**.
2. Arrange these in the scene for players to grab.
3. Create a new script named **StylusGrab.cs** and add new variables:

```
using zSpace.Core;

ZCore _core;
ZCore.Pose pose;          // a variable to cache stylus state info
int grabMask;             // layer assigned to grabbable objects
public bool isGrabbing = false; // a bool to indicate whether grabbing
                             is currently active
float range = 100f;       // how far away an object can be grabbed
float hitDistance;        // how far away the object was grabbed
GameObject grabObject;    // the object grabbed
GameObject grabPt;        // a point to parent grabbed objects to
```

4. Assign references in **Start()**:

```
grabMask = LayerMask.GetMask ("Grabbable");
_core = GameObject.FindObjectOfType<ZCore> ();
grabPt = new GameObject(); //make this a new empty GameObject
```

5. In **Update()**, get the current state of the stylus:

```
pose = _core.GetTargetPose (ZCore.TargetType.Primary,
ZCore.CoordinateSpace.World);

bool btnPressed = _core.IsTargetButtonPressed(ZCore.TargetType.
Primary, 0) && Time.timeScale != 0;
```

6. Cast a ray from the end of the stylus to determine if it intersects with a grabbable object.

```
RaycastHit hit;
Physics.Raycast(pose.Position, pose.Direction, out hit, range,
grabMask);
```

7. Add grab state logic:

```
if (!isGrabbing)
{
    if (btnPressed)
    {
        if (hit.collider != null)
        {
            BeginGrab(hit);
        }
    }
}
else
{
    if (btnPressed)
        UpdateGrab();
    else
        EndGrab();
}
```

8. Add **BeginGrab**, **UpdateGrab**, and **EndGrab** to **Update()**:

```
void BeginGrab(RaycastHit h)
{
    isGrabbing = true;
    grabObject = h.transform.gameObject;
    hitDistance = h.distance;
    grabPt.transform.position = h.point;
    grabPt.transform.rotation = pose.Rotation;

    grabObject.GetComponent<Rigidbody>().isKinematic = true;
    grabObject.transform.parent = grabPt.transform;
}

void UpdateGrab()
{
    grabPt.transform.position = pose.Position + pose.Direction * hitDistance;
    grabPt.transform.rotation = pose.Rotation;
}

void EndGrab()
{
    isGrabbing = false;
    grabObject.transform.parent = null;
    grabObject.GetComponent<Rigidbody>().isKinematic = false;
}
```

9. Save the changes.

## Enable and Test Grabbable Objects

This section describes how to enable and test the **StylusGrab** and **PlayerShooting** scripts and **GrabPoint** prefab.

### Add the StylusGrab Script

1. Add the **StylusGrab.cs** script to a GameObject in the scene. This can be any GameObject. This example uses the root player GameObject.
2. Test the game. Point the stylus at one of the grabbable objects, then click and hold. The object should drop health pickups and move as if linked to the stylus at the distance and position the grab began.

There are still some areas that need improvement. There is no visual cue to indicate when a grabbable object is being cast against. The **crosshairReticle** sinks inside the object. Firing continues while an object is grabbed.

### Add the GrabPoint Prefab

The following changes refine grabbing by adding a distinctive visual cue, and prevent shooting when grabbing an object, or vice-versa.

1. Find the **GrabPoint** prefab and add it to the scene.
2. Open **StylusGrab.cs**.
3. In **Start()**, change the empty GameObject instantiation by replacing the following line:

```
grabPt = GameObject.Instantiate();
```

with this line:

```
grabPt = GameObject.Find("GrabPoint");
```

4. Save the changes.

### Hide Crosshairs While Grabbing

Add a reference to the **crosshairReticle**, so that it is hidden when grabbing.

1. In the class, add:

```
GameObject crosshairReticle;
```

2. In **Start()**, add:

```
crosshairReticle = GameObject.Find("CrosshairReticle");
```

3. At the top of **if(!isGrabbing)** in **Update()**, add:

```
if (!isGrabbing)
{
    if (hit.collider != null)
        UpdateGrabPoint(hit);
    Else
        HideGrabPoint();
}
```

4. Create the new functions in the class:

```
void UpdateGrabPoint(RaycastHit h)
{
    if (crosshairReticle.activeSelf)
        crosshairReticle.SetActive(false);

    if (!grabPt.activeSelf)
        grabPt.SetActive(true);

    grabPt.transform.position = h.point;
    grabPt.transform.rotation = pose.Rotation;
}

void HideGrabPoint()
{
    grabPt.SetActive(false);

    if (!crosshairReticle.activeSelf)
        crosshairReticle.SetActive(true);
}
```

5. Save the changes.
6. Run the game. Note that the crosshairs are replaced with the **GrabPoint** visual cue when the stylus raycast intersects with the block.

### Prevent Simultaneous Grabbing and Shooting

Add some script code to prevent grabbing and shooting from happening at the same time.

1. Open **PlayerShooting.cs** and add a new variable to the class:

```
public bool isShooting = false;
```

2. Change the current shooting code as follows:



```
if(!_core.IsTargetButtonPressed (ZCore.TargetType.Primary, 0) && timer >=
timeBetweenBullets && Time.timeScale != 0)
{
    Shoot ();
}
```

3. Set the Boolean to indicate when the user is firing:

```
if(!_core.IsTargetButtonPressed (ZCore.TargetType.Primary, 0) && timer >=
timeBetweenBullets && Time.timeScale != 0)
{
    Shoot ();
    isShooting = true;
}
else if(!_core.IsTargetButtonPressed(ZCore.TargetType.Primary, 0))
{
    isShooting = false;
}
```

4. Open the **StylusGrab.cs** class and add a reference to the **PlayerShooting** script:

```
PlayerShooting playerShooting;
```

5. In **Start()**, add:

```
playerShooting = GameObject.FindObjectOfType<PlayerShooting>();
```

6. Just before **BeginGrab**, modify the if statement to make sure that the player cannot grab blocks when in a sustained state of firing:

```
if (hit.collider != null && !playerShooting.isShooting)
{
    BeginGrab(hit);
}
```

7. Save the changes.

Conversely, make sure the player cannot fire while grabbing.

1. In **PlayerShooting.cs** add a reference to **StylusGrab.cs**:

```
StylusGrab stylusGrab;
```

2. In **Awake()**, add the following line:

```
stylusGrab = GameObject.FindObjectOfType<StylusGrab>();
```

3. Check for the new Boolean by adding it to the if statement just before calling **Shoot()**:

```
if(_core.IsTargetButtonPressed (ZCore.TargetType.Primary, 0)
    && timer >= timeBetweenBullets && Time.timeScale != 0
    && !stylusGrab.isGrabbing)
{
    Shoot ();
}
```

4. Save the changes.

### Set Script Execution Order

Use project settings to make sure that the scripts execute in the intended order. The code assumes that **StylusGrab.cs** is executed before **PlayerShoot.cs**, which by default is not guaranteed.

1. Click **Edit > Project Settings > Script Execution Order**.
2. Add **StylusGrab.cs** and **PlayerShooting.cs**.
3. Make sure **PlayerShooting.cs** has a higher number than **StylusGrab.cs**.
4. Save the changes.

Now shooting and grabbing cannot happen at the same time.

### Make Grabbable Objects Solid

Notice that bullets go straight through the grabbable objects. To fix this, add the **grabbableMask** to the **shootableMask**.

1. In **PlayerShooting.cs awake()**, modify the **shootableMask** setup as follows.

```
shootableMask = LayerMask.GetMask ("Shootable") | LayerMask.GetMask
("Grabbable");
```

2. Save the changes.

The feature now includes the following behaviors:

- Players can grab and manipulate the block with the stylus.
- Prevents shooting when grabbing and vice-versa.
- Replaces the **crosshairReticle** with a grab point visual cue when pointing at the block.
- Blocks bullets with the grabbable mask.

**Note:** If some of the hearts are not casting light, they may be hitting a dynamic lights limit of forward rendering. You can change the lights limit to **Deferred** in the **Player Settings** rendering path dropdown.

## Run the Compiled Build

To run the new game correctly, it needs to have stereoscopic 3D enabled.

1. Open a command window and run the game with an extra argument.

**BuiltGameName.exe -enable-stereoscopic3d**

2. To make it easier for players, you can create a **.bat** file and display it next to the **.exe** with the extra argument inside the file. Then players only need to run the **.bat** file to start the game with stereoscopic 3D enabled.

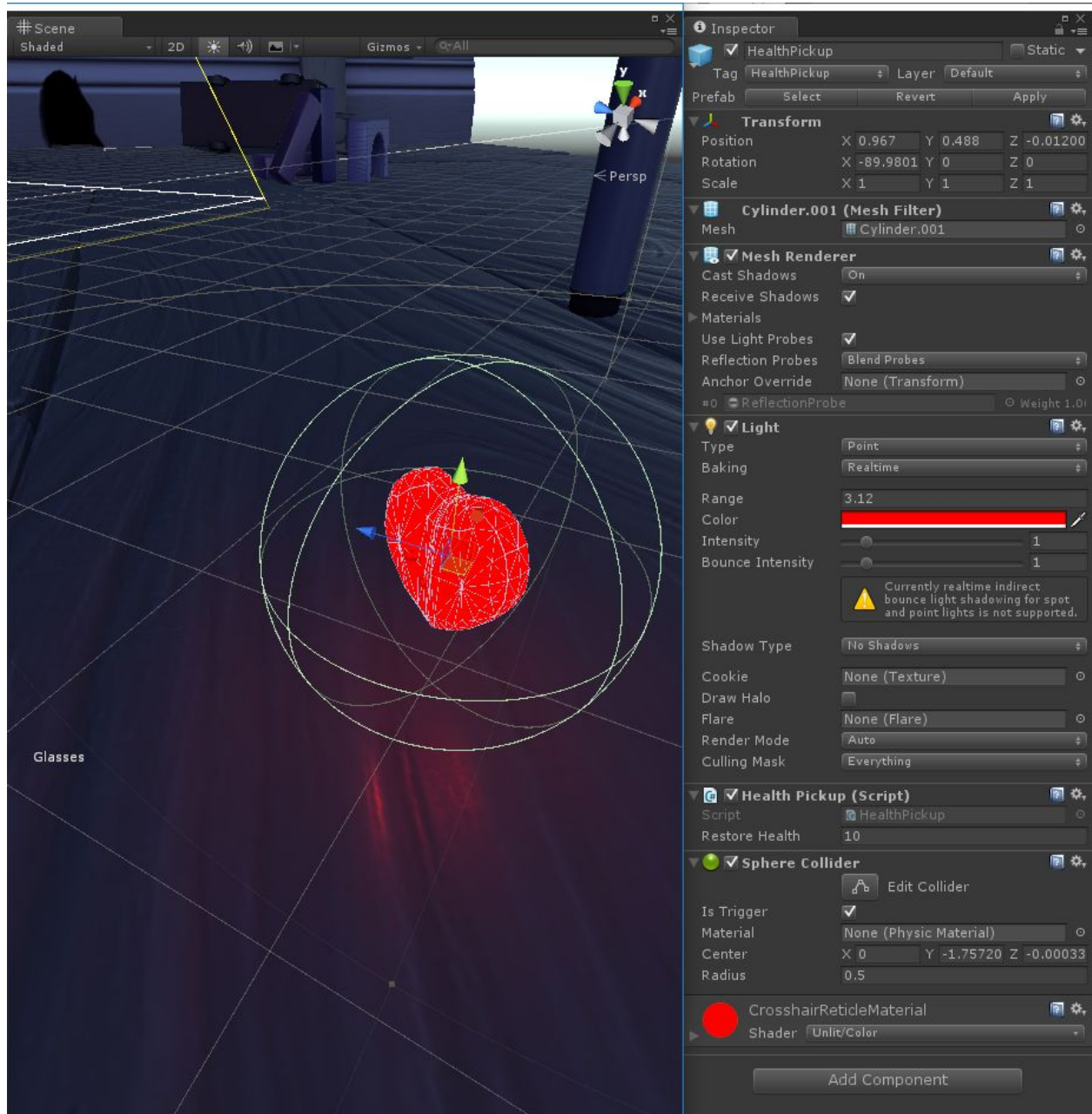
## Health Pickups

**Note:** The steps below are already implemented in the provided assets, and are described here for more detailed reference.

Now, we can give the grab feature a simple purpose. Whenever a block is grabbed, let's have it uncover health pickups for the player to consume if he's low on health.

Find the HealthPickup model and add it to the scene. Give it the CrosshairReticleMaterial, copy the light component from the crosshairReticle GameObject, and give it a sphere collider with "Is Trigger" checked.

Create a new script called HealthPickup and attach this to the model.



1. Add a function to **PlayerHealth.cs**:

```
public void AddHealth(int addedHealth)
{
    currentHealth += addedHealth;
    Mathf.Clamp(currentHealth, 0, startingHealth);
    healthSlider.value = currentHealth;
}
```

2. In the new **HealthPickup.cs** add:

```
public class HealthPickup : MonoBehaviour
{
    public int restoreHealth = 10;
    PlayerHealth playerHealth;

    void Awake ()
    {
        playerHealth =
        GameObject.FindObjectOfType<PlayerHealth>();
    }

    void Update ()
    {
        this.transform.Rotate(Vector3.forward * 5f);
    }

    void OnTriggerEnter(Collider col)
    {
        PlayerHealth playerHealth =
        col.gameObject.GetComponent<PlayerHealth>();

        if (playerHealth == null)
            return;

        // only consume the pickup if the player's health
        // is less than what he started with
        if(playerHealth.currentHealth < playerHealth.startingHealth)
        {
            playerHealth.AddHealth(restoreHealth);
            Destroy(gameObject);
        }
    }
}
```

Test these changes by letting the character incur damage. Then pass through the pickup collider. Make sure the player recovers health.

Enable the grabbable objects to reveal health pickups whenever game play moves the objects out of the way.

1. Create a new script called **HealthDrop.cs** and attach it to the grabbable block.
2. Add the following script:

```
public class HealthDrop : MonoBehaviour
{
    Rigidbody rigid;

    void Start ()
    {
        rigid = GetComponent<Rigidbody>();
    }

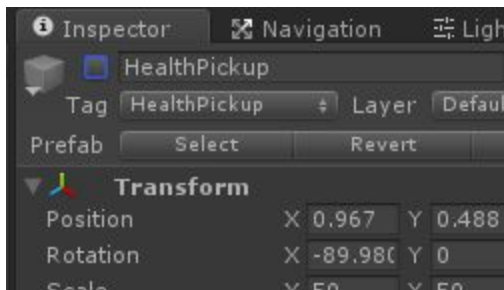
    void Update ()
    {
        // when the object is grabbed,
        // turn on and detach the pickups,
        // then turn the script off
        if (rigid.isKinematic)
        {
            foreach (Transform child in transform)
                child.gameObject.SetActive(true);

            transform.DetachChildren();

            enabled = false;
        }
    }
}
```

Health pickups are now children of the block.

3. In the editor, copy the HealthPickup twice and hide each one inside the geometry of the block evenly spaced, and parent them to the block.
4. Clear the checkbox at the top left of each pickup's inspector so that they are not active until the blocked is grabbed during gameplay.



5. Test health pickups in the game. Move a block away to reveal health pickups and make sure they are consumed when contacted by the player avatar.

## 3.D STANDARDS ALIGNMENT GUIDE

### 3.D.1 PROFESSIONAL STANDARDS FOR INTERACTIVE APPLICATION AND VIDEO GAME CREATION

#### 1. INTERACTIVE APPLICATION AND VIDEO GAME DESIGN

##### 1.1. OVERVIEW OF KEY ASPECTS IN THE DESIGN PROCESS

- 1.1.3. Explain the role of iteration in the design process
- 1.1.13.3. Explain the use of testing methods for sensory and emotional responses
- 1.1.15. Describe common hardware interface devices and their usage (keyboards, controllers, etc.)

##### 1.5. APPLICATION OF PHYSICS FOR DESIGN

- 1.5.4 Apply and manage the use of Colliders
- 1.5.9. Demonstrate the ability to handle object collisions and physics simulations in a realistic manner

##### 1.7. TOOLS AND TECHNOLOGY

- 1.7.9. Demonstrate a working knowledge of game development tools
- 1.7.15. Describe the basic logic, concepts and key structures behind computer programming languages
- 1.7.17. Determine appropriate programming and scripting languages to create desired game mechanics, control the environment, UI and gameplay

#### 2. INTERACTIVE APPLICATION AND VIDEO GAME DEVELOPMENT

##### 2.1. INTERACTIVE / REAL-TIME EDITING

- 2.1.3. Demonstrate successful navigation of 2D and 3D scenes using pan, zoom, orbit, walk-thru modes, etc.
- 2.1.4. Define objects, assets, components and properties and describe their relationships
- 2.1.5. Explain and demonstrate asset importing procedures for images, models, audio and video
- 2.1.7. Demonstrate the use of object preferences and inspector tools
- 2.1.8. Accurately transform objects with respect to coordinate systems (translate, rotate and scale)
- 2.1.9. Describe and change the active status of objects
- 2.1.10. Describe and change the status of components

##### 2.2. ASSET MANAGEMENT AND RESOURCES

- 2.2.1. Organize assets and components
- 2.2.2. Understand and use hierarchical organization structures
- 2.2.6. Create and use prefabs to save and reuse setups

## 2.3. OBJECTS & CHARACTERS

- 2.3.1. Import assets from appropriate file formats

## 2.7. SCRIPTING AND PROGRAMMING

- 2.7.1. Demonstrate an understanding of the mathematical concepts, logic and syntax of programming languages
- 2.7.2. Demonstrate an understanding of “if” and “switch” statements
- 2.7.3. Demonstrate an understanding of loops to manage recurring events
- 2.7.5. Declare and update Fields and Properties with varied access modifiers
- 2.7.6. Demonstrate an understanding of Functions, Constants and Variables
- 2.7.8. Demonstrate the usage of Initialization Functions for reference assignment
- 2.7.9. Demonstrate an understanding of proper use of Classes and Functions
- 2.7.10. Demonstrate an understanding of Object Oriented Programming
- 2.7.10.2. Explain and show examples of Encapsulation
- 2.7.10.3 Explain and show examples of Composition
- 2.7.12. Demonstrate the use of real-time data for interaction detection
- 2.7.13. Demonstrate an understanding of various programming interfaces
- 2.7.17. Use scripting/coding to create and control games objects and events
- 2.7.19. Programmatically solve issues with object interactions and events

## 2.8. HUMAN COMPUTER INTERFACE/GRAPHICAL USER INTERFACE

- 2.8.1. Explain and demonstrate principles of visual communication
- 2.8.3. Using examples, describe key principles behind graphical user interfaces (UIs)
- 2.8.4. Define usability as an objective for user interfaces (UIs)
- 2.8.6. Explain how specific UI characteristics can affect usability
- 2.8.10. Implement a new user interface system, test and evaluate its usability

## 3. INTERACTIVE APPLICATION AND VIDEO GAME DEVELOPMENT

### 3.1. TARGETING ONE OR MORE PLATFORMS

- 3.1.2.1. Demonstrate and explain considerations for user Interface design

### 3.D.2 COMMON CORE STATE STANDARDS (CCSS)

- CCSS.ELA-Literacy.RST.11-12.2 • Determine the central ideas or conclusions of a text; summarize complex concepts, processes, or information presented in a text by paraphrasing them in simpler but still accurate terms.
- CCSS.ELA-Literacy.RST.11-12.3 • Follow precisely a complex multistep procedure when carrying out experiments, taking measurements, or performing technical tasks; analyze the specific results based on explanations in the text.
- CCSS.ELA-Literacy.RST.11-12.4 • Determine the meaning of symbols, key terms, and other domain-specific words and phrases as they are used in a specific scientific or technical context relevant to grades 11-12 texts and topics.



### **3.D.3 STEM CAREER CLUSTERS (SCC)**

- SCC01 Academic foundations: Achieve additional academic knowledge and skills required to pursue the full range of career and postsecondary education opportunities within a career cluster.
- SCC10 Technical skills: Use the technical knowledge and skills required to pursue the targeted careers for all pathways in the career cluster, including knowledge of design, operation, and maintenance of technological systems critical to the career cluster.

### **3.D.5 NEXT GENERATION SCIENCE STANDARDS (NGSS)**

- Science and Engineering Practices:
  - NGSS4: Analyzing and interpreting data
  - NGSS5: Using mathematics and computational thinking
  - NGSS8: Obtaining, evaluating, and communicating information

## **3.E SUGGESTED RESOURCES**

Listed below is a recommendation of resources to consider for this unit:

- zSpace Online Developer Documents: <https://developer.zspace.com/docs/>