Kinamoto 3.0 Custom Scene Creation Guide

Introduction

When paired with the Bertec Head Mounted Display, the Bertec Kinamoto software allows users to create and upload custom scenes to use with their system, offering users a wide array of options. Bertec's Custom Scene Development Kit, which can be downloaded from within the software's user interface, has many guiding elements for users to create a custom scene that works seamlessly inside their Bertec software. This development kit includes:

- A Unity Project with example scenes
- The Bertec Code Library

The Bertec Code Library is required to build a compatible Bertec Kinamoto Scene; the Bertec Kinamoto will not use or deploy any Unity project that was not built using the Library and set up as per the Library Scene Requirements. Two different kinds of scenes can be created from the provided template Unity projects: a **visual flow scene** or a **static scene**.

- A **visual flow scene** has a central path that the subject travels forward through which then makes the surrounding scene elements go past the user in their periphery. An example of this scene type is the Bertec Park Scene.
- A **static scene**, by default, has no moving elements of the scene which the subject can focus on while walking or standing on the treadmill. An example of this scene type is the Bertec Starfield Scene.

Once the custom scene using the Bertec Code Library has been created in Unity and compiled into an Android Package, it can be used with the Bertec Kinamoto software and the Bertec Headset. Follow the remainder of this guide for further guidance on how to create, configure, and upload scenes.

Scene Types

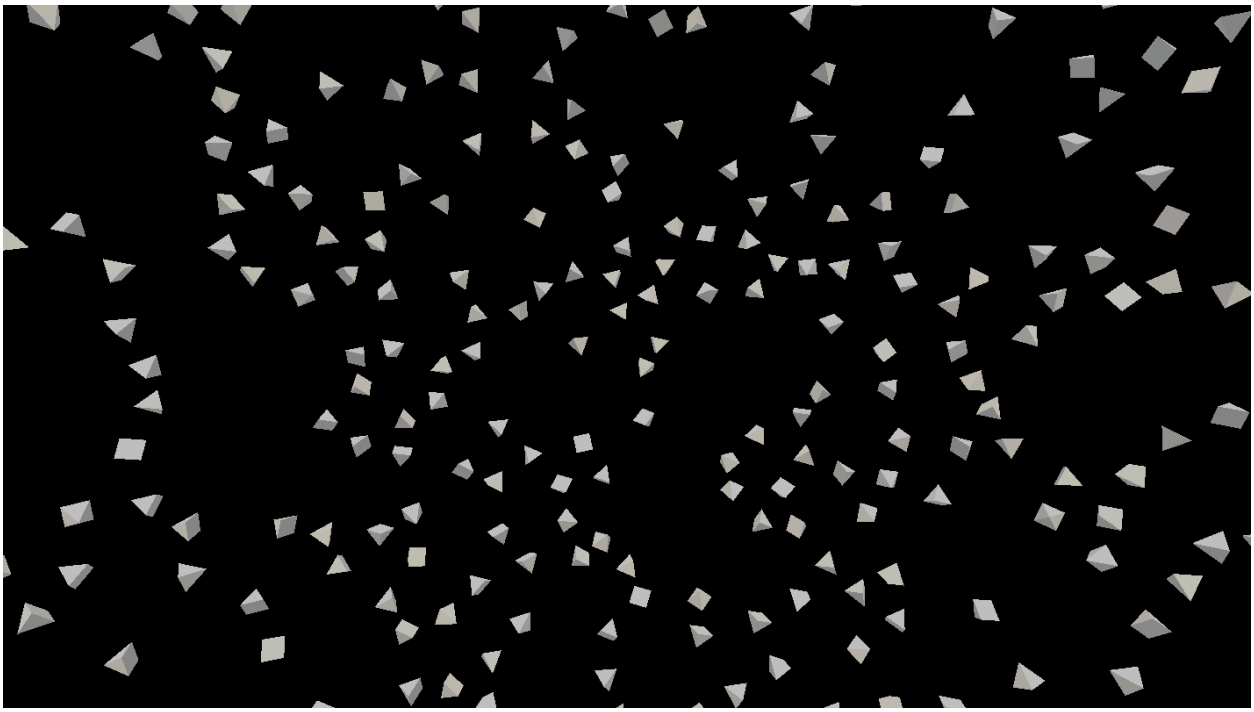
Visual Flow Scenes

Visual flow scenes are designed to immerse the subject in a dynamic, visually stimulating environment with forward movement. The movement can be either a fixed rate or dynamically matched to the Bertec Treadmill belt speed. Visual Flow scenes replicate real-world walking paths, flowing toward the subject as they walk or stand still with the headset on. Users can enhance subject adaptability, variability, and motivation by creating compelling visual conflicts using a library of stimuli that allows real-time modification of parameters. Within visual flow scenes, specific options such as scene rotation can be configured as actions that respond in real-time to conditions set up in Protocol Builder.

Static Scenes

Static scenes are designed as flat backgrounds with 3D or 2D objects layered on top. These scenes can be designed to include any background and objects the user desires. Static scenes differ from visual flow scenes in that they do not move toward the user, and they do not create the sensation of the user traveling through them.

Static scenes can have various items on the visible field, such as geometric shapes or full 3D objects. These items can be in fixed, static positions, or be designed to move in a pattern. In addition, the scene itself can be rotated around the user's field of vision using the SceneRotation feature.



Scene Controls

Visual Flow Scenes

Lighting: In Visual flow scenes, lighting can be tailored by adjusting the properties of the light sources within the scene. Unity presents a variety of light types, such as directional lights, point lights, spotlights, and area lights, each offering customizable parameters to achieve specific lighting effects. Multiple lighting options can be implemented within a scene to accommodate different environments. For instance, if the scene depicts an outdoor setting with a visible 'sky,' users can configure various lighting options to mimic different levels of sunlight at various times of the day.

Periphery: Visual flow scenes typically have objects positioned on both the left and right sides of the path that can be toggled on or off through the scene control panel in the software. These peripheral objects can be static or animated to make them feel more realistic.

Obstacles: Visual flow scenes often have objects displayed in the pathway that the subject navigates around while walking on the treadmill. Obstacles can either be static or dynamic, located on the ground or in space. Users can configure different obstacle modes by modifying frequency and distance settings to customize difficulty levels. Additionally, Audio feedback can be enabled, giving the user an indication of whether the subject successfully passed the obstacle.

Path Texture and Size: The path texture is the material of the pathway the user follows in Visual Flow scenes. Only one path texture option can be displayed at a time. The user can upload any texture to their custom scene path and switch between the types for the desired purpose.

Distractions: Objects that appear intermittently within the Visual Flow scene, designed to divert the subject's focus away from ongoing tasks or the surrounding environment. These objects are strategically introduced to capture the subject's attention, creating interruptions that may disrupt their concentration or engagement with other elements in the scene.

Visual Flow and Static Scenes

Cognitive Tasks: Simple math equations or Stroop tasks that are layered on top of the scene. Subject responses can be recorded manually by clicking a button on the UI. The percentage of incorrect and correct responses is saved.

Grain: When enabled, this visual effect applies a visible speckled texture layered on top of the scene. Different levels of the grain effect can be applied.

Center of Focus: When enabled, this visual effect creates a darker border around the scene, replicating a frame. This effect can create a focus zone for the subject to look at.

Rotation: Static and Visual Flow scenes can be rotated on the Roll, Pitch and Yaw axis. Static scenes can also rotate on the sinusoidal axis replicating an oscillating motion.

Key Point Visualizer: Objects that represent the movement of the key point being tracked. The visualizer can be one object, or a pair of objects. For example, the shoes in the Bertec Visual Flow Park Scene can move based on the movement of the subject's key point marker or COP and the yellow star in the sample visual flow scene project can be configured to function the same way.

Basic Scene Design and Configuration

Getting Started

The Sample Project is pre-configured to give you a starting point building your own Scenes. The Sample Project is already properly set up for Unity VR and the Headset with the required settings and packages so that you can successfully build a Bertec Kinamoto-compatible application.

We strongly recommend using the Sample Project as your starting point instead of creating a brand-new Unity project. Doing so ensures that the project has the proper settings, options, and packages that are needed both for Unity VR and the Headset hardware.

In addition, the Sample Project comes with several preset scenes that you can get started with:

- The **Startup Scene**, which is a blank scene that is used to bootstrap the resulting package into the running environment. This scene will automatically switch to the selected scene from the Bertec Kinamoto desktop UI. This scene should not be modified so that it can work properly with the Bertec Kinamoto.
- An example **Static** scene, with a black or blue background and some static 'bubble' objects on it. This scene itself does not move around but the user can look around themselves. This scene is very simple and is a good starting point for learning the basics. You can freely modify this scene to add your own design and features to it.
- An example **Visual Flow** scene, with some simple objects and options. The scene design and code shows how to implement an "infinite runner" style of scene. This scene also shows how Cognitive Tasks work. You can freely modify this scene to add your own design and features.

Once you have copied the original Sample Project files into a new folder, rename the project and change both the **company name** and **product name** in the Project Settings, under Player. Taken together this will form the default Android package name, which you can further adjust if needed. The package name is further down in the Player settings, under Identification.

Creating a new Scene

While duplicating one of the existing Sample Scenes and modifying it to fit your needs is the suggested path, creating a new scene is straightforward and requires a few simple steps.

Preparing the scene game objects

First, create the scene via right clicking in the Unity Project panel and select Create->New Scene. Rename this scene whatever you wish or leave it as New Scene.

Inside this new scene, select and delete the Main Camera object; this will be replaced shortly. You can leave the default Directional Light alone for now.

Select the **MainBertecController** prefab object in the Packages/Core Visual Protocol Package/Prefabs and drag it into your scene. This prefab contains the main camera, the XR controller for the headset, and other functionality.

Create a new top-level empty Game Object and name it **MainScene**. Move the default Directional Light into this as a child object.

The MainScene game object is now the "anchor" for the scene, and all visual objects should be placed under it as child objects. Failing to do so will impact the passthrough mode and cause game objects to render oddly.

Declaring the Scene

In the Project panel, right click and create a new empty C# script, giving it a name such as **NewSceneController**. Drag this new script into your MainScene object so that Unity adds this as a new component. Alternatively, you can use the Add Component button to select this new object.

Double-click this new scene script to open the editor of your choice and rename the default class name to the name you gave it (ex: NewSceneController).

Inside this new class, add a new class derived from **Bertec.SceneInfo**, and add the following attribute tag the new class:

```
[Bertec.SceneInfo(Key = "NewScene", Name = "New Example Scene", Scene = "@New Scene")]
```

This will inform the Bertec Code Library that this scene now participates in the Bertec Kinamoto ecosystem and is tagged "NewScene". When the Bertec Kinamoto lists the scene for the user to pick, it will be shown as "New Example Scene". The Scene attribute defines where the Unity scene file is at; prefixing it with the @ sign informs the Bertec Code Library to find it by filename instead of full Assets/Some/Folder pathname.

The SceneInfo class can be either enclosed in another class or be a top-level class.

Defining and reacting to Options

Options are items that will be presented to the user in the Bertec Kinamoto and are defined within the SceneInfo class. Each option can be either a simple on/off checkbox, a list of choices (typically presented a drop-down list), a range of values (usually a slider), or a freeform text entry box. A Scene can contain as few or as many options as needed, or none at all. Options can be changed dynamically by the Bertec Kinamoto as part of a test protocol or reaction group.

To define an option, use the AddOption function call from within your SceneInfo constructor, like so:

```
public NewSceneInfo()
{
  var boxcolor = AddOption(new Bertec.SceneOption()
  {
    Type = Bertec.SceneOption.OptionType.Choicelist,
    Key = "boxcolor",
    Name = "Box Color",
    Choices = new List<SceneOptionChoice>()
    {
      { "red", "Red Color" },
      { "green", "Green Color" },
      { "blue", "Blue Color" }
    },
    DefaultChoice = "green"
  });
}
```

This code snippet defines a list of options for a box color of red, green, and blue, with the default color being green. In this example, the boxcolor option will be presented to

the user as Box Color, while the three choices will be shown as Red Color, Green Color, and Blue Color in the desktop UI. Note that while this example appears to be nothing more than a lowercase-uppercase conversion, you can easily declare the choice key values as anything you want such as “4”, “snake”, and “foobar” and the choice ‘labels’ as “Low”, “Medium”, and “Plaid”. The choice key – “red” or “4” – are separate and distinct from the label “Red Color” or “Low”.

Each time the user selects the option and changes the value or the Bertec Kinamoto changes the value while running the protocol, the Bertec Code Library will trigger an OnOptionChanged event with the Option Key (“boxcolor”) and the Choice Value Key (“red”, “green”, or “blue”) as parameters – you do not get the label text, only the value (so “red” not “Red Color”). The OnOptionChanged event is part of the ProtocolOptionChangedMonoContainer, which in turn is a component of the MainBertecController prefab game object in your scene.

Connecting your Scene’s GameObject to the OnOptionChanged event requires nothing more than calling the static ProtocolOptionChangedMonoContainer.Connect function in your Awake function, as shown:

```
void Awake()
{
  Bertec.ProtocolOptionChangedMonoContainer.Connect(OptionChanged);
}

void OptionChanged(string key, object val)
{
  if (key == "boxcolor")
  {
    Color c = Color.black;
    if (val.ToString() == "blue")
      c = Color.blue;
    if (val.ToString() == "green")
      c = Color.green;
    if (val.ToString() == "red")
      c = Color.red;
    // Do something with the color
  }
}
```

Once the Awake call has been completed, each time the user changes the option either directly or under Bertec Kinamoto actions, your OptionChanged function will be called with the option key string and the value.

Multiple options can call your OptionChanged function multiple times with different key:value pairs. An example of how to handle more than one option key is in the Example Visual Flow scene's OptionChanged function.

For more information about the various options and the functionality provided for them, please consult the Bertec Code Library Reference Guide.

Scene Features

Certain levels of functionality are defined as global Scene Features – these are separate from Scene Options in that they are less free-form and are controlled explicitly by both the Bertec Code Library and Bertec Kinamoto.

One of the key Features is the **VisualFlow** flag, which determines if the scene is a Visual Flow (moves with the belt or at a fixed rate) or a Static (does not move) scene. There are several other flags, including **HasPostProcessing** (allows for changing the visual look of the entire display), **ScreenRotation** (allows for things like angle changes and sway movement), and **Cognitive**, which defines types of cognitive tasks the user must perform.

To set the Scene Features, modify your SceneInfo-derived constructor and set the Features object to a new **SceneFeatures** class with the flags you wish to use set. For example:

```
public NewSceneInfo ()
{
  Features = new SceneFeatures
  {
    VisualFlow = true,
    HasPostProcessing = true,
    ScreenRotation = ScreenRotations.RotateXYZ
  };
}
```

This will add the Scene Feature that enables the Belt Speed Control, Post Processing and Screen Rotation panels in the Bertec Kinamoto UI.

For additional Feature information and examples, please consult the Bertec Code Library Reference Guide.

Adding the Scene to Kinamoto

After you complete your scene, you can build it using the Unity Build function under the Build Settings menu item. We recommend leaving the preset options as-is, since this will generate the optimal APK file and program.

After the APK is built to the folder of your choice, connect your headset to the PC using the USB cable and add the Scene to Kinamoto using the Scenes -> Add/Update button [[TODO]]. This will copy the scene from the PC folder into the headset and activate it. Then, select the Scene from the drop-down list in Kinamoto and the headset will switch to it. Any options or Features you have set in your Scene will become available in the Kinamoto UI.

Each time you make a change to your scene, re-add the built APK file to Kinamoto and allow it to synchronize it to the headset. If you do not do the re-add and re- synchronize steps, the updated scene will not be available for use.

Important Considerations

Kinamoto-compatible scenes require the Bertec library and specific scene + project settings. It is always best to start a new scene using the example Scene template project and build upon it from there. We do not recommend starting a new scene or Unity project from scratch unless directed by Bertec Customer Success.

The headset is an Android device that puts limitations on what your program can do. Memory, CPU, and GPU resources are limited so your scene and graphic designs must take that into account.

The maximum size of a compiled scene APK file is 256megs; anything larger will not be usable on the headset.

Like all VR systems, the headset battery life is limited; when not in use we recommend you keep the headset charged.

To maximize battery life, the Android operating system on the headset will prefer to put the headset in “standby” mode when the headset is off the user for about 20 seconds. In this mode the screen will blank but the headset remains connected to the Kinamoto

desktop program. When the headset is not in use for roughly 5 minutes the headset will "sleep" and disconnect from the PC. Note that depending on product use and projected battery life, the headset may put the headset into standby and sleep modes in less than the typical times of 30 seconds and 5 minutes.

Placing the headset back on the user will "wake it up" and reconnect to the PC.

Best Practices

Scenes should be designed to use as little GPU memory and CPU resources as possible. Reducing texture usage and object polygon count improves both performance (FPS) and the user experience and improves battery life. A consistently high frame rate (targeting 60 FPS or above) is crucial for a comfortable VR experience; with careful design, 75 FPS can be achieved. Dropped frames and low frame rates can lead to motion sickness and a less immersive experience. **At a minimum, your scene should run no less than 30 FPS, with 60 FPS being the goal**.

Designing for a PC and designing for VR are vastly different things since VR has a much weaker processor than a regular PC. Designing for VR requires careful consideration of performance to ensure a smooth and immersive experience for users, especially given the limitations of VR hardware.

Model Specifications

Even though VR can now somewhat handle high-resolution models with realistic materials it is always safe to stick to *low poly* models to ensure a smooth experience for users.

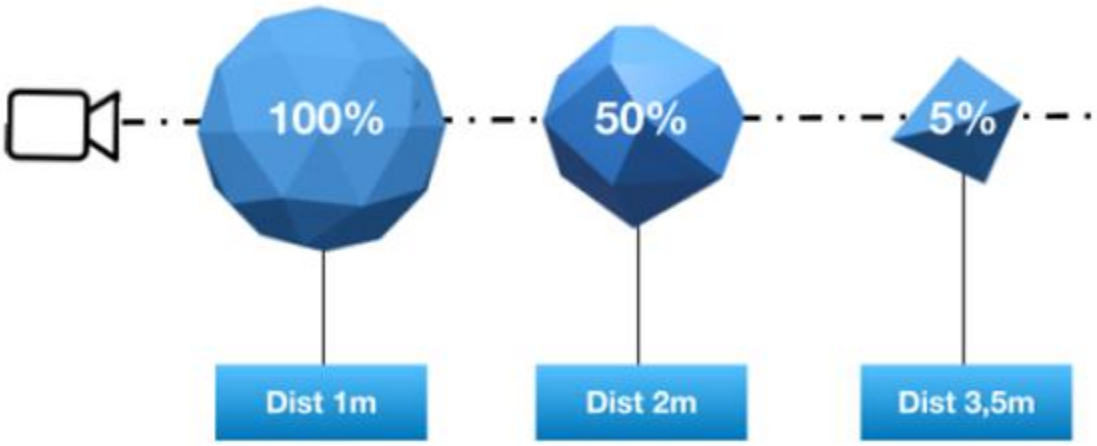
- For *smaller objects* and props that are not the focal point of the scene, aim for polygon counts between 1,000 to 10,000 triangles.
- *Medium complexity* models that are more prominent in the scene or require closer inspection, such as characters or interactive objects, can have polygon counts ranging from 10,000 to 30,000 triangles.
- *High-detail* models, such as characters with intricate designs or key scene elements, may have polygon counts between 30,000 to 50,000 triangles. Due to hardware limitations, strive to keep your usage of high-detail models as low as possible, and restrict them to one or two on the screen at once.

These polygon counts represent maximum levels. For better optimization, always aim to keep polygon counts as low as possible without sacrificing quality. As a general rule, try to keep the entire scene under 100k polygons, optimizing individual models to stay within reasonable limits, such as under 5k polygons for complex objects.

It's essential to prioritize performance and optimize models based on their importance to the scene. Additionally, using techniques like *LODs (Level of Detail), texture baking, and mesh simplification* can help manage polygon counts while maintaining visual fidelity in VR environments.

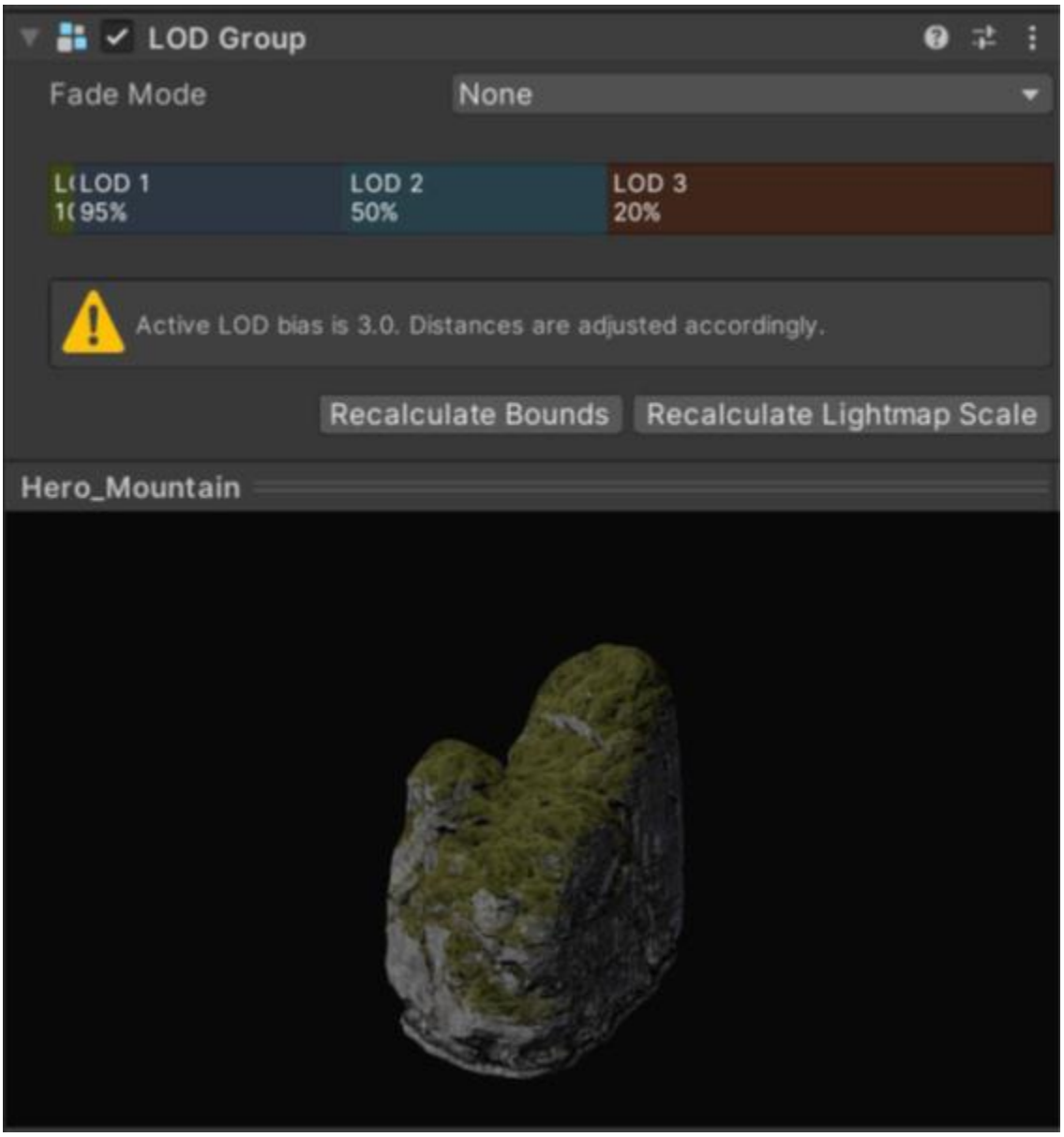
LODs (Level of Detail)

When a Game Object in the scene is far away from the camera, the level of detail is less than a Game Object that is close to the camera. And even though you can't see the detail on a distant Game Object, Unity uses the same number of triangles to render it at both distances. This can cause problems in VR since it runs with a considerably weak processor. *Ensure that your elements will always be higher in detail if the user can move close to it and put fewer details on areas that are for background purposes only. You can even use 2D for objects that are close to the horizon.*

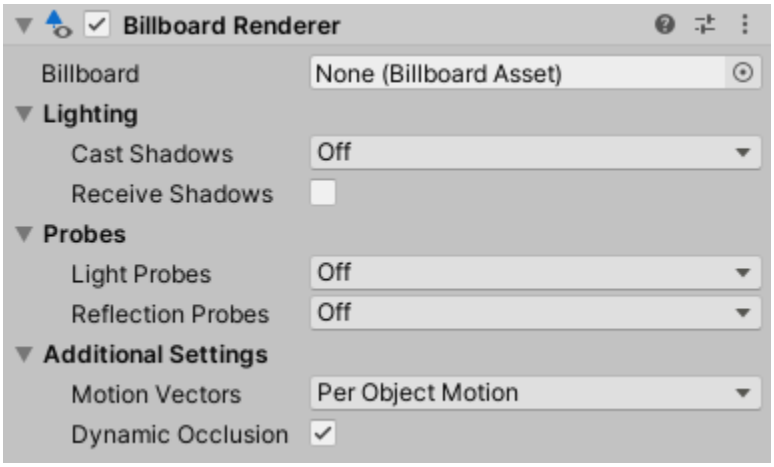


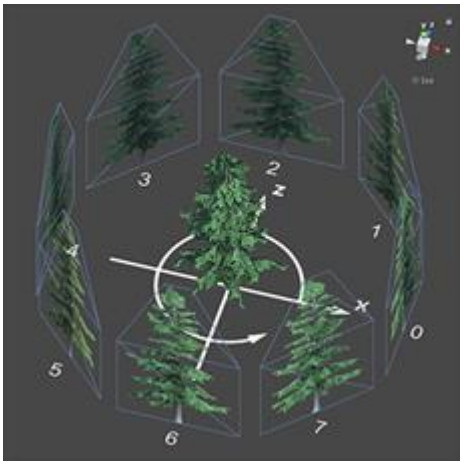
To optimize rendering, you can use the *Level of Detail (LOD)* technique. It allows you to reduce the number of triangles rendered for a Game Object as its distance from the camera increases. You use several meshes and optionally a Billboard Asset, which all represent the same Game Object with decreasing detail in the geometry. Each of the Meshes contains a Mesh Renderer component and represents a 'Mesh LOD level', while

the Billboard Asset has a Billboard Renderer component and represents a 'Billboard LOD level'.



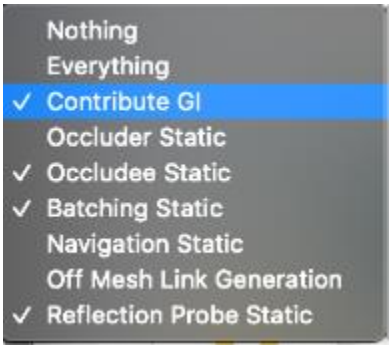
The *Billboard Renderer* renders Billboard Assets. Billboards are a level-of-detail (LOD) method for drawing complicated 3D Meshes in a simpler way when they are far away from the Camera. When a Mesh is far away from the Camera, its size on the screen means there is no need to draw it in full detail. Instead, you can *replace the complex 3D Mesh with a 2D billboard representation*.

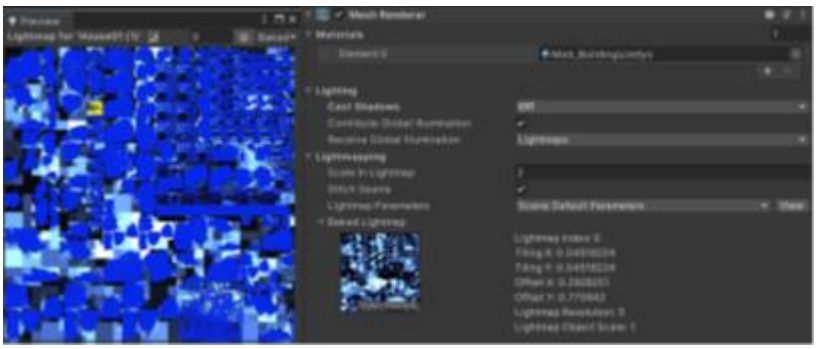




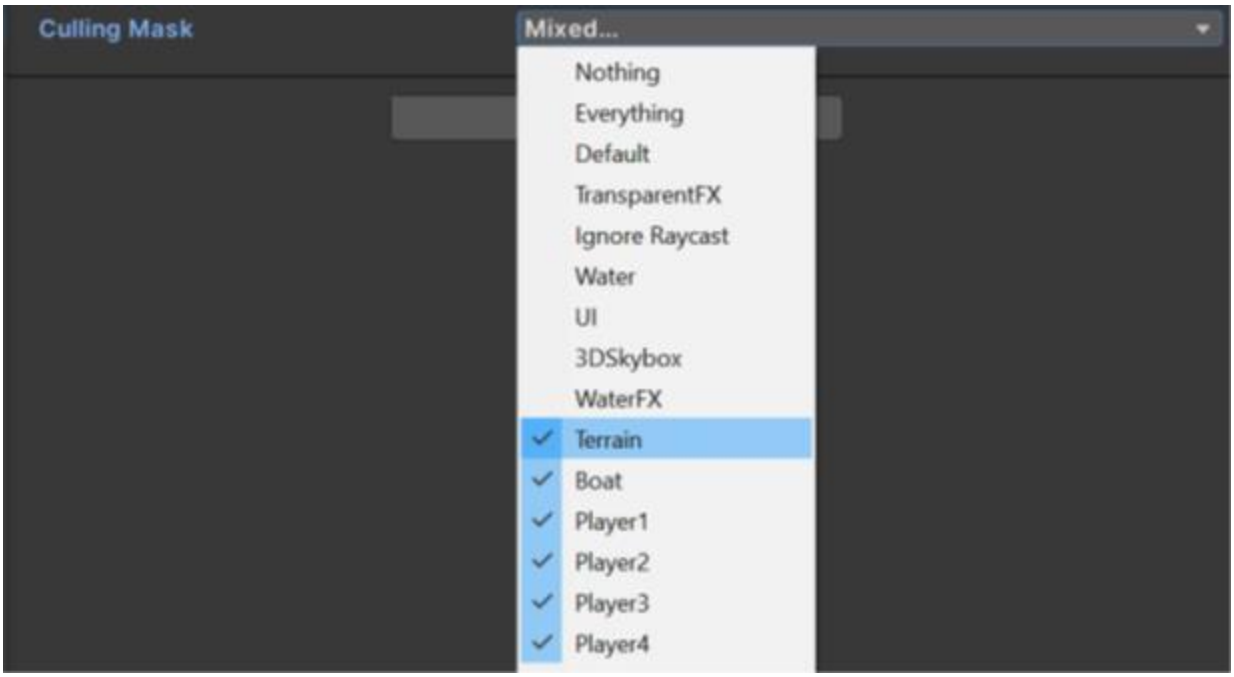
Scene Lighting

For optimizing static scenes, consider baking lighting into Lightmaps. You can add dramatic lighting to your static geometry using *Global Illumination (GI)*. Mark objects with Contribute GI so you can store high-quality lighting in the form of Lightmaps. Baked shadows and lighting can then render without a performance hit at runtime. *The Progressive CPU and GPU Lightmapper can accelerate the baking of Global Illumination.*





For complex scenes with multiple lights, separate your objects using layers, then confine each light's influence to a specific culling mask.



Light Probes store baked lighting information about the empty space in your scene while providing high-quality lighting (both direct and indirect). They use Spherical Harmonics, which calculates quickly compared to dynamic lights. This is especially useful for moving objects that normally cannot receive baked lightmapping. Layers can limit your light's influence to a specific culling mask.

Light Probes can apply to static meshes as well. In the MeshRenderer component, locate the Receive Global Illumination dropdown, and toggle it from Lightmaps to Light Probes. Continue using lightmapping for your prominent level geometry but use probes for smaller details. Light Probe illumination does not require proper UVs, saving you the extra step of unwrapping your meshes. Probes also reduce disk space since they don't generate lightmap textures.

Material & Texture Optimization

In VR, the complexity of materials can have a significant impact on performance. High-quality materials often use multiple shaders and textures, which require more processing power. Since VR scenes are rendered twice (once for each eye), the strain on the GPU is even higher than in standard applications. To optimize materials, simplify their structure by using fewer shaders and reducing the number of textures involved. Unity's Standard Shader is a good choice, but be sure to adjust its settings for performance, avoiding expensive features like reflections or transparency unless absolutely necessary. Reducing the number of materials applied to a single Game Object

also minimizes draw calls, which helps maintain a stable frame rate. Where possible, reuse materials across multiple objects to further reduce the workload on your system.

High-resolution textures can greatly enhance visual quality, but they come at a cost in terms of memory usage and processing power. In VR, where performance is critical, it's important to balance texture quality with efficiency. Aim to use compressed textures whenever possible and avoid unnecessarily high resolutions—1k or 2k textures are generally sufficient for most objects in VR. Additionally, always generate mipmaps for your textures. Mipmaps are smaller versions of the texture that Unity can use when the object is far from the camera, reducing the workload on the GPU. This technique can prevent texture aliasing and improve overall performance, particularly in scenes with many distant objects.

Occlusion Culling

Objects hidden behind other objects can potentially still render and cost resources. Use Occlusion Culling to remove hidden objects. While frustum culling outside the camera view is automatic, occlusion culling is a baked process. Simply mark your objects as Static Occluders or Occludees, then bake via *Window > Rendering > Occlusion Culling*. Though not necessary for every scene, culling can improve performance in specific cases, so be sure to profile before and after enabling occlusion culling to check if it has improved performance.

Memory Management

Managing memory efficiently is essential in VR development, as excessive memory usage can lead to crashes or noticeable performance drops. Unity provides tools like the Profiler to help you track memory allocation and identify potential issues. One of the key areas to focus on is asset optimization—compress textures, models, and audio files to reduce their size. Additionally, make sure to unload assets that are no longer needed in the scene. Unity's Resources.UnloadUnusedAssets() function can help with this. Be mindful of how much data you're loading into memory at any given time, and aim to keep your memory footprint as low as possible, especially on lower-end VR hardware.

Draw Calls

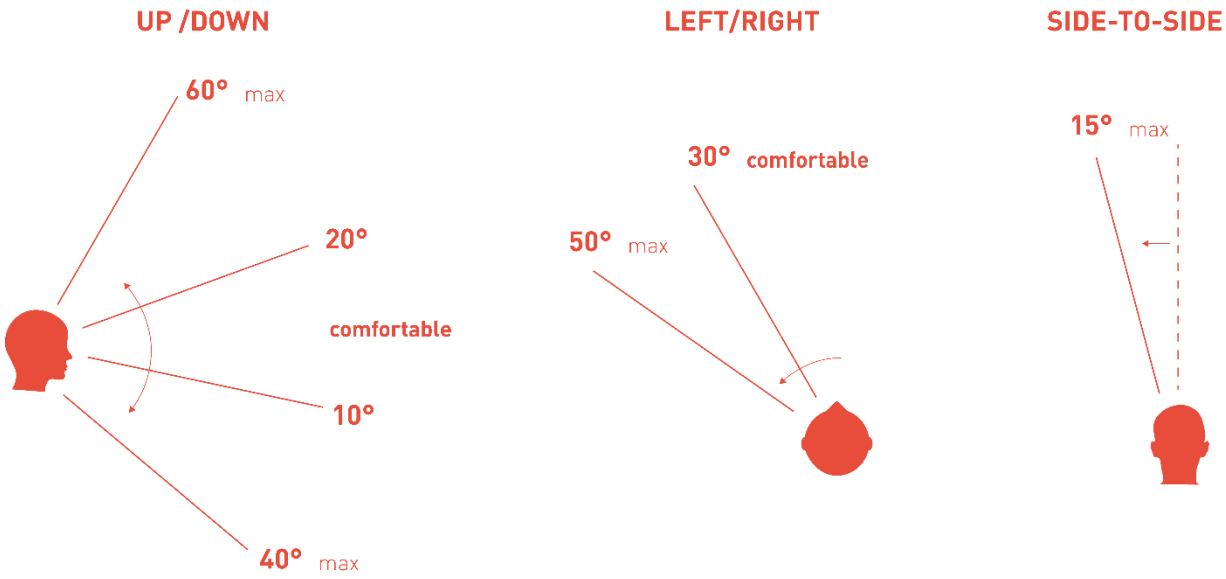
In VR, minimizing draw calls is crucial for maintaining a high frame rate. A draw call is a command sent to the GPU to render an object, and too many draw calls can quickly overwhelm the system, leading to frame rate drops. To reduce draw calls, combine meshes that share the same material whenever possible, as this allows Unity to render them in a single draw call. Static batching can also be used for objects that don't move, which reduces the number of draw calls by combining them at runtime. For dynamic objects, consider using dynamic batching. Additionally, using texture atlases—large textures that contain multiple smaller textures—can reduce the number of material switches, further optimizing draw calls. Always monitor your draw calls using Unity's Profiler to ensure they're within an acceptable range for VR performance.

User Experience

Viewing Zones and Placement of the Components

VR allows creators to utilize a full 360 canvas, with depth, so presenting information and guiding the user can be a bit trickier. There are certain comfort zones to consider for people to view information without straining too much physically or making movements that go beyond the limitations of the human body.

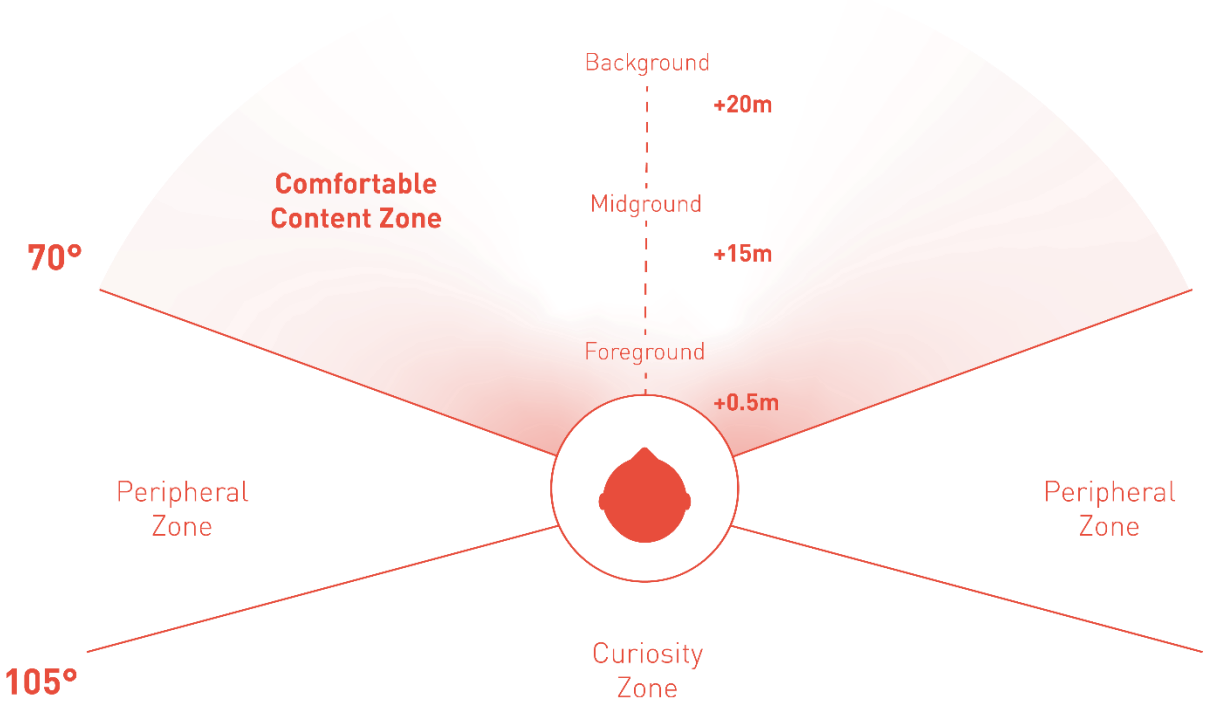
The field of view in a VR head-mounted display (HMD) is around 94 degrees. Assuming you are in a seated position, you can comfortably rotate your head 30 degrees to the side, up to a maximum of 50 degrees. These numbers increase if you're standing or in a swivel chair with a wireless HMD.



The minimum comfortable viewing distance for UI, before a user starts going cross-eyed, is 0.5m. Beyond 15 meters the sense of 3D stereoscopic depth perception diminishes rapidly until it is almost unnoticeable beyond 20 meters. This gives us a sweet spot between 0.5 meters to 15.0 meters where we can place important content.

- 0.5m - Minimum Comfortable Viewing Distance
- 15m - Strong Stereoscopic Depth Perception
- 20m - Limit of Stereoscopic Depth Perception

VIEWING ZONES



In the Visual Flow Scene, the surrounding scene elements and the other scene components are placed in consideration of the viewing zones so that it won't strain the eyes and neck.

- CoP/Key Points – If the user is viewing the scene in the third-person view, they should be placed at eye level right between the midground and foreground.
- Obstacles – Obstacles should be inside the comfortable content zones and the animations start playing when they reach midground so that the user would both be able to identify them and have time to react before they reach them.
- Cognitive Tasks – Like obstacles, cognitive tasks have to be inside the comfortable viewing zones but since it is more comfortable on the eyes and neck to look up than down, they can be placed above the eye level while not exceeding comfortable viewing zones.
- Distractions – Distractions can be placed both in the comfortable content and peripheral zones but in both cases, they should not be at the center blocking the user's point of view. For the best performance, they can be dynamic coming from the peripheral zone and entering the comfortable content zone.

- Surrounding Scene Elements – These elements should cover the whole viewing angle so that they can create an immersive experience but the important elements like peripheral density options should be placed such that they can move from the comfortable zone to the peripheral zone and exit the scene while the scene is moving.

Positioning the Subject

In the HMD our goal is to create an immersive experience and how the user views the scene is very important. Two commonly used viewing angles in VR are first-person and third-person viewing angles.

- First-Person Viewing Angle: In first-person locomotion, the user's viewpoint corresponds directly to the in-game character's perspective. Movement in the VR environment is controlled by the user's physical movements or using input devices such as hand controllers. First-person locomotion can provide a sense of immersion but may induce motion sickness in some users, particularly during rapid or continuous movement.
- Third-Person Viewing Angle: Third-person locomotion involves controlling a character or avatar from a perspective external to that character/avatar, typically positioned behind or above them. Users navigate the VR environment by

controlling the character's movements, while maintaining a fixed viewpoint relative to the character. Third-person locomotion can reduce motion sickness by providing a stable reference frame and minimizing disorientation, making it a preferred option for users sensitive to motion sickness.

In HMD scenes we mainly use 3rd person viewing because it is important for the user to see their movement inside the scene. This creates a better experience and gives subjects room to think before facing obstacles or cognitive questions. In particular, the Visual Flow Scene can be straining for some users if they constantly look down for the obstacles while on the treadmill.

However, in some scenes, it is possible to use more than one viewing angle option to create different test experiences since first-person is more immersive than third-person. In those cases, it is important to consider the camera placements for a better user experience.

Visual Accuracy: Proportions and Textures

Proportions: In the realm of VR, where immersion is paramount, attention to scaling and proportions is vital to creating a believable and captivating experience. Ensuring that objects within the virtual environment align with real-world dimensions not only enhances immersion but also contributes to usability and comfort for users. In order to ensure that there are two things you need to do:

- Reference Real-world Dimensions: Use real-world measurements as a reference when designing objects and environments in VR. Consider the average human height and proportions to ensure that objects align with users' expectations.
- User Scaling Options: Provide users with options to adjust their virtual avatar's height or scale to better match their real-world dimensions. This customization enhances comfort and immersion for users of varying heights.

Models and Textures: Creating realistic and immersive visuals in a VR scene is crucial for enhancing users' sense of presence and embodiment. One key aspect of achieving this immersion is the use of appropriate models and textures within the virtual environment.

The use of realistic human models and body parts inside the scene might enhance the immersion since the user would be able to recognize and embody those elements. However, the use of super realistic human models and body parts can strain the processor of HMD and losing the realism by using an unrealistic model can be off-putting for the user.

The use of mesh models and faded materials are a better solution to create a sense of immersion and embodiment inside virtual reality. In this case, the human brain identifies the object in front of it and fills the empty image on its own.

Interactions: Feedback and Consistency

Feedback: In the immersive world of VR, the user's experience hinges not only on visual aesthetics and interactivity but also on the seamless communication of actions and consequences. In the virtual world the user has no ability to literally touch and feel things – this is where feedback plays a pivotal role. Feedback serves as the bridge between user actions and their consequences within the virtual environment. Whether it's interacting with objects, navigating through spaces, or triggering events, users should receive instant feedback – either visual or auditory - to validate their actions.

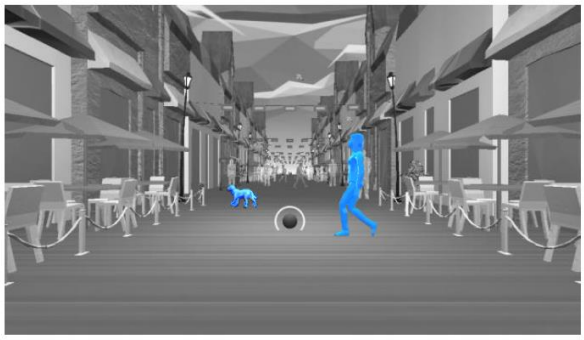
- Visual Feedback: Utilize color changes, animations, particle effects, or UI overlays to convey information and highlight interactive elements.

- Auditory Feedback: Incorporate sound cues or spatial audio to provide context, alerts, or confirmation of actions.

Consistency: In the design of a VR scene, it is crucial to maintain consistency in the design language across different elements and feedback styles. Each of these elements serves a unique purpose and should adhere to a consistent visual style within its respective environment. Consistency enhances the environment's aesthetic appeal and promotes clarity and ease of interaction for users. Inconsistencies in the design language of different scene elements such as variations in color, transparency, or visual style, can lead to confusion and hinder users' ability to navigate and interact with the environment effectively.

DO

DON'T



DO

DON'T

FAQ

1. Settings Configuration:

- Why am I encountering **build errors or issues** with my VR project?

  You may encounter build errors or issues if you haven't switched the **build target to Android** in Unity's Build Settings. Ensure that the build target is correctly set to Android to ensure compatibility with the HMD. Additionally, verify that the necessary VR SDKs, such as the Pico SDK, are imported and configured in Unity to avoid compatibility issues and ensure access to VR features specific to the Pico 4 platform. The sample project comes preconfigured with the necessary Unity and Bertec packages and should be used as a starting point when creating your own scenes or projects.

- What should I do if I'm experiencing compatibility issues or missing VR features in my Unity project for the Pico 4 headset?

  If you're experiencing compatibility issues or missing VR features in your Unity project for the Pico 4 headset, it may be due to not importing or configuring the necessary VR SDKs. Make sure that you have imported and configured the Pico SDK in Unity to ensure compatibility and access to Pico 4-specific VR features. Additionally, check for updates to the Pico SDK and firmware to ensure compatibility with the latest Unity versions and Pico 4 hardware.

2. Asset Integration and Management:

- Why is the size of my imported model incorrect and how can I fix it?

  The size of your imported model might be incorrect due to different export settings or scale factors used in various modeling software or formats. Unity uses a 1:1 scale where 1 unit equals 1 meter. To ensure your models fit correctly in the provided sample scene, check and adjust the export settings in your modeling software to match this scale. Additionally, in Unity, you can modify the scale of imported models through the Import Settings in the Inspector by adjusting the "Scale Factor." Compare your model with known references in the scene to verify proper sizing, and ensure all transformations (scaling, rotation, translation) are reset or applied in the modeling software before export. Consistently managing these factors will help maintain the correct scale and proportion of all your scene components.

- Why are some assets appearing with pink (magenta) material in my Unity scene?

  The pink material indicates that the shader or material assigned to the object is missing or not found. To resolve this issue, ensure that the shader or material used by the asset is properly imported and assigned in Unity. Check the asset import settings and re-import the asset if necessary to ensure all required files are included.

- Why do imported assets appear white or with no materials applied in the scene, even though they have materials included?

  This issue may occur due to incorrect import settings or compatibility issues with Unity versions. Double-check the import settings for the asset and make sure that materials are correctly assigned upon import. If the problem persists, consider updating Unity to a compatible version or re-importing the asset with adjusted settings. If the place you downloaded the asset came with the materials in a different folder, make sure that you also import those materials into your Unity project.

- How can I ensure that materials are correctly assigned to objects in my VR scene?

  To avoid incorrect material assignments, double-check that materials are properly assigned to objects in the Unity Inspector window. Ensure that materials are assigned to the appropriate slots (e.g., Albedo, Normal Map) and that the correct shader is selected for the desired visual effect. Additionally, use prefabs or material presets to maintain consistency and streamline material assignment across multiple objects.

- Why do my textures appear distorted or do not render properly in VR?

  Texture import settings can affect how textures are displayed in the VR scene. Check the import settings for your textures and ensure that compression settings, alpha channel handling, and texture size are appropriate for your project requirements. Experiment with different import settings to achieve the desired visual quality and performance in VR.

- Assets appear differently in my VR scene compared to the Unity Editor. What could be causing this inconsistency?

  Discrepancies between the scene view and VR environment may result from improper lighting and reflection settings. Adjust lighting conditions and reflection probes in your VR scene to match the desired look and feel. Consider using baked

lighting and optimizing real-time lighting calculations to improve performance and visual fidelity in VR.

- How can I avoid unintended changes or inconsistencies when working with prefab instances and variants in my VR project?

  When modifying prefabs, be mindful of whether you are working with prefab instances or prefab variants. Prefab instances maintain a link to their original prefab, so changes made to instances will affect all instances of the prefab. Prefab variants allow for variations of a prefab without affecting the original prefab or other variants. Use prefab variants for asset variations and modifications to prevent unintended changes and maintain project organization in your VR scene.

- Why are my assets not appearing or working correctly in the scene?

  Assets may not appear or function correctly in the scene due to various factors, including the use of high-polygon models, large textures, or inefficiently designed assets that can impact performance in VR environments. When downloading assets, always check if they are compatible with VR and consider factors such as polygon count and texture resolution to ensure they meet the performance requirements of your VR project.

- Why are objects in my VR scene appearing too large or too small?

  Ensure correct scale and proportions in Unity when designing VR scenes by using Unity's standard cube (1x1x1) to represent the size of a person. Scale objects and environments relative to this standard to maintain accurate proportions and provide a realistic user experience. Considerations for proportions are essential for creating immersive and comfortable VR environments.

3. Accessibility and Inclusivity:

- Are there alternative locomotion methods for users sensitive to motion sickness?

  Yes, for users sensitive to motion sickness, consider offering alternative locomotion methods such as first-person and third-person viewing angles. Implementing teleportation, blink movement, or comfort modes can help mitigate motion sickness and provide a more comfortable VR experience for users.

- What are some considerations for designing inclusive experiences in VR?

  When designing VR experiences, it's important to consider accessibility for all users. One crucial aspect to consider is color blindness. It is highly recommended to use a black-and-white filter to check the contrast ratios of the scenes and determine if the objects and colors are distinguishable for accessibility. By ensuring sufficient contrast between elements in the VR environment, you can improve visibility and usability for users with color vision deficiencies. Additionally, consider providing alternative visual cues or text-based indicators to convey information effectively regardless of color perception. Prioritizing accessibility features in your VR experience can enhance usability and inclusivity for a wider range of users.