

Unity VR Essentials: A Beginner's Guide

Introduction

Unity, created by **Unity Technologies**, is a cross-platform game engine—a set of well-established, reusable tools primarily designed for game development¹. However, the tools provided by the software offer a wide range of applications beyond game development, including architecture, interior design, advertising, and animation production².

At the beginning of the 21st century, the booming gaming industry introduced affordable and superior alternative tools, breaking the monopoly of outdated software specialized in Virtual Reality (VR). Unity exemplifies this trend, excelling in VR by extending its effectiveness in video game development. It offers a variety of valuable tools for designing environments that can be used for commercial and scientific purposes, as demonstrated by Lv et al. (2013)³.

Mastering all the tools and features available in Unity would take years. Moreover, there are extensively detailed guides for each of the engine's concepts and tools. To support further efforts and contributions to ongoing and future VR projects, this wiki is designed to serve as a concise guide to some of Unity's essential tools, providing a foundational understanding for a quick start with VR development.

This wiki is based on Unity version 2020.3. Please note that some instructions may differ depending on the software version used.

Why Unity?

Considering the compelling argument that the physical world plays a significant role in the development of human intelligence⁴, the development of Artificial Intelligence (AI) algorithms is highly dependent on the research community's ability to create realistic environments and present increasingly complex challenges². This provides common ground for researchers and game developers, as they face similar challenges when trying to understand human behavior. Although this understanding can be employed to answer questions about the fundamentals of brain function or to increase customer engagement with a product⁵, there is a growing emphasis on the need for a beginner-friendly, high-performing simulator that can facilitate the creation of increasingly realistic environments².

Juliani et al. (2018)² proposed a taxonomy that classifies existing game engines based on their ability to create complex scenery across multiple dimensions. Notably, Unity was identified as the sole engine in the general platform class, which includes engines capable of creating environments with arbitrarily complex visuals, physical and social interactions, and tasks. This class is distinguished by its ability to replicate environments created by other engines while also facilitating the creation of novel, complex environments. It must be noted that this paper was authored by a team affiliated with Unity Technologies.

According to a survey of over 350 participants conducted by Unity Technologies and published in their 2021 Unity Gaming Report, Unity was the preferred engine for 61% of game developers. The engine's cross-platform capabilities are highly valued by users⁶, supporting deployment on more than 25 platforms.

Unity's popularity fosters a large user base, which accumulates extensive experience and generates a wealth of learning materials and resources. This has led to the formation of an active community that provides tried-and-tested solutions for common issues and innovative ideas for tackling complex and novel problems. Notably, Unity's community thrives despite the fact that Unity is not entirely open-source, although it does offer some open-source resources and packages. One of Unity's significant advantages is its extensive community support, combined with an array of tools that provide seamless control over various building blocks and notable support during implementation, such as the brief pop-up documentation available for each building unit.

Unity is also compatible with several physics engines, such as PhysX and Havok, which facilitate the creation of realistic movements. Additionally, it offers the High-Definition Render Pipeline (HDRP), a preprogrammed tool that allows developers to quickly create realistic graphics. Furthermore, the engine is characterized by a lack of restrictions during the creation process, making it a perfect tool for both simple and complex scenery.

The competition between **Unity** and **Unreal Engine** is a common point of discussion. Both engines are industry leaders, serving as cornerstones for developing renowned games across different genres, including 2D, 3D, and VR. While both engines offer extensive learning resources, Unity provides a more accessible entry-level programming experience due to its support for C#, a programming language generally considered easier to learn than C++, which Unreal Engine supports. Additionally, Unreal Engine relies on engine-specific code that may not always align with general C++ applications, potentially requiring developers to adapt their skills.

In the taxonomy proposed by Juliani et al. (2018)², Unreal Engine was not included in the survey, despite theoretically meeting the criteria for the general platform class. This omission was due to the lack of fundamental predefined functions and VR frameworks, which would require additional effort to establish the necessary tools for creating VR environments, thereby significantly increasing the workload.

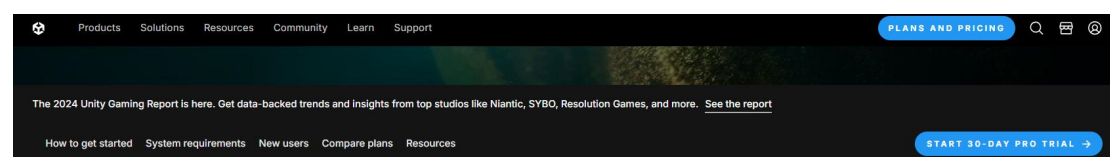
It is worth noting that Unreal Engine excels in producing high-fidelity, triple-A level visuals, making it superior for projects requiring exceptional graphical quality. However, this comes at the cost of higher processing power requirements, and its user interface can be daunting for beginners. While Unity does not offer graphical tools as advanced as Unreal's, it still allows developers to achieve high-quality visuals, albeit with more effort. Unreal's advantages over Unity extend beyond graphical capabilities, particularly in complex projects, and are not limited to basic non-commercial Extended Reality (XR) experiments.

In summary, Unity is a cost-effective, user-friendly, and time-efficient platform for creating high-quality environments, thanks to its intuitive User Interface (UI), extensive cross-platform support, and robust development tools.

Installing Unity

1) Download the **Unity Hub** from the link below:

<https://unity.com/download>



Create with Unity in three steps

1. Download the Unity Hub Follow the instructions onscreen for guidance through the installation process and setup. Download for Windows Download for Mac Instructions for Linux	2. Choose your Unity version Install the latest version of Unity, an older release, or a beta featuring the latest in-development features. Visit the download archive	3. Start your project Begin creating from scratch, or pick a template to get your first project up and running quickly. Access tutorial videos designed to support creators, from beginners to experts. Access our Pro Onboarding Guide
---	---	--

Figure 1: Unity's download website

The hub acts as a management portal for all the different projects. From there, it is possible to launch existing projects and create new ones. The hub allows the user to start individual projects using

different Unity versions at the same time. Verifying and changing the versions allocated to the projects can be done from inside the hub.

2) In the Unity download archive, it is possible to choose the desired version to download.

<https://unity.com/releases/editor/archive>

User Interface (UI)

The main UI consists of four primary panels (Fig. 2), each handling a different aspect of managing and editing **GameObjects**, which are the fundamental building blocks of any Scene. GameObjects are essentially empty placeholders with three main attributes: position, scale, and rotation. These placeholders can be filled with various **components** that give GameObjects their functionality by defining their behavior, appearance, and interaction with other GameObjects. Every file used in the creation process is referred to as an **asset**, regardless of its type (e.g., images, sounds, textures, etc.), and can be either created within Unity or externally imported. **Prefabs** are a special type of asset (see importing prefabs). Additionally, users can create new C# **scripts** and attach them to GameObjects to achieve desired behaviors (Video 1, timestamp: 7:05).

Please note that in some tutorials, the interchangeable use of the terms "components," "assets," and "prefabs" can lead to confusion.

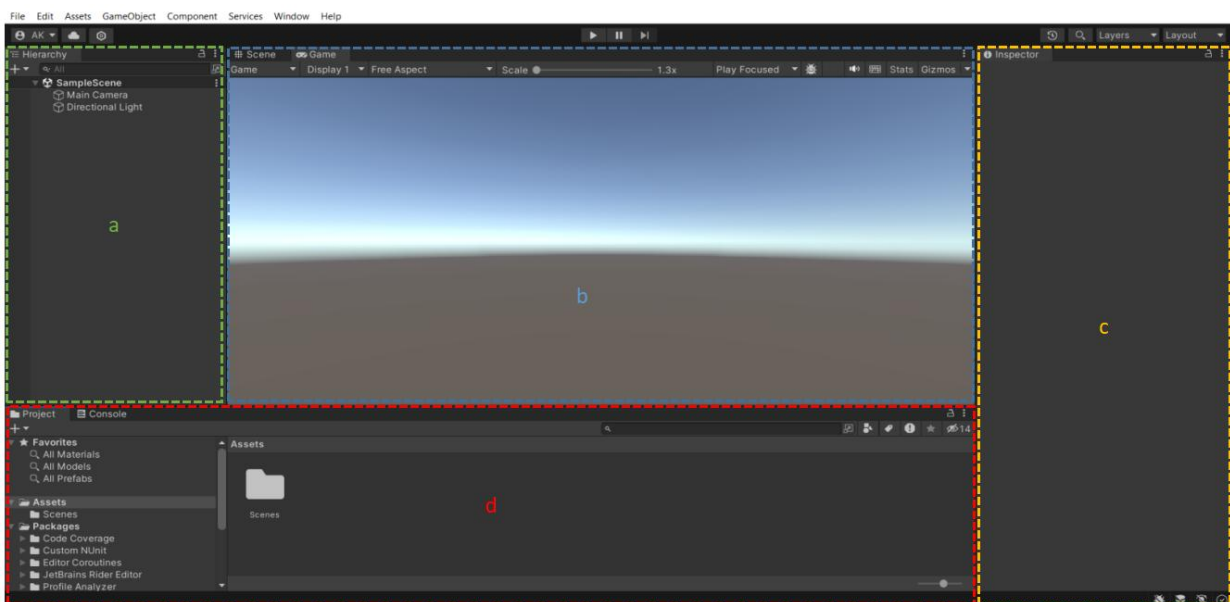


Figure 2: Unity's main interface. a) Hierarchy. b) scene. c) inspector. d) project

- A. **Hierarchy**: In this panel, GameObjects are organized, and their inter-relationships are managed. The structure resembles a tree with various branches and sub-branches, allowing some GameObjects to be parented to others, thereby creating a structural **Hierarchy**. A **parent** GameObject can include multiple **child** objects. This structuring offers a high degree of organization and allows for managing and editing multiple objects as a single unit, where editing the parent object affects all its children.
- B. **Scene**: This panel provides a dynamic, interactive view of the current scene, allowing for the manipulation of GameObjects directly within the scene's space. Through the Scene panel, the user can select, move, rotate, and scale objects while receiving direct feedback on how the applied changes influence the entire scene context. The panel offers different **camera** and lighting angles, and the user is able to freely move within the space using the arrow keys or change the viewing angle (not the scene camera) by holding the right mouse button.
- C. **Inspector**: As the name suggests, this panel allows for a close inspection of the different building blocks in the scene individually. Upon selecting an object (which can be a script, a GameObject, etc.), the components and properties of this object are displayed in this panel and can be modified. The modifications occur in real-time, and the changes can be directly observed in the Scene panel. Additionally, this panel is where components are added or removed from the different objects. Please note that any changes made during an active scene are lost once the scene is stopped.
- D. **Project**: This panel provides a management system for all the assets and files within a project. It displays the entire directory structure of the project's assets folder, including scripts, textures, models, audio files, prefabs, scenes, and other resources. For ease of use, a drag-and-drop function is available, which allows assets to be directly integrated into the Scene or **Hierarchy** panels.

Video.1 (exploring the interface and creating new objects and scripts)

GameObjects: <https://docs.unity3d.com/6000.0/Documentation/Manual/GameObjects.html>

Components:

<https://docs.unity3d.com/Manual/Components.html#:~:text=Components%20are%20the%20functional%20pieces,components%20and%20GameObjects%2C%20see%20GameObjects>

Assets: <https://docs.unity3d.com/Manual/AssetWorkflow.html#:~:text=An%20asset%20is%20any%20item,as%203D%20models%2C%20textures%2C%20sprites>

Creating a scene

A scene is a fundamental unit that organizes GameObjects and components in a specific context. It represents a specific environment, level, or menu within a game or an experiment.

Below are some of the basic and optional steps to create a VR scene.

Importing assets

Importing assets into Unity is a crucial step in the development process, enabling the use of external resources such as 3D models, textures, audio files, and animations as template building blocks. Unity supports a wide range of file formats. Once imported, assets are automatically converted into a format that Unity can efficiently use. Assets can then be organized within the **Project** window, applied to GameObjects in the **Hierarchy** or directly to the Scene view, and their properties configured in the Inspector.

Note: Please ensure that the assets are properly managed to ensure project organization and workflow optimization. This is crucial for facilitating a smoother workflow, better program performance, and more efficient debugging if needed.

Asset can be imported in two main ways:

◆ Importing new assets

Assets can be imported into Unity by either dragging and dropping files directly into the **Project** window or by clicking on the "Import New Asset" option, which can be accessed from the Assets tab or by right clicking on "Assets" in the **Project** window.

Importing Assets: <https://learn.unity.com/tutorial/importing-assets#604e8e8eedbc2a0b4c3a8925>

◆ Importing from an existing project

The desired asset needs to be exported from the existing project first by right-clicking on it and choosing "Export," then saving the file in a desired location. It is possible to export individual assets as well as the whole assets' [package](#). Then, in the new project, right-click in the **Project** window and select "Import Package". Select the previously exported assets from their saved location. Note that you can import individual assets or the entire assets' package from the **Import** window.

Reusing assets between projects: <https://docs.unity3d.com/510/Documentation/Manual/HOWTO-exportpackage.html>

Supported files: <https://docs.unity3d.com/2020.1/Documentation/Manual/3D-formats.html>

Prefabs

A prefab is a type of asset that contains GameObjects that have been edited or configured. One of the most notable features of a prefab is its reusability across multiple scenes or different projects. Prefabs are essentially clones of an original asset and remain linked to it, which enhances efficiency. Editing the original asset automatically updates all clones, allowing for broad changes without needing to edit each instance individually. Conversely, it is also possible to edit a single clone and then choose to apply those edits to other clones or the original asset if needed. Additionally, after making changes to a clone, it can be reverted to its original form, which is based on the original asset.

Prefabs are created automatically when an object is dragged from the **Hierarchy** into the **Project** window, and they have a `.prefab` extension. In the **Hierarchy**, prefabs are represented in blue text and are accompanied by a blue cube icon.

Video.2 (creating prefabs)

Prefabs: <https://docs.unity3d.com/Manual/Prefabs.html>

Adding Components

Components are the core of what gives GameObjects their functionality and behavior. They are modular pieces of code, often prewritten by Unity, that can be added to or removed from GameObjects. Also, custom components can be created by writing scripts (Video 5, timestamp: 4:47. See also Video 8, timestamp: 0:26).

Adding Unity Components

The simplest method to add prewritten component is by navigating to the **Inspector** window and clicking on the “Add Component” option. This will open a browser tab where the name of the desired component can be entered and then added by clicking on its name (Fig. 3).

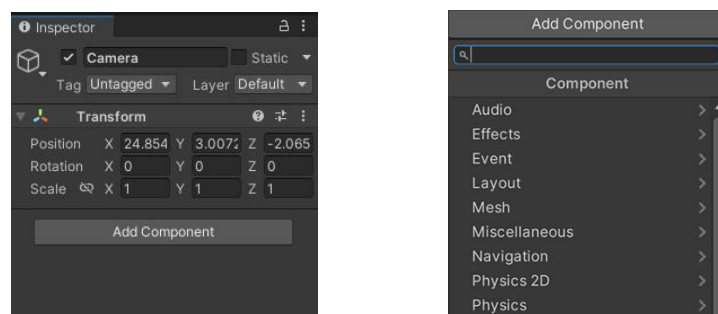


Figure 3: The Component browser can be accessed in the **Inspector** window by clicking on "Add Component" at the bottom of the Inspector, below the last added component.

Adding Custom Components

Custom components cannot be directly created in the Unity Editor. Instead, they must be written as scripts in C#. Once the script is saved, it can be attached to a GameObject as a component to define its behavior. Any public variables defined in the script will appear in the Inspector as part of the newly added component.

Component: <https://docs.unity3d.com/2022.2/Documentation/ScriptReference/Component.html>
Use components: <https://docs.unity3d.com/6000.0/Documentation/Manual/UsingComponents.html>

Importing packages

As long as the package is not custom-made, it is recommended to import packages through the [package manager](#), which is a window that allows the user to find, install, and manage Unity packages. Please refer to (Video.3 Timestamp: 0:17) for a practical example on how to import the [Probuilder](#) package.

Creating a basic room layout

For certain experiments, achieving a sense of immersion is crucial. The test subject should perceive the virtual environment as a seamless extension of their physical surroundings. To accomplish this, an enclosed space can be created within the virtual environment to simulate the physical room accurately. This can be achieved by either importing a prebuilt room (prefab) and customizing it to closely resemble the intended space or by utilizing the ProBuilder package.

ProBuilder is a versatile tool that enables the creation, editing, and texturing of 3D models directly within the scene. It allows users to design complex geometric structures, modify existing [meshes](#), and apply materials efficiently, eliminating the need for external 3D modeling software.

Video.3 (creating walls using the probuilder package)

ProBuilder: <https://docs.unity3d.com/Packages/com.unity.probuilder@4.0/manual/index.html>

Colliders and rigidbodies (component)

A [Collider](#) is a component that carries shape information of the GameObject it is attached to. It also dictates the outcome of collision interactions with other GameObjects.

Colliders carry shape information but do not possess physical properties intrinsically (e.g., mass, gravity effects, etc.). These properties and the effects of applied external forces are handled by the [rigidbody](#) component, which connects GameObjects to the physics engine.

There are many types of colliders. They are categorized either by shape complexity or the presence or absence of the **Rigidbody** component. In the shape category, primitive colliders are the most elementary collider type. They take the shape of simple geometrical forms (e.g., cube, sphere, capsule) and can be resized and positioned to fit the shape of the GameObject they are attached to (Fig. 4). For GameObjects with complex geometrical structures, it is possible to break the structure down into multiple primitive colliders, creating a compound collider, where each primitive collider is a child node of the parent GameObject. This allows the compound structure to behave as a unit. However, if a shape is exceedingly complex, a mesh collider can be used for an accurate representation. This collider draws its information directly from the asset's mesh (i.e., information that defines the surface polygons of an object).

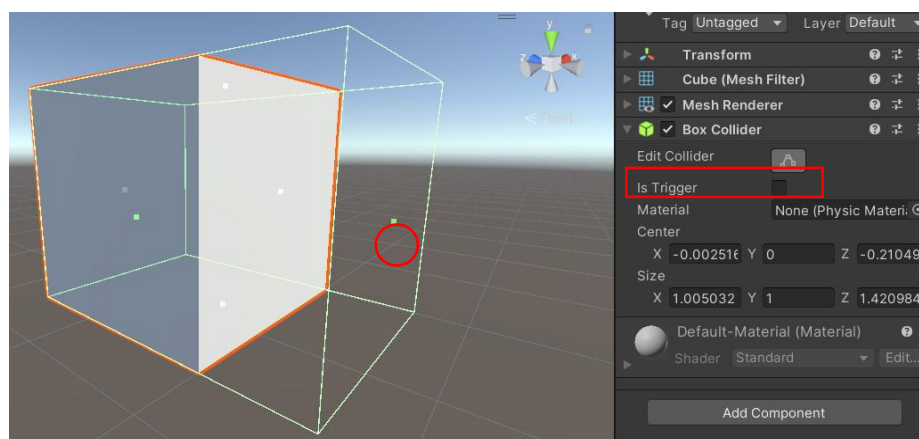


Figure 4: The **Box Collider** component can be resized manually by clicking on the "Edit Collider" option (red rectangle). The desired face of the collider can be dragged by holding the left mouse button on the corresponding knob (red circle).

If a collider has a **Rigidbody** component attached to it, it is called a dynamic collider or Rigidbody collider, and physics is applied to it. A static collider is a collider with no Rigidbody attached. This is mainly used to shape an environment, as this collider allows for collisions but remains stationary.

The **collider** component changes the behavior of the object in two different ways, depending on whether the "Is Trigger" property is active or inactive:

- **Inactive:** The collider behaves like a solid object, and objects with **Rigidbody** components will interact with the collider according to the rules of physics, to a degree that depends on whether a Rigidbody is attached to the collider.

- **Active:** The collider becomes a trigger zone, and other colliders can pass through it, which in turn activates "OnTrigger" events.

Note: When scripting, use scripts that either alter the physics of a GameObject or its [Transform](#) values, but never both, as this can lead to bugs or crashes.

Introduction to collision: <https://docs.unity3d.com/6000.0/Documentation/Manual/CollidersOverview.html>
 Colliders: <https://docs.unity3d.com/560/Documentation/Manual/CollidersOverview.html>
 Box collider (primitive): <https://docs.unity3d.com/Manual/class-BoxCollider.html>
 Mesh collider: <https://docs.unity3d.com/6000.0/Documentation/Manual/class-MeshCollider.html>
 Rigidbody: <https://docs.unity3d.com/2021.2/Documentation/Manual/class-Rigidbody.html>

Canvas and Text

The [Canvas](#) is a GameObject that functions as a container for displaying text, images, or user interface elements. Within the **Hierarchy**, the element to be displayed is assigned as a child of the Canvas GameObject. The positioning of the Canvas can either be within the scene or as an overlay above it. Its placement is determined by the selected render mode (Fig. 5), which consists of three distinct options:

- World space: The Canvas is in the scene and can be treated like any other GameObject.
- Screen Space-overlay: The canvas is placed as an overlay above the scene.
- Screen Space-Camera: The canvas is placed as an overlay above the scene and within a fixed distance from the camera.

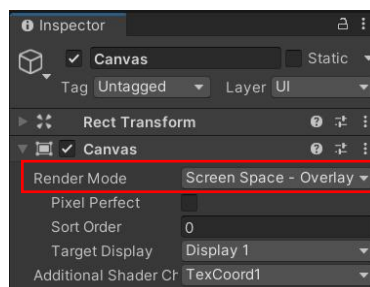


Figure 5: The Render mode dictates the positioning of the canvas

Text can then be added by creating a [TextMesh Pro](#) (TMP) GameObject as a child of the Canvas. The text can then be edited from the **Inspector** window. The size and location of the TMP can also be changed indirectly in the **Inspector** window or directly in the scene by using the [Rect Tool](#) (Fig. 6).

Note: Please refer to (Video 6, timestamp: 5:56) for a practical demonstration of how Canvas and TMP can be incorporated in the experiment's logic.

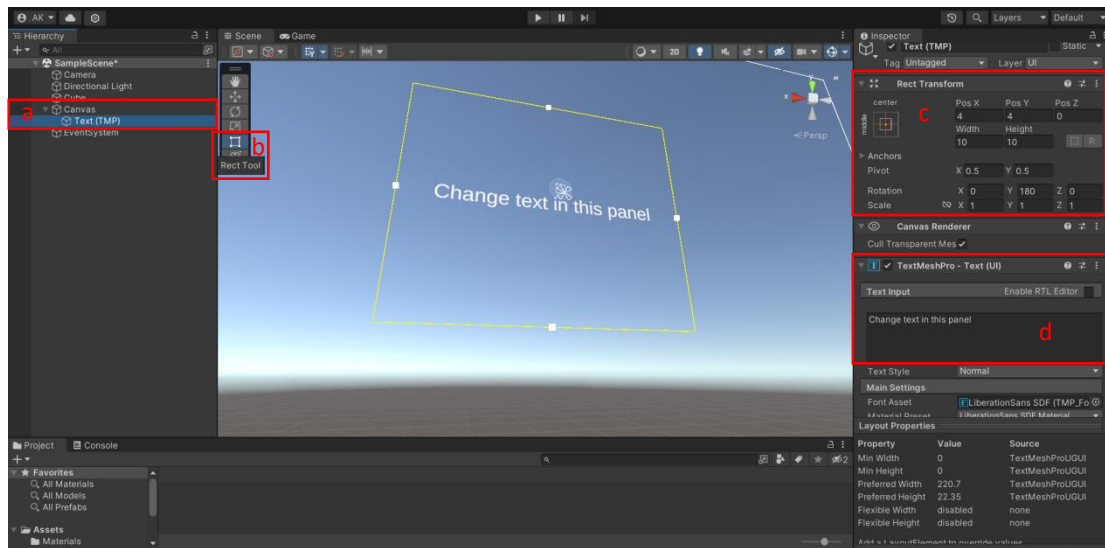


Figure 6: Formatting a TMP GameObject. a) The TMP is typically a child of the Canvas. If TMP is added directly into the **Hierarchy**, a parent Canvas GameObject will be automatically added with it. b) The Rect Tool can be selected for fluid control over the size and coordinates of the TMP in the scene directly. c) The **Rect transform** component is an alternative way for manipulating the TMP's coordinates and scale. d) The **Text(UI)** component allows for inserting and formatting the text displayed in the scene.

Canvas: <https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/UITCanvas.html>
TextMesh Pro: <https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/TextMeshPro/index.html>
<https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/TextMeshPro/TMPObjects.html>

Implementing VR interactions

XR Origin (camera setup)

XR Origin is a **Hierarchy** of GameObjects that acts as mediator between the tracked data from the real world and their representation in the XR environment. In other words, it translates real-world information into the virtual scene. Given that a Camera GameObject is part of the **Hierarchy**, the XR origin serves as a replacement for the standard camera when deployed in the scene. According to the recommendations in the user's manual, the **Hierarchy** should additionally include a GameObject that serves as an anchoring point for the other GameObjects (XR Origin), a GameObject that balances out the cameras position (Camera Offset), and virtual controllers/hand-held devices (i.e., user's input devices for interacting with the environment) (Fig. 7).



Figure 7: The recommended **Hierarchy** configuration of the XR origin is the default setting. Please note that "XR origin" refers to both the parent anchor GameObject and the **Hierarchy** in its entirety.

The "origin" in XR origin refers to the parent GameObject that sets an anchor point in the real world, which will then be translated into the virtual scene. This anchor serves as a reference point for representing the tracked data from all physical objects (e.g., headset, hand-held devices, etc.). The tracked physical objects are represented in the scene by GameObjects, which are organized as children of the XR origin parent GameObject. This will set the transform values (e.g., position) of these child GameObjects relative to the anchor point dictated by the parent GameObject.

By default, the anchor point is set to XR origin GameObject itself and can be changed through the **origin base game object** component attached to the XR origin GameObject itself.

In older Unity XR packages, XR Origin was referred to as XR Rig. This practice however has been deprecated in newer version. Please see the Unity manual for more information:

XR Origin:

1. <https://docs.unity3d.com/2020.3/Documentation/Manual/xr-origin.html>
2. <https://docs.unity3d.com/Packages/com.unity.xr.core-utils@2.5/manual/xr-origin.html>

Detailed description of the XR Components: <https://docs.unity3d.com/Packages/com.unity.xr.core-utils@2.3/manual/xr-origin-reference.html>

XR interactions toolkit

This toolkit is a package made up of a set of components that are designed to make the creation process of interactive XR applications more intuitive, thereby reducing the complexity of building XR interactions from scratch. It offers preprogrammed interactors (e.g., ray interactors) and interactable components (e.g., grabbable objects or buttons), along with an interaction system that manages how the interactions are processed between the components.

Video.4 (Setting up XR origin and the interactions toolkit).

XR Interaction Toolkit: <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.3/manual/index.html>

Grab Interactions - Video timestamp (min 1:30 - 7:16) - Credits to Valem - Source: https://www.youtube.com/watch?v=0xt6dACM_1I

Adding hand models

To enhance immersion, the hand held devices can be rendered in the scene. There are various methods to achieve this, and two of these will be addressed in detail here:

- Using Unity assets:

Video.5: (Adding Hand models).

Download the Oculus hand models: <https://drive.google.com/file/d/10b39lekUdpBHlcTslZ-BINRyH5uqPUe1/view>

- Using the **SteamVR** plugin:

Developed by **Valve**, the company behind **Steam**, this plugin offers a wide set of preprogrammed components for designing an XR environment, which helps streamline complex tasks.

Although this method is quick to implement, it can sometimes lead to compilation errors when combined with other components.

1. Download the **SteamVR** plugin by adding it to your Unity account either from the asset store or directly from the website (See [Importing packages](#)).
2. In the **project manager**, import the plugin to the project.
3. Click "Import" in the pop-up window to import all the plugin contents. It is possible to opt out of some contents, it is however advised to import all contents to avoid potential errors.
4. In the **Import** window, click "accept all" (it may be necessary to repeat this step multiple times until this window disappears).
5. In the **Inspector**, search for the **Camera Rig** prefab (in some versions, it is referred to as "Player") and import it into the scene. This construct is similar to the XR origin and includes GameObjects for both the left and right hands. Each of these hands has a model hand already attached.
6. The hand models are now active, and the corresponding GameObjects can be seen when the **Camera Rig** prefab is expanded in the **Hierarchy**.

Note: this method is not compatible with the **XR Origin**, which is already equipped with a camera rig.

SteamVR Plugin: <https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647>

Raycasts (Component)

Raycasting is a technique that involves projecting an invisible ray from a specific point, such as a controller or input device, into the virtual environment to detect or interact with objects in the scene. The ray detects any objects it intersects with, enabling users to point at, select, or manipulate objects from a distance. This concept is similar to using a laser pointer in the real world, but in this case, the ray can also function as an extended virtual arm when needed.

A raycast on its own is invisible; however, it can be made visible using the **Line Renderer** component.

Raycasting from the controller:

1. Add **XR Ray Interactor** component to your controller object (see [Adding Unity Components](#), [XR Origin](#), and Fig. 7). This component will cast a ray from the virtual controller into the scene.
2. Configure the Ray properties:
 - **Interaction layer mask:** specify which layers the ray should interact with and which objects belong to those layers. If allocated to a certain layer, the ray will interact only with objects included in that layer (objects can be set to specific layers through their own **XR Interactor** component).
 - **Enable interaction with UI GameObjects:** enables interaction with UI elements.
 - **Force grab:** enables the ray to move objects directly to the user's hand during interactions.
 - **Line type:** choose between a straight line, projectile curve, or bezier curve for the ray.
 - **Max raycast distance:** adjust the ray's reach.
 - **Ray origin transform:** set the starting position and direction of the ray (usually based on the controller's pose).
3. Add interaction events through the **XR Ray Interactor** component.
 - Example: check the box next to **Audio events** --> **on select entered** then add an audio file. The audio file will play when the raycast is used to select an item (Fig. 9).

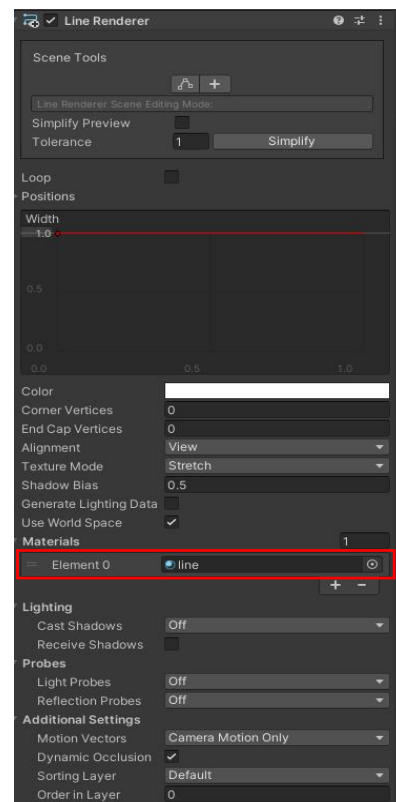


Figure 8: The **Line Renderer** and its parameters are crucial for rendering lines in the scene. The red rectangle highlights the **Materials** field, where a material needs to be assigned for the line to render correctly. Without a material, the line will not appear as intended.

Notes:

- Make sure that the interaction buttons are correctly configured for a successful interaction (Video 4, timestamp: 3:08. See also Video 5, timestamp: 7:41).
- Using scripts, audio clips can be programmed to play when certain conditions are met (Video 8, timestamp: 4:39).

4. Format the line visuals:

- Use the **Line Renderer** and **XR Interactor Line Visual** components. There are many options for configuring the visuals, including thickness, color and reticle (a targeting-assistance asset that shows up at the end of the rendered line and can be assigned).

Note: Make sure that the line renderer has a Material. The line will not materialize correctly otherwise (Fig. 8).

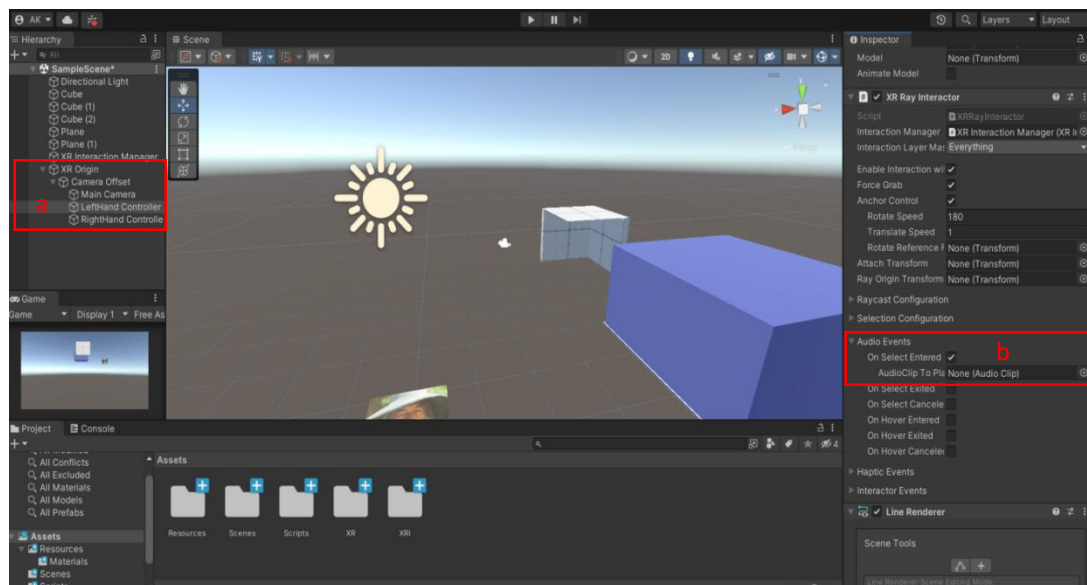


Figure 9: Adding interaction events through the **XR Ray Interactor** component. **a)** The left and right hand components can be found by expanding the **XR origin** GameObject and then the Camera Offset GameObject. **b)** In the **XR Ray Interactor** component, it is possible to upload audio files and play them at certain events, such as when an object is selected or hovered over. Haptic (vibration) feedback can also be set to trigger on similar events.

Line renderer: <https://docs.unity3d.com/6000.0/Documentation/Manual/class-LineRenderer.html>

Reticle: <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/xr-interactor-reticle-visual.html>

Ray Interactions - Credits: *Fist Full of Shrimps* - Source: <https://www.youtube.com/watch?v=SC1e16ZlvUk>

Unity experiment framework (UXF)

The Unity Experiment Framework ([UXF](#)) is an open-source toolkit designed to facilitate the creation and management of scientific experiments within the Unity environment. It is particularly useful for conducting behavioral experiments and studies involving human-computer interactions⁷.

Unity, primarily a game engine for graphics scene creation, lacks the specific tools needed to organize experiments into sessions, blocks, and trials. UXF fills this gap by providing customizable design patterns that can be reused and adjusted, thereby reducing the need for extensive coding to structure experiments⁷. The framework also allows for the manipulation of experimental conditions and sensory input at various stages during an experiment, offering high control over independent variables. The experiment's workflow is automated in a cascading manner (Fig. 10). This architecture can be dynamically modified through structural variables, such as setting the number of trials, configuring conditions, and variables.

UXF enables the collection of various types of data, including user-input data, timing data, physiological data, and environmental variables. The data is automatically collected based on its type and stored in separate folders (Table 1).

When combined with some of Unity's basic tools, UXF provides shortcuts for structuring experiments, allowing researchers to focus on the experiment's logic and accelerating the overall experiment generation process.

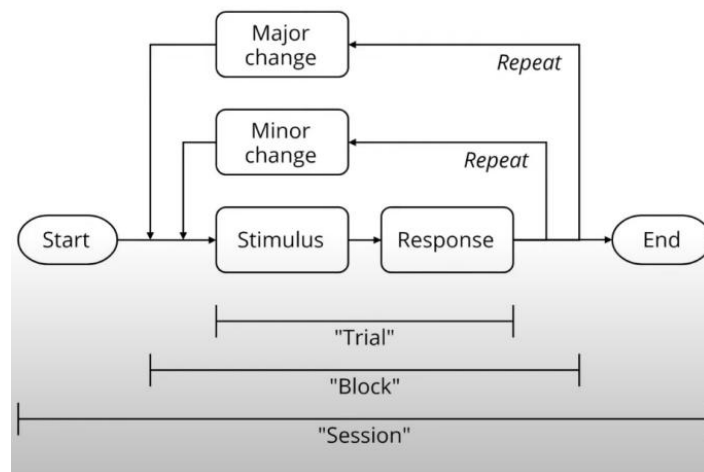


Figure 10: UXF automates trial runs, manages variable control, and organizes the rhythm of trial and block repetitions, storing the resulting data in a structured **Hierarchy**. The flowchart is adapted from (Brooks et al., 2020).

File(s)	Folder	Description
<code>trial_results.csv</code>	(none)	The main behavioural results file, with one row per trial. It also contains references (relative paths) to other trial-level data files such as tracker files, so you can read in the data and associate it with a trial.
<code>participant_details.csv</code>	/session_info	A copy of the participant's details (typically the data that are collected along with participant ID using the UI). Stored as a single row.
<code>log.csv</code>	/session_info	A copy of all <code>Debug.Log</code> calls during the session, as well as any other custom data saved under datatype <code>SessionLog</code> .
<code>settings.json</code>	/session_info	A copy of all settings applied as the session begins.
Trackers e.g. <code>head_movement_T001.csv</code>	/trackers	A copy of tracker data, stored with one file per trial. Tracker data is continuous data, the most common will be tracking the movement of an object (e.g. head/hands) with the <code>PositionRotationTracker</code> component.
Other data	/other	Any other custom data stored manually , associated with a trial or a session.

Table 1: The sorting order of automatically collected data. Adapted from the UXF Github repository.

UXF wiki: <https://github.com/immersivecognition/unity-experiment-framework/wiki>

UXF Github repository: <https://github.com/immersivecognition/unity-experiment-framework>

Quick start tutorial by Dr. Jack Brookes (creator of UXF): <https://www.youtube.com/watch?v=1GGXz5XwPkk&t=2s>

Key words:

Asset: A general term for any resource file used in creating a project, regardless of the type of file.

Camera: A GameObject that renders the scene from a specific viewpoint, determining what is visible to the player.

Canvas: A GameObject in Unity that acts as a container for UI elements, rendering them in screen space or world space.

Component: An element that defines the behavior and functionality of a GameObject. Components can be built-in (e.g., Rigidbody, Collider, Light) or custom-made using C# scripts.

Collider: A component that defines the physical boundaries of an object.

GameObject: The fundamental building block of scenes in Unity. Each GameObject is defined by its components and its hierarchical relationship to other GameObjects.

Hierarchy: A UI panel in Unity where GameObjects are organized and structured in a parent-child relationship.

Inspector: A UI panel in Unity that is used for modifying GameObject properties.

Mesh: A component that stores the shape of a 3D object using a collection of geometrical elements, such as vertices, edges, and faces, to define its structure.

Package: A collection of assets, features, and customization options bundled together for easy download and integration.

Package Manager: A window that allows users to manage packages, including importing, updating, and removing features, assets, and tools from the Unity Registry or custom sources.

Parent/Child: A hierarchical relationship between GameObjects. A parent GameObject can have one or more child GameObjects, which inherit transformations and properties from the parent while maintaining their own unique attributes.

Physics Engine: The system handling physics interactions (e.g., PhysX, Havok).

Prefab: a type of an asset that is preconfigured and customizable. A prefab can be used in multiple projects.

ProBuilder: A tool for creating and editing 3D models directly in Unity.

Raycasting: A method for detecting objects by projecting an invisible line from a specific point in the virtual space. This method is used for selection, interaction, and collision detection.

Rect Tool: A tool used to manually manipulate the transform values of UI elements and 2D objects by adjusting their rectangular boundaries in the scene directly.

Rigidbody: A component that allows physics-based interactions.

Scene: A collection of GameObjects representing a level or environment.

Script: A C# file that controls GameObject behavior.

Shader: A program that defines how an object is rendered, including its appearance, lighting, and surface properties.

TextMesh Pro: A preconfigured GameObject that renders high-quality text with advanced styling and better performance compared to the default Text component.

Transform: Defines an object's position, rotation, and scale.

UXF (Unity Experiment Framework): A toolkit for managing scientific experiments in Unity.

XR: An abbreviation for “Extended Reality”, which includes virtual, mixed, and augmented reality.

XR Origin: A complex of GameObjects that defines the position and movement of VR cameras and controllers.

Abbreviations:

AI: Artificial Intelligence

HDRP: High-Definition Render Pipeline

TMP: TextMesh Pro

UI: User Interface

UXF: Unity Experiment Framework

VR: Virtual Reality

XR: Extended Reality

References:

¹ Lewis, M., & Jacobson, J. (2002). Game engines. *Communications of the ACM*, 45(1), 27.

² Juliani, A. (2018). Unity: A general platform for intelligent agents. *arXiv preprint*, arXiv:1809.02627.

³ Lv, Z., Tek, A., Da Silva, F., Empereur-Mot, C., Chavent, M., & Baaden, M. (2013). Game on, science—How video game technology may help biologists tackle visualization challenges. *PLoS ONE*, 8(3), e57990. <https://doi.org/10.1371/journal.pone.0057990>

⁴ Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2017). Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40, e253. <https://doi.org/10.1017/S0140525X16001837>

⁵ Laird, J., & VanLent, M. (2001). Human-level AI's killer application: Interactive computer games. *AI Magazine*, 22(2), 15.

⁶ Hussain, A., Shakeel, H., Hussain, F., Uddin, N., & Ghouri, T. L. (2020). Unity game development engine: A technical survey. *University of Sindh Journal of Information and Communication Technology*, 4(2), 73–81.

⁷ Brookes, J., Warburton, M., Alghadier, M., & others. (2020). Studying human behavior with virtual reality: The Unity Experiment Framework. *Behavior Research Methods*, 52(2), 455–463.
<https://doi.org/10.3758/s13428-019-01242-0>