Meta Horizon Worlds Technical Specification

> This is an in-development (Jan '25) **community-written** document.
> For questions contact *wafflecopters*.

Created by the Horizon Community. Written by wafflecopters (Ari Grant) with contributions from PigeonNo12, Shards632, and Tellous and help from HomeMed, SeeingBlue, and UnravelWinter.

Overview

Meta Horizon Worlds (called "Horizon" for the rest of this document) is a Metaverse content platform where people can find and create 3D immersive content to play, explore, and socialize in. Horizon calls each experience a **world**. The content can be accessed on:

Supported platforms: mobile, web, Windows PCs, and VR.

Use the Horizon creation tools you can create team-vs-team shooter games, fantasy fighting games, social deception games, hang out spaces and clubs, art exhibits, simulation games, battle royale games, dungeon crawlers, obstacle courses, puzzle games, talk shows, adventure games, stories, party games, improv clubs, and whatever else you can imagine.

The tools support many features for managing and scripting players, physics, 3D mesh import, projectiles, purchases, grabbable items, wearable items, player inputs, lights, UI, NPCs, and more.

Desktop Editor: on Windows the Horizon executable can be launched in "Edit" mode (TODO - explain) which opens app in a set of tools where you can create and edit worlds.

VR Editor: in Quest VR devices the Meta Horizon Worlds app contain an edit mode that allows for creating and editing worlds inside of VR. It offers a natural and intuitive experience where you can place object directly with your hands and immerse yourself in your creations. The VR editor does not provide access to all tools that the desktop has.

TypeScript and Code Blocks. Horizon uses TypeScript as its scripting language. TypeScript scripts can only be edited in the Desktop Editor. Horizon also has a custom block-based scripting system (where you write scripts by combining blocks together) that it calls **Code Blocks**. Codeblock Scripts can only be edited in the VR Editor.

Worlds

You use the Desktop Editor to edit worlds, adding content and scripts to build out your ideas. The publishing menu enables you to configure worlds settings and publish the world when ready. Worlds are saved in "files" called world snapshots which allow rollback. A published world may be running many instances at once.

Creating a World

TODO

Metadata and Publishing

Let's *NOT* document all of what is below. These are here for reference to see which ones we want to document.

TODO

Publish World "box"
Status: unpublished
Name
Description
World rating (a flow that result in: TODO)
Comfort rating: Not Rated, Comfortable, Moderate, Intense
(World) Tags (select N from list)
Mute Assist (boolean)
Visible to public (boolean)
Members-only world (boolean)
Boolean (boolean)
Available Through Web and Mobile
Compatible with Web and Mobile

Player Settings
VOIP: Global vs Local
Maximum Player Count (4 to 32)
Suggested Minimum Player Count (1 up to Max)
Emotes (boolean)
Emotes Audio (boolean)
Can Hands Collide With Physics Objects
Can Hands Collide With Static Objects
Custom Player Movement
Generate Instant Replays
Frame Budget Boost (Early Access) -- (Default, On, Off)
Spawn Nearby (boolean)
Footsteps Volume
Footsteps Min Distance
Footsteps Max Distance
Hide Action by Default (boolean)
-- mobile only setting, can be overridden by setting icon in Properties
Disable Dynamic LOD Toggles on Avatar (makes it easier to increase player count)
Enable Max Quality Avatar

Name, description, comfort setting, player count, etc.

Editor Roles

The **owner** is the person who created the world. Once a world is created, there is no way to change the owner. Other people, called **collaborators**, can than be added to (and removed from) the world via the Collaborators menu. When adding a collaborator, you choose whether they are an editor or tester.

Role	Can travel to editor instances?	Can enter build mode, edit scene, and edit scripts?	Can publish the world?	Can edit persistence settings (create and edit leaderboards, quests, and PPVs)?	Can assign editor roles?
Owner	✅	✅	✅	✅	✅
Editor	✅	✅	❌	❌ (Exception: editing Quests *are* allowed)	❌
Tester	✅	❌	❌	❌	❌

World Snapshot

When you create a new world, Horizon creates a new "file" on their servers which contains all the information and data for the world. Horizon calls this a **world snapshot**. Every time you update the world, a new snapshot is created. You can manage all the saves snapshots via the backups feature.

> **The world snapshot**
>
> Whenever this document refers to **the world snapshot** it is referring to the specific snapshot that you have loaded the world from (which is the last one saved, unless you did a rollback).

World Backups

The editor regularly "auto saves" the world, creating a new world snapshot that is calls a **backup**. You can manually create a snapshot as a via the "Save Backup" option.

The list of all previous saved snapshots are viewable in the "Backups" menu. This menu allows you to see the list of backups, see when background were created, modify the name and description, or to **restore** a backup to be the current snapshot.

For instance, before starting a major change to the world, you could create a backup, and then if you run into issues, you could restore that "safe" backup back to when the world was "unchanged". You can also look back into backups to investigate when a certain bug appeared, or to go back and make an asset or copy a script that you have since modified or deleted.

> **Source Control**
>
> Currently there is no way to put a whole world into an external source control system, such as git, but it is possible to put the scripts into source control.

Instances

Horizon maybe have multiple *copies* of a world running at the same time. For example if the maximum player count is set to 20 and there are 100 people "in the world" then they would be spread out across *at least* 5 separate copies. These copies are called **instances**.

> **Horizon sometimes refers to Instances as "Sessions"**
>
> In all technical documentation, Horizon uses the word *instance*. Given that this is a somewhat technical term, it refers to them as **sessions** within the user-facing side of the product. For example, a person can "create a new session".

Instance Lifetime

Creation: When a player travels to a world (to play it or edit it), Horizon finds or creates an instance of the right type.

Longevity: The instance then remains running as long as players stay in it. Even when all players leave, and the instance becomes empty, it may stay running for some time, in case any players try to return or new players arrive.

Destruction: When there are no players in an instance it will be destroyed, after some timeout threshold. In rare instances a server error may also cause an instance to be destroyed (which will send all players in it back to the app-launch state).

> **Destroyed instances are permanently gone and so is their data.**
>
> When an instance is destroyed there is no way for players to get back that specific instance. Any data they had "acquired" in that instance is permanently lost. You can **use Horizon persistence to track data across instances and visits**.

Instance Types

There are two types of instances: **published instances** and **editor instances**. The editing tools, for modifying a world, are only available inside of an *editor instance*. There is no way to turn one into the other; when Horizon starts up a new instances, based on how the player is traveling, and then the type never changes, for as long as the instance is alive.

Instance Type	How do you travel to one?	Can you open the editing tools?	How many instances are allowed?
Published	Use the "Visit World" button, or travel to a friend, travel via a door.	No	No limit
Editor	Use the "Edit World" button if you are the world owner, editor, or a tester.	Yes, if you are the owner or a editor.	1

Visitation Modes: Edit, Preview, and Publish

"Visiting" a world in Horizon is done in one of three modes: edit, play, and publish. In a published instance, all players are always in "publish mode". In an editor instance, the creator and editors can switch back and forth between edit and preview modes; testers are always in preview mode.

Mode	Description	Instance Type	Required Role
Edit	Experience the world **as an editor** where you can modify the world.	Editor Instance	Editor

Mode	Description	Instance Type	Required Role
Preview	Experience the world **as a player** from within the editable instance.	Editor Instance	Editor or Tester
Publish	Experience the world **as a player** in a published instance.	Published Instance	n/a

> **Debug Console Gizmo Visibility**
>
> The Debug Console Gizmo has setting to control which visitation mode(s) it is visible in.

Available Instances

A player can only travel to an instance if that instance is **available for the player**. Availability is determined by three criteria, all of which must be met:

1. **Isn't at maximum player count**: a player can only travel to a world if there as at least one index available. If the capacity is set to 20 and there are 19 people there, then 1 more can travel to the world. It is then unavailable for all players until at one player leaves.
2. **Is Safe**: Horizon has an undisclosed, and evolving, set of rules for what it deems *safe*, regarding travel. These rules may include: which players have blocked one another (and how recently), if the traveling player has recently been voted out of that instance, if the instance has a moderated event running, and more.
3. **Instance is Open**: all published instances exist as either *open* or *closed*. An **open instance** can be joined by an player (if the above criteria are met). A **closed instance** can only be joined by players who are explicitly invited by players already in the instance.

Open and Closed Instances

New instances default to open. When a new instance is created via the "Travel" button it is **open**.

Player can create closed instances. When a player explicitly creates a new instance, via "Create New Session", they can choose whether the instance is open (allowing anyone to join) or closed (allowing only invited-players to join).

Openness can be changed with scripting. You can change whether or not the current instance is open via TypeScript with

```
this.world.matchmaking.allowPlayerJoin(isOpen)
```

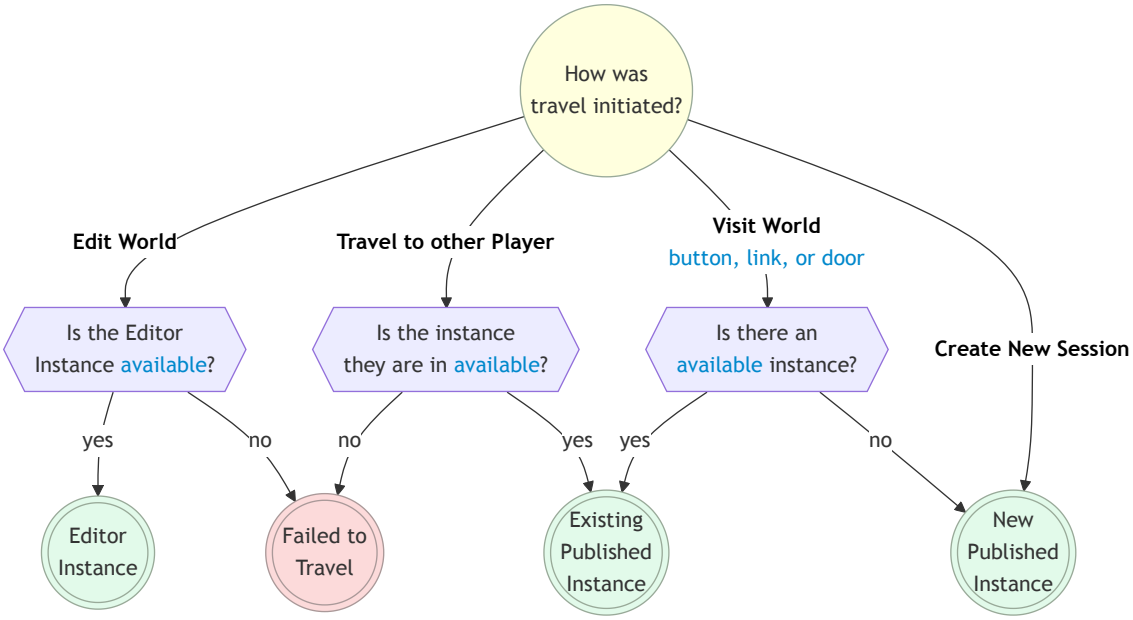
which returns a `Promise<void>` to signal when the change has taken effect.

TODO: when calling `allowPlayerJoin(false)`, can players join by invite or is the instance actually LOCKED vs Closed?

Instance Selection

When a player travels to a world, Horizon will determine which instance to send them to (if there are multiple) or create a new instance if needed (if all are full, none exist, or the player specifically created a new one).

> **The Editor Instance**
>
> There is only ever (at most) one **editor instance** of a given world. When that one instance is full, no other editors can load the world to edit.



Travel, Doors, and Links

TODO:

- Doors act like an in-experience Hyperlink
- Travel to friend
- Instruction how to get an actual link...

Resetting an Instance

TODO stop / play / pause and `world.reset()`.

Scene Graph

Every world in Horizon is made out of entities each of which has an intrinsic type such as being a mesh or a particle effect. Entities can be configured to have *behaviors* (such as being grabbable or attachable) and be have other entities as their children (or as a parent). The collection of all of these entities, their attributes, and relationships is called the **scene graph**. When you modify the scene graph in the editor, those changes are saved in the world snapshot.

Hierarchy

Any entity can be set as the child of another entity. For example, you might make a robot's forearm a Mesh Entity that is a child of the upper arm Mesh Entity. Or you might put a steering wheel inside a car. The main reasons to create parent-child relationships are:

1. To have the transform of one entity impact another (e.g. moving a car also moves the steering wheel within it).
2. To create "layers" or "folders" in the editor (e.g. putting all trees in a "collection" to make them easier to manage).

When an entity has no parent it is called a **root entity**.

Ancestors

We call the collection of an entity's parent, grandparent, great-grandparent, etc the entity's **ancestors**. If the entity has no parent, we say it has 0 ancestors. If it has just a parent and then grandparent, it would have 2.

We call the children, and their children, and their children, etc of an entity its **descendants**.

Empty Object and Groups

Empty Objects and Groups are two methods of "collection" entities together. They are similar in most regards, with only a few differences:

Collection Type	Pivots	Interactive Entity Children	Projectile Launcher	Child Count
Group	Always at the **center of all their children**. Meaning that moving one child will move the pivot point.	Children have their **interaction disabled**.	Projectile collisions happen **on the group**.	1+
Empty Object	The **center of the Empty Object** is always the pivot point.	Children **can be Interactive Entities**, if the Empty Object's `Motion` is `None`.	Projectile collisions happen **on a child**.	0+

Empty Objects and Groups behave identically in regards to collisions and triggers in all cases other than projectiles launched from the projectile gizmo.

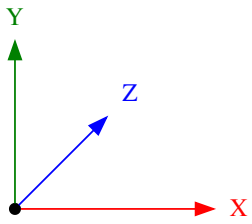
TODO - explain how collisions and triggers both do the algorithm of "start with the colliding leaf object and walk up the ancestor chain until you find the first with a matching tag and then immediately stop".

Coordinate System

Axes. Following standard convention, the editor uses *red* for the *x-axis*, *green* for the *y-axis*, and *blue* for the *z-axis* when displaying "manipulation handles" to move, rotate, or scale an entity.

Y-up. The positive-y axis is *up*.

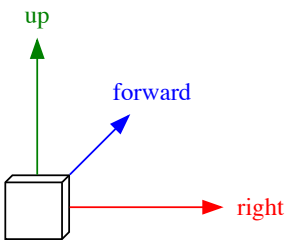
Left-handed. The coordinate system is *left-handed*, meaning that if position the camera so that the positive y-axis is pointing up and the positive x-axis is pointing right then the positive z-axis points forward.



Local coordinates. Every entity and every player and player body part has a set of local axes called: **right**, **up**, and **forward** which have an origin at the pivot point, if an entity, and at the center of the body part if it is a body part (example: player center is the hips; head center is literally the center of the head). Local coordinates are used for moving entities around in the Desktop editor (if enabled) and are used when interacting with local transforms.

Local Coordinates Example

The *forward* axis of *a player head* is always pointing away from their face (parallel to their nose), its *right* axis is always point "out" their right ear, and its *up* axis is pointing out from the top of the skull. When the entity or player body part moves, the origin of these axes move; likewise the axes rotate along with the entity (so that the *right* axis always points out from the right ear).



Meters. Distances and positions in Horizon are referenced using meters. For example, the position `(0, 1, 0)` is 1 meter (roughly 3.28 feet) up from the center of the world. Avatars in Horizon are approximately 1.8 meter tall (5 feet 11 inches).

Origin. The editor has the origin `(0,0,0)` at the center of the grid. The origin cannot be moved.

Transforms

Entities have three transform properties: position, rotation, and scale. You can use the Properties panel or the "manipulation handles" to manipulate these properties. Editing these values determines how entities are transformed when a new instance starts. **Within the Horizon editor you can only configure initial position, rotation, and scale**. If you want these values to change while the world is running, you will need to modify the values using scripting.

In the desktop editor you can switch quickly between transform tools via the keyboard.

Manipulation Tool	Keyboard Shortcut
Move	W
Rotate	E
Scale	R

Entities can be transformed globally and locally, they have pivot points, and can be transformed relative to other entities or players.

No Arbitrary Matrix Transforms

Horizon does not currently allow matrix transforms. You can achieve some skew effects by rotating an entity inside a non-uniformly scaled one. Arbitrary matrix transforms are not exposed to the developer.

Position

Positions are specified as 3-dimensional vectors, represented as the `Vec3` type in TypeScript. In the editor these are written as a "triple" such as `(0, 0, 0)`.

The `position` property on an entity determines where in 3D space the pivot point of the entity is, in relation to the origin of the world. Often the pivot is just the center of the entity, and so typically the position of an entity is where its center point is.

Setting a position

Position is a read-write property on the `Entity` class. To get the current position of an entity, do:

```
entity.position.get()
```

To move an entity to be 3 meters up from the origin and 4 meters forward, do:

```
entity.position.set(new Vec3(0, 3, 4))
```

Setting the `position` property is not influenced by the position of any ancestors.
See local transforms for setting position relative to a parent entity.

An entity position cannot have a value outside of `[-10,000, 10,000]`

When an entity moves (via `position.set` or via physics) to a location where any of its x-, y-, or z-values are outside the range `[-10,000, 10,000]`, then instead, the **entity will be automatically moved to the location it had at world start** (or at spawn-time if it was spawn). If it is a physics entity then it will also have its velocity cleared out.

Players do not auto-move / respawn when they are too far away from the origin.

Entities can be spawned farther than 10,000 away from the origin.

It is a bug that entities can be spawned outside the bounds of the world.

Rotation

Rotations are specified using a mathematical object called a `Quaternion`. Whenever you see the word "Quaternion" you can just think it means "rotation". This isn't mathematically true but is sufficient for nearly all uses.

Rotations in the editor are specified using Euler Angles which are a robust way of specifying yaw, pitch, and roll. The default **Euler Order** order in Horizon is **YXZ** meaning that entity does a *yaw*, then a *pitch*, and then a *roll* (when specifying Euler Angles). Euler angles are specified in **degrees**.

Rotations are tricky!

Rotations, Quaternions, Euler Angles, etc are all rather tricky and subtle concepts. It will take a lot of time to build an intuition for them. Be patient and don't worry if rotations seem complex (they are)!

The `rotation` property on an entity determines how much the entity is rotated around its pivot point. This rotation is specified *globally*, meaning that it is measured with respect to the world. A zero-rotation will have an entity's up-axis align with the world's y-axis, it's right-axis align with the world's x-axis, etc.

Setting a rotation

Rotation is a read-write property on the `Entity` class. To get the current rotation of an entity, do:

```
entity.rotation.get()
```

To rotate an entity so that it yaws 45 degrees and then rolls 90 degrees, do:

```
entity.rotation.set(Quaternion.fromEuler(new Vec3(0, 90, 45)))
```

Default Rotation ("Not rotated")

If you want an entity to be "not rotated", set its rotation to be `(0, 0, 0)` in the editor. In Typescript you can use any of these lines (they all do the same thing):

```
entity.rotation.set(Quaternion.fromEuler(new Vec3(0, 0, 0)))

entity.rotation.set(Quaternion.fromEuler(Vec3.zero))

entity.rotation.set(Quaternion.one)
```

Setting the `rotation` property is not influenced by the rotation of any ancestors.
See local transforms for setting rotation relative to a parent entity.

Scale

Scales are specified as 3-dimensional vectors, represented as the `Vec3` type in TypeScript. In the editor these are written as a "triple" such as `(0, 0, 0)`.

Inherent Size: All entities have their own inherent size. For instance, a SubD cube is inherently 1 meter long on each side. Mesh assets have a size based on how they were authored. The inherent size of an entity is the size it is when it is *unscaled*.

The `scale` property determines the fraction an entity should be of its inherent size. For instance, a SubD cube is inherently 1 meter long on each side. If you set its scale to be `(1, 0.5, 2)` then the cube will be 1 meter long on its right-axis, 0.5 meters long on its up-axis, and 2 meters long on its forward-axis. In this example, the object has been "shrunk" along its up-axis, and "expanded" along its forward-axis.

Setting a scale

Scale is a read-write property on the `Entity` class. To get the current scale of an entity, do:

```
entity.scale.get()
```

To scale an entity so that it is 3 times bigger on its up axis (than its inherent size), do:

```
entity.scale.set(new Vec3(1, 3, 1))
```

Since the default scale is `(1,1,1)`, you can set any part of a scale to `1` to leave the entity "un-scaled" along that axis.

Setting the `scale` property is not influenced by the rotation of any ancestors.
See local transforms for setting scale relative to a parent entity.

Mesh Primitives Have Unexpected Inherent Sizes

The built-in mesh primitives have an inherent scale of 150 meters on each side (as of Feb 2025). Thus if you wanted to use a built-in mesh cube and have it be 1 meter long on each side, you would need to give it a scale of (1/150, 1/150, 1/150). This is a longstanding bug.

Offsets - Move, Rotate, and Scale

When you want to set the position of an entity in relation to the current position we call this **offsetting** the position. There is no built-in API for doing this (as of Feb 2025) but it can be accomplished easily with the pattern of *get-modify-set*.

Offsetting position and scale

To move an entity up 2 meters from its current location you can do:

```
const offset = new Vec3(0, 2, 0)

const pos = entity.position.get()
const newPos = pos.add(offset)
entity.position.set(pos)
```

Offsetting scale works similarly.

Offsetting rotation

To rotate an entity 90 degrees around the world's y-axis, from its current rotation, you can do:

```
const offset = Quaternion.fromEuler(new Vec3(0, 90, 0))

const rot = entity.rotation.get()
const newRot = offset.mul(rot)
entity.rotation.set(newRot)
```

Note that `mul()` is used to combine rotations.

If instead you wanted to rotate an entity 90 degrees around its own up-axis you would do:

```
const offset = Quaternion.fromEuler(new Vec3(0, 90, 0))

const rot = entity.rotation.get()
const newRot = rot.mul(offset)
entity.rotation.set(newRot)
```

where the order of the Quaternion multiplication has been flipped. See Quaternions for more explanation.

Transform Property

Each entity has a transform property that can be accessed via `entity.transform`.

```
class Entity {
  readonly transform: Transform
  // ...
}
```

Position, rotation, and scale can all be accessed through a `Transform`. The following two lines behave identically.

```
entity.position.set(p)
entity.transform.position.set(p)
```

Additionally, the `Transform` object can be used to access **local** position, rotation, and scale. See the next section for more information.

Local Transforms

Entities have a `localPosition`, `localRotation`, and `localScale` that can be accessed via the transforms (e.g. `entity.transform.localPosition.get()`). These properties specify values in relation to a parent entity (or to the world if there is no parent), specified in the parent's local coordinates.

Throughout this doc, other than this section, we omit the word *global*. When you see "position" it means "global position".

Local Position Example

Let `parent` be an entity that has not been rotated nor scaled with `child` as one of its children.

If `parent`'s **global position** is `(3, 0, 0)` and `child`'s **global position** is `(8, 1, 0)` then `child`'s **local position** will be `(5, 1, 0)`. The `child`'s local position is how much it is moved from its parent.

Note: if the `parent` were rotated or scaled then you can't just "subtract the positions".

Global values "cascade down" the hierarchy.

An entity's global position/rotation/scale influences the global position/rotation/scale of its children (which then cascades to their child too!). If you have a plate on a table on a boat and the boat moves globally then so do the table and the plate; if the table moves then so does the plate (and everything on it!)

Local values exist in the transformed local coordinate system of the parent.

Rotating and/or scaling an entity causing it axes to rotate and scaled as well. We call these the *transformed axes*.

> A child with local position of `(0, 6, 0)` is moved 6 units **from the global position** of its parent **along the parent's transformed up-axis**. If there is no parent then this is just 6 meters up the world's y-axis.

Pivot Points

The transformation origin point of an entity is called its **pivot point**. It rotates around its pivot point, it scales around its pivot point, and when you move an entity its pivot point end ups at the position specified.

1. **Mesh entities** have their pivot points specified when they are authored (e.g. in Blender)
2. **Empty objects** have their pivot points at the center of the gizmo (the grey cube)
3. **Group entities** compute their pivot point to be at the center of their "bounding box" **in edit mode**. For example if you move a child in a group in edit mode then when click off the group it will recompute its pivot point to be at the center of all of its children. *This only happens in edit mode. The pivot of a group doesn't auto-change when the world is running (even if its children move around).*
4. **All other entities** (e.g. door, text gizmo, box collider gizmo, etc) have a built-in pivot point (usually at their center).

> ⚠ **In the desktop editor the manipulator handles don't always render at the pivot points!**
>
> The desktop editor lets you choose to put the "manipulator handlers" at either the `Center` or `Pivot` of entities. Check that dropdown if you aren't seeing the pivots as you expect. This dropdown has no effect on how the world *runs* and is simply there to help with *editing*.

Transform Relative To

TODO

```
/// Entity
lookAt(target: Vec3, up?: Vec3): void;
moveRelativeTo(target: Entity, relativePosition: Vec3, space?: Space): void;
moveRelativeToPlayer(player: Player, bodyPart: PlayerBodyPartType, relativePosition: Vec3, space?: Space): void;
rotateRelativeTo(target: Entity, relativeRotation: Quaternion, space?: Space): void;
rotateRelativeToPlayer(player: Player, bodyPart: PlayerBodyPartType, relativeRotation: Quaternion, space?: Space): voi
```

Billboarding

TODO

Entities

Every "thing" in the Horizon scene is an *entity* (a grabbable item, a mesh, a light, a particle effect, a sound, a group of other entities, etc).

> **Entity and Object mean the same thing (except in TypeScript)**
>
> Horizon calls these **objects** in the Desktop Editor and VR Tools but calls them **entities** in TypeScript. This document tries to consistently call them entities, except when quoting places where Horizon explicitly uses the word "object", but may accidentally call them objects on occasion.
>
> In TypeScript `Object` is a built-in for managing data, whereas `Entity` is a Horizon-specific class.

TODO

Entity Types

Every entity in Horizon has an underlying **intrinsic type** determined by how the entity was originally created (e.g. whether you instantiated a Sound Gizmo, Text Gizmo, Mesh Asset, etc).

Additionally, an entity can have (multiple) **behavior types** based on settings in the Properties panel (such as being grabbable, attachable, etc).

For example, a *hat mesh that is grabbable and attachable* has a intrinsic type of MeshEntity and two behavior types: GrabbableEntity and AttachableEntity.

Static vs Dynamic Entities

All entities in Horizon are either **static** or **dynamic**.

Static entity: A static entity can never change in any way (other than being spawned in and out). A static entity's position, rotation, color, etc never change. Horizon computes more detailed lighting on static entities. Scripts can *read* the data of a static entity (such as getting position) but can never change the values. Static entities **cannot** have behaviors An entity **is static when `Motion` is set to `None` in the Properties panel**.

Dynamic entity: A dynamic entity is one that changes. It may move and rotate, have its color changed, have forces applied, be grabbed, be attached to an avatar, etc. A dynamic entity has simpler lighting than static entities. Dynamic entities *can* have behaviors. An entity **is dynamic when `Motion` is set to `Animated` or `Interactive` in the Properties panel**

- When `Motion` is set to `Animated` you can record a "hand animation" on the entity.
- When `Motion` is set to `Interactive` you can make the entity grabbable, physics-simulated, or both.

> **Parents don't affect static vs dynamic.**
>
> A static entity can have a dynamic parent and vice versa.

Intrinsic Entity Types

The table below lists all intrinsic types, which are subclasses of `Entity` . Note that some intrinsic types don't have an associated subclass and thus are access simply as `Entity` instances. Every entity only has **one intrinsic type** which can be accessed via the entity.as() method.

The intrinsic type classes (in the table below) all subclass `Entity`. All the entity properties are available on all of them.

Intrinsic entity types are organized in the desktop editor into a few top-level categories:

- **Shapes**: built-in mesh "primitive" shapes (such as cube, sphere, torus, cylinder, etc) all of which instantiate Mesh Entities.
- **Gizmos**: entities that have in-world behavior (such as for spawning a player at a location, showing UI, rendering a particle effect, launching a projectile, and so much more). These are all listed in the table below and enumerated in full detail below.
- **Colliders**: mesh-less entities that still have a "shape" that can be collided with (such as sphere, cube, and capsule). It's type is just `Entity`.
- **Sounds**: a large library of pre-made sound effects; you can also create more using the AI sound feature. These all instantiate sounds gizmos (which have the type `AudioGizmo`).
- **Empty Object**: a special "collection" entity. It's TypeScript type is just `Entity`.
- **Group**: another special "collection" entity. It's TypeScript type is just `Entity`.
- **Sublevel**: an abstract entity containing information for spawning in portions of levels.

There is a full list of all intrinsic entity types and their documentation below.

Behavior Entity Types

A dynamic entity can have **multiple behavior types** which can be accessed via the entity.as() method.

The behavior type classes (in the table below) all subclass `Entity`. All the entity properties are available on all of them.

Behavior Type	Description	TypeScript Class	How to Enable
Animated (Recording)	An entity that has a recording on it.	`AnimatedEntity`	Set `Motion` to `Animated`. Use the `Record` button in the Properties panel.
Attachable	An entity that can be attached to a Player.	`AttachableEntity`	Set `Motion` to `Animated` or `Interactive`. Set `Avatar Attachable` to `Sticky` or `Anchor` in the Properties panel.
Grabbable	An entity that can be grabbed and held.	`GrabbableEntity`	Set `Motion` to `Interactive`. Set `Interaction` to `Grabbable` or `Both`. Interaction can also be changed with `entity.interactionMode.set(...)`.
Physics-Simulated	An entity that can respond to forces and torques.	`PhysicalEntity`	Set `Motion` to `Interactive`. Set `Interaction` to `Physics` or `Both`. Interaction can also be changed with `entity.interactionMode.set(...)`

Entity as() method

You can convert an entity instance into its intrinsic or behavior types using the entity `as()` method.

For example:

```
const particleEffect: ParticleGizmo = entity.as(ParticleGizmo)
```

Once you call `as()` on an entity, you can store that "casted" entity (in a `let`, `const`, or `class` member) and you don't need to call `as()` on it again.

Note that `as()` returns the same entity back, preserving equality. Thus after the line above `particleEffect === entity` would evaluate to `true`.

> **`as()` always succeeds! Do not cast to the wrong type!**

The `as()` method will always return an instance of the requested type. This means that you can convert a text gizmo entity into an `AudioGizmo` without error or warning. However if you then attempt to use it as an `AudioGizmo` you will get errors, warnings, and unexpected behavior. Don't cast entities, with `as()` to classes they are not. **This is a brittle part of Horizon's TypeScript API that has no workaround.**

Do not use TypeScript's built-in `as` operator on an `Entity`.

The `as()` method on `Entity` actually does work at runtime; it is not just a type-cast. That means the following two lines are **not the same**:

```
✅ const sound = entity.as(AudioGizmo)
❌ const sound = entity as AudioGizmo
```

Animated Entities

An `AnimatedEntity` is an entity whose **Motion** is set to **Animated** and has a "hand-recorded animation" (created with the "Record" button) which can be played, paused, and stopped.

Animated Entity has these properties in the Properties panel:

- **Animation [Play/Stop/Record]** - Animations can be recorded without scripting. To record an animation in the desktop editor or in VR, set **Motion** to **Animated**, press **"Record"**, adjust the entity's position, rotation, and/or scale, and then press **"Stop"**. Press **"Play"** to preview the recorded animation.
- **Play on Start** - To play/stop an animation on the first frame on world start, enable/disable **Play on Start**.
- **Loop** - Controls whether an animation loops again (forever) after it finishes playing.
  - **Never** - After an animation finishes playing, do nothing.
  - **Continuously** - After an animation finishes playing, replay the animation again from the first frame.
  - **Back and Forth** - After an animation finishes playing, replay the animation in the opposite direction, starting from the current frame. When that animation finishes, play it again in the forward direction. Repeat alternating forward and backward playback.
- **Speed** - Playback speed of the animation. Defaults to 1. A 0.5 speed would take twice as long to play back.

Use the `AnimatedEntity` class to control recorded animations.

Method	Description
`play()`	Play the animation from the current frame, or from the beginning if the animation last completed.
`pause()`	Pause the animation at the current frame. Playing again will resume, starting at this frame.
`stop()`	Reset the animation to the first frame, restoring the entity's position/rotation/scale to its initial state.

`AnimatedEntity` is not yet "active" `preStart()` and `start()`.

Calling `play()` in `preStart()` or `start()` doesn't always work. If you always want to play at start, use the **Play on Start** setting. If you want to do it conditionally, then use a small timeout to delay it.

You cannot directly transform an `AnimatedEntity` with a recorded animation.

An animated entity ignores any calls to `set()` its position, rotation, or scale.

An `AnimatedEntity` performs its recorded animation locally when it has a parent. Thus you can essentially move, rotate, or scale an `AnimatedEntity` by putting it in a group or empty object and transforming that parent.

Recorded animations can be nested.

Since an `AnimatedEntity` performs its recorded animation locally, entities with recorded animations can be children of other `AnimatedEntity` s.

This means you can hand-animate a wheel to rotate, duplicate the wheel, set the wheels as children to car, and then hand-animate the car to drive around. You can also script your car to animate on cue by calling `start()` on the car and its wheels on the same frame.

Interactive Entities

When a dynamic entity's `Motion` is set to `Interactive` in the Properties panel it can be used for grabbing, physics, or both. We call these **interactive entities**.

An interactive entity's behavior types can be changed at runtime

```
entity.interactionMode.set(EntityInteractionMode.Grabbable)
```

with any of the following options:

Value	Behavior
`EntityInteractionMode.Grabbable`	The entity is a GrabbableEntity
`EntityInteractionMode.Physics`	The entity is a PhysicalEntity
`EntityInteractionMode.Both`	The entity is both a GrabbableEntity and a PhysicalEntity
`EntityInteractionMode.Invalid`	The entity is neither grabbable nor interactive. It remains dynamic.

Be careful putting Interactive Entities inside of hierarchies. Interactivity may be disabled!

If you want to have an interactive entity be within a hierarchy (e.g. child of another entity) then all of its ancestors should be *Empty Objects* or *Mesh Entities*. All ancestors should have `Motion` set to `None` .

If `Motion` is `Animated` or `Interactive` on any of its ancestors then interactivity will be disabled.

If any of its ancestors are a Group Entity then interactivity will be disabled.

If there are any ancestors other than Mesh Entities, Empty Objects, and Group Entities then it is undefined whether or not interaction is disabled.

Entity Properties

All `Entity` instances have the class properties in the table below. Additionally, entities have methods for managing visibility, transforming relative to an entity or player, and checking if an entity exists.

`Entity` Class Member	Type	Description
Scene Graph		
id	`bigint`	A unique value representing this entity in this instance. `id`s are not reused (within an instance).
name	`ReadableHorizonProperty<string>`	The name the Entity has in Properties panel.
parent	`ReadableHorizonProperty<Entity | null>`	The entity's parent (if there is one).
children	`ReadableHorizonProperty<Entity[]>`	The entity's children.
tags	`HorizonSetProperty<string>`	The array of tags on the entity.
Transform		
position	`HorizonProperty<Vec3>`	The entity's *global* position.
rotation	`HorizonProperty<Quaternion>`	The entity's *global* rotation.
scale	`HorizonProperty<Vec3>`	The entity's *global* scale.
transform	`Transform`	The entity's transform instance (containing properties for local and global values).
Local Coordinates		
forward	`ReadableHorizonProperty<Vec3>`	The entity's local positive z-axis.
up	`ReadableHorizonProperty<Vec3>`	The entity's local positive y-axis.
right	`ReadableHorizonProperty<Vec3>`	The entity's local positive x-axis.
Rendering		
color	`HorizonProperty<Color>`	The color the entity renders as. This is *only supported with the SubD rendering system*. To change the color of a MeshEntity use tinting.
visible	`HorizonProperty<boolean>`	The top-level control for visibility. Read the rules for when an entity is visible.
Behavior		
collidable	`HorizonProperty<boolean>`	If the entity has its collider active. This impacts grabbability, physics collision, trigger detection, if a play can stand on an entity (or is blocked by it), etc.
interactionMode	`HorizonProperty<EntityInteractionMode>`	The kind of interactive entity the entity is. This only works when `Motion` is set to `Interactive`.
simulated	`HorizonProperty<boolean>`	Whether the entity is impacted by physics (if its position and rotation are updated in the physics calculations of the frame).

`Entity` Class Member	Type	Description
Ownership		
owner	`HorizonProperty <Player>`	The owner of the entity. Changing this property executes an ownership transfer.

exists() method: When an entity is depawned it's `Entity` instances will then have `exists()` return `false`. Additionally, in Horizon's code block system it is possible to create an `Entity` variable, never set it to anything, and then send it in an event. TypeScript will also see this as an `Entity` instance with `exists()` returning `false`. Non-existent entities return "default values" (e.g. position returns the origin); you should not `set()` any properties on one.

Simulated

The **simulated** property is only available in scripting (as a `boolean` read-write Horizon property). The property controls whether the entity is updated in the physics calculations of the frame.

When an entity has **`simulated` set to `false`**:

- It **cannot be grabbed** ❌ (even if grabbable). If a held entity has it's `simulated` set to `false` it *will force release*.
- It **cannot have forces applied** ❌ (even if it is physical).
- It **can be attached via scripting** ✅ (if it is attachable) though it may push the player (if `collidable` is `true`). If an attached entity has it's `simulated` set to `false` it *will NOT detach*.
- It **can be moved** ✅ via `position.set(...)` and `rotation.set(...)` (if it is dynamic).

The `simulated` property defaults to `true`.

Entity Tags

Entities in Horizon can be assigned a list of **tags** which are used of "classifying" entities. Tags are just `string`s. Tags can be assigned in the Properties panel; they can also be modified in scripting with the `Entity` property `tags: HorizonSetProperty<string>`.

When `entity.tags.get().contains(thing)` returns `true` we say that the **`entity` has the tag** `thing`.

Tags (currently) have three primary use cases:

1. **Controlling triggers**: Trigger gizmos has a Properties panel setting that lets you specify a *tag* so that the trigger will only receive trigger enter and exit events for entities that have that tag.
2. **Controlling collisions**: Entities have a Properties panel setting that lets you specify a *tag* that the entity will receive collision events from. The entity will only receive collision events if it collides with another entity which has the specified tag.
3. **Finding entities**: Horizon has a method on the World class to get all entities in the instance which match a given "query":

```
// World
getEntitiesWithTags(
  tags: string[],
  matchOperation?: EntityTagMatchOperation
): Entity[];
```

The method takes an list of tags (a string array) and optionally a match operation (the enum `EntityTagMatchOperation` has `HasAnyExact` and `HasAllExact`) which defaults to `HasAnyExact` if not specified.

- `HasAnyExact`: return all entities in the world that have at least one of the tags in the `tags` argument.
- `HasAllExact`: return all entities in the world that have all of the tags in the `tags` argument.

> ⚠ **Horizon does not auto-check for duplicate tags.**

> Be careful not to give an entity the same tag more than once. Doing so may result in certain events happening more than once or the entity appearing multiple times in lists.

Entity Visibility

Entities can be rendered ("visible") or not rendered ("invisible"). When an entity is rendered for a specific player we say that it is *visible to that player*. Visibility is controlled in the world snapshot by setting the **visible** property in the Properties panel. Visibility at runtime is controlled by the `visible` Horizon property on `Entity` and by **player visibility permissions**.

Invisibility cascades down: If an entity is invisible then so are all of its children (and all descendants).

Visible=false overrides permissions: When `visible` is set to `false`, the entity is invisible to all players, regardless of player permissions. When `visible` is set to `true`, the entity is visible to players according to the per-player permissions (which default to being visible for everyone).

Permissions persist when Visible=false: Changing the `visible` property does not change the visibility permissions. When `visible` is changed to `false` the entity becomes invisible to everyone, but the per-player permissions are intact and will begin acting again if the entity has `visible` changed to `true`.

Entity Visibility Permissions

Initially an entity's permissions allow the entity to be seen by all players (when `visible` is set to `true`). You can then modify the settings with the entity method:

```
// Entity
setVisibilityForPlayers(
  players: Array<Player>,
  mode: PlayerVisibilityMode
  ): void;
```

where `PlayerVisibilityMode` has the values `VisibleTo` and `HiddenFrom`. When you call this method, it sets the entire permissions for the entity.

VisibleTo: only the specified players can see the entity (when it is `visible`) until the permissions are modified again.

HiddenFrom: everyone other than the specified players can see the entity (when it is `visible`) until the permissions are modified again.

Resetting permissions: you can reset the permissions to be "visible to everyone" via `entity.resetVisibilityForPlayers()` which acts the same as `entity.setVisibilityForPlayers([], PlayerVisibilityMode.HiddenFrom)`.

Checking permissions: `entity.isVisibleToPlayer(player)` allows you to check if `player` can see the `entity` according to the *permissions*. This method is unaffected by the `visible` property; it is telling you if the `player` can see the `entity` *when the entity has* `visible` *set to* `true`.

> **Visibility and Collidability are separate.**
>
> Making an entity invisible (by setting `visible` to `false` or by using per-player visibility controls) does not impact collidability. Even if an entity is invisible it can still be collided with (if it has an active collider). If you want an invisible entity to not be a "blocker" then set `collidable` to `false` as well. At this time **there is no per-player collidability**.

> **Example**
>
> Let `entity` be a cube with `visible` set to `true` in the Properties panel. Let `playerA` and `playerB` be the two players in the instance.

```
// Initially, both players can see it.
entity.visible.set(false) // now no one can see it
entity.visible.set(true) // both can see it

// This makes it so only playerA can see it
entity.setVisibilityForPlayers([playerA], PlayerVisibilityMode.VisibleTo)

entity.visible.set(false) // no one can see it
entity.visible.set(true) // only player A can see it

entity.visible.set(false) // no one can see it

// This changes the rules, but still no one can see it
entity.setVisibilityForPlayers([playerB], PlayerVisibilityMode.VisibleTo)

entity.visible.set(true) // only player B can see it
```

All Intrinsic Entity Types

All intrinsic entity types are listed in the table below, each of which link to detailed documentation.

Intrinsic Type	TypeScript Class
Box Collider	`Entity`
Capsule Collider	`Entity`
Custom UI	`Entity`
Debug Console	`Entity`
Door	`Entity`
Dynamic Light	`DynamicLightGizmo`
Empty Object	`Entity`
Environment	`Entity`
Group	`Entity`
In-World Item	`IWPSellerGizmo`
Media Board	`Entity`
Mesh	`MeshEntity`
Mirror	`Entity`
Navigation Volume	`Entity`
NPC	`AIAgentGizmo`
ParticleFx	`ParticleGizmo`
Projectile Launcher	`ProjectileLauncherGizmo`
Quests	`AchievementsGizmo`
Raycast	`RaycastGizmo`
Script	`Entity`

Intrinsic Type	TypeScript Class
Snap Destination	`Entity`
Sound	`AudioGizmo`
Sound Recorder	`AudioGizmo`
Spawn Point	`SpawnPointGizmo`
Static Light	`Entity`
Sphere Collider	`Entity`
Sublevel	`SublevelEntity`
Text	`TextGizmo`
TrailFx	`TrailGizmo`
Trigger Zone	`TriggerGizmo`
World Leaderboard	`Entity`
World Promotion	`Entity`

Collider Gizmo

Description: Represents a collision field in your world. Used to stop players, objects, and/or projectiles.

Property	Type	Description
Collidable	`boolean`	Determines whether collision is applied to this collider.
Collision Layer	`Everything` , `Objects Only` , or `Players Only` . Default is `Everything`	Determines which layers will collide. With `Objects Only` , projectiles and other objects will be blocked, but players can pass through and with `Players Only` everything can pass through except players.

TypeScript: Collider Gizmos are references as `Entity` instances with no additional scripting capabilities.

Custom UI Gizmo

Description: Presents a custom UI (User Interface) to your players. Also see Custom UI

Property	Type	Description
Display mode	`Spatial` or `Screen Overlay`	Determines how your UIs will be seen. `Spatial` means the UI is 3D object somewhere in your world. `Screen Overlay` means it will appear on top of the players screen.
Raycast	`boolean`	Determines if the raycast will appear for VR players. If disabled, VR players cannot interact, web and mobile still can unless `Focus Prompt` is disabled.
Raycast distance	`number`	Controls the distance within which a player can interact with the UI panel if `Raycast` is enabled.
Mipmap	`boolean`	? TODOFinish this
Focus Prompt	`boolean`	?
Focus prompt distance	`number`	?

TypeScript: Custom UI Gizmos are referenced as the `UIGizmo` class from `horizon/ui` with no properties or methods. For more information on `horizon/ui` see Custom UI

Limitations:

Debug Console Gizmo

Description: Allows creators to monitor the console for messages in Play and Publish visitation modes.

Property	Type	Description
Visibility	`Edit Mode Only` , `Edit and Preview Mode` , or `In Published World`	Determines which visitation modes testers, editors, and the the owner can see the Debug Console Gizmo.

TypeScript: Debug Console is referenced as `Entity` instances with no additional scripting capabilities.

Door Gizmo

Description: Showcase selected public (or unlisted) worlds and allow players to easily travel to them.

Property	Type	Description
Door	Existing World	Which will the door should showcase.
Visible	`boolean`	Whether the door is visible or not.

TypeScript: doors are references as `Entity` instances with no additional scripting capabilities.

Limitations:

- Cannot be transformed by script. You can put a door in a group or empty object if you want to script its movement, make it grabbable, etc.
- Performance intensive due to VFX - use sparingly.
- Doors play a "shimmering sound" on loop that are audible near them; there is no way to disable the sound. The only partial workaround is to put the door in a group or empty object and move that parent from the "play area"; the sound will then be too far away to hear.

Dynamic Light Gizmo

Description: Casts movable and changing light during runtime. It can move, rotate, change intensity, etc. If you don't need the light to change, use a static light for better performance.

Property	Type	Description
Light Type	`Point` or `Spot`	Type of light. A point light is a point of light emitting in all directions (like a small bulb). A spot like is a "cone" of focused light (just like real spotlights).
Color	`Color`	RGB values between 0.0 - 1.0
Intensity	`number`	Light brightness (0-10).
Falloff Distance	`number`	Distance light travels (0-100).

TypeScript: dynamic light gizmos are referenced as the `DynamicLightGizmo` class with the following properties (light type and color are *not* modifiable in scripts):

```
// DynamicLightGizmo
enabled: HorizonProperty<boolean>;        // Enable/disable the light
falloffDistance: HorizonProperty<number>; // Travel distance (0–100)
intensity: HorizonProperty<number>;       // Brightness (0–10)
spread: HorizonProperty<number>;          // Spot light spread (0–100)
```

Limitations:

- Maximum of 20 Dynamic Lights per world
- Performance intensive due to per-frame light/shadow processing

Environment Gizmo

Overview

Allows creators to make changes to the properties of their world like skydome, lighting, fog, voip settings, etc...

Manual Properties

- Active
  - ON/OFF Toggle
- Skydome Type
  - Cubemap
  - Custom Gradient
- Texture
  - Daytime
  - Sunrise
  - Sunset
  - Overcast
  - Night
  - Midnight Black
  - Twilight
  - Misty Marsh
  - Winter Sky
  - Twilight Clouds
  - Day Clouds
  - Day Panorama
  - Night Panorama
  - Star Field
- Texture Rotation
  - Value between 0 - 360
- Exposure
  - Value between 0.0 - 2.0
- Custom Light Intensity
  - ON/OFF Toggle
  - Light Intensity
    - Value between 0.0 - 2.0
- Custom Fog Color
  - ON/OFF Toggle
  - Fog Color
    - RGB values between 0.0 - 1.0
- Fog Density
  - Value between 0.0000 - 0.1000
- Show Grid
  - ON/OFF Toggle

- VOIP Settings (link to player audio)
  - Environment - TODO (VOIP=Env will pass back to last env gizmo)
  - Default
  - Nearby
  - Extended
  - Whisper
  - Mute

Typescript API

- None

> **Multiple Environment Gizmos Allowed**
>
> Multiple Environment Gizmos are allowed, but only one can be active at a time. You can use asset spawning to change the environment dynamically.

> **Spawning Multiple Environment Gizmos**
>
> When spawning multiple Environment Gizmos, the original Environment Gizmo may not reactivate when all other gizmos despawn. It might be safer to respawn your original Environment Gizmo when needed.

> **Known Issues**
>
> - TODO list known issues or delete

In-World Item Gizmo

Description: Used to sell In-World Items to users in your worlds. Also see In-World Purchases

Overview

Used to sell In-World Items to users in your worlds.
TODO Needs a lot more explaining

Manual Properties

- Visible
  - ON/OFF Toggle
- In-world Item
  - Dropdown list of all available Items
- Customize Purchase Dialog Position
  - ON/OFF Toggle
  - If ON, Purchase Dialog Position is available
    - Vector(X,Y,Z)
- UI Property
  - Trigger
  - Button
  - Icon

TypeScript: In-World Item Gizmos are referenced as the `IWPSellerGizmo` class with the following methods:

```
consumeItemForPlayer(player, item) //Consumes a specific item owned by the player.
playerHasConsumedItem(player, item) //Indicates whether a player used a specific item.
playerOwnsItem(player, item) //Indicates whether the player owns a specific item.
quantityPlayerOwns(player, item) //Gets the number of the items that the player owns.
timeSincePlayerConsumedItem(player, item, timeOption) //Gets the time since a player consumed the item.
```

Codeblock Events

```
OnItemPurchaseStart: CodeBlockEvent<[player: Player, item: string]>;
OnItemPurchaseComplete: CodeBlockEvent<[player: Player, item: string, success: boolean]>;
OnItemConsumeStart: CodeBlockEvent<[player: Player, item: string]>;
OnItemConsumeComplete: CodeBlockEvent<[player: Player, item: string, success: boolean]>;
OnItemPurchaseSucceeded: CodeBlockEvent<[player: Player, item: string]>;
OnItemPurchaseFailed: CodeBlockEvent<[player: Player, item: string]>;
OnPlayerConsumeSucceeded: CodeBlockEvent<[player: Player, item: string]>;
OnPlayerConsumeFailed: CodeBlockEvent<[player: Player, item: string]>;
OnPlayerSpawnedItem: CodeBlockEvent<[player: Player, item: Entity]>;
```

All events in the table below are 🔊 server-broadcast CodeBlockEvents; you can connect to any server-owned entity to receive them.

Built-In CodeBlockEvent	Parameter(s)	Description
🔊 `OnItemPurchaseStart`	`player: Player` `item: string`	Sent when a player has opened a purchase menu. The parameters give you a reference to the `Player` and the item id(as a `string`).

🔊 `OnItemPurchaseComplete`	`player: Player`
	`item: string`
	`success: boolean`
		Sent when a player has closed a purchase menu. The parameters give you a reference to the `Player`, the item id(as a `string`), and a `boolean` that tell us if the purchase was successful.

🔊 `OnItemConsumeStart`	`player: Player` `item: string`	Sent when a player has attempted to consume a consumable item. A player has opened a purchase menu. The parameters give you a reference to the `Player` and the item id(as a `string`).

🔊 `OnItemConsumeComplete`	`player: Player`
	`item: string`
	`success: boolean`
		Sent when a player has finished attempting to consume a consumable item. Item consumptions must be recognized and approved or they will fail (see In-World Purchases). The parameters give you a reference to the `Player`, the item id(as a `string`), and a `boolean` that tell us if the consumption was successful.

🔊 `OnItemPurchaseSucceeded`	`player: Player` `item: string`	Sent when a player successfully purchases an item. The parameters give you a reference to the `Player` and the item id(as a `string`).
🔊 `OnItemPurchaseFailed`	`player: Player` `item: string`	Sent when a player fails to purchase an item. The parameters give you a reference to the `Player` and the item id(as a `string`).
🔊 `OnPlayerConsumeSucceeded`	`player: Player` `item:string`	Sent when a player successfully consumes an item. The parameters give you a reference to the `Player` and the item id(as a `string`).
🔊 `OnPlayerConsumeFailed`	`player: Player` `item:string`	Sent when a player fails to consume an item. The parameters give you a reference to the `Player` and the item id(as a `string`).
🔊 `OnPlayerSpawnedItem`	`player: Player` `item:Entity`	Sent when a player spawns a Durable item into the world from their personal Horizon Inventory. The parameters give you a reference to the `Player` and the item (as an `Entity`).

Limitations: Only owners can make test purchases while in Preview mode.

Media Board Gizmo

Description: Allows players to scroll through pictures that have been shared to the world and approved by the creator.

Property	Type	Description
LoD Radius	`number`	Determines at what distance(in meters) the media board will appear for players.
Panel UI Mode	`Light Mode` or `Dark Mode`	Set the view for the gizmo.
Pinned Page	`number`	TODO
Deterministic Ranking	`boolean`	If enabled, all players will see the same images. If disabled, images players see will be tailored to them.

TypeScript: Media Board Gizmos are referenced as the `Entity` class with no properties or methods.

Mirror Gizmo

Description: A stationary gizmo that allows players to see a reflection of themselves and the world. Can be used to edit avatars and take pictures.

Property	Type	Description
Visible	`boolean`	Sets whether the Mirror Gizmo is visible to players.
Photo Capture	`boolean`	Sets whethers players can take pictures using the Mirror Gizmo.
Name Tag Visibility	`Show` or `Hide`	Sets whether player name tags will appear in the Mirror Gizmo, including pictures.
Has Edit Avatar Button	`boolean`	Sets whether players can use the Mirror Gizmo to edit their avatars.
Has Frame	`boolean`	Sets whether the Mirror Gizmo has a border around the edge.
Aspect Ratio	`9:16` `16:9`	Determines the Mirror Gizmo's aspect ratio, making it appear in landscape or portrait mode.
Render Radius	`number`	Determines how close(in meters) players or objects must be before the mirror will render them.
Near LOD Radius	`number`	Determines how close a player must be to see the best level of detail.
Far LOD Radius	`number`	Determines how far a player must be to see the lowest level of detail.
Near Resolution	`240p` , `340p` , `480p` , `540p` , `720p` , `1080p` , `1440p` , or `2160p`	Determines the reflection resolution when viewed at the best level of detail.
Far Resolution	`240p` , `340p` , `480p` , `540p` , `720p` , `1080p` , `1440p` , or `2160p`	Determines the reflection resolution when viewed the lowest level of detail.
Near Camera FPS	`number`	Determines the framerate of the reflection when viewed the best detail.
Far Camera FPS	`number`	Determines the framerate of the reflection when viewed the lowest detail.

TypeScript: Mirror Gizmos are referenced as the `Entity` class with no properties or methods

Limitations: Mirror Gizmos are costly, recommend only one per world and becareful about how much geometry it reflects in your world to avoid performance issues.

Navigation Volume

Description: Allows the creation of navigation meshes thats NPCs can use to walk. Also see NPCs

Property	Type	Description
Volume Type	`Inclusion` or `Exclusion`	Sets whether the volume is considered an area where NPCs can walk or not walk.
Navigation Profile	dropdown	Contains a list of all the navigation profiles created in the world.

TypeScript: Navigation Volume Gizmos are referenced as the `Entity` class with no properties or methods

NPC Gizmo

Description: Represents an NPC Avatar(bot) and its spawning location. NPCs act like real Players. They get a player id and have events like Player Enter. Also see NPCs

Property	Type	Description
Character Name	`string`	Sets the NPC's name.
Spawn on Start	`boolean`	Determines whether the NPC spawns into the world when the world is started.
Appearance	`Edit Avatar` and `Refresh` buttons.	Allows you to edit the avatar's appearance and refresh that appearance in the world.

TypeScript: NPC Gizmos are referenced as the `AvatarAIAgent` class from the `horizon/avatar_ai_agent` with the following properties and methods.

```
//Properties
agentPlayer: ReadableHorizonProperty<Player | undefined>; //The player this agent is associated with.
readonly grabbableInteraction: AgentGrabbableInteraction; //The grabbable interaction capabilities of the agent.
readonly locomotion: AgentLocomotion; //The Locomotion capabilities of the agent.

//Methods
despawnAgentPlayer(): void; //Removes the player embodied by this agent from the world.
static getGizmoFromPlayer(player: Player): Entity | undefined; //Returns the AIAgentGizmo that is associated with the p
spawnAgentPlayer(): Promise<AgentSpawnResult>; //Spawns a player to be embodied by this agent from the world.

export declare enum AgentSpawnResult //The result of a player spawn request
/*
0 = Success
1 = AlreadySpawned
2 = WorldAtCapacity
3 = Error
*/
```

Limitations: Costly to performance. Considered the same cost as a real player.

ParticleFx Gizmo

Description: Play built-in particle effects (smoke burst, water spray, muzzle flare, camp fire, etc). Available from two places in the editor:

1. Gizmo ParticleFX: Created via Build Menu's Gizmos section
2. Asset ParticleFX: Created via Asset Library's VFX category

Gizmo ParticleFX Properties:

Property	Type	Description
Play on Start	`boolean`	Auto-play when the worlds starts (or when it is spawned in)
Looping	`boolean`	Repeat effect continuously
Preset	Dropdown	Select from predefined particles
Preview	Button	Test effect in the desktop editor

Asset ParticleFX Properties:

Property	Type	Description
Prefab Name	Dropdown	Select from predefined particles
Play on Start	`boolean`	Auto-play when world loads
Looping	`boolean`	Repeat effect continuously
Preview	Button	Test effect
Custom FX Properties	Various	Effect-specific settings (e.g., fire color)

TypeScript: particle effect gizmos are referenced as the `ParticleGizmo` class with the following methods:

```
/// Particle Gizmo
play(options?: ParticleFXPlayOptions): void;
stop(options?: ParticleFXStopOptions): void;
```

which support optional configuration via the types:

```
type ParticleFXPlayOptions = {
  fromStart?: boolean;     // Resource allocation priority
  players?: Array<Player>; // Target specific players
  oneShot?: boolean;       // Override looping property
};

type ParticleFXStopOptions = {
  players?: Array<Player>;// Target specific players
};
```

The `player` property defaults to all players, if not specified. `oneShot` allows you to override the looping property.

`fromStart` is a nuanced expert-level feature; it defaults to `true`. Effects have limited resources (such as a maximum number of particles). When playing an effect while it is already playing the `fromStart` property lets you specify whether the "already playing" or the "new play" gets more priority:

- `fromStart: true` prioritizes the new effect instance.
- `fromStart: false` prioritizes completing the current effect.

`oneShot` is currently being ignored.

Currently (Feb 2025) the `oneShot` property has no impact on whether an effect loops or not. It will also use the value in the Looping setting. This is a bug.

TrailFx Gizmo

Description: Emits a colored line behind moving objects with a configurable length, width, and color gradient.

Property	Type	Description
Play on Start	`boolean`	Auto-start trail effect when the world starts (or the effect is spawned in)
Length	`number`	Trail length in meters
Width	`number`	Trail width in meters
Start Color	`Color`	RGB values (0.0-1.0) at trail start
End Color	`Color`	RGB values (0.0-1.0) at trail end
Preset	`Simple Trail` or `Tapered Trail`	Trail style preset to determine if the trail gets narrower toward the tail (tapered) or stays the same width throughout (simple)

TypeScript: trail effect gizmos are referenced as the `TrailGizmo` class with the following members:

```
// TrailGizmo
length: HorizonProperty<number>;  // Trail length in meters
width: HorizonProperty<number>;   // Trail width in meters
play(): void;                     // Start trail effect
stop(): void;                     // Stop and remove trail
```

Limitations:

- Performance intensive - use sparingly
- Stopping the effect removes the entire drawn trail

Projectile Launcher Gizmo

Description: Launches configurable particles with customizable physics properties. Ideal for weapons and launchers.

Property	Type	Description
Projectile Preset	`string`	Predefined particle effect to launch
Speed	`number`	Launch speed in meters per second
Player Collision	`No Players`, `All Players Except Owner`, or `All Players`	Determines which players the projectile can collide with. Defaults to `All Players Except Owner`.
Object Collision	`No Objects`, `All Objects Except Launcher's Group`, or `All Objects`	Determines which objects the projectile can collide with. Defaults to `All Objects Except Launcher's Group`.
Static Collision	`boolean`	Enable/disable collision with static objects

Property	Type	Description
Gravity	`number`	Percent of gravity force applied to projectile (value of 1 is standard gravity). Defaults to 0.
Scale	`number`	Size multiplier for the projectile. Default to 0.1.
Trail Length Scale	`number`	Length of particle trail in meters. Defaults to 1.
Projectile Color	`Color`	RGB values (0.0-1.0) for projectile tint. Defaults to (1,1,1) white.

TypeScript: projectile launcher gizmos are referenced as the `ProjectileLauncherGizmo` class with the following members:

```
// ProjectileLauncherGizmo
projectileGravity: WritableHorizonProperty<number>;  // Gravity force applied to projectile

// Launch a projectile with optional parameters
launch(options?: LaunchProjectileOptions): void;
```

where launch can be configured with

```
type LaunchProjectileOptions = {
  speed: number;
  duration?: number;
}
```

Built-In CodeBlockEvents: the following events are sent to a `ProjectileLauncherGizmo`:

Built-In CodeBlockEvent	Parameter(s)	Description
`OnProjectileLaunched`	`launcher : Entity`	Sent when a projectile is launched from a launcher with a reference to the launcher.

`OnProjectileHitPlayer`	`playerHit: Player`
	`position: Vec3`
	`normal: Vec3`
	`headshot: boolean`
		Sent when a projectile collides with a `Player`. `playerHit` gives us a reference to the `Player` that was hit by the projectile. `position` is where the collision happened. `normal` is the direction of the surface or face that was hit. `headshot` tells us whether it collided with the players head.

`OnProjectileHitObject`	`objectHit: Entity`
	`position: Vec3`
	`normal: Vec3`
		Sent when a projectile hits a dynamic entity (Motion is `Animated`, or `Interactive`).

`OnProjectileHitWorld`	`position: Vec3` `normal: Vec3`	Sent when a projectile hits a static entity (Motion is `None`). This event is only sent if `Static Collision` is enabled in the Properties panel.

`OnProjectileExpired`	`position: Vec3`
	`rotation: Quaternion`
	`velocity: Vec3`
		Sent when a projectile despawns by duration limit only.

Limitations:

- Requires setting Projectile Preset before use
- Maximum 10 active projectiles per launcher (the 11th launch causes the oldest alive to "vanish")
- OnProjectileHitObject only triggers for Animated or Interactive entities
- OnProjectileHitWorld only triggers for entities with Motion set to None

- High projectile speeds may cause collision detection issues

Quests Gizmo

Description: Displays a list of Quest avaliable in your world for players to track their progress. Also see Quests

Property	Type	Description
Displayed Title	`string`	Displayed a title at the top of the Quest Gizmo windows.
Number of Entries Per Page	`number`	Determines how many questions will be displayed on a single page. Values can been between 1 and 6.
Panel UI Mode	`Light Mode` or `Dark Mode`	Sets the Quest Gizmos display mode.
LoD Radius	`number`	Sets the distance(in meters) that the Quest Gizmo will appear for players.
Visible	`boolean`	Sets whether the Quest Gizmo is visible to players.

Typescript: Quest Gizmos are referenced as the `Entity` class with no members.

Built-In CodeBlockEvent	Parameter(s)	Description
onAchievementComplete	`player:Player` `scriptID:string`	Sent when an achievement is completed by a player.

Raycast Gizmo

Description: Used to cast a laser capable of returning information about what it hit, like distance, hit point, hit normal, and more.

Property	Type	Description
Collide With	`Players` , `Objects Tagged` , or `Both`	Sets which layers the raycast will interact with.
Object Tag	`string`	Sets the tag required for the raycast to interact with an object.
Raycast Distance	`number`	Determines how far the raycast will travel.

Typescript: Raycast Gizmos are referenced as the `RaycastGizmo` class with the following method.

```
//Casts a ray from the Raycast gizmo using the given origin and direction and then retrieves collision information.
raycast(origin: Vec3, direction: Vec3, options?: {
    layerType?: LayerType;
    maxDistance?: number;
}): RaycastHit | null;

//The type of layer in the world.
enum LayerType {
  Player = 0, //The layer is for both players and objects.
  Objects = 1, //The layer is for objects.
  Both = 2 //The layer for players.
}

//The results of a raycast collision
export declare type RaycastHit = StaticRaycastHit | EntityRaycastHit | PlayerRaycastHit;

enum RaycastTargetType {
  Player = 0,
  Entity = 1,
  Static = 2
}

type BaseRaycastHit = {
  distance: number; // meters
  hitPoint: Vec3;
  normal: Vec3;
};

type StaticRaycastHit = BaseRaycastHit & {
  targetType: RaycastTargetType.Static;
};

type EntityRaycastHit = BaseRaycastHit & {
  targetType: RaycastTargetType.Entity;
  target: Entity;
};

type PlayerRaycastHit = BaseRaycastHit & {
  targetType: RaycastTargetType.Player;
  target: Player;
};
```

How to Raycast

TODO

Limtations: Raycasting too often in a short period of time can hurt performance.

Script Gizmo

See FBS or Script API

Snap Destination Gizmo

Description: Designed to help position and orientate players that land on it using teleport.

Property	Type	Description
Apply Orientation	`boolean`	Applies a rotation on the player that teleports onto the Snap Destination Gizmo

Typescript: Snap Destination Gizmos are referenced as the `Entity` class with no methods.

Sound Recorder Gizmo

Description: Sound Recorders allow you to record audio for playback, but that's not the only type of audio gizmo in Horizon.
We have 3 different types:

- `Sound Recorder` found in the Gizmo menu. Lets creators record up to 20 minutes of their own audio.
- `Pre-made sound` found in the Sounds menu. Collection of Horizon provided sound effects, background audio, and music.
- `Audio Graph` generated by Gen AI. Allows you playback audio generated by the Gen AI in the Desktop Editor.

Sound Recorder

Property	Type	Description
Sound	`Play` and `Record` button	`Play` will attempt to play any audio on the Sound Record Gizmo. `Record` will start recording any sounds coming through your headset mic.
Loop	`boolean`	Determines if the sound will repeat after it is finished.
Play on Start	`boolean`	Determines if the sound will start to play when the world starts.
Volume	`number`	Sets the volume of the Sound Recorder Gizmo. Values are between 0.0 and 1.0.
Pitch	`number`	Sets the pithc of the Sound Record Gizmo. Values are between -24 and 24.
Global	`boolean`	Determines whether the Sound Recorder Gizmo will play where everyone in the world can hear it.
Minimum Distance	`number`	Sets the distance from the Sound Recorder Gizmo before the volume levels starts to fade. Values between 0.0 and 1000.0
Maximum Distance	`number`	Sets the distance from the Sound Recorder Gizmo before the volume completely fades out. Values are between 0.0 and 1000.0.
Send Audio Complete	`boolean`	Determines whether the Sound Record Gizmo sends an event when the audio is finished.

Pre-made Sound

Property	Type	Description
Preview	`Play` button	Lets creators hear a preview of the sound in Edit Mode.
Play on Start	`boolean`	Determines if the sound will start to play when the world starts.
Volume	`number`	Sets the volume of the Pre-made Sound Recorder Gizmo. Values are between 0.0 and 1.0.
Pitch	`number`	Sets the pithc of the Pre-made Sound Record Gizmo. Values are between -24 and 24.
Global	`boolean`	Determines whether the Pre-made Sound Recorder Gizmo will play where everyone in the world can hear it.
Minimum Distance	`number`	Sets the distance from the Pre-made Sound Recorder Gizmo before the volume levels starts to fade. Values between 0.0 and 1000.0

Property	Type	Description
Maximum Distance	`number`	Sets the distance from the Pre-made Sound Recorder Gizmo before the volume completely fades out. Values are between 0.0 and 1000.0.
Send Audio Complete	`boolean`	Determines whether the Pre-made Sound Gizmo sends an event when the audio is finished.

Audio Graph

Property	Type	Description
Preview	`Play` button	Lets creators hear a preview of the sound in Edit Mode.
Loop	`boolean`	Determines if the sound will repeat after it is finished.
Play on Start	`boolean`	Determines if the sound will start to play when the world starts.
Volume	`number`	Sets the volume of the Pre-made Sound Recorder Gizmo. Values are between 0.0 and 1.0.
Volume Randomness	`number`	Randomally adjust the Audio Graph Gizmo volume each play. Values are between 0.0 and 1.0.
Pitch	`number`	Sets the pithc of the Pre-made Sound Record Gizmo. Values are between -24 and 24.
Pitch Randomness	`number`	Randomally adjust the Audio Graph Gizmo pitch each play. Values are between 0.0 and 4.0.
Global	`boolean`	Determines whether the Pre-made Sound Recorder Gizmo will play where everyone in the world can hear it.
Minimum Distance	`number`	Sets the distance from the Pre-made Sound Recorder Gizmo before the volume levels starts to fade. Values between 0.0 and 1000.0
Maximum Distance	`number`	Sets the distance from the Pre-made Sound Recorder Gizmo before the volume completely fades out. Values are between 0.0 and 1000.0.
Low-Pass Cutoff	`number`	Reduces the amplitude of higher frequency signals. Values are between 1 and 20000.
Send Audio Complete	`boolean`	Determines whether the Pre-made Sound Gizmo sends an event when the audio is finished.

Typescript: Sound Gizmos are referenced as the `AudioGizmo` class with the following properties and methods.

```
//Properties
pitch: WritableHorizonProperty<number>; //The audio pitch in semitones, which ranges from -24 to 24.
volume: WritableHorizonProperty<number, AudioOptions>; //The audio volume, which ranges from 0 (no sound) to 1 (full v

//Methods
play(audioOptions?: AudioOptions): void; //Plays an AudioGizmo sound.
stop(audioOptions?: AudioOptions): void; //Stops an AudioGizmo sound.
pause(audioOptions?: AudioOptions): void; //Pauses an AudioGizmo sound.

//Provides AudioGizmo playback options for a set of players.
export declare type AudioOptions = {
    fade: number; //The duration, in seconds, that it takes for the audio to fade in or fade out.
    players?: Array<Player>; //Only plays the audio for the specified players.
    audibilityMode?: AudibilityMode; //Indicates whether the audio is audible to the specified players.
};

enum AudibilityMode {
    AudibleTo = 0
    InaudibleTo = 1
}
```

Built-In CodeBlockEvent	Parameter(s)	Description
`OnAudioCompleted`		Sent when an Sound Gizmo is finished playing.

Limitations: Due to memory cost of storing audio data and CPU cost of spatial audio processing it is recommended 10 max audio graphs in scene.

Spawn Point Gizmo

Description: Used to move players instantly to predetermined locations, includes a brief black transition scene. Can also affect camera view, player gravity, and speed.

Property	Type	Description
Spawn on start	`boolean`	Determines if the Spawn Point Gizmo will be used to spawn players as they join the world.
Set Position Only	`boolean`	Determines if the Spawn Point Gizmo will rotate the player to match its rotation when it spawns them.
Player Gravity	`number`	Sets the gravity of each player to this value when this spawn is used. Values between 0.0 and 9.81.
Player Speed	`number`	Sets the speed of each player to this value when this spawn is used. Values between 0.0 and 45.
Force HWXS Camera	`None` , `Third Person` , `First Person` , `Orbit` , and `Pan`	Determines which camera view web and mobile players will have after using the spawn.

Typescript: Spawn Point Gizmos are referenced as the `SpawnPointGizmo` class with the following properties and methods.

```
//Properties
gravity: HorizonProperty<number>; //The gravity for players spawned using this gizmo.
speed: HorizonProperty<number>; //The speed for players spawned using this gizmo.

//Methods
teleportPlayer(player: Player): void; //Teleports a player to the spawn point.

//Example
this.entity.as(SpawnPointGizmo).gravity.set(9.81)
this.entity.as(SpawnPointGizmo).speed.set(4.5)
this.entity.as(SpawnPointGizmo).teleportPlayer(player)
```

Notes:

- If no spawn points have `Spawn on start` enabled then a spawn point will be picked at random.
- The blue button above the spawn point can be used to set a default spawn for yourself in Edit mode.

Static Light Gizmo

Description: Emits static light that cannot be moved during run-time. Improved performance over Dynamic Light Gizmo.

Property	Type	Description
Shape	`Cuboid` , `Ellipsoid` , `Disk` , or `Rectangle`	Determines the shape of the static light, affecting how the light is casted onto the surrounding geometry.
Color	`Color`	Sets the color of the light coming from the Static Light Gizmo.
Intensity	`number`	Sets the intensity of the light emitted from the Static Light Gizmo. Values between 0.0 and 100.00

Typescript: Static Light Gizmos are referenced as the `Entity` class with no members.

Text Gizmo

Description: The text gizmo is a 2D surface on which text can be rendered. It supports a wide variety of markup commands that allows changing color, size, font, bold, italics, underline, vertical and horizontal offsets, line height, alignment, and more.

Property	Type	Description
Text	`string`	Sets the text displaying on the Text Gizmo.
Auto Fit	`boolean`	Automatically determines the size of the font. If disabled, you can set the size manually.
Fized Font Size	`number`	Sets the font size of the text when `Auto Fit` is disabled.
Visible	`boolean`	Determines if the Text Gizmo is visible to players.

Typescript: Text Gizmos are referenced as the `TextGizmo` class with the following property.

```
//Properties
text: HorizonProperty<string>; //The content to display in the text label
```

Using a Text Gizmo

The initial text of a text gizmo can be set in the Properties panel. Changing the text after that can be done via the `text` read-write property on the `TextGizmo` class, such as:

```
this.entity.as(TextGizmo).text.set('Hello World')
```

> **Auto Fit Property**
>
> The text gizmo has the property **auto fit**, which is only settable in the Properties panel. When it is set to `true`, the font size will change to fit the text. This is useful for making signs, for example; but, it can look weird to have all signs using slightly different text sizes. You'll have more control of the text, and have more consistency in the world, if you **turn auto fit *off***.

> **Text gizmos contribute to draw calls.**

Limitations

The total length of the text, including all markup, cannot be longer than 1000 characters. If the text is longer than 1000 characters, the text will be truncated.

The text gizmo only supports the English characters (essentially whatever can be typed on an English keyboard without any modifier keys). This means that the text gizmo is not capable of displaying any of the following: á ê ï o ū ç ñ ¿ 月 😂, for example.

Text Gizmo Markup

Horizon exposes Unity's TextMeshPro markup. The rest of this guide is a summary of Unity's TextMeshPro documentation.

Text Gizmo Tags

Text markup is able to modify the contents (e.g. making all letters uppercase), styling (such as size or color), and layout (such as alignment, rotation, and spacing) of the text. Markup is specified using tags, which are a word surrounded in angle brackets (e.g. `<b>`). Once a tag is specified, all text that comes after it will have that attribute applied, until that tag "closes" by specifying the tag with a slash before the name (e.g. `</b>`).

> **Example**
>
> `this is <b>bold</b> text`
>
> will render as
>
> > this is **bold** text

Tags that are never *closed* stay active. The bold attribute starts being applied once `<b>` is encountered and stops when `</b>` is encountered. The closing tag is optional, and if it is omitted, the attribute will continue to be applied until the end of the text.

> **Example**
>
> `this is <b>bold text`
>
> will render as
>
> > this is **bold text**

Text Gizmo Tag Parameters

Some tags accept a parameter, which is specified after the tag name and an equals sign.

Example

```
this is <size=75%>small</size>
```

will render as

> This is small

Supported Text Gizmo Tags

Tag	Description	Example
Text Decoration		
`b`	Make text bold.	`This is <b>bold</b> text.` ⬇ This is **bold** text.
`u`	Make text underlined.	`This is <u>underlined</u> text.` ⬇ This is underlined text.
`i`	Make text italicized.	`This is <u>italicized</u> text.` ⬇ This is *italicized* text.
`s`	Make text have a strikethrough.	`This is <u>strikethrough</u> text.` ⬇ This is ~~strikethrough~~ text.
`color`	Set the text color. *Parameter:* A hex RGB value with either 1 or 2 digits per component (e.g. #ff0000 or #f00) or RGBA (e.g. #ff0000ff or #f00f). You can also specify a color by name. The following are supported: black, blue, green, orange, purple, red, white, and yellow.	`This is <color=#f00>red<color=#0000ff> and blue</color> text.` ⬇ This is red and blue text.
`mark`	Make text highlighted (*on top of* the text). *Parameter:* A hex RGB color with 2 digits per component (e.g. #ff0000) or RGBA (e.g. #ff0000ff).	`This is <mark=#ffff007f>highlighted</mark> text.` ⬇ This is text.
`alpha`	Set the text alpha. *Parameter:* A hex two-digit value (e.g. #ff).	`This is <alpha=#4f>transparent</alpha> text.` ⬇ This is transparent text.
`size`	Change the font size of text. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`<size=75%>small</size>, <size=18>medium</size>, and <size=2em>large</size>` ⬇ small, medium, and large

Tag	Description	Example
`font`	Change the font of text. *Parameter:* One of the following anton sdf bangers sdf electronic highway sign sdf liberationsans sdf **oswald bold sdf** roboto-bold sdf default	`This is <font=bangers sdf>vibrant</size> text.` ⬇ This is vibrant text.
`material`	Change the material of text. *Parameter:* The name of the material used to render the current font. The following combinations are supported: *anton sdf* • anton sdf - drop shadow • anton sdf - outline *bangers sdf* • bangers sdf - drop shadow • bangers sdf - outline • bangers sdf glow • bangers sdf logo *liberationsans sdf* • liberationsans sdf - metalic green • liberationsans sdf - overlay *roboto-bold sdf* • roboto-bold sdf - drop shadow • roboto-bold sdf - surface	`<font=bangers sdf>bangers sdf <material=bangers sdf glow>bangers sdf glow <material=bangers sdf logo>bangers sdf logo` ⬇ 
`gradient`	Render a gradient over text. If the text is not white, the gradient will be "blended" with the text color. *Parameter:* One of the following	`This is <gradient=Yellow to Orange – Vertical>stylish</gradient> text.` ⬇ This is stylish text.
`uppercase`	Make text uppercase.	`This is <uppercase>biG</uppercase> text.` ⬇ This is BIG text.
`lowercase`	Make text lowercase.	`This is <lowercase>SmAlL</lowercase> text.` ⬇ This is small text.
`smallcaps`	Make text uppercase but small.	`This is <smallcaps>biggish</smallcaps> text.` ⬇ This is BIGGISH text.
`sup`	Make text superscript.	`Math like x<sup>2</sup> is fun!` ⬇ Math like x^2 is fun!

Tag	Description	Example
`sub`	Make text subscript.	`Chemistry like H<sub>2</sub>O if cool!` ⬇ Chemistry like H_2O is cool!
`rotate`	Rotate the letters within text. *Parameter:* An angle in degrees (e.g. 45).	`This is <rotate=-20>rotated</rotate> text.` ⬇ This is *rotated* text.
Vertical Layout		
`br`	Insert a line break.	`This is a line break.` ⬇ This is a line break.
`line-height`	Set the line height for the current line and those that follow.	`<line-height=200%>line A <line-height=100%>line B <line-height=50%>line C line D` ⬇ line A line B line C line D
`voffset`	Shift the "cursor" vertically up or down (impacting the text that comes next).	`do<voffset=2em>mi<voffset=1em>re` ⬇ mi re do
Horizontal Layout		
`align`	Set the alignment of the current line and those that follow. *Parameter:* One of the following: left, center, right, justified, or flush.	`<width=150><align=right>hello <align=left>world` ⬇ hello world
`width`	Set the width of the current line and those that follow. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`<width=150> <align=right>hello <align=left>world` ⬇ hello world

Tag	Description	Example
`indent`	Set the indent level for this line and all lines after it. It applies to lines created with or due to wrapping. If you only want to indent lines made with , use line-indent. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`<indent=25%>This is a kind of boring sentence.  <indent=0%>Followed by a less interesting one.` ⬇ This is a kind of boring sentence. Followed by a less interesting one.
`line-indent`	This is the same as "indent" but it only applies to manual line breaks made with and not line breaks causes from wrapping (e.g. when using the width attribute). *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`<line-indent=25%>This is a kind of boring sentence.  Followed by a less interesting one.` ⬇ This is a kind of boring sentence. Followed by a less interesting one.
`pos`	Set the position of text cursor for the rest of the line. When the tag closes with </pos> the cursor returns back to where it was. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`And a step<pos=3em>to the right!` ⬇ And a step to the right!
`space`	Insert whitespace. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`Let me...<space=3em>think` ⬇ Let me think
`margin`	Set the margin for the current line and those that follow. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`This is a kind of boring sentence.  <margin=4em>Followed by a less interesting one.` ⬇ This is a kind of boring sentence. Followed by a less interesting one.

Tag	Description	Example
`margin-left`	Set the left margin for the current line and those that follow. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`This is a kind of boring sentence. <margin=4em>Followed by a less interesting one.` ⬇ This is a kind of boring sentence. Followed by a less interesting one.
`margin-right`	Set the right margin for the current line and those that follow. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`This is a kind of boring sentence. <margin=4em>Followed by a less interesting one.` ⬇ This is a kind of boring sentence. Followed by a less interesting one.
`cspace`	Modify the spacing between letters. A positive value spreads them out and a negative value brings them closer together. *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`This is <cspace=1em>crazy` ⬇ This is c r a z y
`mspace`	Modify the width of each letter, turning the font into a monospace font (meaning every character takes up the same horizontal space) *Parameter:* A measurement value in pixels (e.g. 10), font units (e.g. 2em), or percent (e.g. 50%). Percents are relative to the initial size of the font.	`This is <mspace=1em>whimsy</mspace> text.` ⬇ This is wh i ms y The letters are equal-width. See 'w' vs 'i'. This is wh i ms y
Glyphs / Sprites		
`sprite`	Render a builtin sprite. Notice that this tag takes a second attribute called "index" that specifies white sprite to render in the set. *Parameter:* Only "dropcap numbers" is currently supported.	`<sprite="dropcap numbersz" index=3>` ⬇ 
Parsing		

Tag	Description	Example
`noparse`	Disable parsing on text.	`Is <noparse>this <b>bold</b> </noparse> or is <b>this</b>?` ⬇ Is this bold or is **this**?

Trigger Gizmo

TODO - Enable And disable trigger and note about costly to performance.

Description: Detects when a player or object enters or exits an area.

Property	Type	Description
Enabled	`boolean`	Determines whether the Trigger Gizmo will detect any events.
Trigger On	`Players` or `Objects Tagged`	Sets whether the triggers response to players or objects with a specific tag.
Object Tag	`string`	If `Trigger On` is set to `Objects Tagged` then this is the required tag for an object to trigger an event.
Selectable in Screen Mode	`boolean`	Determines whether web and mobile users will see an interaction option when near the Trigger Gizmo.

Typescript: Trigger Gizmos are referenced as the `TriggerGizmo` class with the following property.

```
//Properties
enabled: WritableHorizonProperty<boolean>; //Whether the Trigger is enabled.

//Example of connecting to a trigger entered event.
this.connectCodeBlockEvent(this.entity, CodeBlockEvents.OnPlayerEnterTrigger, (enteredBY) => {
    console.log('Player entered the world.', enteredBY.name.get());
})
```

Built-In CodeBlockEvent	Parameter(s)	Description
OnPlayerEnterTrigger	`enteredBy: Player`	Sent each time a player has entered the trigger area.
OnPlayerExitTrigger	`exitedBy: Player`	Sent each time a player has exited the trigger area.
OnEntityEnterTrigger	`enteredBy: Entity`	Sent each time an object has entered the trigger area.
OnEntityExitTrigger	`player: Player`	Sent each time an object has exited the trigger area.
"occupied"†	`by: Player | Entity`	Sent when the first player (or entity) enters a trigger area. This is not a built-in codeblock like the others, you must create this as custom codeblock event.
"empty"†	`by: Player | Entity`	Sent when the last player (or entity) exits the trigger area. This is not a built-in codeblock like the others, you must create this as custom codeblock event.

†: The *occupied* and *empty* events in the table above are not currently exposed through `CodeBlockEvents`. To use them you you must allocate the events yourself, e.g.

```
import { CodeBlockEvent, Entity, Player, PropTypes } from "horizon/core"

const ExtraTriggerCodeBlockEvents = {
  PlayerOccupied : new CodeBlockEvent<[Player]>(
    'occupied',
    [PropTypes.Player]
  ),
  PlayerEmpty : new CodeBlockEvent<[Player]>(
    'empty',
    [PropTypes.Player]
  ),

  EntityOccupied : new CodeBlockEvent<[Entity]>(
    'occupied',
    [PropTypes.Entity]
  ),
  EntityEmpty : new CodeBlockEvent<[Entity]>(
    'empty',
    [PropTypes.Entity]
  )
}
```

Limitations: Using too many Trigger Gizmos can affect performance.

World Leaderboard Gizmo

Description: Used to display player scores in your world.

Property	Type	Description
Leaderboard	dropdown	Contains a list of all the avaliable leaderboards in your world.
Displayed Title	`string`	Sets the title of the Leaderboard Gizmo window.
Number of Entries Per Page	`number`	Sets how many scores you can see per page. Value is between 1 and 10.
UI Anchor Style	`Static` or `Billboard`	`Static` sets the Leaderboard Gizmo stay in one place when viewed by the player. `Billboard` causes the Leaderboard Gizmo to rotate to always face the player who is viewing it.
Panel UI Mode	`Light Mode` or `Dark Mode`	Determines how the Leaderboard Gizmo is displayed to players.
Entry Display Mode	`Raw Value` or `Time in Secs`	Determines how the data will be displayed. Leaderboard Gizmos only accept numberical data and it can be displayed as a number or time.

Typescript: Leaderboard Gizmos are referenced as the `Entity` class with no methods. Although there is one related method from the `World` class:

```
setScoreForPlayer(leaderboardName: string, player: Player, score: number, override: boolean): void; //Sets the leaderbo
```

TODO

- Kind of data allowed
- Player opt-out
- Creation
- Using the Gizmo

- APIs
- Resetting
    - Daily / Weekly / Monthly

Assets

TODO need some kind of "collection asset" when you select items and make an asset (separate from an Asset Template)

Mesh Asset

type SetTextureOptions = {
players?: Array;
};

type SetMaterialOptions = {
materialSlot?: number | string;
}

type SetMeshOptions = {
updateMaterial?: boolean;
};

class MeshEntity extends Entity {
style: EntityStyle;

```
 setTexture(texture: TextureAsset, options?: SetTextureOptions): Promise<void>;

 setMesh(mesh: Asset, options: SetMeshOptions): Promise<void>;

 setMaterial(materialAsset: MaterialAsset, options?: SetMaterialOptions): Promise<void>;
```

}

Mesh Style

interface EntityStyle {
/**
* @example
*
`* // Augment base color as such: * * outColor.rgb = lerp(inColor.rgb, Luminance(inColor.rgb) * tintColor,`
/
/*
* Color in the RGB range of 0 - 1; defaults to 1, 1, 1 (no tint color).
/
tintColor: HorizonProperty;
/*
* Tint strength in the range of 0 - 1; where 0 is no tint and 1 is fully tinted; defaults to 0.
/
tintStrength: HorizonProperty;
/*
* Brightness in the range of 0 - 100; where 0 is black, 1 is no adjustment, and 100 is very bright; defaults to 1.
*/
brightness: HorizonProperty;
}

Data Asset (Text and JSON)

```
DefaultFetchAsDataOptions:false
type FetchAsDataOptions = {
    skipCache: boolean;
};
fetchAsData(options?: Partial<FetchAsDataOptions>): Promise<AssetContentData>;
```

Texture Asset

Material Asset

Asset Template

E.g. only root-level properties and scripts are maintained in an update.
You CAN nest.

Custom Model Import

Overview

Assets, imports, templates, updates.

SubD vs Custom Models

Uploads

- Explain collection of FBXs and PNGs.
- Each FBX will be a new asset.
- Texture rules
- Suffix rules
- Pivots
- Limits
- Colliders

Errors

List and explanation of all possible errors

Tinting

Textures

- Formats: `png` s; Horizon will ingest any valid png and convert it as necessary to its own internal representation
- Any size is allowed but power-of-2 is better for perf
- Does Horizon de-dupe textures for download?
- Horizon does not currently support mipmaps
- Materials can be emissive insofar as they are "unlit" but they don't contribute to the light probes
- Horizon used packed textures for different material attributes; see Materials
- Can we verify that Horizon uses ASTC 2.0 (Adaptive Scalable Texture Compression)

Materials

> **No post-processing**
>
> Current Horizon has no post-process rendering options which makes things like bloom, motion blur, sepia, etc impossible.

Performance

Draw Calls

Challenge question (for the doc): are draw calls really ever the primary issue? Is this information truly used and needed by 1p and 2p? A lot of Horizon's behavior is "like other 3D engines". What specific things (about Horizon) do we actually need to document, assuming that someone is technically savvy (enough) already?

- Do not rely on Horizon to do any draw call batching. Meaning each instantiated asset is at least 1 draw call.

- Hypothesis / guess: UI Gizmos are rendered into textures on the *CPU* and then rendered as single quads with a texture on the GPU (don't know about batching...). What about name tags?
- Theory: 1 draw call per avatar, 1 draw call per UI Gizmo, 1+ draw calls per instantiated asset, 1 draw call per FX/trail gizmo that is running, 1 draw call per emotes (per player that is emoting), 1+ draw call per NPC
    - In build mode: 1 draw call per gizmo

Element	Draw Call	Notes
Player	3+ each	Avatar, name tag, emotes
Entities	1+ each	Per instantiated asset
UI Gizmo	1 each	back-face / occlusion?
Particle / Trail Gizmo	1 each	occlusion-culled?
Text Gizmo	1 each	THESE MAY BE BATCHED!
Door Gizmo	1 each	occlusion-culled?
Leaderboard / Quests / Media Board / Purchase UI	1 each	occlusion-culled?
Mirror Gizmo	2x total draw call count?	... REALLY?!...
Pop-ups	1 per visible	occluded?
Projectile Launcher	1 per visible	

There are draw calls outside a creator's control

Things like the sky, personal UI, the wrist UI, teleport visuals, onscreen controls, and many other elements may add to the "base number" of draw-calls.

Group entities with the same materials together into an asset when possible

If you have 50 bricks with the same material all in 1 asset Horizon will batch that to be 1 draw call. If those are instead a single brick duplicated 50 times then that will be at least 50 draw calls.

If you have an asset with 25 bricks of material A and 25 of material B then this will be 2 draw calls. If instead they were all duplicated then there would be 50 draw calls.

Multi-material assets increase draw call count

If an asset has multiple materials or material textures then the draw call count will increase by the number of them.

Vertices, Polygons, and Entities

...

Memory

...

Horizon Lighting

GI overview and tips.

General Tips

Triangulate. Normals direction.
Workflows / advice for greyboxing.

Scripting

Scripts are how you create dynamism in worlds. You use them to create interactivity and movement. You use scripts to make something simple like a door that opens when you approach it as well as the most complex things, such as an entire complex team-vs-team shooter game (which would use many separate scripts).

TypeScript: Scripts are written in TypeScript. They can be edited in the Desktop Editor as well as the scripts web tool (click here and then select a world and then select "Scripts").

Code Blocks: Horizon also has a drag-and-drop scripting system called "Code Blocks" that are only editable in VR (and outside the scope of this document).

Components and Files: In scripts you define Component classes that you can attach to `Entities` in the Desktop editor. You can specify properties ("props") in the `Components` that will show in the Properties panel in the Desktop editor, allowing you to set and change the properties in the editor, per-entity. Scripts can contain other code too, which is executed when files are loaded. Components have a detailed lifecycle that execution through the frame.

Core types: Component instances communicate with one another and the world by sending and receiving events. There are many types in Horizon, but you'll most often use the core game types: Entity, Player, Asset, Component, and World; the core data types: Vec3 (for position and scale), Color, and Quaternion (for rotations); and the event types: LocalEvent, and NetworkEvent.

Creating and Editing Scripts

You can create scripts by using the create button in the scripts dropdown or simply creating a new file in the scripts folder. Click the [≡] button in the editor and then "Launch TypeScript Editor". You can then create and edit files.

> ⚠ **Editing a script while the editor is playing a world reloads that file.**
>
> This is often useful, but it can cause surprises with only part of the world reloading. You may need to restart the world for certain efforts. Read about file execution for more information.

Syncing Scripts

When you create, edit, or delete scripts in a world's scripts folder, Horizon automatically tracks and syncs those edits to the world snapshot.

> 🔥 **If syncing doesn't appear to be working, delete the *.editor* file, leave world, and come back.**

Scripts in Source Control

When you open a world in the editor, Horizon checks to see if the scripts match what is saved in the world snapshot. If they don't match, it will ask if you want to update the world with the scripts you have in the folder. Thus, the scripts folder acts like an "auto-sync directory". This means that you can put a git repo in the scripts folder, share the same repo with different "forks" of a world, use submodules to share scripts, and more!

Horizon Properties

Most data in the Horizon scene graph is accessed via *Horizon Properties*:

```
const pos = entity.position.get()
otherEntity.position.set(pos)
```

Properties can be

1. **read-only**: only a `get()` method. Uses `ReadableHorizonProperty`.
2. **write-only**: only a `set()` method. Uses `WritableHorizonProperty`.
3. **read-write**: `get()` and `set()` methods. Uses `HorizonProperty`.

There is an additional class `HorizonSetProperty` which is used for managing properties that act like Sets such as entity tags. You can `get()` and `set(values)` the values; additionally there are methods for `length()`, `contains()`, `clear()`, `add(value)`, and `remove(value)`.

> **`Set` versus `Array`**
>
> The `HorizonSetProperty` is meant to act like a `Set`. However the `get()` and `set()` methods both use arrays, for convenience and simplicity.

> **Horizon Property Example**
>
> Here is some of the `PhysicalEntity` class:
>
> ```
> class PhysicalEntity extends Entity {
>   gravityEnabled: WritableHorizonProperty<boolean>
>   locked: HorizonProperty<boolean>
>   velocity: ReadableHorizonProperty<Vec3>
>   // ...
> }
> ```
>
> From the above definition we can discern that we can do the following with a physical entity:
>
> - `set` if is gravity enabled, as a `boolean`, but can't get the current setting.
> - `set` if it is locked, as a `boolean`, and `get` if it is locked.
> - `get` the current velocity, as a `Vec3`, but not set the velocity (instead, you must use forces).

> **Writable properties may accept a second argument.**
>
> Consider this snippet of the AudioGizmo class
>
> ```
> class AudioGizmo extends Entity {
>   pitch: WritableHorizonProperty<number>;
>   volume: WritableHorizonProperty<number, AudioOptions>;
>   // ...
> }
> ```
>
> We can `set` the current pitch and volume (but not get them):
>
> ```
> audioGizmo.pitch.set(12)
> audioGizmo.volume.set(0.5)
> ```

The `set()` on `volume` takes an additional second parameter of type `AudioOptions`, allowing further configuration (duration of the fade, in this case).

```
audioGizmo.volume.set(0.5, { fade: 1 })
```

Horizon Property Subtleties

Horizon Property's `set()` is not immediate.

When you `set` a Horizon property the value is not immediately saved back to the scene graph. This allows different scripts to all "see the same state of the world" regardless of what order they run in. This means that if you `get` a value right after setting it that you will still get the old value.

In the following code `pos1` and `pos2` will have the same value.

```
const pos1 = entity.position.get()
entity.position.set(Vec3.zero)
const pos2 = entity.position.get()
```

Read about the frame sequence to learn more about when properties update.

Track your scene graph value updates when needed.

If you need to know values after `set`ting but before they are committed to the scene graph, you should track the values manually (such as in a class variable).

Never modify the result of `get()`. Create a new instance, or clone(), first.

Never mutate the result of a `get()`. Don't do direct field mutation (e.g. `v.x += 4` on a vector) or use mutating methods such as `v.addInPlace(w)` or `v.copy(w)`. These **will "corrupt" the property value**.

Horizon properties cache their values until the scene graph is updated (see frame sequence for when). This means that `get()` keeps returning the same value until the update occurs. The following code is then dangerous:

```
// BAD #1!
const p = entity.position.get()
p.x += 10

// BAD #1!
const p = entity.position.get()
p.addInPlace(new Vec3(0, 10, 0))
```

If any code now reads that object's position it will get the wrong value, until the next time the property is updated in the frame sequence.

You should always `clone` a Horizon property's value before modifying it (or add a method that creates a new value).

```
// OK #1
const p = entity.position.get().clone()
p.x += 10

// OK #2
const p = entity.position.get().add(new Vec3(0, 10, 0))
```

TypeScript's class property setters and getters do not work with Horizon properties.

Horizon does not use standard TypeScript properties. You must do

```
entity.position.set(newPosition)
```

and will get an error if you do:

```
entity.position = newPosition
```

Types

There are many TypeScript types in Horizon; however, there are a few that form the backbone of most scripts:

Type	Description
Component	Add interactivity and logic to a world (by creating a subclasses and attaching it to an entity).
World	Information and methods related to the current instance.
Entity	A node in the scene graph with intrinsic attributes and behavior. There are many subtypes available via entity.as().
Player	A player in the world (instance), including the "server player" and NPC players.
Asset	Data that lives outside the scene graph (such as text blobs, textures, and "prefabs").
Vec3	A "3D quantity" which can be used to represent position, velocity, acceleration, force, torque, scale, and more.
Quaternion	An abstract mathematical object primarily used for representing *rotations*.
Color	An RGB Color with each component between 0 and 1.

Construction / new: `Component`s, `Entity`s, `Player`s, and the `World` are all created by the system. You should never instantiate these directly with `new`. You can (and will) instantiate `Vec3`, `Quaternion`, and `Color`. You can allocate `Asset`s directly with their asset ids.

Equality comparison: `Entity` and `Player` can be compared directly with `===` and `!==`; these have been implemented to compare their underlying `id`s. All other types will use built-in TypeScript equality checks. `Vec3`, `Quaternion`, and `Color` implement Comparable

Comparable Interface

`Vec3`, `Quaternion`, and `Color` implement `Comparable<T>` which provides the methods `equal(other: T): boolean` and `equalApprox(other: T, epsilon?: number): boolean`.

Epsilon: represents the maximum distance the two can be apart and still be considered equal (it defaults to a small number around `1e-6` which is 0.000001).

All three classes also implement static versions of these methods. For clarity here is a subset of the `Vec3` class:

```
class Vec3 implements Comparable<Vec3> {
  equals(vec: Vec3): boolean;
  equalsApprox(vec: Vec3, epsilon?: number): boolean;

  static equals(vecA: Vec3, vecB: Vec3): boolean;
  static equalsApprox(vecA: Vec3, vecB: Vec3, epsilon?: number): boolean;

  // ...
}
```

Example: Comparing Vec3 instances

```
const a = new Vec3(1, 2, 3)
const b = new Vec3(1, 2, 3)

console.log(a === b)            // false ❌
console.log(a.equal(b))         // true ✅

const c = new Vec3(1, 2, 3.000000001)

console.log(a.equal(c))         // false ❌
console.log(a.equalApprox(c))   // true ✅
```

If you compare `Vec3`, `Quaternion`, and `Color` with `===` or `!==` then you are doing *referential* equality which means "is this the exact same instance?". In the above example `a === b` is `false` because `a` and `b` are different class instances (even though they represent the same vector); however `a === a` is `true` because both sides are the same class instance (the same *reference*).

Copying vs Mutating Methods

`Vec3`, `Quaternion`, and `Color` all implement a number of patterns around mutability. This section only refers to those 3 classes.

1. **inPlace methods**: methods that end with the suffix `...InPlace` will mutate the `this` they are called on. Thus `vec.mulInPlace(2)` is the same as `vec.x *= 2; vec.y *= 2; vec.z *= 2`. In place methods return `this` for convenience (chaining operations).
2. **new by default**: in contrast to the above point, if a method *does not end with `...InPlace` then it creates a *new* instance. Thus `vec.mul(2)` is the same as `new Vec3(vec.x * 2, vec.y * 2, vec.z * 2)`. The one exception is `copy()`.
3. **copy(input)**: this method should really be called `copyInPlace`. It takes the `input` argument and makes `this` be `equal()` to it. It returns `this` again for convenience (chaining operations). Thus `vec.copy(other)` is the same as `vec.x = other.x; vec.y = other.y; vec.z = other.z`.
4. **clone()**: creates a new equivalent instance (which would return true with `equal()` but not `===`). Thus `vec.clone()` is the same as `new Vec3(vec.x, vec.y, vec.z)`.
5. **out argument**: some methods (e.g. `Vec3.add`) take an optional `out` argument. If the `out` argument is provided then the result is "created" inside of that argument (similar to `out.copy(result)`) and also returned. If the *out* argument is not provided then a new instance is created and returns. Thus `Vec3.add(a, b)` is the same as `a.add(b)`, whereas `Vec3.add(a, b, result)` is equivalent to `result.copy(a).addInPlace(b)`.

Don't mutate the result of a Horizon Property `get()`.

Mutating the result of a property `get()` -- either by mutating a field, such as `v.x += 4` on a vector or via an *in place* method such as `v.addInPlace(w)` or `v.copy(w)` -- will cause the property to report the wrong value to future `get()`s.

Vec3

The `Vec3` class represents a 3-dimensional quantity which is usually a 3D position. It can also be used to represent an offset, velocity, force, torque, scale, Euler Angles, and more.

Vector Creation

`Vec3s` can be created in several ways:

```
// Direct construction
const vec = new Vec3(1, 2, 3)

// Static convenience vectors
const origin = Vec3.zero      // ( 0,  0,  0)
const unit = Vec3.one         // ( 1,  1,  1)
const right = Vec3.right      // ( 1,  0,  0)
const up = Vec3.up            // ( 0,  1,  0)
const forward = Vec3.forward  // ( 0,  0,  1)
const down = Vec3.left        // (-1,  0,  0)
const down = Vec3.down        // ( 0, -1,  0)
const forward = Vec3.backward // ( 0,  0, -1)
```

Vector Properties

- `x: number` - The magnitude along the X axis
- `y: number` - The magnitude along the Y axis
- `z: number` - The magnitude along the Z axis

Vector Operations

`Vec3` has the `equal(other: Vec3)` and `equalApprox(other: Vec3, epsilon?: number)` methods from Comparable, and their `static` counterparts. It also has `copy(other)` and `clone()` methods.

`Vec3` s support many common mathematical operations which support both copying and mutating versions:

```
const v1 = new Vec3(1, 0, 0)
const v2 = new Vec3(0, 1, 0)

// Addition and subtraction
const sum = v1.add(v2)       // (1,  1, 0)
const diff = v1.sub(v2)      // (1, -1, 0)

// Scalar multiplication and division
const doubled = v1.mul(2)    // (2,   0, 0)
const halved = v1.div(2)     // (0.5, 0, 0)

// Length operations
const length = v1.magnitude()           // 1
const lengthSqr = v1.magnitudeSquared() // 1

// Normalization (makes length 1)
const normalized = v1.normalize() // (1, 0, 0)

// Vector products
const dot = v1.dot(v2)       // 0
const cross = v1.cross(v2)   // (0, 0, 1)

// Get distance between vectors
const dist = v1.distance(v2)           // 1.414...
const distSqr = v1.distanceSquared(v2) // 2
```

The following table assumes that `v`, `w`, `r`, and `out` are `Vec3`s; `s` and `n` are `number`s.

Operation	Code	Math	Description

Addition	`r = v.add(w)`
	`v.addInPlace(w)`
	`Vec3.add(v, w, out)`
		$\vec{v} + \vec{w}$	Get the sum of two vectors.

Subtraction	`r = v.sub(w)`
	`v.subInPlace(w)`
	`Vec3.sub(v, w, out)`
		$\vec{v} - \vec{w}$	Get the difference of two vectors.

Scalar Multiplication	`r = v.mul(w)`
	`v.mulInPlace(w)`
	`Vec3.mul(v, s, out)`
		$\vec{v}s$	Scale a vector.

Scalar Division	`r = v.div(w)`
	`v.divInPlace(w)`
	`Vec3.div(v, s, out)`
		$\vec{v}\frac{1}{s}$	Inverse-scale a vector.

Component-wise Multiplication	`r = v.componentMul(w)` `v.componentMulInPlace(w)`	$\vec{v} \otimes \vec{w}$	Multiply the two vectors component-by-component.
Component-wise Division	`r = v.componentDiv(w)` `v.componentDivInPlace(w)`	$\vec{v} \oslash \vec{w}$	Divide the two vectors component-by-component.
Magnitude (Length)	`n = v.magnitude()`	$\lvert\vec{v}\rvert$	Compute the Euclidean magnitude (length or norm) of the vector.
Magnitude (Length) Squared	`n = v.magnitudeSquared()`	$\lvert\vec{v}\rvert^2$	Compute the Euclidean magnitude (length or norm) squared of the vector.

Operation	Code	Math	Description

Normalize	`r = v.normalize()`
	`v.normalizeInPlace()`
	`Vec3.normalize(v, out)`
		$\frac{\vec{v}}{\lvert\vec{v}\rvert}$	Divide a vector by its length to product a vector with a length of 1 (in the same direction). Exception: the zero-vector normalizes to the zero-vector.

Distance	`n = v.distance(w)`	$\lvert\vec{v} - \vec{w}\rvert$	The (Euclidean) distance between `v` and `w`.
Distance Squared	`n = v.distanceSquared(w)`	$\lvert\vec{v} - \vec{w}\rvert^2$	The squared (Euclidean) distance between `v` and `w`.
Dot Product	`n = v.dot(w)` `n = Vec3.dot(v, w)`	$\vec{v} \cdot \vec{w}$	The dot product of `v` and `w`.

Cross Product	`r = v.cross(w)`
	`v.crossInPlace(w)`
	`Vec3.cross(v, w, out)`
		$\vec{v} \times \vec{w}$	The cross product of `v` and `w`.

Reflection	`r = v.reflect(w)` `v.reflectInPlace(w)`	$\vec{v} - \frac{2(\vec{v} \cdot \vec{w})}{\lvert\vec{w}\rvert^2}\vec{w}$	Reflect `v` across `w`.
Lerp	`Vec3.lerp(v, w, s, out)`	$\vec{v} + (\vec{w} - \vec{v})s$	Compute the linear interpolation `v` and `w` the amount `s`.

Dot Product

The **dot product** is a fundamental vector operation that multiples the lengths of the two vectors together and then multiples in a "sameness value"; that value is 1 if they are parallel, 0 if they are perpendicular, and -1 if the are facing opposite directions (anti-parallel). If the two vectors both have length 1 then the dot product gives you a number between -1 and 1. The length of the dot product in general is

$$|\vec{v} \cdot \vec{w}| = |\vec{v}||\vec{w}| \cos\theta$$

where θ is the angle between the two vectors.

Common uses for dot product include:

- Determining if vectors are **perpendicular** (dot product is 0)
- Testing if vectors are pointing in **similar directions** (dot product > 0)
- Testing if vectors are **pointing apart** (dot product < 0)
- Finding the **angle between** vectors. The angle (in radians) between `v` and `w` is

  $$\arccos\left(\frac{\vec{v} \cdot \vec{w}}{|\vec{v}|\,|\vec{w}|}\right)$$

  which in code is

```
Math.acos(v.dot(w).div(v.magnitude() * w.magnitude()))
```

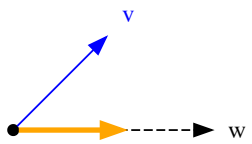
- Calculating the **projection** of one vector onto another. To find the vector you get when "flattening" `v` onto `w` you do

  $$\frac{\vec{v} \cdot \vec{w}}{|\vec{w}|^2}\vec{w}$$

  which in code is

```
w.mul(v.dot(w) / w.magnitudeSquared())
```

The diagram below shows the "projection of `v` only `w`" as the orange arrow; it's like the "shadow" of `v` on `w`.



Cross Product

The cross product produces a new vector that is perpendicular to both input vectors. It's length is

$$|\vec{v} \times \vec{w}| = |\vec{v}||\vec{w}| \sin\theta$$

where θ is the angle between the two vectors.

This is especially useful for:

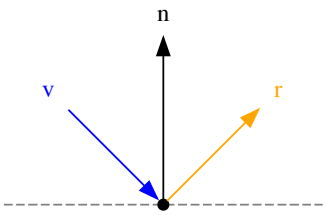
- Finding perpendicular directions
- Determining surface normals
- Creating coordinate systems (e.g. finding the `right` vector from `up` and `forward`).

> **Cross product order matters.**
>
> The cross product is not commutative: `up.cross(forward)` produces `right`, whereas `forward.cross(up)` produces `left`.

```
const right = new Vec3(1, 0, 0)
const up = new Vec3(0, 1, 0)

// Cross product returns a vector perpendicular to both inputs
const forward = right.cross(up)    // (0,  0,  1)
const backward = up.cross(right)   // (0,  0, -1)

// Building a coordinate system
const normal = surfaceNormal.normalize()
const tangent = normal.cross(Vec3.up).normalize()
const bitangent = normal.cross(tangent)

// Static method with optional output vector
const result = new Vec3(0, 0, 0)
Vec3.cross(right, up, result)  // Stores result in existing vector
```

Vector Reflect

Vector reflection, via `v.reflect(n)`, calculates how a vector bounces off a surface. Given a vector `n`, it acts like that vector is pointing directly out from a surface. Then it takes the `v` and gives the direction after `v` "bounces" off the surface.



Vector reflection is most often used for:

- Collision calculations
- Physics simulations
- Ray calculations

The code `v.reflect(n)` is equivalent to

```
v.sub(n.mul(2 * v.dot(n) / n.magnitudeSquared()))
```

which in math is:

$$\vec{v} - 2\frac{\vec{v} \cdot \vec{n}}{||\vec{n}||^2}\vec{n}$$

Vector Linear Interpolation (Lerp)

Linear interpolation is the act of smoothly blending between two values (linearly) and is usually abbreviated as "Lerp" (Linear interpolation).

Blend value: Lerp takes a start a value, an end value, and a "blend" number. If the blend value is 0 then you get back the start value. If the blend value is 1 then you get the end value. A blend value of 0.5 gives you the value halfway between the start and the end. A blend value of 0.25 gives the point one-fourth of the way from the start to the end. And so on.

```
const start = new Vec3(0, 0, 0)
const end = new Vec3(10, 0, 0)

const mid = Vec3.lerp(start, end, 0.5)        //  (5, 0, 0)
const nearStart = Vec3.lerp(start, end, 0.3)  //  (3, 0, 0)
const atEnd = Vec3.lerp(start, end, 1.0)      // (10, 0, 0)
```

Color

The `Color` class represents an RGB color where each component (red, green, blue) is stored as a floating-point number between 0 and 1. It supports color space conversions (from HSV), hex colors, supports operations that can be used for many effects, such as blending and filtering.

No alpha: The `Color` class does not have an `alpha` component (transparency). It is possible to use alpha Custom UI and with meshes but those alpha values are *not* part of the `Color` class.

Creation

Colors can be created in several ways:

```
// Direct construction with RGB values (0-1)
const red = new Color(1, 0, 0)
const purple = new Color(0.5, 0, 0.5)

// Static convenience colors
const r = Color.red    // (1, 0, 0)
const g = Color.green  // (0, 1, 0)
const b = Color.blue   // (0, 0, 1)
const w = Color.white  // (1, 1, 1)
const k = Color.black  // (0, 0, 0)

// From hex string
const fromHex = Color.fromHex("#ff0000")  // (1, 0, 0)

// From HSV (hue, saturation, value)
const fromHSV = Color.fromHSV(new Vec3(0, 1, 1))  // red
```

The `fromHex` method only supports 6-digit colors which must be preceded by the `#` mark.

Color Properties

- `r: number` - The red component (0 to 1)
- `g: number` - The green component (0 to 1)
- `b: number` - The blue component (0 to 1)

Color Operations

`Color` has the `equals(other: Color)` and `equalsApprox(other: Color, epsilon?: number)` methods from Comparable, and their `static` counterparts. It also has `copy(other)` and `clone()` methods.

Colors support several common operations which have both copying and mutating versions:

```
const c1 = new Color(1, 0, 0)
const c2 = new Color(0, 1, 0)

// Addition and subtraction
const sum = c1.add(c2)       // (1, 1, 0)
const diff = c1.sub(c2)      // (1, -1, 0)

// Scalar multiplication and division
const darker = c1.mul(0.5)   // (0.5, 0, 0)
const brighter = c1.div(0.5) // (1, 0, 0) - clamped to 1

// Component-wise multiplication
const mixed = c1.componentMul(c2)  // (0, 0, 0)

// Color space conversions
const hsv = c1.toHSV()       // (0, 1, 1)
const hex = c1.toHex()       // "#ff0000"
const vec = c1.toVec3()      // Vec3(1, 0, 0)
```

The following table assumes that `c`, `d`, and `out` are `Color`s; `s` is a number; `v` is a `Vec3`.

Operation	Code	Description

Addition	`r = c.add(d)`
	`c.addInPlace(d)`
	`Color.add(c, d, out)`
		Adds two colors component-wise.

Operation	Code	Description

Subtraction	`r = c.sub(d)`
	`c.subInPlace(d)`
	`Color.sub(c, d, out)`
		Subtracts colors component-wise.

Scalar Multiplication	`r = c.mul(s)`
	`c.mulInPlace(s)`
	`Color.mul(c, s, out)`
		Multiplies each component by a scalar.

Scalar Division	`r = c.div(s)`
	`c.divInPlace(s)`
	`Color.div(c, s, out)`
		Divides each component by a scalar.

Component-wise Multiplication	`r = c.componentMul(d)` `c.componentMulInPlace(d)`	Multiplies two colors component-by-component. Useful for color filtering, e.g. `color.componentMul(new Color(0, 0.5, 1))` "deletes" the red component, cuts the green component in half, and leaves the blue component alone.
To Vec3	`v = c.toVec3()`	Convenience method to move the `r`, `g`, and `b` values into a `Vec3`'s `x`, `y`, and `z`, respectively.

Color Space Conversions (HSV)

The `Color` class supports conversions between RGB and HSV (Hue, Saturation, Value):

```
// RGB to HSV
const red = new Color(1, 0, 0)
const hsv = red.toHSV() // Vec3(0, 1, 1)
                        // x: hue (0-1)
                        // y: saturation (0-1)
                        // z: value (0-1)

// HSV to RGB
const purple = Color.fromHSV(new Vec3(0.83, 1, 1))
```

Hex Colors

Hex color codes are a common way to specify colors using six hexadecimal digits representing RGB values. The `Color` class supports direct conversion to and from hex format:

```
// From hex to Color
const red = Color.fromHex("#ff0000")
const purple = Color.fromHex("#800080")
const navy = Color.fromHex("#000080")

// From Color to hex
const color = new Color(1, 0, 0)
const hex = color.toHex()  // "#ff0000"
```

Note that hex colors must include the `#` prefix and use two digits for each of red, green, and blue (with `00` being the lowest value and `ff` being the highest for each component).

Color Blending

Colors can be blended using the various mathematical operations:

```
// Mix two colors equally
const color1 = new Color(1, 0, 0)  // red
const color2 = new Color(0, 0, 1)  // blue
const purple = color1.mul(0.5).add(color2.mul(0.5))

// Create a darker version
const darker = color1.mul(0.5)

// Color filtering using component multiplication
const filter = new Color(1, 0.8, 0.8)  // slight red tint
const filtered = color1.componentMul(filter)
```

Quaternion

The `Quaternion` class represents an abstract mathematical object which is often used to represent rotations in 3D space, as they avoid issues like gimbal lock that can occur with Euler angles.

Quaternion Creation

`Quaternion` s can be created in several ways:

```
// Direct construction
const quat = new Quaternion(0, 0, 0, 1)

// Static convenience quaternions
const zero = Quaternion.zero     // (0, 0, 0, 0)
const identity = Quaternion.one  // (0, 0, 0, 1)
const xRot = Quaternion.i        // (1, 0, 0, 0)
const yRot = Quaternion.j        // (0, 1, 0, 0)
const zRot = Quaternion.k        // (0, 0, 1, 0)

// From Euler angles (in degrees)
const fromEuler = Quaternion.fromEuler(new Vec3(90, 0, 0))

// From axis-angle (angle in radians)
const fromAxisAngle = Quaternion.fromAxisAngle(Vec3.up, Math.PI/2)

// From forward and up vectors
const lookRot = Quaternion.lookRotation(Vec3.forward)
```

> ⚠ **`Quaternion.one` means "no rotation".**
>
> Do not use `Quaternion.zero` when you mean "a rotation of 0 degrees". It turns out that `Quaternion.zero` isn't even a rotation and should (nearly) never be used unless you truly know what you are doing math-wise.

> 🔥 **Don't `new` a `Quaternion`, unless you know the math!**
>
> `Quaternion` s are complex beasts. You can't just pass any 4 values into the constructor (you might not even get a valid rotation). Instead always create a rotation from euler angles, axis and angle, or a look rotation.

Quaternion Properties

- `x: number` - The x component of the quaternion
- `y: number` - The y component of the quaternion

- `z: number` - The z component of the quaternion
- `w: number` - The w component of the quaternion (scalar part)

Quaternion Operations

`Quaternion` has the `equals(other: Quaternion)` and `equalsApprox(other: Quaternion, epsilon?: number)` methods from Comparable, and their `static` counterparts. It also has `copy(other)` and `clone()` methods. Note that quaternion equality is special: two quaternions are considered equal if their components are equal OR if the negation of their components are equal, since both represent the same rotation. Only every compare with the methods above.

`Quaternion`s support several common operations which have both copying and mutating versions:

```
const q1 = new Quaternion(0, 0, 0, 1)
const q2 = new Quaternion(0, 1, 0, 0)

// Multiplication (combines rotations)
const combined = q1.mul(q2)

// Inversion (reverses rotation)
const inverse = q1.inverse()

// Normalization (ensures unit length)
const normalized = q1.normalize()

// Conjugate
const conjugate = q1.conjugate()

// Convert to Euler angles (in degrees)
const euler = q1.toEuler()

// Get rotation axis and angle
const axis = q1.axis()
const angle = q1.angle() // in radians
```

The following table assumes that `q`, `r`, and `out` are `Quaternion`s; `v` and `w` are `Vec3`s; `t` is a `number`.

Operation	Code	Math	Description
Multiplication	`r = p.mul(q)` `p.mulInPlace(q)`	pq	Combines two rotations. Order matters: `p.mul(q)` means apply `q` rotation first, then `p`.

Inversion	`r = q.inverse()`
	`q.inverseInPlace()`
	`r = Quaternion.inverse(q)`
		q^{-1}	Creates a quaternion that represents the opposite rotation (negative angle around the same axis).

Normalization	`r = q.normalize()`
	`q.normalizeInPlace()`
	`Quaternion.normalize(q, out)`
		$\frac{q}{\lvert q\rvert}$	Ensures the quaternion has unit length. *If it doesn't have unit length then it isn't a valid rotation!*

Conjugate	`r = q.conjugate()`
	`q.conjugateInPlace()`
	`Quaternion.conjugate(q, out)`
		q^*	Negates the x, y, and z components. For unit quaternions, this is the same as `inverse`.

Vector Rotation	`w = Quaternion.mulVec3(q, v)`	$q\vec{v}q^*$	Rotates a `v` by `q`, producing a new `Vec3`.
Interpolation	`Quaternion.slerp(q, p, t, out)`	$\left(pq^{-1}\right)^t q$	Smoothly interpolates between two rotations via Spherical Linear Interpolation (Slerp)

Operation	Code	Math	Description
Vector Quaternion	`q = Quaternion.fromVec3(v)`	$(\vec{v}, 0)$	Create a quaternion with values `(v.x, v.y, v.z, 0)`. Note that this will likely not be a valid rotation and is only useful for advanced math techniques.

Euler Angles

Euler angles are one intuitive way to think about rotations (they are similar to "yaw, pitch, and roll"). When you type a rotation into the desktop editor as 3 values, you are describing a rotation via Euler angles.

The `Quaternion` class provides methods to convert between the quaternions and Euler angles:

```
// Convert from Euler angles (in degrees)
const quat = Quaternion.fromEuler(
  new Vec3(90, 0, 0),
  EulerOrder.XYZ
)

// Convert to Euler angles (in degrees)
const euler = quat.toEuler(EulerOrder.XYZ)
```

Rotation order matters: rotating around the y-axis by 90 degrees and then the x-axis by 90 degrees is *not* the same as doing those rotations in the other order. So when you specify Euler angle, it is important to specify the order the 3 numbers are applied in. The desktop editor uses the order: rotate around y ("yaw"), then rotate around x ("pitch"), then rotate around z ("roll"). This order is called "YXZ" since it matches the order the values are applied.

The `EulerOrder` enum specifies the order in which rotations are applied. The default value for `fromEuler` and `toEuler` is `EulerOrder.YXZ`. All possible orders are supported: `XYZ`, `XZY`, `YXZ`, `YZX`, `ZXY`, and `ZYX`.

Axis and Angle

Every rotation can be represented as rotating around some axis by some angle. Given an `axis` vector and an `angle` (in radians) you can create a quaternion. The `axis` does *not* need to be normalized.

```
const axis = new Vec3(0, 1, 0)  // rotate around y-axis
const angle = Math.PI / 4       // rotate by 45 degrees

const q = Quaternion.fromAxisAngle(axis, angle)
```

which will create a quaternion with the components:

```
const a = axis.normalized()
const s = Math.sin(angle/2)
const c = Math.cos(angle/2)
const q = new Quaternion(a.x * s, a.y * s, a.z * s, c)
```

You can also go the other way - extracting the axis and angle from a quaternion:

```
const quat = Quaternion.fromAxisAngle(Vec3.up, Math.PI/2)

const axis = quat.axis()    // (0, 1, 0)
const angle = quat.angle()  // ~1.57 radians (90 degrees)
```

Axis-angle representation is particularly useful for:

- Understanding rotations geometrically
- Rotating around specific axes like joints or hinges
- Creating smooth circular motion by incrementing the angle
- Converting between different rotation representations

Look Rotation

Look rotation is particularly useful for cameras and objects that need to orient themselves to face a particular direction. You specify which direction you want the `forward` to point in and then what direction the `up` should point in (or point as closely as possible to). The `up` value is optional and defaults to `Vec3.up`. The two input vectors *do not need to be normalized*.

```
// Look down the world's x-axis with your up pointing along the world's z-axis
const quat = Quaternion.lookRotation(Vec3.right, Vec3.forward)
```

This is commonly used to:

- Orient cameras to look at targets
- Make objects face their direction of movement
- Align objects with surface normals

Spherical Linear Interpolation (Slerp)

Similar to lerp, **Slerp** (Spherical Lerp) provides smooth interpolation between two quaternion rotations. Like vector lerp, it takes a blend value between 0 and 1:

```
const start = Quaternion.fromEuler(new Vec3(0, 0, 0))
const end = Quaternion.fromEuler(new Vec3(0, 90, 0))

const halfway = Quaternion.slerp(start, end, 0.5)    // 45 degree rotation
const quarter = Quaternion.slerp(start, end, 0.25)   // 22.5 degree rotation
const complete = Quaternion.slerp(start, end, 1.0)   // 90 degree rotation
```

World Class

The `World` class represents the currently running instance and the world's persistent data.

`World` Class Member	Description
System Events	
`static onUpdate`	The built-in LocalEvent for subscribing to the on-update frame event.
`static onPrePhysicsUpdate`	The built-in LocalEvent for subscribing to the pre-physics frame event.
Instance Management	
`reset`	Reset the instance.
`matchmaking`	Manage the instance's open setting.
Instance Players	
`getPlayerFromIndex`	Find which, if any, Player has the given index.
`getPlayers`	Get all current players in the instance.
`getServerPlayer`	Get the player representing the server.
`getLocalPlayer`	Determine which player's client is running the current code.

`World` Class Member	Description
`ui`	Show a popup or tooltip UI to players.
World Entities	
`getEntitiesWithTags`	Find entities in the instance.
`spawnAsset`	Spawn an asset into the instance.
`deleteAsset`	Delete spawned entities from the instance.
World Data	
`id`	Get the unique id for this world.
`leaderboards`	Manage player leaderboard scores.
`persistentStorage`	Manager persistence (player saved data).

Components

Components are the powerhouse of scripting in Horizon. They contain the logic and behaviors for reacting to events in the world and making stuff happen in the world (such as transforming entities, activating gizmos, and more).

The **primary steps for scripting** are:

1. Create a new file (or add to an existing one)
2. Create a new Component class
3. Attach the Component to an entity (or many entities)
4. Add property definitions that will appear in the Properties panel
5. Connect code to run when system (or user) events occur

The steps above are the "main path" but there are also many more parts of scripting:

- Sending events
- Creating timers and async code
- Creating local scripts and transferring ownership for low-latency interactions
- Running code every frame
- Interacting with the physics system
- Rendering UI
- Creating tooltips and popups
- Spawning assets
- Tinting and modifying meshes
- Managing NPCs
- Creating and updating persistence

  and so much more!

Component Class

Scripting an entity requires attaching a component to it. Creating Components is the core of scripting. To create a component, you subclass `Component`, override the `start` method (code that runs when the entity "awakens"), and register the component. A minimal component looks like:

```
class BasicComponent extends Component<typeof BasicComponent> {
  override start() {}
}
Component.register(BasicComponent)
```

The most unusual part is the `<typeof BasicComponent>` part. You must always put the component's class name within the angle brackets (<>) after `Component` (this is a trick Horizon uses for component props).

Attaching Components to Entities

Registering: the code `Component.register(BasicComponent)` above must be called for every component subclass that you create. This call results in the component appearing in the **Attached Script** dropdown in the desktop editor. The component will appear with the format *filename:componentName*. So if the `BasicComponent` above is inside the TypeScript file `Demo` then the script will appear in the Attached Script dropdown as `Demo:BasicComponent` . This allows you to put **multiple components in a file**.

The `Component.register` method takes an optional second argument to override the name in the dropdown. So if the following line of code is in a file `Obstacles` :

```
Component.register(SpinningTurntableComponent, 'Spinner')
```

then the class will appear in the Attached Script dropdown as *Obstacles:Spinner*. If you want to override the name in `Component.register` , only use letters, numbers, and underscore in the name.

Attaching: once a component subclass is registered, you can click an entity in the desktop editor, open the Properties panel, and choose it from the Attached Script dropdown. That entity will now run the code in that component. Once you attach a script, the Properties panel will show all the component's properties as editable fields. The next section explains how to add properties to a component.

Component Properties

Components can define properties that appear in the Properties panel by implementing the optional static method `propsDefinition` . When you attach the component, these properties will be configurable in the UI and accessible via the component's `props` field.

Component Properties Example

Here's a simple example showing how to define properties:

```
import {Color, Component, PropTypes} from 'horizon/core'

class ExampleComponent extends Component<typeof ExampleComponent> {
  static propsDefinition = {
    name : {type : PropTypes.String},
    color : {type : PropTypes.Color, default: new Color(1, 0.5, 0)}
  }

  override start() {
    console.log(this.props.name, this.props.color.toString())
  }
}
Component.register(ExampleComponent)
```

This example creates:

- A name field that shows as a text input
- A color field that shows as a color picker, defaulting to orange (1, 0.5, 0)

The static `propsDefinition` object defines your properties. Each property needs:

- A *key* that will become the property name in `this.props`
- An *object* value containing:

- `type` : *Required*. a value from PropTypes (note that not all kinds of `PropTypes` are useful in a `propsDefinition` ; see the limitations below).
- `default` : *Optional*. Initial value for the property in the Properties panel.

`PropTypes` Value	Results In	Default Value	Notes
`Number`	`number`	`0`	-
`String`	`string`	`''`	-
`Boolean`	`boolean`	`false`	-
`Vec3`	`Vec3`	`(0, 0, 0)`	-
`Color`	`Color`	`(0, 0, 0)` Black	RGB values between 0 and 1.
`Entity`	`Entity | null`	`null`	Cannot specify default
`Quaternion`	`Quaternion`	`(0, 0, 0)`	Properties panel value is edited as YXZ Euler angles
`Asset`	`Asset | null`	`null`	Cannot specify default

Important Type Distinctions

Entity is a type used in code like:

```
const e: Entity = ...
```

While `PropTypes.Entity` is data used in `propsDefinition` or CodeBlockEvents.

No Type Checking

TypeScript does not type-check the `static propsDefinition` object. Verify your property definitions carefully.

Limitations

- **Player**: the `PropTypes` object includes `PropTypes.Player` but there is no way to make use of it for `propsDefinition` .
- **Arrays**: the `PropTypes` enum includes array versions of all types (like `NumberArray`), but there is no way to make use of them for `propsDefinition` .

  Nullable Types: Properties using `PropTypes.Entity` or `PropTypes.Asset` will always be nullable: `Entity | null` or `Asset | null` , respectively. You must check for `null` before using these properties:

```
static propsDefinition = {theEntity: {type: PropTypes.Entity}}

// ...

override start() {
  if (this.props.theEntity !== null) {
    // Safe to use this.props.theEntity here
  }
}
```

Component Lifecycle

Components follow a strict, sequential lifecycle with 3 key parts. All components are **prepared** and then all are **started** (this is useful for event subscriptions). Then all components are "active", running in the world. If, or when, the editor stops, the component's entity

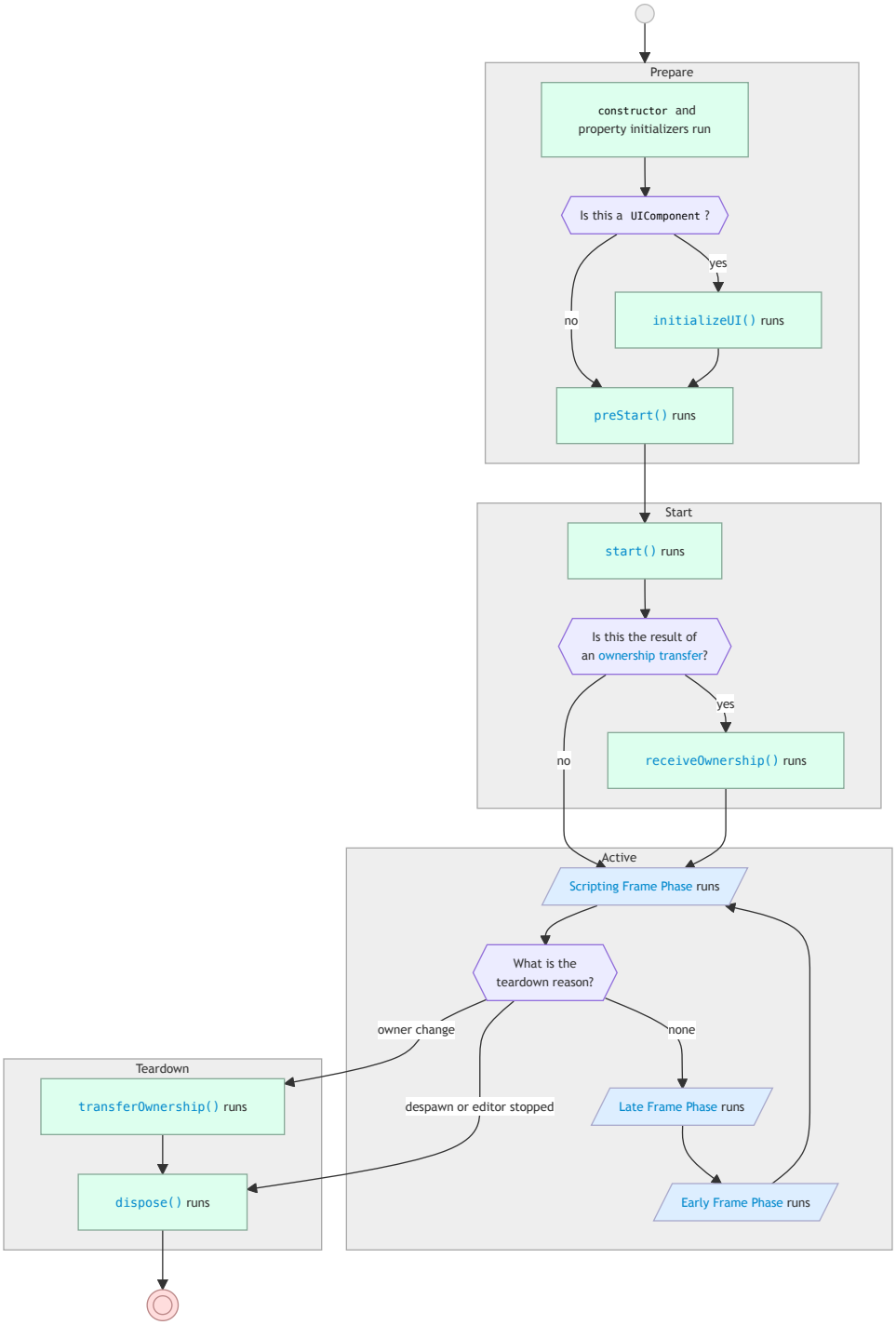
despawns, or the component's entity prepares to change owner then they are **torn down**.

Likewise, when a group of entities are spawned, all them are prepared; then, all of them are started.

1. **Preparation** - When components are created (via instance start, spawning, or after an ownership transfer):
   - Component allocation occurs
   - Constructor executes
   - Property initializers run
   - `initializeUI()` executes (UIComponents only)
   - `preStart()` executes
2. **Start** - After preparation:
   - `start()` executes
   - `receiveOwnership()` executes (only during ownership transfers)
   - Component becomes "active" (begins processing events and timers)
3. **Teardown** - When the editor stops, component despawns, or an before an ownership transfer:
   - `transferOwnership()` executes (only during ownership transfers)
   - Component is disposed, meaning that `dispose()` executes and all callbacks registered with `registerDisposeOperation` run, except for the ones where the `DisposeOperationRegistration` was already canceled or ran.
   - All [async timeouts and intervals] created with the component are canceled.
   - All event subscriptions created with the component are disconnected.

> **Component Initialization Sequence**
>
> 1. Property initializers run first
> 2. All components execute `preStart()` and `initializeUI()`
> 3. All components execute `start()`
>
> **Start of world**: The above order is true for all components at the start of the world.
> **After spawn**: It is also true for all components resulting from a spawn.
> **After ownership transfer**: It is also true for all components created with a new owner (e.g. if in a frame the current client) has 3 new components then all 3 will be `preStart` ed before any are `start` ed.

The diagram below shows the full lifecycle of a component. All green rectangle boxes are TypeScript code executing during the Scripting Frame Phase. The full breakdown a frame gives another view into when all these actions occur.



Connect to events in `preStart`. Send events in `start`.

Do *not* connect in `start`. Do *not* send in `preStart`. See the explanation in the events section for a detailed. explanation.

Property initializers run before `props` are available.

```
// ❌ Incorrect: Props not available during initialization
class BadComponent extends Component<typeof BadComponent> {
    private color = this.props.color  // Will throw an error!
}

// ✅ Correct: Initialize in preStart
class GoodComponent extends Component {
    private color: Color = new Color(0, 0, 0)  // Default value if needed

    override preStart() {
        this.color = this.props.color  // Safe to access props here
    }
}
```

> ⚠ **Do not attempt to override the component constructor.**
>
> The base Component class handles critical setup that could break if the constructor is overridden. So, instead:
>
> - Use property initializers for data (without accessing props)
> - Use `preStart()` for event registration
> - Use `start()` for initialization behavior

Async (Delays and Timers)

There are two ways to delay code (to run it later):

- **timeouts**: code that will run once after a delay (unless canceled before it runs).
- **intervals**: code that will run after a delay, and then again after that same delay, and so on forever (unless it is canceled).

Canceling a timeout or an interval is called **clear**ing it. These naming conventions are consistent with standard JavaScript.

Component instances have a member `async` that provides access to functions for creating async code. For example if you have a `component` you might write:

```
component.async.setTimeout(() => console.log('ready!'), 1000 /* ms */)
```

to execute the given `console.log(...)` after 1000 milliseconds(1 second). The following methods are provided in `Component`'s `async` member:

`Component` `async` member	Description

`setTimeout(`
`  callback: TimerHandler,`
`  timeout?: number,`
`  ...args: unknown[]`
`) => number`
	Schedule the `callback` to fire after `timeout` milliseconds. If `args` are provided then they will be passed into `callback` when it is called. `setTimeout` returns an `id` that can be passed to `clearTimeout` to cancel running `callback` (if it is cancelled before it does so). If `timeout` is omitted it will be treated as 0 (see below).

`clearTimeout(`
`  id: number`
`) => void`
	Cancel the timeout with the given `id` (if it hasn't run yet).

`setInterval(`
`  callback: TimerHandler,`
`  timeout?: number,`
	Schedule the `callback` to fire after `timeout` milliseconds (and then again after the same delay, and again, and so on). If `args` are provided then they will be passed into `callback` every time that it is called. `setInterval` returns an `id` that can be passed to `clearInterval` to cancel

`Component async` member	Description
`...args: unknown[]` `) => number`	running `callback` (so that it doesn't run any more times). If `timeout` is omitted it will be treated as 0 (see below).

`clearInterval(`
`id: number`
`) => void`
	Cancel the interval with the given `id` (it will not run any more times).

Component disposal: the `id`s returned from `setTimeout` and `setInterval` are automatically registered with the component's disposal to be cleared. Thus if you write:

```
component.async.setInterval(() => console.log('hi!'), 1000 /* ms */)
```

then if, or when, `component` is torn down, the interval will be automatically canceled.

Late frame phase: the callbacks passed to `setTimeout` and `setInterval` are executed in the late frame phase when async callbacks are checked for readiness.

> **Timeouts and Intervals are not precisely timing. They make a "best attempt" at the delay (but may wait slightly *longer*).**
>
> The methods above are not precise in when the callback runs. They will wait at least as long as the requested `timeout` value and then run at *the next convenient time* after that. So, for example, if you create a `timeout` with a 0 millisecond delay, it won't run immediately; it will run "super soon" (likely during the next late frame phase). If you create an interval with a timeout of 0 milliseconds, it may only run it a few times (or even just once) every frame, to prevent hurting perf.

> **Use underscores to make numbers more readable.**
>
> JavaScript (and therefore TypeScript) allows underscores to be inserted into numbers solely for readability. That means `123` and `1_2_3` are the same value. You can thus use underscores to make numbers more readable. So instead of writing `10000` to mean 10,000 milliseconds, you can write `10_000`!

Run Every Frame (PrePhysics and OnUpdate)

Async intervals are not effective for **running code every frame** because they are difficult to align the timing of (due to being imprecise).

The World has two static members exposing Local Events that are broadcast every frame (to all clients):

`World` class `static` member	Type	Description

`onPrePhysicsUpdate`	`LocalEvent<{`
	`deltaTime: number`
	`}>`
		A built-in local event that is broadcast every frame **before physics runs** to all clients device and server.

`onUpdate`	`LocalEvent<{`
	`deltaTime: number`
	`}>`
		A built-in local event that is broadcast every frame **after physics runs** to all clients device and server.

```
const subscription = component.connectLocalBroadcastEvent(
  World.onUpdate,
  (info) => {
    // runs every frame before physics updates
    console.log(`${info.deltaTime} seconds since last frame`)
  }
)
```

Callbacks registered with `onPrePhysicsUpdate` run before physics computations occur in the frame. Callbacks registered with `onUpdate` run after physics computations. **onUpdate is usually what you need**. See the description of prePhysics vs onUpdate for more information.

The callback provides a single argument of type `{deltaTime: number}` which contains the amount of time that has passed since the event was last broadcast. See the section on receiving events to learn about `connectLocalBroadcastEvent` and the `EventSubscription` that it returns.

BuiltInVariableType

`BuiltInVariableType` represent "primitive Horizon data". It is used in CodeBlockEvents and defining props with PropTypes. The different types in `BuiltInVariableType` are available as a TypeScript object in PropTypes.

```
type BuiltInVariableType =
  | string      | string[]
  | number      | number[]
  | boolean     | boolean[]
  | Vec3        | Vec3[]
  | Entity      | Entity[]
  | Quaternion  | Quaternion[]
  | Color       | Color[]
  | Player      | Player[]
  | Asset       | Asset[]
```

PropTypes

`PropTypes` is an enum representing all the same values in BuiltInVariableType. `PropTypes` is used in creating the component propsDefinition and in creating CodeBlockEvents.

```
const PropTypes = {
  Number: "number";
  String: "string";
  Boolean: "boolean";
  Vec3: "Vec3";
  Color: "Color";
  Entity: "Entity";
  Quaternion: "Quaternion";
  Player: "Player";
  Asset: "Asset";
  NumberArray: "Array<number>";
  StringArray: "Array<string>";
  BooleanArray: "Array<boolean>";
  Vec3Array: "Array<Vec3>";
  ColorArray: "Array<Color>";
  EntityArray: "Array<Entity>";
  QuaternionArray: "Array<Quaternion>";
  PlayerArray: "Array<Player>";
  AssetArray: "Array<Asset>";
};
```

SerializableState

`SerializableState` represents the type of data that can be packaged up to be **sent over the network**. It is used in NetworkEvents and ownership transfer.

```
type SerializableState =
  | { [key: string]: SerializableState }
  | SerializableState[]
  | PersistentSerializableStateNode
  | TransientSerializableStateNode

type PersistentSerializableStateNode =
  | Vec3 | Entity | Quaternion | Color
  | number | boolean | string
  | bigint | null;
type TransientSerializableStateNode = Player;
```

`PersistentSerializableState` is data that can be packaged up to store in persistent data. It is the same as `SerializableState` *except* that it *does not include Player* (since player instances are ephemeral to the instance).

Communication Between Components

The primary way in which components communicate with one another, and react to occurrences in the world, is by **sending** and **receiving** events through entities (or through players). If two components are running on the same client then you can have them interact directly without using events.

There are multiple kinds of events with different purposes. Event listeners are usually connected in `preStart`. Events can be sent at any time.

`LocalEvent`s and `NetworkEvent`s can be sent to specific entities or be broadcast to all listeners in the world. Many "system actions" (such as players entering the world, an entity being grabbed, a collision occurring, etc) are sent as built-in CodeBlockEvents; there are many built-in CodeBlockEvents.

All event types can be instantiated with **custom user-made types** by simply calling `new` on the event type and making one. When creating custom events, export them so that you can create them once and share them across files. If you keep instantiating the same

event repeatedly across your code, you are likely to make an error (and make refactors more difficult).

Event	Purpose	Timing	Payload
CodeBlockEvent	Listen to built-in CodeBlockEvents. Communicate with Codeblock scripts.	*Asynchronously* run in the next scripting frame phase if sent to the same client. Otherwise, it runs after a network trip on the receiving client.	Tuple of BuiltInVariableTypes.
LocalEvent	Communicate with a TypeScript scripted entity on the same client. Supports broadcast.	Delivered *synchronously* (immediately).	*Anything*
NetworkEvent	Communicate with a TypeScript scripted entity on any client. Supports broadcast.	*Asynchronously* run in the next scripting frame phase if sent to the same client. Otherwise, it runs after a network trip on the receiving client	SerializableState

Receiving Events

Connecting Events: Events are "subscribed" to using a `Component` method starting with `connect...`, such as

```
const subscription = component.connectLocalEvent(
  entity, event, callback
)
```

which will result in `callback` being run every time `event` is sent to `entity` (if `entity` has the same owner as `component.owner`, since the line above used a *local* event).When using broadcast, there is no specified entity to listen to.

```
component.connectLocalBroadcastEvent(event, callback)
```

Many "system actions" are communicated by sending built-in CodeBlockEvents to entities. There are also some built-in local-events.

You can also connect to events sent to players.

Disconnecting Events: All `connect...` events return an `EventSubscription`, a type with a single `disconnect(): void` method. Calling

```
subscription.disconnect()
```

would make it so that `callback` is no longer called. It's good practice to disconnect listeners when you are done with them.

> ⚠ **Connect to events in `preStart`. Send in `start`.**
>
> Imagine the following scenario: `ComponentA`, in its `start`, sends an event to `ComponentB`'s entity; but `ComponentB` doesn't register to listen to the event until its `start`. Does `ComponentB` get the event? It depends on which component `start`ed first!
>
> This is a subtle problem. To address this: **all components `preStart` before any `start`**. This means that if `ComponentA` sends an event in `start` and `ComponentB` subscribes to that event in `preStart` then `ComponentB` is guaranteed to to be ready to receive the event by time `ComponentA` sends it!
>
> **Never `connect` in `start`. Never `send` in `preStart`**. This can cause events to get missed!

Sending Events

There are many events sent by the system (built-in CodeBlockEvents and built-in local-events). You can also `new` your own events and then send and receive them as well.

To send an event you simply use the `Component` `send...` method matching the event type:

```
component.sendNetworkEvent(entity, networkEvent, data)
```

The `data` will then be passed into any callback that are connected to that event on that entity. If you are sending a broadcast even then you don't specify the receiver:

```
component.sendNetworkBroadcastEvent(networkEvent, data)
```

You can also send events to players.

Cannot Cancel: once an event is sent there is no way to revoke it.

Routing Events through Players

The (non-broadcast) `connect...` and `send...` methods on Component allow either an Entity or a Player for the `target` argument.

Routing events through players follows the same rules as entities.

The example below shows a script that has props for a trigger and an index. It listens to the OnPlayerEnterTrigger event to know when a player enters the trigger. When one does, a network event is sent **to the player** to announce that they were "captured" by the trigger with the given index. Anyone listening to the `playerCaptured` event on that player will receive the event.

Note that the `playerCaptured` event is `export` ed from the file so that other scripts can listen to it. Also, this particular example could have also been achieved with a broadcast event.

```
import {
  NetworkEvent, Component, PropTypes, CodeBlockEvents
} from "horizon/core"

export const playerCaptured = new NetworkEvent<{ index: number }>(
  'didCapturePlayerInTrigger'
)

class TriggerCaptureComponent extends Component<
  typeof TriggerCaptureComponent
> {
  static propsDefinition = {
    trigger : { type : PropTypes.Entity },
    index : { type : PropTypes.Number }
  }

  preStart() {
    const {trigger, index} = this.props
    if (trigger) {
      this.connectCodeBlockEvent(
        trigger,
        CodeBlockEvents.OnPlayerEnterTrigger,
        (player) => {
          this.sendNetworkEvent(player, playerCaptured, {index})
        }
      )
    }
  }

  start() {}
}
Component.register(TriggerCaptureComponent)
```

Code Block Events

`CodeBlockEvent`s are a **legacy event type** used for listening to built-in events and for communicating with Codeblock scripts. **Do not create custom `CodeBlockEvent`s** as they can conflict with built-in events and cause unexpected behavior (unless you are communicating with Codeblock scripts, then you *must* use `CodeBlockEvent`s).

Creation:

```
const cbEvent = new CodeBlockEvent<[food: string, count: number]>(
    'registerGroceries',
    [PropTypes.String, PropTypes.Number]
)
```

A `CodeBlockEvent` requires:

- A name string (e.g. `'registerGroceries'`)
- A tuple of parameter types passing in as PropTypes and as generics (in the `<...>`)

Usage example (where the event is sent to the component's entity):

```
import {
  CodeBlockEvent, Component, PropTypes
} from "horizon/core"

const cbEvent = new CodeBlockEvent<[food: string, count: number]>(
  'registerGroceries',
  [PropTypes.String, PropTypes.Number]
)

class ExampleComponent extends Component<typeof ExampleComponent> {
  preStart() {
    this.connectCodeBlockEvent(this.entity, cbEvent, (food, count) => {
      console.log(`I need to buy ${count} ${food}!`)
    })
  }

  start() {
    this.sendCodeBlockEvent(this.entity, cbEvent, 'apples', 5)
  }
}
Component.register(ExampleComponent)
```

Properties:

- **Client Support**: Can be sent and received across clients, meaning that you can send to an entity with a different owner than the sender, and likewise connect to an entity with a different owner than the connector. One exception: built-in broadcasted CodeBlockEvents *cannot* be received on a different client than the event is emitted on.
- **Execution**: Runs in the next scripting frame phase after receipt (which maybe be on a different client after a "network trip")
- **Data Format**: Requires a tuple of BuiltInVariableTypes
- **Event Disambiguation**: System checks both name and parameterTypes before executing listeners

Built-In Code Block Events

The system uses `CodeBlockEvent`s for many built-in actions. For example, when an entity enters a trigger zone with matching tags, the system sends `CodeBlockEvents.onEntityEnterTrigger` to the trigger. See the list of built-in CodeBlockEvents for more info.

Broadcasted `CodeBlockEvents`: some built-in `CodeBlockEvent`s are "broadcast" meaning that you can *listen to any entity to receive them* (as long the receiver is executing on the same client the event is emitted on). The list built-in CodeBlockEvents includes information on which ones are *broadcast*. Throughout this document, 📡 denotes a *server-broadcast* `CodeBlockEvent`; 🏠 denotes a *device-broadcast* `CodeBlockEvent`.

For example, to listen to `CodeBlockEvents.onPlayerEnterWorld`, you can listen to it on *any entity* (though it has to be server-owned). There is no way to *send* a broadcast event yourself.

```
// on a Component
this.connectCodeBlockEvent(
  entity, // any entity owned by the server
  CodeBlockEvents.onPlayerEnterWorld,
  callback
)
```

> **CodeBlockEvents are the only way for TypeScript and Codeblock scripts to communicate**
>
> Codeblock scripts are unable to use either LocalEvents or NetworkEvents, thus the only method of communication between them is via CodeBlockEvents.

Sending a CodeBlockEvent with an Asset parameter to scripts with 'local' execution mode will throw a silent error.

If you do try to send an Asset in a CodeBlockEvent from a script executing on the server to one executing on a player device, the server script execution will be silently killed at the point where you try to send the event. Recall that unless you are communicating with Codeblock scripts, the **advice is to never use custom CodeBlockEvents**. Use LocalEvents and NetworkEvents for all your event sending and receiving.

A TypeScript connecting to an entity with an attached Codeblock script can lead to issues.

Some times TypeScript and Codeblock scripts will "fight each other" when both listening to events from the same entity. It is a subtle bug that doesn't appear in all cases. However we recommend: **do not have TypeScript and Codeblock scripts listen to events from the same entity**.

Example: Imagine a trigger that has a Codeblock script attached to it that listens to trigger enter on `self` (Codeblock's version of `this.entity`). If you then have a TypeScript script also connect to the trigger enter event then it turns out that neither script will receive trigger events for that trigger.

Network Events

`NetworkEvent`s are the **recommended alternative to `CodeBlockEvent`s** for communication between components with different owners.

Creation:

```
const networkEvent = new NetworkEvent<{ code: number }>(
  'setCodeWithNumber'
)
```

A `NetworkEvent` requires:

- A name string (highly specific to avoid conflicts)
- A payload type satisfying `SerializableState`

Usage example (where the event is sent to the component's entity):

```
import {
  NetworkEvent, Component
} from "horizon/core"

const networkEvent = new NetworkEvent<{ code: number }>(
  'setCodeWithNumber'
)

class ExampleComponent extends Component<typeof ExampleComponent> {
  preStart() {
    this.connectNetworkEvent(this.entity, networkEvent, ({code}) => {
      console.log(`I got ${code}!`)
    })
  }

  start() {
    this.sendNetworkEvent(this.entity, networkEvent, {code: 42})
  }
}
Component.register(ExampleComponent)
```

Properties:

- **Client Support**: Can be sent and received across clients, meaning that you can send to an entity with a different owner than the sender, and likewise connect to an entity with a different owner than the connector.
- **Execution**: Runs in the next scripting frame phase after receipt (which maybe be on a different client after a "network trip")
- **Data Format**: Accepts any SerializableState
- **Event Disambiguation**: ⚠️ The system only checks the event name - use highly specific names to avoid conflicts between different `NetworkEvent` s!

Local Events

`LocalEvent` s are designed for communication between entities on the same client, offering maximum flexibility with minimal overhead. They execute essentially as **synchronous function calls**.

Creation:

```
const doorEvent = new LocalEvent<{open: boolean, date: Date}>(
  'setDoorState'
)
```

A `LocalEvent` requires:

- An optional name string (useful for debugging)
- A type parameter for payload; it can be any type whatsoever (since there is no network serialization). Note that the example above *cannot* be done with NetworkEvent because `Date` is not compatible with SerializableState.

Usage example (where the event is sent to the component's entity):

```
import {
  LocalEvent, Component
} from "horizon/core"

const doorEvent = new LocalEvent<{open: boolean, date: Date}>(
  'setDoorState'
)

class ExampleComponent extends Component<typeof ExampleComponent> {
  preStart() {
    this.connectLocalEvent(this.entity, doorEvent, (info) => {
      console.log(`I got ${info.open} on ${info.date}!`)
    })
  }

  start() {
    this.sendLocalEvent(this.entity, doorEvent, {
      open: true, date: new Date()
    })
  }
}
Component.register(ExampleComponent)
```

Properties:

- **Client Support**: Local only - events cannot cross client boundaries
- **Execution**: Immediate synchronous execution on the local client
- **Data Format**: Accepts any arbitrary type as payload type
- **Event Disambiguation**: Uses referential equality - no risk of name conflicts.

> ⚠️ **You must use the exact same `LocalEvent` instance for both `sendLocalEvent` and `connectLocalEvent`.**
>
> Since `LocalEvent`s are disambiguated *referentially* you **must use the same `LocalEvent` instance**. In the code below, the first example will run `callback`. The second example does not.
>
> ```
> // Example 1: ✅ Right!
> const evt = new LocalEvent('jump')
> this.connectLocalEvent(entity, evt, callback)
> // ... later ...
> this.sendLocalEvent(entity, evt, {})
>
> // Example 2: ❌ Wrong!
> // The send is using a different event than the connect.
> this.connectLocalEvent(entity, new LocalEvent('jump'), callback)
> // ... later ...
> this.sendLocalEvent(entity, new LocalEvent('jump'), {})
> ```

Built-In Local Events

There are currently two groups of built-in LocalEvents:

- The World class has static members onPrePhysicsUpdate and onUpdate for running code every frame (before and after physics, respectively). These are broadcast on all clients.
- The PlayerControls class has static members for `onFocusedInteractionInputStarted`, `onFocusedInteractionInputMoved`, `onFocusedInteractionInputEnded`, and `onHolsteredItemsUpdated` which are all broadcast on the player device that own the

controls.

Broadcast events

NetworkEvents and LocalEvents both support broadcasting. Instead of sending events to and listening to events on a specific entity, you instead simply listen for the event being "broadcast", and then any registered listener can receive it.

Note: events are simply LocalEvents, NetworkEvents, or CodeBlockEvents. There is no special "broadcast event type". Instead, broadcast refers to the way that events are sent and received.

> ⚠ **Me mindful not to overuse broadcast events.**
>
> Broadcast events decouple senders from receivers without requiring specific entity or player targets. However, overuse can lead to performance issues when many listeners run unnecessarily to check if events are relevant to them. If this happens, consider either splitting the event into more specific events or routing through entities or players instead.

For example, if you create the following `LocalEvent`:

```
const evt = new LocalEvent<{value: number}>()
```

then you send it like this (there is no `entity` receiver arg)

```
component.sendLocalBroadcastEvent(evt, {value: 10})
```

and it will be received by all listeners on the same client (since it's a local event). To register one of those listeners, do:
then you can register to listen to it being sent from

```
component.connectLocalBroadcastEvent(evt, callback)
```

LocalEvent Broadcast: `sendLocalBroadcastEvent` will synchronously (immediately) call all registered listeners on the same client. The order that the callbacks are called in is undefined and should not be relied on. There are some built-in LocalEvents which are broadcasted.

NetworkEvent Broadcast: `sendNetworkBroadcastEvent` is asynchronous (delayed). Any listeners on the same client will process the event in the next scripting frame phase. Listeners on other clients will wait until they receive the event (over the network) and then will process the event in their next scripting frame phase.

> ℹ **NetworkEvent broadcast has an extra optional parameter for fine-grained control.**
>
> The `sendNetworkBroadcastEvent` method takes an extra, optional, final parameter: `player?: Player[]` which allows you to limit which clients the event is sent to. This allows for fine-grained optimization, but should ⚠ only be used if you absolutely understand what you are doing.

CodeBlockEvent Broadcast: There are no methods for sending or receiving broadcasted `CodeBlockEvent`s. However there are some built-in CodeBlockEvents that can be connected to using *any entity* and thus act kind of like a broadcast.

Limit receiving players: all `send...` methods (except for CodeBlockEvents) take an additional optional final parameter: `players?: Player[]` which allows you to specify that the event will only be sent to those players' clients. This is an *expert-level* feature; **only use it if you truly know what you are doing**.

Converting Between Components and Entities

When two components are running on the same client they can directly call one another's functions (instead of going through entities and the event system). There are 2 ways to "find component instances on the local client:

1. **components attached to entities**: you can do `entity.getComponents()` to get all components on an entity. If only one component is attached to the entity then the array will have 1 element in it. You can also pass in a class `entity.getComponents(ExampleComponent)` to get an array of `ExampleComponent` instances attached to the entity (which, again, will be at most one).
2. **all component instances**: you can run `Component.getComponents(ExampleComponent)` to get an array of all instances of `ExampleComponent` on the local client.

> **`getComponents` cannot be used until start.**
>
> You cannot call `entity.getComponents(...)` or `Component.getComponents(...)` in a property initializer, `initializeUI`, or in `preStart`. This information isn't ready until after the prepare state of component instantiation.

> **Calling a method on a component.**
>
> In this example we find all `ListenerComponent`s in the local client from within the `SpeakerComponent`. We are then able to directly access the `props` and the `hear` method on `ListenerComponent`.

```
import {Component, PropTypes} from 'horizon/core'

class ListenerComponent extends Component<typeof ListenerComponent> {
  static propsDefinition = {
    name: { type: PropTypes.String }
  }

  start() {}

  hear(message: string) {
    console.log('I heard: ' + message)
  }
}
Component.register(ListenerComponent)

class SpeakerComponent extends Component<typeof SpeakerComponent> {
  start() {
    const listeners = Component.getComponents(ListenerComponent)

    for (const listener of listeners) {
      listener.hear('Hello, ' + listener.props.name)
    }
  }
}
Component.register(SpeakerComponent)
```

Disposing Objects

The `DisposableObject` interface represents a TypeScript object with a `dispose()` method which can be called to do "cleanup". Additionally a `DisposableObject` must also have the `registerDisposeOperation` method, which allows callbacks to be registered to also be run when the object is `dispose`d.

> **currently only Component implements the `DisposableObject` interface.**

```
interface DisposableObject {
  dispose(): void;
  registerDisposeOperation(
    operation: DisposeOperation
  ): DisposeOperationRegistration;
};

type DisposeOperation = () => void;

interface DisposeOperationRegistration {
  // run the dispose operation
  run: () => void;

  // cancel the operation so it never runs
  cancel: () => void;
};
```

When you call `registerDisposeOperation` you get back a `DisposeOperationRegistration` which has methods `run` , to manually dispose (even earlier), and `cancel` to stop the passed in `operation` from ever running.

> **`PlayerControls` takes a `DisposableObject`**
>
> In the PlayerControls class, the static method `connectLocalInput` takes a `DisposableObject` object as an argument. The controls will be *unregistered* when the disposable object disposes.

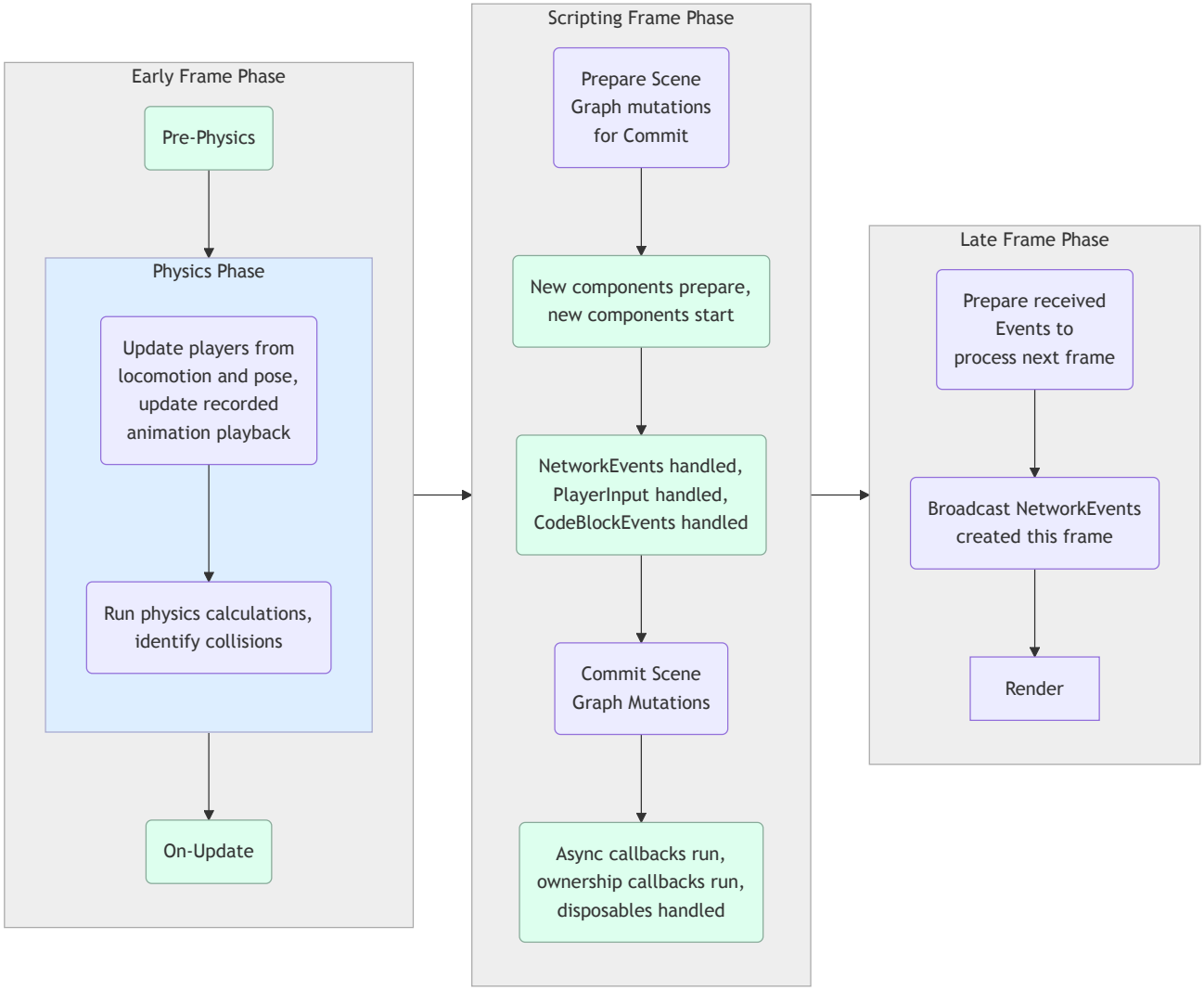
Frame Sequence

TODO

NOTE: a pre-physics handler in code blocks scripts runs before start

`async` runs AFTER default.

> On Frame N during PrePhysics:
> moving object into a trigger
>
> On Frame N+1 during Events:
> triggerEnter is handled
>
> I think the rule is as simple as:
>
> Any CODE BLOCK EVENT generated in a frame is process the next frame, no exceptions.



Proved: preStart and start run in "frame -1". Code blocks "start" event is handled in frame "0" (after frame 0's prePhysics and default).

Proved: async runs at end of frame. An interval / timeout of 0ms (or some other tiny value) is allowed to run many times per frame (but is capped via some max time - meaning that the async phase will deplete the timer queue only for so long).

Proved: creating an async function ANYWHERE in a frame (EVEN during that frame's async phase) is eligible to be run during that frame's async phase (as long as it's delay is small enough and we haven't hit the "budget").

Proved: creating a timeout / interval in start() with timeout of 0ms will run in that frame before prePhysics.

Proved: 0ms is the default time when omitted.

Proved: a single async timeout that takes too long gets killed with an error in the console. Throwing (and the associated error allocation) take so much CPU time that basically no other async handlers will run this frame.

Proved: The `deltaTime` in the PrePhysicsUpdate and OnUpdate has a maximum of 0.1. Horizon always runs all the code in a frame. When the code in a frame takes too long to run, the framerate on the device executing the script will drop. For example, an OnUpdate handler running at 20fps in a server-executed script will cause all server-executed scripts and all server-owned physics objects to update at 20fps. A player-device executed script running at 18fps causes that player's entire experience in the Horizon app to run at 18fps until the player exits that world.

Proved: if `sendCodeBlockEvent` is called 2048 times (or more) in a frame you get an error (and none of the events are processed that frame). Note 2047 times is allowed; 2048 is not. There is a bug where the thrown error implies that 2048 is allowed (it's not!).

Proved: code block event handlers will eventually timeout but it seems to be upward of some 10s of seconds. Even if each event takes a long time, Horizon will do its best to process the entire queue every frame. It never punts events to a future frame. Meaning... it may stall the JS thread for 10s+ (frame isn't stuck, just JS thread) to process all the events.

Proved: each code block event handler is wrapped in a try.

Early Frame Phase

TODO

- PrePhysics Phase
- Physics Phase
- OnUpdate Phase

Scripting Frame Phase

TODO

- Component Initialization
- Network Events Handling
- Player Input Handling
- Code Block Events Handling
- Committing Scene Graph Mutations
- Async Handling

Late Frame Phase

TODO

- Network Sync
- Render (if a player device)

TODO

Component Inheritance

It is not recommended to create deep hierarchies of components. We recommend you prefer **composition over inheritance** and use the general guidance of: **only subclass an abstract class**.

If you want to make your own component subclass that is meant to be further subclassed, then this pattern should suffice (note that the abstract Parent is *not* registered).

```
abstract class Parent<T> extends Component<typeof Parent & T> {
  static propsDefinition = {
    name: {type: PropTypes.String},
  }
  start() {}
  abstract greeting() : string
}

class Child<T> extends Parent<typeof Child> {
  static propsDefinition = {
    ...Parent.propsDefinition,
    favoriteNumber: {type: PropTypes.Number},
  }
  start() {}
  greeting() {
    return this.props.name + ' ' + this.props.favoriteNumber
  }
}
Component.register(Child)
```

Script File Execution

TODO
Auto-Restart on Script Edit

restarts
spin up (once per file per client - except on edit / reset)
transfer

Helper Functions

Horizon has a few helper functions in `horizon/core`:

- **clamp**: ensures that a given number stays within a specified range. `clamp(value: number, min: number, max: number): number`
  - If `value` is less than `min`, it returns `min`.
  - If `value` is greater than `max`, it returns `max`.
  - Otherwise, it returns `value` unchanged.
  - *Examples*:
    - `clamp(15, 10, 20)` is `15`
    - `clamp(5, 10, 20)` is `10`
    - `clamp(25, 10, 20)` is `20`
- **assert**: throws an error if the given condition is false. `assert(condition: boolean): void`
  - This is typically used for debugging and enforcing invariants.
  - Example: `assert(user !== null)` // Throws if user is null
- **radian to degree conversion**: converts an angle from radians to degrees. `radiansToDegrees(radians: number): number`
  - Uses the formula: $\text{degrees} = \text{radians} \frac{180}{\pi}$.
  - Example: `radiansToDegrees(Math.PI)` is `180`
- **degree to radian conversion**: converts an angle from degrees to radians. `degreesToRadians(degrees: number): number`
  - Uses the formula: $\text{radians} = \text{degrees} \frac{\pi}{180}$.
  - Example: `degreesToRadians(180)` is `3.141...`

Network

Clients (Devices and the Server)

Horizon Worlds instances run as a *distributed systems* with multiple machines involved. Each machine is called a **client**. Clients have the full scene graph, can run code, and have a Player associated with them.

There are two types of clients:

- **Player Devices**: a client associated with a human player. These clients receive player input, can run local scripts, render the world from their player's camera / eyes every frame, and synchronize data with Meta's servers. For a mobile player the device is their phone or tablet, for a PC or web-based player it is the computer and for a VR user this is their headset (or their computer if they are tethered).
- **Server**: a special client that lives on Meta's servers. It's associated player is the special server player. The server client runs all default scripts and local scripts on entities owned by the server player. The server operates just like player devices except that it skips rendering at the end of each frame.

Some built-in CodeBlockEvents can only be connected to on the server (such as OnPlayerEnterWorld) whereas others can only be connected to on a player device (such as OnPlayerEnteredFocusedInteraction).

TODO Other APIs that have client-affinity?

Simulation frame rate differences

Worlds run fixed sequence of actions each frame. The number of frames that occur per second is called the **frame rate** and is abbreviated "fps" for "frames per second". The time it takes to run each frame is the *frame time* (the time per frame).

$$\text{frame rate} = \frac{1}{\text{frame time}}$$

Clients don't all have the same frame rate! For example, the server (typically) runs at 60 frames per second and some VR headsets run at 72 frames per second. It's possible, and very likely, that **scripts execute *more frequently* on player devices than they do on the server**.

If you need to know the frame time, e.g. to run your simulations or animations, **do not rely on a specific frame rate or frame time**. Use the `deltaTime` provided by onPrePhysicsUpdate and onUpdate to get the time, in seconds, since the last frame (the last frame time).

Entity Ownership

Each entity in the world is owned by exactly one client. An entity's owner:

- **Runs local scripts**: The owning client runs the attached script on the entity (if there is one and if it is set to *local execution mode*).
- **Has scene graph authority**: The owning client is the *authority* for that entity's core attributes (such as position, visibility, and collision settings). When a client wants to modify an entity it doesn't own, it must send a message to the owning client requesting the change.

When an instance starts (or assets / sublevels spawn in) **all entities begin owned by the server** until their ownership is changed. When the owner changes, the local components attached to the entity move. When a player leaves, all entities owned by them are transferred back to the server.

Local and Default Scripts

In the Script dropdown in the desktop editor, scripts can be marked as *default* or *local* **execution mode**. All scripts are originally created with a *default* execution mode, and must be manually changed to *local* if so desired.

The *default* vs *local* terminology is a bit confusing. The execution mode setting describes what happens to a component when the entity it is attached to changes owner.

- **Default script execution mode**: all components in the file will always executes on the server, regardless of ownership of the entity they are attached to.
- **Local script execution mode**: components defined in the file will "move" to execute on the client matching the new owner of the entity (every time the owner changes).

Component execution mode: Even though execution mode is an aspect of files, we borrow the term for components, according to the execution mode of file they are defined in. So, we say *a component has default execution mode when the file it is defined in does*.

We often abbreviate the terms as: *default component*, *local component*, *default script*, and *local script*.

A script file's execution mode (local or default) affects how its components run:

1. **One execution mode per file**: All components in a script file share the file's execution mode. However:
   - A *local component* can run on any client.
   - Different components in the same *local script* can run on different clients.
   - Different instances of the same component class in a *local script* can run on different clients.
2. **"Local" means "movable"**: The term "local" means the component *can* run on player devices, not that it *must*:
   - *Local components* run on whichever client owns their entity.
   - If the server player owns the entity, its *local component* runs on the server.
3. **Ownership transfer creates new components**: *Local components* don't actually *move*. When their entity changes owner, an ownership transfer occurs:
   - The old component is disposed.
   - A new component is instantiated on the new owner.
   - The old component may pass data to the new one.
4. **Mixed execution on one entity**: Some creators have the ability to attach multiple scripts to an entity.
   - Only *local components* "move" when the entity owner changes.
   - One entity may have some components running on the server and others running on a player device.

Why Local Scripts and Ownership Matter: Network Latency

When a client modifies an entity it owns, the changes happen immediately at the end of the current frame. But when a client modifies an entity owned by a different client:

1. The change is sent over the network to the owning client (through the server)
2. The owner applies the change
3. The owner broadcasts the new state to all clients
4. Other clients receive and apply the new state

This process takes at least a few frames (or more if slow networks are involved). Using local scripts on player-owner entities can make actions feel instantaneous to the local player (with no real impact to the other players). See the example below.

> **Example**
>
> Imagine a player holding a flashlight and pressing a button to turn it on. The table below shows what happens if the script controlling the flashlight is running on the player's device vs the server. Note that **in the player-owned case, the player sees it immediately. In both cases others see it after 2 network trips**.

In the table "📡" means that a network trip occurs.

Flight Owner	Steps	When The Player Sees	When Other Players See
Player	1. Player presses button on their device 2. Device enables light 3. Light's state sent to server📡; from there it's sent to other clients📡	End of frame	2 network trips
Server	1. Player presses button on their device 2. Button press sent to server📡 3. Server enables light 4. Light's state sent to all clients📡	2 network trips	2 network trips

Authority and Reconciliation

Derived attributes depend on parents

An entity's owner is the authority on its intrinsic attributes, but there are many attributes that are *derived* from the entity's parent and ancestors. For example an entity's position is computed from its local position and it's parent's position.

When any of the simulation runtimes wants to make a change to the intrinsic state of an entity, a network message needs to be sent to the owner runtime of that entity to actually effect the change. When that owner receives change requests, it will apply them in the order received to update the state, and then broadcast the new state out to all other runtimes. Once those runtimes receive the new state, its effect will reflect in future rendering and interactions on that runtime.

Changes made to entities owned by the same runtime have no network latency

State change compression may occur if multiple changes handled at once

If there are multiple state change requests for the same state element received in a given frame, the value broadcast to other runtimes at the end of the frame will only be the *last* value processed. For instance, if you have a playing TrailFX and you stop and play it within the same frame, the 'stop' will likely not get broadcast to other runtimes and they will only see that it remains playing, and will thus not delete its old trail. If the entity is also owned by the runtime doing the state change requests, it may see intermediate states (e.g. it will stop the TrailFX, deleting its trail, and then start it again at its current location).

Ownership transfers are not instantaneous

The transfer of authoritative entity intrisnic state ownership between runtimes requires network communication between runtimes, and takes an undefined amount of time, tho usually less than 500 msec. This is a significant delay relative to the rendering frame rate, and needs to be accomodated by any game logic.

Entity intrinsic state authority is undefined during an ownership transfer

During the ownership transfer period, attempted changes to the intrinsic state of the entity will likely be lost or reverted when the transfer to the new owner completes. It is important to not try to change state on an entity after an ownership transfer has been initiated until it is clear that the new owner has acquired authority over its state, such as by sending events indicating this is the case, or by polling the owner of the entity to see it has changed to the desired target.

Ownership does not cascade to children

When you transfer ownership of an entity, the ownership is *not* automatically transferred for the children (or their descendents). If you want children to be transferred as well, you must manually transfer ownership of everything you care about.

Example

```
anEntity.owner.set(newOwner)
anEntity.children.get().forEach(c => c.owner.set(newOwner))
```

This transfers ownership of an entity and its children but not their children. Rather than just recursively transferring everything, instead consider what needs to actually be transferred (many entities states are not interacted with via scripts)!

TODO - What is the entity's relationship to the server upon instantiation?

- done, I think?
  How does the local script affect the entity?
- ???
  Explain the involvement of a manager (server objects that delegate ownership of entities that should be locally owned)
- ???
  Explain framerate(cycle speed?) changes between local and server
- done?
  Explain the relationship of local to server modules and wired references
- ???
  Link to network/codeblock events
- is this describe elsewhere? maybe I am duplicating work
  Maybe ownership cleanup tip (transfer to server on exit world during edit)
- I think covered?

Ownership Transfer

Entity ownership transfer happens through either programatic action or automatically via certain built-in behaviors.

When an ownership transfer is initiated, the current owner stops acting as an authority for the intrinsic state of the entity. The receiving owner will use the last broadcast intrinsic state prior to the ownership transfer being initiated as the intrinsic state to manage for that entity when it acknowledges its ownership authority. In the meantime, any state change requests sent by any runtimes will likely be routed to the incorrect owner and may get either lost or reverted when the new owner asserts its authority over the entity state.

- API overview of `transferOwnership` and `receiveOwnership` and `SerializableState`.
- Full-details sequencing diagrams.
- Clarify how scripts are instantiated per-owner as part of entity transfer.

Discontinuous Ownership Transfers

When ownership of an entity is transferred from one Player to another, including possibly the server player, we say that the transfer is a **continuous ownership transfer**. Whenever the "from" `Player` is unavailable we say that it is a **discontinuous ownership transfer**.

Discontinuous transfers occur when:

1. A component initializes for the first time in an instance (either because the instance started or it just spawned in).
2. A player quits the app or crashes then the player's runtime is no longer available to package up the data.

The

> ⚠ **Not handling non-'orderly' ownership transfers of 'local' execution mode Components is a frequent source of bugs**
>
> When a player runtime abruptly shuts down, there will not be any OnGrabEnd, OnAttachEnd, or OnPlayerExitWorld events delivered to the Components running in that runtime prior to the ownership transfer occurring. Unless the script has been written to have 'failsafe' behavior to return the intrinsic state of the Entity to a known default state when it is restarted in the server runtime, that Entity can become unusable due to having an inconsistent scripting state relative to its actual intrinsic state.

Automatic Ownership Transfers

There are a number of situations where an entity's ownership is changed automatically. These situations act exactly if the ownership was changed via `entity.owner.set(...)` :

1. When an entity is grabbed by or attached to a player
   - This ensures frame-accurate position updates when tracking player movement.
2. When an entity collides with another entity or player (if "preserve ownership on collision" is disable in the Property panel)
   - This makes it easy to have the collided entities act with low latency for the player from then on.
3. When a player leaves the world
   - The entities they own transfer to the server.
   - This is discontinuous since the departing owner can't participate.

> 🐞 **Exiting build preview does not automatically transfer ownership.**
>
> When in build mode, exiting from preview back to edit mode does *not* automatically transfer ownership of any Entities owned by the build player back to the server player even though the player exited.
>
> Components with *local* execution mode will continue to run in the player's device runtime. This can be confusing if they are scripted to track player avatar location, as they will start to follow the "big" build avatar around. It is best to handle `OnPlayerExitWorld` events and explicitly transfer ownership of all scripted Entities owned by the departing player back to the server player.

Transferring Data Across Owners

During continuous ownership transfers, components with local execution mode can transfer state from the old component to the new one.

In order to create a component that transfers data during an ownership transfer:

1. **Define a type** representing the data that will be transferred. The date must adhere to the SerializableState type.

   ```
   type Ammo = {count: number};
   ```

2. Pass the type into the **second generic slot when extending** `Component` .

```
class Weapon extends Component<typeof Weapon, Ammo> { ... }
```

3. Implement **`transferOwnership`** on the component to package up data that will be transferred. The `transferOwnership` method also passes in the players involved in the transfer. It's possible, and likely, for one of them to be the server player.

```
transferOwnership(from: Player, to: Player): Ammo {
  return {count: this.ammo};
}
```

4. Implement **`receiveOwnership`** on the component to use the data packaged up by the previous owner. The `receiveOwnership` method also passes in the players involved in the transfer. It's possible, and likely, for one of them to be the server player. The state will be *`null`* if the transfer is discontinuous (first awakening, or from a player that disconnected).

```
receiveOwnership(state: Ammo | null, from: Player, to: Player) {
  if (state !== null) {
    this.ammo = state.count
  }
}
```

Example

The example below shows a simple script that tracks how much ammo is in a weapon. When the gun is transferred, it maintains the same ammo count. Note that if the transfer is discontinuous the amount of ammo will be the value in the props.

```
import { Component, Player, PropTypes } from "horizon/core"

type Ammo = {count: number};

class Weapon extends Component<typeof Weapon, Ammo> {
  static propsDefinition = {
    initialAmmo: {type: PropTypes.Number, default: 20},
  };

  // props are immutable; we store ammo in a private field
  private ammo: number = 0;

  start() {
    this.ammo = this.props.initialAmmo;
  }

  receiveOwnership(state: Ammo | null, from: Player, to: Player) {
    if (state !== null) {
      this.ammo = state.count
    }
  }

  transferOwnership(from: Player, to: Player): Ammo {
    return {count: this.ammo};
  }
}
Component.register(Weapon);
```

Collision Detection

When an entity collides with another entity or player (unless "preserve ownership on collision" is enabled in the Property panel)

Colliders

Trigger Entry and Exit

- Colliding with dynamic vs static.
- Colliding with player vs entities.
- Collider gizmo.
- Can control if ownership transfer on collision (see Network!)
- collision events: need to change "Collision Events From" since the default value is `Nothing`. You need to set a `Object Tag` or you won't get any events either.

TODO: CodeBlockEvents

```
/**
 * The event that is triggered when a player collides with something.
 */
OnPlayerCollision: CodeBlockEvent<[collidedWith: Player, collisionAt: Vec3, normal: Vec3, relativeVelocity: Vec3, loca
/**
 * The event that is triggered when an entity collides with something.
 */
OnEntityCollision: CodeBlockEvent<[collidedWith: Entity, collisionAt: Vec3, normal: Vec3, relativeVelocity: Vec3, loca
```

Collidability

Mesh entities and Collider gizmos have **colliders** that are used by the physics system (for collisions, trigger detection, grabbing, avatars standing, etc).

A **collider is active** when the following true

1. Its entity has `collidable` set `true`
2. Its `parent` (and all their parents) have `collidable` set to `true`
3. It is not occluded by other colliders in the world. *Occlusion is typically from a specific direction*. Example: if you want to grab an object but it is behind a wall then the wall's collider will occlude the object (from the vantage point of the player trying to grab it).

and is otherwise ignored by the physics system. For example if the floor's collider is inactive an avatar will fall through it. If a grabbable entity's collider is inactive you cannot grab it.

> **In order for a group to be seen by the physics system it must have at least one active collider within it (however deep).**
>
> For example if all the colliders in a group are inactive then that group cannot be grabbed, it will not been seen by any triggers, it cannot be stood on, etc.

Controlling Collisions

- Turn collidable on / off
- Control can collide with players, entities, or both

Collision Events

Triggers

Trigger detection is done at the *collider* level. When a collider enters/leaves a trigger then (if it is an entity-detecting trigger) Horizon starts with the entity and traverse up the ancestor chain until it finds the first entity with a matching tag, send it the event, and then STOPS the traversal.

This means that whenever it seems both a parent and a child could get a trigger event at the same time then the child always gets it first.

Physics

TODO random notes

VelocityChange: velocity += arg
Force: velocity += arg/mass * deltaTime
Impulse: velocity += arg/mass

applyLocalForce(force, mode)
assumes force is in the coordinate system of the object

applyForceAtPosition(force, position, mode)
.Force not supported
.Impulse works
.VelocityChange not supported

Computes torque as standard r x F

DO NOT PHYSICS ON EMPTY OBJECTS: - behavior is highly inconsistent across force / torque APIs

Facts that should be ignored because

Don't run physics on empty objects; only use physical at the root OR groups
applyForce on empty object fails silently for Force/Impulse and is ok for VelChange

Empty object center-of-mass is at the bounding box center for torque / localTorque
But it is at the empty object's position for forceAtPosition

Overview

High-level framing of what Horizon is capable of. Example: there are no constraints (no hinges, springs, connecting rods, etc)

Somewhere: force vs impulse vs velocity change

Units

Name	Unit	Description
Time	Seconds when using onUpdate event	Amount of time passed since last frame
Position	Measured in Meters	The location of an entity
Velocity	Meters/Seconds	The location change over time
Acceleration	Meters/Seconds^2	Velocity change over time
Mass	Measured in Kilograms	Entity's resistance to acceleration when a force is applied
Gravity	Simply denoted as gravity	acceleration due to gravity (approximately 9.81m/s2 on Earth)
Weight	Newtons or (Mass * Gravity)	The the force needed to accelerate one kilogram of mass by one meter per second squared.

Angular force

Name	Unit	Description
Angle	Degrees in Properties panel	Amount of rotation

Torque

> **Scripting torque allows for the use of radians**
>
> See rotational force API

TODO - Shouldn't the in-between frames be ignored? In other words, how is it doing FixedUpdate?

Creating a Physical Entity

For an entity to become a physical entity:

1. Have an active collider
2. Set `Motion` to `Interactive`
3. Set `Interaction` to `Physics` or `Both`
4. All ancestors, if any, are Meshes and Empty Objects with Motion set to None.

Force Properties	Description
Gravity	
Mass	
Drag	
Angular Drag	
Dynamic Friction	
Static Friction	
Bounciness	
Center-of-Mass	

> **Set the mass of a physical entity when first creating a physical entity**
>
> This will ensure that physics calculations with other entities work as expected when you start experimenting with other physical properties and functions.

TODO - Collision type: discrete, continuous - figure out horizon way(Any guarantees?)
TODO - Average?, min?, max? - friction and bounciness calculation (Any guarantees?)

PrePhysics vs OnUpdate Events

Simulated vs Locked Entities

PhysicalEntity Class

```
export declare class PhysicalEntity extends Entity {
    /**
     * Gets a string representation of the entity.
     * @returns The human readable string representation of this entity.
     */
    toString(): string;
    /**
     * Whether the entity has a gravity effect on it.
     * If `true`, gravity has an effect, otherwise gravity does not have an effect.
     */
    gravityEnabled: WritableHorizonProperty<boolean>;
    /**
     * `true` if the physics system is blocked from interacting with the entity; `false` otherwise.
     */
    locked: HorizonProperty<boolean>;
    /**
     * The velocity of an object in world space, in meters per second.
     */
    velocity: ReadableHorizonProperty<Vec3>;
    /**
     * The angular velocity of an object in world space.
     */
    angularVelocity: ReadableHorizonProperty<Vec3>;
    /**
     * Applies a force at a world space point. Adds to the current velocity.
     * @param vector - The force vector.
     * @param mode - The amount of force to apply.
     */
    applyForce(vector: Vec3, mode: PhysicsForceMode): void;
    /**
     * Applies a local force at a world space point. Adds to the current velocity.
     * @param vector - The force vector.
     * @param mode - The amount of force to apply.
     */
    applyLocalForce(vector: Vec3, mode: PhysicsForceMode): void;
    /**
     * Applies a force at a world space point using a specified position as the center of force.
     * @param vector - The force vector.
     * @param position - The position of the center of the force vector.
     * @param mode - The amount of force to apply.
     */
    applyForceAtPosition(vector: Vec3, position: Vec3, mode: PhysicsForceMode): void;
    /**
     * Applies torque to the entity.
     * @param vector - The force vector.
     */
    applyTorque(vector: Vec3): void;
    /**
     * Applies a local torque to the entity.
     * @param vector - The force vector.
     */
```

```
    applyLocalTorque(vector: Vec3): void;
    /**
     * Sets the velocity of an entity to zero.
     */
    zeroVelocity(): void;
    /**
     * Pushes a physical entity toward a target position as if it's attached to a spring.
     * This should be called every frame and requires the physical entity's motion type to be interactive.
     *
     * @param position - The target position, or 'origin' of the spring
     * @param options - Additional optional arguments to control the spring's behavior.
     *
     * @example
     * ```
     * var physEnt = this.props.obj1.as(PhysicalEntity);
     * this.connectLocalBroadcastEvent(World.onUpdate, (data: { deltaTime: number }) => {
     *  physEnt.springPushTowardPosition(this.props.obj2.position.get(), {stiffness: 5, damping: 0.2});
     * })
     * ```
     */
    springPushTowardPosition(position: Vec3, options?: Partial<SpringOptions>): void;
    /**
     * Spins a physical entity toward a target rotation as if it's attached to a spring.
     * This should be called every frame and requires the physical entity's motion type to be interactive.
     *
     * @param rotation - The target quaternion rotation.
     * @param options - Additional optional arguments to control the spring's behavior.
     *
     * @example
     * ```
     * var physEnt = this.props.obj1.as(PhysicalEntity);
     * this.connectLocalBroadcastEvent(World.onUpdate, (data: { deltaTime: number }) => {
     *  physEnt.springSpinTowardRotation(this.props.obj2.rotation.get(), {stiffness: 10, damping: 0.5, axisIndependent:
     * })
     * ```
     */
    springSpinTowardRotation(rotation: Quaternion, options?: Partial<SpringOptions>): void;
}
```

Projectiles

Applying Forces and Torque

Player Physics

Setting player position (locally) in pre-physics results in that position being used during Physics in the same frame. If not local you are waiting on a network send. In that frame's physics phase the position may then further be updated. If you update a position of a player (locally) in PrePhysics then that position will be reported for the rest of the frame (even though there is a new physics-based position, which will start being reported at the start of the next frame).

Player positions are committed to the scene graph after prePhysics (and used in physics), onUpdate, codeBlockEvent (and likely not after network events)
NOT async

When set position of player (locally) in async: the value is used in that frame's physics calculation to get a new physics value is not seen until prePhysics of the next frame; in the meantime, the new (scene graph position) that you just is seen the rest of the frame.

Note: player position refers to the location in the world of the "center of the hips" (it is the hip joint in the skeleton)

Setting a player's position will require a network trip from server to player since player's are authoritative over their own position and pose.

Springs

Spring physics allows entities to move and rotate as if they were attached to a spring. This system provides smooth, natural motion that can be adjusted using stiffness and damping parameters. The spring helper methods are **intended to be called every frame**.

Spring Options: spring behavior is controlled through the `SpringOptions` type, which defines key parameters for spring-based movement.

```
type SpringOptions = {
  stiffness: number;
  damping: number;
  axisIndependent: boolean;
};
```

- `stiffness` : The stiffness of the spring, which controls the amount of force applied to the object. Higher values represent a spring that "pulls harder".
- `damping` : The damping ratio of the spring, which reduces oscillation and prevents excessive bouncing. Higher values reduce in a faster "loss of energy".
- `axisIndependent` : If `true` , the object's motion is parallel to the push direction; if `false` , rotation and movement may interact.

Default Spring Options: if no options are provided, the following defaults are used.

```
const DefaultSpringOptions: SpringOptions = {
  stiffness: 2,
  damping: 0.5,
  axisIndependent: true
};
```

These values are intended to provide a balanced spring motion that feels natural without excessive oscillation.

Spring Push

`springPushTowardPosition` pushes an entity toward a target position as if attached to (and pulled by) a spring. This is intended to be called every frame. The entity must have **Motion=Interactive** and should have (TODO: verify) **simulated=true**.

```
// PhysicalEntity
springPushTowardPosition(position: Vec3, options?: Partial<SpringOptions>): void;
```

- `position` : the target position, which acts as the origin of the spring force.
- `options` (optional): overrides the spring behavior (stiffness, damping, and axis independence).

Example

```
const physEnt = this.props.obj1.as(PhysicalEntity);
this.connectLocalBroadcastEvent(World.onUpdate, (data: { deltaTime: number }) => {
  physEnt.springPushTowardPosition(this.props.obj2.position.get(), {stiffness: 5, damping: 0.2});
});
```

Spring Spin

`springSpinTowardRotation` rotates an entity toward a target rotation as if attached to a spring. This is intended to be called every frame. The entity must have **Motion=Interactive** and should have (TODO: verify) **simulated=true**.

```
// PhysicalEntity
springSpinTowardRotation(rotation: Quaternion, options?: Partial<SpringOptions>): void;
```

- `rotation` : the target rotation, represented as a quaternion.
- `options` (optional): overrides the spring behavior.

Example

```
const physEnt = this.props.obj1.as(PhysicalEntity);
this.connectLocalBroadcastEvent(World.onUpdate, (data: { deltaTime: number }) => {
  physEnt.springSpinTowardRotation(this.props.obj2.rotation.get(), {stiffness: 10, damping: 0.5, axisIndependent:
});
```

Players

TODO - Velocity, locomotion speed, jump speed, player device type

The `Player` class represents a person in the instance, an NPC in the instance, or the "omnipotent player" (the server).

Max player count: Each world has a maximum player count that controls the maximum number of players allowed in each instance. The count is configured in world settings.

Construction / New: `Player` instances are allocated by the system; you should *never allocate them directly* (never use `new Player`).

Equality Comparison: `Player` instances can be compared referentially `aPlayer === bPlayer` which is the same as `aPlayer.id === bPlayer.id`.

Server Player: There is a special "Server `Player`" instance that represents the server. It's primary use is in checking or setting which player "owns" an entity (it's the "server player" if none of the human players do). The server player does not count against the maximum player count being reached.

Id and Index: Each `Player` has an `id` and an `index` which serve different purposes (see below). From a `Player` instance you can access `PlayerBodyBart`s, e.g. `aPlayer.leftHand` or get their name `aPlayer.name.get()`. There are many `CodeBlockEvents` associated with players (such as entering/exiting a world, grabbing entities, and much). All aspects of players are described in detail in the next sections.

Identifying Players

Players in Horizon all have a global "account id". There is no way to access this id directly, although Horizon uses it under the hood for persistence (player variables, leaderboards, and quests). Within an instance players can be referenced by the `id` or the `index` they are assigned on entry. Player `index`es are reused when players leave; `id`s are not.

Player ID

Each `Player` instance has a `readonly id: number` property.

> **Entering an instance assigns a new ID (for that instance).**
>
> When a person enters an instance they are assigned an `id` that has not yet been used in that instance. If a player switches devices, or leave the instance and later return, they will be given a new `id`.

> **IDs are per-instance. Do not persist them.**
>
> The `id` that a player gets in one instance of a world has nothing to do with the `id` they might get in another instance. If a person gets assigned `id` 42 in one instance then the moment they leave that instance you should no longer associate them with the `id`.

> **IDs should be used rarely.**
>
> Since you can compare two `Player` instances directly with `===` and `!==` there is little reason to use the `id` property. You can even use `Player` instances as keys in a `Map`. If you have a reason to use the `id` field, be mindful that the association between a person and their `id` only exists until they leave that instance.

Player Indices

When a player (human or NPC) enters a world they are also assigned an `index`. The `index` will be a number between `0` and `n-1`, where `n` is the maximum number of players allowed in an instance. When a player enters an instance they are assigned an `index` value that is not currently used by any other player. When they leave that value becomes available again.

For example: if three players arrive in an instance they may be assigned `index` values of `0`, `1`, and `2`. If they player with `index` `1` leaves then the next player that arrives may get index `1` again.

You can read a player's index with

```
player.index.get()
```

and use

```
world.getPlayerFromIndex(index) // Player | null
```

to find out which player currently, if any, has a given index.

> **Do not rely on the order indices are assigned**
>
> There are no guarantees that a player gets the *smallest* available `index`. Any available value maybe be assigned to a new player.

> **Example: per-player entities**
>
> A common use of `index`es is managing per-player entities. For instance, if you want every player to have a shield when they spawn in. Then you could have an array of shield `Entity`s and when a player enters the world, assign them the shield from that array that matches their `index`.

Listing All Players

The `World` class has the method:

```
getPlayers() : Player[]
```

which returns the current list of players in the world (human and NPC, but does not include the server player). Note that the order of this array should not be relied upon. The order may change between calls and there is no relation to the `index` property described above.

Server Player

There is a special instance of the `Player` class that represents the *server*. It has an `id` but no meaningful `index`. All `Player` APIs work for the server player, but return default values (example: the location will return the origin; name will return the empty string).

The **server player does not count as one of the human player**s:

- it does not get assigned an `index`
- it does not count toward the maximum player count being reached
- it is not included in the getPlayer() array

The server player owns all entities when the world starts (or when entities are spawned in).

The `World` class has the method

```
getServerPlayer(): Player
```

which can be used to access the server player. The primary use cases are:

1. Transferring ownership to the server

```
entity.owner.set(world.getServerPlayer());
```

2. Checking if an entity is owned by the server

```
if (entity.owner.get() === world.getServerPlayer()) { ... }
```

3. Checking if a script is running locally

```
if (world.getLocalPlayer() === world.getServerPlayer()) { ... }
```

Local Player

Every script is run on an execution client associated with a `Player` (see Network for more info). If the script is set to *default* mode, then it is always running on the server. If the script is set to *local* then is can be transferred to and from the servers and the local devices of players.

If a script is running locally on a human-player's device then that player is the *local player* for that script. If the script is running on the server then the *server player* is the *local player* for that script.

The `World` class has the method

```
getLocalPlayer() : Player
```

for determining which `Player` 's device the current script is running one. This method with return a human-player in the world or the *server player*.

Player Entering and Exiting a World

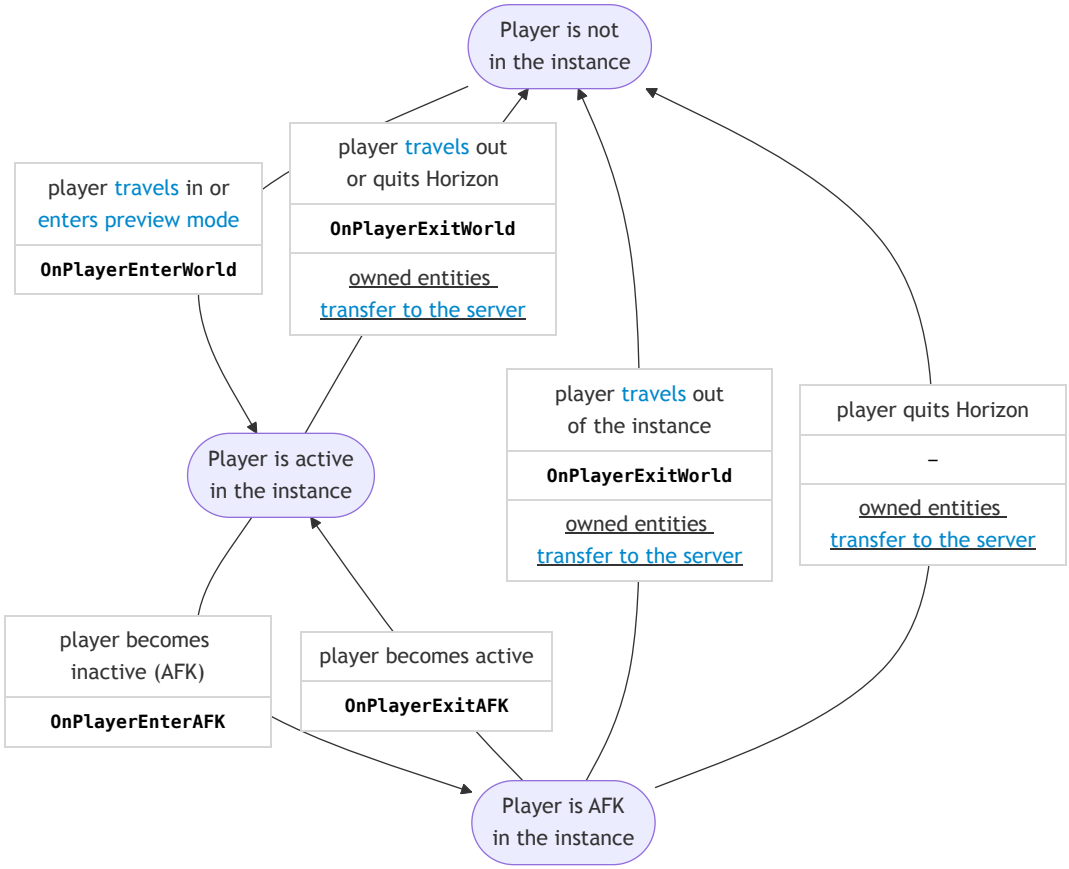
When a player (human or NPC) enters an instance they are assigned a player id and a player index. The built-in CodeBlockEvent `OnPlayerEnterWorld` is then sent to all component instances that have registered to receive to it. Likewise `OnPlayerEnterWorld` is sent when a player leaves the instance.

Both events in the table below are 🔈 server-broadcast CodeBlockEvents; you can connect to any server-owned entity to receive them.

Built-In CodeBlockEvent	Parameter(s)	Description
🔈 `OnPlayerEnterWorld`	`player: Player`	Sent when a player enters the instance. This occurs when a **player travels to the instance**; it also happens when a player goes from **edit mode to preview mode** in the editor. The player is already in getPlayers() when this event is sent.
🔈 `OnPlayerExitWorld`	`player: Player`	Sent when a player exits the instance. This occurs when a **player travels away from the instance** or quits Horizon Worlds; it also happens when a player goes from **preview mode to edit mode** in the editor. The player is no longer in getPlayers() when this event is sent (unless they are in build mode; then they remain in the array).

When a player exits the world, all entities owned by them are transferred to the server.

The flow of events are shown in the diagram below. Ovals represent the *state* the entity is in. The boxes represent what happens when the entity goes from one state to another; in the box, *italics text is the action* that caused the change, **bold text is built-in CodeBlockEvents** that are sent (in the order top-to-bottom if there are multiple in a box), and underlined text is a system action that occurs.



Entities are not transferred to the server when leaving preview mode.

When a player exits the instance all the entities they own are automatically transferred to the server player. However this does not happen when going from preview to edit mode (when in an editor instance). This can result in unusual behavior where entities continue to react to a player that is in build mode. To avoid this, listen to the `OnPlayerExitWorld` event and assign entities back to the server player.

`OnPlayerEnterWorld` and `OnPlayerExitWorld` are sent to only server-owned entities.

If an entity is owned by a player then the two code blocks above *are not* sent to it. Any component connected to receive those events from that entity will not get them.

In build mode, `OnPlayerEnterWorld` can occur twice in succession for one player id.

In published mode, `OnPlayerEnterWorld` occurs only once per player id. In build mode, a player on the desktop editor triggers `OnPlayerEnterWorld` twice when they enter Preview mode from a stopped instance. This means that if you're tracking a list of all players using `OnPlayerEnterWorld`, add the new player to a set or dictionary instead of an array.

Player Enter and Exit AFK

A player in an instance can become **inactive**. Horizon calls this inactive state: **AFK** (standing for Away From Keyboard). The exact rules for inactivity are not documented and are subject to change. Roughly speaking:

Becoming inactive (AFK): A mobile player becomes inactive when they go for a while without touching the screen or when they temporarily switch to a different app. A VR player goes inactive when they take off their headset (or even raise it to their forehead) or when they open the Quest OS menu while in the app.

Becoming active (no longer AFK): A mobile player becomes active when they foreground the app and begin touching the screen. A VR player becomes active when they put their headset back on or close the OS menu.

There are two built-in code block events associated with inactivity / AFK. Both are 🔈 server-broadcast CodeBlockEvents; you can connect to any server-owned entity to receive them.

Built-In CodeBlockEvents	Parameter(s)	Description
🔈 `OnPlayerEnterAFK`	`player: Player`	Sent when a player becomes inactive.
🔈 `OnPlayerExitAFK`	`player: Player`	Sent when a player is no longer inactive.

See the diagram in the player enter / exit section for a detailed overview of when the above two events are sent.

> ⚠ **`OnPlayerEnterAFK` and `OnPlayerExitAFK` are sent to only server-owned entities.**
>
> If an entity is owned by a player then the two code blocks above *are not* sent to it. Any component connected to receive those events from that entity will not get them.
> TODO if a local entity connects to a server owned one, is it forward these 2 events?

> 🐞 **If a player kills the app after going AFK, `OnPlayerExitWorld` is not triggered.**
>
> In complex worlds it is a good habit to regularly check getPlayers() to see which players are present and handle cleanup for any that left but were missed.

Pose (Position and Body Parts)

The Player class has properties for `position` and `rotation`. These are Horizon properties and so you must call `get()` (e.g. `player.position.get()`). The `position` properties returns the world location of the player's center point (which is near the middle of their hips).

There are additional properties for reading the positions and rotations of the head, torso, feet, left hand, and right hand.

Player Body Parts

A player has a number of properties for accessing body parts: `head`, `torso`, `foot`, `leftHand`, and `rightHand`; each return an instance of the class `PlayerBodyPart` (or the more specific `PlayerHand`). They are Horizon properties and so you must use `get()`:

```
const torso = player.torso.get()
```

The `foot` body part is an "abstract" location in between the two feet (directly below the avatar center point near the hips).

Each body part has a has the standard global transform properties: position, rotation, and scale as well as local versions: `localPosition`, `localRotation`, and `localScale`. There is also `forward` and `up`.

Additionally you can use `bodyPart.player` to identify which player the part belongs to and `bodyPart.type` to identify which part of the body it is (e.g. `player.leftHand.get().type` returns `PlayerBodyPartType.LeftHand`).

Body parts have two helper methods: `getPosition` and `getRotation` that let you conditionally pass in an instance of the `Space` enum:

- `bodyPart.getPosition(Space.World)` is the same as `bodyPart.position.get()`.
- `bodyPart.getPosition(Space.Local)` is the same as `bodyPart.localPosition.get()`.

> **Getting a body part's local *right* vector.**
>
> Unlike for entities, there is no built-in `right` property to get the local position x-axis direction. You can compute it yourself with:
>
> ```
> const torso = player.torso.get()
> const torsoUp = torso.up.get()
> const torsoForward = torso.forward.get()
>
> const torsoRight = torsoUp.cross(torsoForward)
> ```
>
> We did "up cross forward" because Horizon is left-handed; "forward cross up" gives the local *left* axis instead.

Player Hand

`PlayerHand` is a subclass of PlayerBodyPart, thus inheriting all of the behaviors and properties outlined above.

`PlayerHand` also has a property `handedness`, returning either `Handedness.Left` or `Handedness.Right`.

Additionally, `PlayerHand` has the method `playHaptics` which is used to make a VR player's controllers vibrate.

VOIP Settings

Horizon has the ability control *who can hear a player and from how far away*.

It call this the **VOIP Setting** which can be configured with a number of values: **mute, whisper, nearby, *default*, extended, global** which representing increases ranges of being heard. Mute means that no one can hear the player, global means that everyone can hear the player (and with full volume). The other values represent a spectrum in between mute and global, with **default** being the recommended setting for most experiences (people in your general vicinity can hear and so can people farther away if you are loud). There is one more special VOIP Setting **environment**, which is described below.

> **There are no team-based voip settings (there are no "voice channels").**
>
> There is (currently) no way to configure voip settings between two specific players or to configure voice channels. Specifically you can't make a team game where a whole hears each other globally but the other team hears them whisper. When a player is set to global or whisper, **the setting controls how everyone else hears them**.

The **environment gizmo** allows you to set a VOIP Setting for the whole world. All players will be assigned this value upon joining.

You can change the VOIP setting for a given player in TypeScript with:

```
player.setVoipSetting(VoipSetting.Whisper)
```

and at any point in time you can "cancel" that custom setting and return the player back to whatever setting is on the environment gizmo via:

```
player.setVoipSetting(VoipSetting.Environment)
```

> **Spawning Environment Gizmos**
>
> when a new environment gizmo is spawned, all players will be updated to have its VOIP setting. If the spawned gizmo is set to `Environment` then all players will be returned to the setting in the last active gizmo.

World's Player Settings' VOIP Setting: there is a top-level setting in *Player Settings* called *VOIP Settings* that can be set to `Global` and `Local`. When set to `Global` every player has **global** as their setting, it is not possible to change any VOIP settings further (all gizmos and TypeScript related to VOIP are ignored). The `Local` setting gives the world the **default** setting, which can then be further changed by environment gizmos and TypeScript.

> **Never use the World's Player Settings' `Global` VOIP Setting.**
>
> The World's Player Settings' VOIP Settings toggle has bugs. We recommend that you **set it to `Local`** (or just never touch it after creating a new world).

Haptics

A VR player's controllers can be made to vibrate to add immersion to an experience. There is currently no way to vibrate a mobile device.

To vibrate a VR player's controllers, choose a player hand and then call the `playHaptics` method on it with a duration (in seconds), a strength (via the `HapticsStrength` enum), and a sharpness (via the `HapticsSharpness` enum). For example:

```
player.leftHand.playHaptics(0.5, HapticStrength.Medium, HapticSharpness.Sharp)
```

Strength: the overall intensity of the vibration. For example, imagine a constant unchanging vibration; that vibration should occur at different levels of "intensity" or "volume". This is the *strength* of the haptic effect.

Sharpness: the texture or "shape" of the vibration. It's about if the effect ramps up slowly or starts immediately at full strength and if it's a fast vibration vs a slower "pulsing".

> **Analogy: think of strength like "volume" and sharpness like "melody".**

The supported values for haptics strength are:

`HapticsStrength`	Meaning
`VeryLight`	A barely noticeable vibration, just enough to create a faint tactile response without drawing attention.
`Light`	A subtle vibration, likely intended for gentle feedback, such as indicating a soft touch or a minor interaction.
`Medium`	A more noticeable vibration, suitable for standard feedback like button presses or interactions that require a stronger confirmation.
`Strong`	A powerful vibration, used for significant events, such as impacts, collisions, or urgent notifications.

The supported values for haptics sharpness are:

`HapticsSharpness`	Meaning
`Sharp`	A high-frequency vibration with a quick onset and quick decay. Ideal for rapid alerts or precision interactions.
`Coarse`	A moderate vibration with a slightly longer duration, striking a balance between sharp and soft. Ideal for interactions that need to feel distinct without being abrupt.
`Soft`	A low-frequency, smooth vibration that builds and fades gradually, creating a more diffuse, gentle sensation. Ideal for subtle cues or immersive environmental effects.

Throwing

```
export declare type ThrowOptions = {
    speed?: number | null;
    pitch?: number | null;
    yaw?: number | null;
    playThrowAnimation?: boolean | null;
    hand?: Handedness | null;
};

export declare const DefaultThrowOptions: ThrowOptions;

// Player
throwHeldItem(options?: Partial<ThrowOptions>): void;
```

Grabbing and Holding Entities

TODO overview.
TODO actions (onscreen buttons) and inputs.
TODO explain that there is no current want to know who is holding something or if it is held (other than to track it yourself).

Creating a Grabbable Entity

Select an entity and then in the Properties panel set its `Motion` to `Interactive` and `Interaction` to `Grabbable` or `Both`. The entity *must* be a root entity or it will not actually be allowed to be grabbed. Ensure that `collidable` is `true` and that (if it is a group) there is an active collider within it.

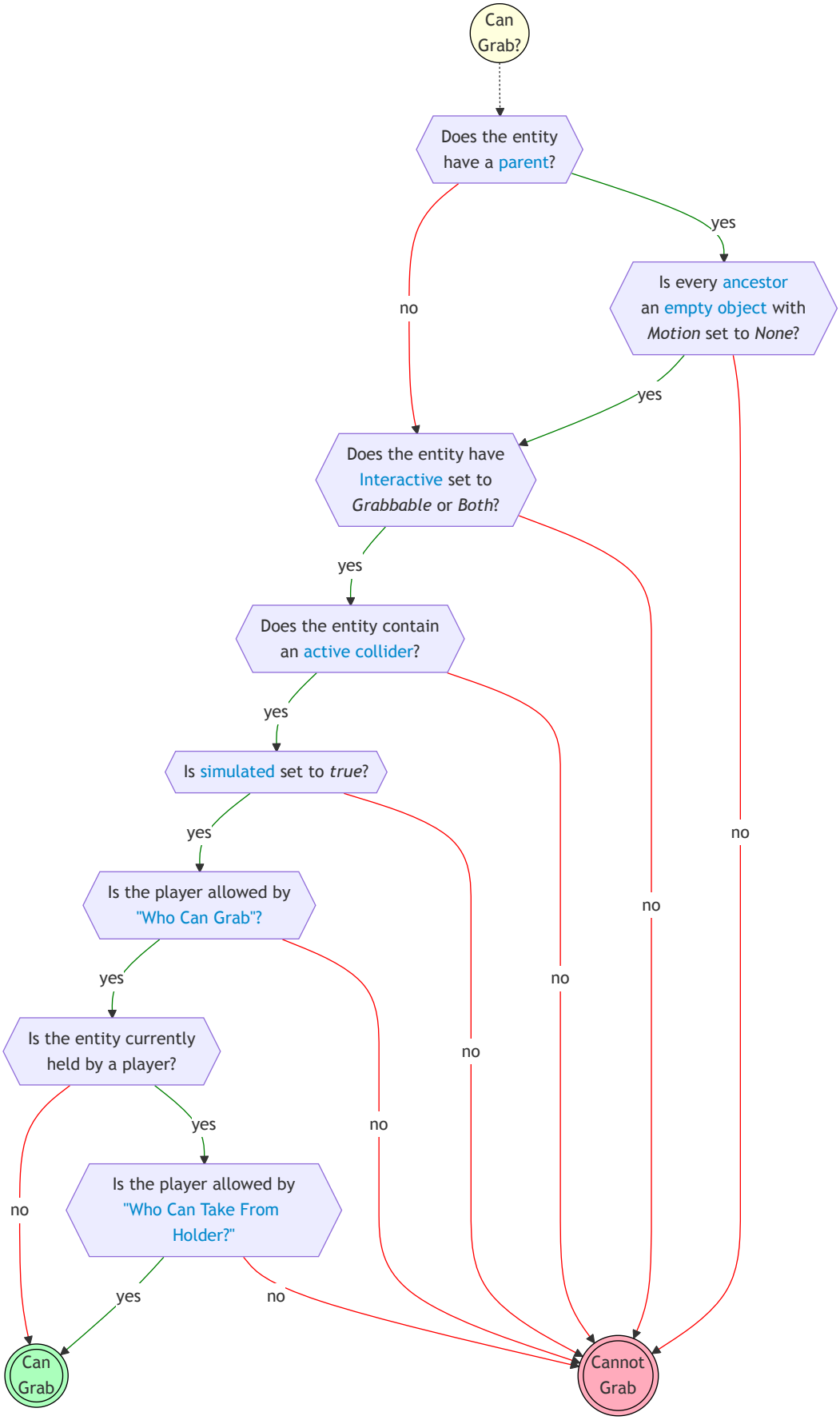
> **Grabbables cannot be inside dynamic objects**
>
> A grabbable entity must be a root entity (it can only have Static Objects in its ancestor chain).

> **Entities must be collidable to be grabbed!**
>
> If a grabbable entity is not `collidable` then it cannot be grabbed. If it is a group and none of the colliders within it are active then it cannot be grabbed, even if the root is collidable!

Can Grab

For an entity to be grabbable it needs:

1. To be a grabbable entity
    i. `Motion` to be `Interactive`
    ii. `Interaction` to be `Grabbable` or `Both`
    iii. All ancestors, if any, are Meshes and Empty Objects with Motion set to None.
2. To be currently grabbable
    i. `simulated` set to `true`
    ii. At least one active collider within it (which is not occluded from the perspective of the player)
3. To be grabbable by this player
    i. Match the rules of "Who Can Grab"
    ii. If it is currently held, match the rules of "Who Can Take From Holder"


Can Grab?
Does the entity have a parent?
yes
no
Is every ancestor an empty object with *Motion* set to *None*?
yes
no
Does the entity have Interactive set to *Grabbable* or *Both*?
yes
no
Does the entity contain an active collider?
yes
no
Is simulated set to *true*?
yes
no
Is the player allowed by "Who Can Grab"?
yes
no
Is the entity currently held by a player?
yes
no
Is the player allowed by "Who Can Take From Holder?"
yes
no
no
Can Grab
Cannot Grab

Entities with grab anchors can be grabbed even when collidable is set to false.

There is currently a bug where when an entity has a grab anchor it can still be grabbed even when collidable is set to false. If you want to make an entity, with a grab anchor, "disappear" you should move it far away (instead of just setting visibility and collidability to false).

TODO does grab anchor override any other rules?

Setting "Who Can Grab?"

`Interactive` entities have a setting in the Properties panel called "Who Can Grab?" with the following options controlling who can grab the entity.

Setting	Behavior
Anyone	Any player is eligible to grab the entity.
First To Grab Only	If an entity has never been grabbed then any player can grab it. Once a player grabs it, only that player can re-grab it until they exit the world instance. Then anyone can grab the entity, and only next player to grab it can re-grab it until they exit the instance, and so on.
Script Assignee(s)	A player is only eligible to grab the entity if they are in the list of allowed players assigned with `entity.setWhoCanGrab(listOfPlayers)`.

When the **Who Can Grab** setting is set to **Script Assignee(s)**, no one can grab the entity until `setWhoCanGrab` is called with an array of some players. You can pass an empty array to make an entity not grabbable.

When the **Who Can Grab** setting is *not* **Script Assignee(s)**, the `setWhoCanGrab` method does nothing when called.

First To Grab Only can cause an entity to be grabbable by no one, even after the player is no longer in the world.

If a player kills the app after going AFK, OnPlayerExitWorld is not triggered. When that happens, the entity will be ungrabbable unless that player re-enters the same world instance, thereby triggering OnPlayerExitWorld on that player and releasing held and attached entities. Our recommendation is to not use **First to Grab Only** because there would be no way to reset who can grab using scripts. Instead, set **Can Grab** to **Anyone**, or to **Script Assignee(s)** and `forceRelease` any held entity when a player is AFK.

`setWhoCanGrab` does not auto-update.

There is no way to have it auto-update when new players join the instance (example: everyone except one player can grab the entity). If you want to include a newly-joined player in the list then you must call the API again.

Setting "Who Can Take From Holder?"

`Interactive` entities have a setting in the Properties panel called "Who Can Taken From Holder?" with the following options controlling what can happen to the entity while it is held.

Setting	Can the holder grab it out of their own hand using their other hand?	Can another player take it from the player that is holding it?
No One	No	No

Setting	Can the holder grab it out of their own hand using their other hand?	Can another player take it from the player that is holding it?
Only You	Yes	No
Anyone	Yes	Yes (*if* the person can grab the entity)

Grab Distance

> **Grab distance varies between platforms.**
>
> For example mobile players can grab entities when much farther away than VR players

> **Grab-distance cannot be configured.**
>
> You cannot explicitly control from how far away an entity can be grabbed; however you can use a trigger to control grabbability (for example: make an entity grabbable by a specific play when they are in that trigger).

Grabbing Entities

When a VR player grabs an entity is stays grabbed until they release the trigger. The entity is only held as long as they are holding the entity.

A screen-based player uses an onscreen button to grab and then (later) a different onscreen button to release.

When a player grabs an entity, ownership is transferred to that player.

Grab Lock

When an entity is grabbable there is a setting its Properties called `Grab Lock`. When it is enabled a VR player no longer needs to keep the trigger (on their VR controller) pressed to hold the entity (which gets tiring after a while!). When `Grab lock` is enabled a VR player presses (and releases) the trigger to grab. When they release the trigger the entity *stays held*. When they later again press and release the trigger again, the entity is released.

Force Holding

An entity can be forced into the hand of a player used the TypeScript API:

```
// GrabbableEntity
forceHold(player: Player, hand: Handedness, allowRelease: boolean): void;
```

It allows you to specify which player to have hold it, which hand they should hold it in, and whether or not that can *manually* release it. If `allowRelease` is `false` then the entity can only be released by force release or by distance-based release. When `allowRelease` is set to `true` a VR player can release the entity by pressing the trigger on their VR controller; a screen-based player can release it using the onscreen release button.

> **Giving players a weapon when the game starts**
>
> A common use case for force-holding is a game where every player has a sword, for example. When the round starts, you given all players a weapon by force-holding it. If you don't want them to let go then set `allowRelease` to `false`. Then you can force

release the entities at the end of the game.

> **A force-held item can be released "accidentally"**
>
> Even if an entity is force-held with `allowRelease` set to `false`, it is possible for the entity to be released by distance-based release. If you want to ensure that players are always holding an entity during a game, then you should listen for the grab-release event and have the player force-hold the entity again.

Releasing Entities

Manual release

If an entity was manually grabbed or it was force-held with `allowRelease` set to `true`, then a player can manually release it. If an entity was force-held with `allowRelease` set to `false` then a player will not be able to manually release the entity and instead must wait on it (eventually) being done for them.

Force release

A held entity can be forced out of a player's hand at any time by calling

```
entity.forceRelease();
```

on the held object. If the entity was **force held** then this is how you remove the entity from their hand.

> **Some actions automatically force release.**
>
> There are a number of ways in which a grabbable entity can be "automatically" force released:
>
> 1. **`Simulated` is set to `false`** - the entity is force released and then remains ungrabbable until `simulated` is set to `true` again.
> 2. **Entity is attached.** When an entity is attached to a player it is forced released (after attaching to the player, meaning that it is momentarily held *and* attached at the same time).
> 3. **Entity moves too far away from player** - either via scripting, animation, or physics "knocking it out of the hand".
> 4. **Player moves too far away entity** - either via scripting, physics, or player movement input "walking away while grabbing physics locked object".
> 5. (Not recommended) **Ownership is changed while held** - changing the owner of a held entity will cause it to be force-release from the player. The `grabEnd` event will *not* be sent in this case.
>
> > **Disabling collidability does *not* cause a force release.**

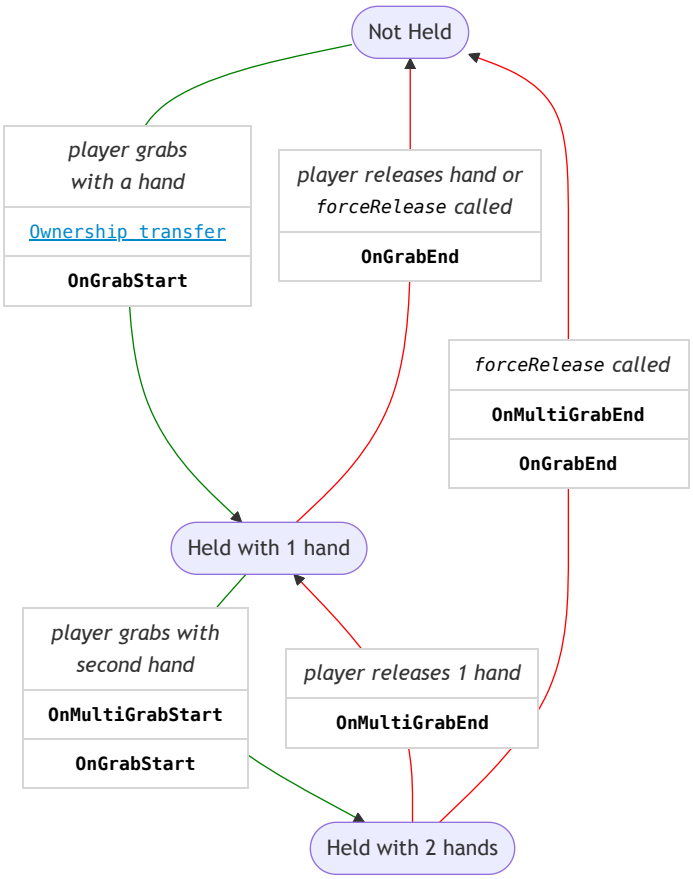
> **Despawning a held object does not send a grab release event!**
>
> This is a bug that may be fixed in the future. Be mindful of despawning assets that contain grabbable entities (you may need to clean up manually).

Grab Sequence and Events

There are a number of events associated with grabbing and holding. The diagram below shows how the state of an entity changes with user-actions (highlighted in blue). Actions have associated `CodeBlockEvent`s that are sent. If a box contains multiple events then they are sent in the top-down order shown.

Built-In CodeBlockEvents	Parameter(s)	Description
`OnGrabStart`	`isRightHand: boolean` `player: Player`	Sent when a player grabs an entity (it is sent *both* for the first hand grabbing *and* the second hand grabbing in a *multi-grab*).
`OnGrabEnd`	`player: Player`	Sent when a player stops holding the entity (both hands off, for a multi-grab).
`OnMultiGrabStart`	`player: Player`	Sent when a player adds their second hand to a multi-grab entity.
`OnMultiGrabEnd`	`player: Player`	Sent when a multi-grab entity is no longer held with 2 hands (either because it is now held by 1 or by none).

The flow of events are shown in the diagram below. Ovals represent the *state* the entity is in. The boxes represent what happens when the entity goes from one state to another; in the box, *italics text is the action* that caused the change, **bold text is built-in CodeBlockEvents** that are sent (in the order top-to-bottom if there are multiple in a box), and underlined text is a system action that occurs.



Hand-off (Switching Hands or Players)

When an entity is transferred from one hand to another or from one player to another then the entity is *fully released* by the first player before being grabbed by the second player. This means there is a moment where the entity is held by no one. An entity is never held by 2

players (not even momentarily); and if it is not a multi-grab entity then it is never held by 2 hands (not even momentarily).

> ⚠ **`OnGrabEnd` is sent during a "hand-off".**
>
> The `OnGrabEnd` event may mean that an entity is about to grabbed by a different hand or player.

Moving Held Entities

Normally the position and rotation of a held object is determined by the position and orientation of the player hand that is holding it (during the physics stage of the frame).

It is some times useful to invert that and instead have **the position and rotation of the held entity influence the position and rotation of the hand that is holding it**.

This can be achieved due to the fact that

```
player.leftHand.position.get()
```

returns where the *player's hand is **supposed** to be*, but not where the *avatar's* hand is. That means that you can move a held entity, which will move the avatar hand holding it, but can still check where the hand is supposed to be (if you hadn't moved it).

There are two approaches for moving a held entity:

Moving a Held Entity Locally in Relation to the Hand

In a gun-recoil animation you want the player hand to be able to move freely, yet have the gun apply an additional local rotation "on top of it". If you set the position / rotation of the entity when a user takes an action (such as firing the gun) then that change will only last for one frame (which might be ok for a quick recoil effect) because the entity's position / rotation will be immediately updated the next frame from the avatar's hand.

If you want a multi-frame or ongoing effect then you need to set the position / rotation of the entity repeatedly in an OnUpdate handler. In summary: **every frame in which you want the entity change from where the avatar want it, you must set it yourself**.

Moving a Held Entity Globally in Relation to the World

When building a lever, for example, you want the avatar hand to "lock onto" the lever. In this case you want to completely control the position of the avatar hand. To do this, set `locked` to `true` on the grabbable entity. This will prevent the entity from being moved by physics or by the avatar. Then you can move the entity by setting its `position` and `rotation`. The avatar hand will then be moved to match.

In this lever example, you could get `player.leftHand.position.get()` every frame to identify where the avatar's hand is *supposed* to be, constrain that position to a "valid position" and then rotate the level according. This is an advanced use case that likely requires trigonometry.

Note that if the grabbed entity gets too far away from the avatar hand you will get a force release.

Here is a simple example of a grabbable entity that is constrained to move along the y-axis (you can only move it up and down).

```
import {CodeBlockEvents, Component, PhysicalEntity, Player, World} from 'horizon/core'

class AxisYConstrainedGrabbable extends Component<typeof AxisYConstrainedGrabbable> {
  private grabInfo?: {isRightHand: boolean, player: Player}

  override start() {}

  override preStart() {
    // Lock the entity so it can't be moved by an avatar hand or by physics
    this.entity.as(PhysicalEntity).locked.set(true)

    // Record which player and which hand grab
    this.connectCodeBlockEvent(this.entity, CodeBlockEvents.OnGrabStart, (isRightHand, player) => {
      this.grabInfo = {isRightHand, player}
    })

    // Forget about the grabber when they release
    this.connectCodeBlockEvent(this.entity, CodeBlockEvents.OnGrabEnd, () => {
      this.grabInfo = undefined
    })

    // Update the entity every frame
    this.connectLocalBroadcastEvent(World.onUpdate, () => {
      if (this.grabInfo) {
        // Get the y-value of the *intended* player hand location
        const {player, isRightHand} = this.grabInfo
        const playerHand = isRightHand ? player.rightHand : player.leftHand
        const handPosition = playerHand.position.get()
        const handY = handPosition.y

        // Get the current location of the entity
        const grabbablePosition = this.entity.position.get()

        // Change the y-value in the vector
        grabbablePosition.y = handY

        // Set the new location of the entity
        this.entity.position.set(grabbablePosition)
      }
    })
  }
}
Component.register(AxisYConstrainedGrabbable)
```

Grabbables and Ownership

Transfer-on-grab: Ownership of a grabbable entity is transferred to the grabbing player, every time it is grabbed. The ownership transfer is visible in the grab sequence diagram. The ownership transfer occurs *before* the `OnGrabEvent`. When the `OnGrabEvent` is sent, the entity will already have the new owner. If the entity is released while the transfer is occurring you will get both `OnGrabStart` and `OnGrabEnd`.

No transfer-on-release: When the grabbable entity is released, the owner continues to be that player (unless explicitly transferred or when that player leaves the instance).

> **Don't change the owner of a held object**

When you change the owner of a grabbable entity while it is held, it will be force released. However, the `OnGrabEnd` event **will not** be sent. If you are tracking which entities are and are not held (by the `GrabStart` and `GrabEnd` events), this is likely to "break" your ability to correctly track the entity.

Attaching Entities

Entities can be attached to players.
Entity must be an interactive entity and have an active collider.
Entity must have `Avatar Attachable` set to `Sticky` or `Anchor` in Properties panel.

Creating an Attachable

Attachable By

This setting defines the permissions of which players can *manually* attach the entity (by releasing the entity while holding it over their body). This setting does not affect scripted attach with `attachToPlayer`.

Attachable By	Description
Owner	Only the person holding the attachable entity is permitted to attach it to themselves.
Everyone	Anyone holding the attachable entity is permitted to attach it to themselves or anyone else.

Avatar Attachable

Attaching an entity to player can be done by the following:

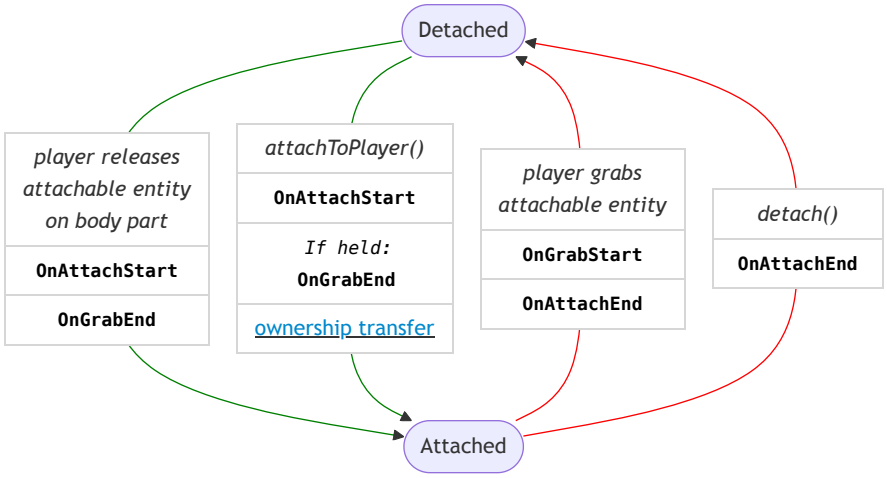
Attach Method	Description
Release on body part	Upon releasing the held entity, the entity checks if collision has occurred between the active collider and the body part of the Attachable By permitted player.
Script	See attachables API.

TODO - Explain what happens when multiple attached

Attaching entities involves two built-in code block events being sent to the attachable. If the player attached or detached by hand (only possible for a VR player) then there are also events related to grabbing.

Built-In CodeBlockEvents	Parameter(s)	Description
`OnAttachStart`	`player: Player`	?
`OnAttachEnd`	`player: Player`	?

The flow of events are shown in the diagram below. Ovals represent the *state* the entity is in. The boxes represent what happens when the entity goes from one state to another; in the box, *italics text is the action* that caused the change, **bold text is built-in CodeBlockEvents** that are sent (in the order top-to-bottom if there are multiple in a box), and underlined text is a system action that occurs.



Transitioning between Held and Attached results in being both at the same time.

An entity goes *from being held to being attached* when `attachToPlayer` is called. An entity goes *from being attached to held* when a player grabs or when `forceGrab` is called. In both of these cases the entity is **momentarily held and attached at the same time**.

Events ordering:

- From Held to Attached: the `OnAttachStart` is sent and then `OnGrabEnd`.
- From Attached to Held: the `OnGrabStart` is sent and then `OnAttachEnd`.

Scripted Attach

Entities can be attached to players and detached from players in scripting using `Entity`'s `attachToPlayer` and `detach`, respectively. The `attachToPlayer` method is not not restricted by the Attachable By and Anchor To settings set in Properties panel; those settings only impact a player manually grabbing an entity and attaching it to themselves (in VR). In order for `attachToPlayer` and `detach` to work the Avatar Attachable property must be enabled in the Properties panel.

Attach and ownership: When `entity.attachToPlayer(player, anchor)` is run, the `entity` is attached to `player` at the `anchor` and the ownership `entity` is automatically transferred to `player`. When `detach()` is called (or a VR player manually removes an item) there is *no* ownership transfer; ownership of the `entity` stays with the player.

Anchor: The anchor is specified by the `AttachablePlayerAnchor` enum which currently has values for `Head` and `Torso`. See socket attachment for changing the exact position and rotation of where an attachable attaches.

Non-grabbable collidable attachables can continuously push the player when they are attached.

When a `attachToPlayer` is called on a **Collidable** entity with **Motion=Animated** or **Interaction=Physics** or `simulated` set to `false`, the entity can continuously push the player (forever). To mitigate this, disable collision on the entity before calling `attachToPlayer`.

Socket Attachment

By default, attachables anchor their pivot point to the attach point with no local rotation (e.g. attaching a hat to a head will have the hat's up vector aligned with the head's up vector, and likewise for right and forward vectors).

You can modify the attachment position and rotation (expressed as a local offset) in the Properties panel by setting `Anchor Position` and `Anchor Rotation` on the attachable entity. In scripting, you can get and set the attachment offsets with `socketAttachmentPosition : HorizonProperty<Vec3>` and `socketAttachmentRotation : HorizonProperty<Quaternion>` on the `AttachableEntity` class.

> **Attach 1 meter in front of a player's torso.**

The code below will attach `attachable` to `player` on their torso. Moving the socket position forward a meter, via `new Vec3(0, 0, 1)`, moves `attachable` to always be 1 meter forward from `player`'s torso.

```
attachable.attachToPlayer(player, AttachablePlayerAnchor.Torso)
attachable.socketAttachmentPosition.set(new Vec3(0, 0, 1))
```

Sticky

Whereas attachable entities may have their `Motion` set to `Animated`, `Sticky` entities work best when set to `Grabbable`. Upon releasing the held entity, it will attach to where the collision occurs between the active collider and the Attachable By permitted player.

Stick To

The following is a list of player body parts that the attachable entity may stick to.

Body Part Setting	Sets the stickiness to
Any	Both head and torso
Head	Anywhere on the player's head
Torso	Anywhere on the player's torso

> **Using code to attach a sticky entity does not place the entity at the center of the body part.**
>
> Wherever the entity is located upon calling `attachToPlayer()` will be where the entity will begin to follow the body part.
>
> Set `Avatar Attachable` to Anchor to reposition the entity to the body part when doing a scripted attach.

Anchor

When attached, an anchored entity will position its pivot point at a specified anchor position.

The anchor position is a body part specified in Anchor To. Anchor position can be altered by setting socket attachment offset values.

By default an anchored entity's rotation matches the rotation of the body part it is attached to. E.g. by default a hat's forward vector will match the head's forward vector (assuming the hat was attached the head with no socket attachment offsets).

> **Once attached, the entity will be affixed to the body part defined in `Anchor To` until detached from player.**

TODO - Explain detach via a grab by a "Who Can Grab?" permitted player and detach via code.

Anchor To

The following is a list of player body parts that the attachable entity may anchor to.

Body Part Setting	Sets the attachment point to
Head	Front-center of the player's forehead.
Torso	Front-center-bottom of the player's ribcage.
Left Hip	Left side of the bottom of the player's pelvis. Note that the attachment will be rotated to "look down", simulating a "holstered" item.
Right Hip	Left side of the bottom of the player's pelvis. Note that the attachment will be rotated to "look down", simulating a "holstered" item.

This image illustrates the local coordinate axes of 4 attachables attached at each of the 4 anchors:

> ⚠ **As of 1/15, `Left Hip` or `Right Hip` are not available as a `AttachablePlayerAnchor`**
>
> Use socket attachments with `AttachablePlayerAnchor.Torso` to get around this.

Auto Scale to Anchor

This settings currently has no effect on the attachable entities.

Attach to 2D screen

This toggle causes the attachable entity to become **screen-attached**. This means the entity's transformation will match the camera transformation. The transformation can be offset by setting the 2D Screen Position, 2D Screen Rotation, and 2D Screen Scale.

Attach to 2D screen can be toggled on for both `Sticky` and `Anchor` attachable types.

> **Attach to 2D screen is meant for cross-screen players**
>
> A VR player who attaches the entity will see the attachable attach to their body as expected.

> **Screen-attached entities will appear on every cross-screen player's screen by default**
>
> Consider setting who can see an entity to avoid this issue.

> **VR players will see other players' screen-attached incorrectly**
>
> The attachable follows their camera's position, but the orientation will be wrong.

Holstering Entities

Grabbable and Attachable

Player Input

Actions on Held Items

Onscreen Controls

Player Controls

TODO

- PlayerControls.onFocusedInteractionInputStarted{
  interactionInfo: InteractionInfo[];
  }
- PlayerControls.onFocusedInteractionInputMoved{
  interactionInfo: InteractionInfo[];
  }
- PlayerControls.onFocusedInteractionInputEnded{
  interactionInfo: InteractionInfo[];
  }
- PlayerControls.onHolsteredItemsUpdated{
  player: Player;
  items: Entity[];
  grabbedItem: Entity;
  }

`PlayerControls`

Focused Interaction

Persistence

Overview

- Cloning a world
- World persistence does not exists

Quests

- Overview
  - Tracked
- Creation
- Using the Gizmo
  - Which are visible
- APIs
- Resetting

TODO- Can be moved to dedicated quest reference
Exactly like achievements on Steam, Xbox, Playstation. Quests help direct visitors around your experience

Must be created in Quest tab in creator menu

Simple means you can complete the quest through an achievement event

Tracked means you can set a player persistent variable to a specified number to complete it

```
class AchievementsGizmo extends Entity {
    /**
     * Displays the achievements.
     *
     * @param achievementScriptIDs - List of achievement script IDs.
     */
    displayAchievements(achievementScriptIDs: Array<string>): void;
}


// CodeBlockEvents
    /**
     * The event that is triggered when an achievement is completed.
     */
    OnAchievementComplete: CodeBlockEvent<[player: Player, scriptId: string]>;
```

Player Persistent Variables (PPV)

- Overview
  - Variable Groups
  - Types: `number` and JSON-serializable `object`.
- Creation
- Read / Write
- Resetting

In-World Purchases (IWP)

TODO

NPCs

TODO

Spawning

Entities and hierarchies can be saved as an asset. Assets are like packages of entities, property configurations, and scripts.

Assets must have an `Asset Type` and `Folder`.

Saved Asset will receive an ID that is used for spawning.

Asset Types	Description
Template Asset	Similar to Prefabs in Unity. Allows for publishing, versioning, and local editing/overriding.
Legacy Asset Group	Simple collection of entities, property configurations, and/or scripts.

Assets can be used across worlds, but using a Template Asset simplifies the process of updating reused assets.

Asset Folders can be shared with other users.

Simple Spawning

```
/**
   * Asynchronously spawns an asset.
   * @param asset - The asset to spawn.
   * @param position - The position where the asset is spawned.
   * @param rotation - The rotation of the spawned asset. If invalid, is replace with `Quaternion.one` (no rotation).
   * @param scale - The scale of the spawned asset.
   * @returns A promise resolving to all of the root entities within the asset.
   */
  spawnAsset(asset: Asset, position: Vec3, rotation?: Quaternion, scale?: Vec3): Promise<Entity[]>;
```

Despawning

Advanced Spawning

```
export declare class SpawnControllerBase {
    /**
     * The ID of the asset that is currently being spawned. This is
     * a protected version of the {@link spawnID} property.
     */
    protected _spawnId: number;
    /**
     * The ID of the asset that is currently being spawned.
     */
    get spawnId(): number;
    /**
     * A list of entities contained in a spawned asset.
     */
    readonly rootEntities: ReadableHorizonProperty<Entity[]>;
    /**
     * The current spawn state of the spawn controller asset.
     */
    readonly currentState: ReadableHorizonProperty<SpawnState>;
    /**
     * The spawn state the spawn controller asset is attempting to reach.
     */
    readonly targetState: ReadableHorizonProperty<SpawnState>;
    /**
     * An error associated with the spawn operation.
     */
    readonly spawnError: ReadableHorizonProperty<SpawnError>;
    /**
     * Loads asset data if it's not previously loaded and then spawns the asset.
     *
     * @returns A promise that indicates whether the operation succeeded.
     */
    spawn(): Promise<void>;
    /**
     * Preloads the asset data for a spawn controller.
     *
     * @returns A promise that indicates whether the operation succeeded.
     */
    load(): Promise<void>;
    /**
     * Pauses the spawning process for a spawn controller.
     *
     * @returns A promise that indicates whether the operation succeeded.
     */
    pause(): Promise<void>;
    /**
     * Unloads the spawn controller asset data. If the spawn controller
     * isn't needed after the data is unloaded, call {@link dispose}.
     *
     * @returns A promise that indicates whether the operation succeeded.
     */
    unload(): Promise<void>;
    /**
     * Unloads the asset data of a spawn controller, and performs cleanup on
     * the spawn controller object.
```

```
     *
     * @remarks
     * This method is equivalent to {@link unload}, except afterwards the spawn controller
     * is no longer available for use and all of its methods throw errors. Call
     * `dispose` in order to clean up resources that are no longer needed.
     *
     * @returns A promise that indicates whether the dispose operation succeeded.
     */
    dispose(): Promise<unknown>;
}
```

Sublevels

horizon/world_streaming

```
/**
 * A sublevel of a world that you can stream independently from the rest of
 * the world at runtime.
 *
 * @remarks
 * Sublevels are a way to break up a world into smaller pieces that you can
 * stream separately from other portions of the world. for more information,
 * see the {@link https://developers.meta.com/horizon-worlds/learn/documentation/typescript/asset-spawning/world-strea
 */
export declare class SublevelEntity extends Entity {
    /**
     * Creates a human-readable representation of the SublevelEntity.
     * @returns A string representation of the SublevelEntity.
     */
    toString(): string;
    /**
     * Gets the current state of the sublevel.
     */
    readonly currentState: ReadableHorizonProperty<SublevelStates>;
    /**
     * Gets the state the sublevel is attempting to reach.
     */
    readonly targetState: ReadableHorizonProperty<SublevelStates>;
    /**
     * Loads the sublevel's asset data if not already loaded and makes it active in the world.
     *
     * @returns A promise that resolves when the sublevel is active.
     */
    activate(): Promise<void>;
    /**
     * Despawns the sublevel and preloads the sublevel's asset data so it can be re-activated later.
     *
     * @returns A promise that resolves when the sublevel is loaded.
     */
    hide(): Promise<void>;
    /**
     * Preloads the sublevel's asset data so it can be activated later.
     *
     * @returns A promise that resolves when the sublevel is loaded.
     */
    load(): Promise<void>;
    /**
     * Pauses the sublevel's asset data loading.
     *
     * @returns A promise that resolves when the sublevel is paused.
     */
    pause(): Promise<void>;
    /**
     * Despawns the sublevel's asset data.
     *
     * @returns A promise that resolves when the sublevel is unloaded.
     */
    unload(): Promise<void>;
}
```

Tooltips and Popups

```
export interface IUI {
    /**
     * Shows a popup modal to all players.
     * @param text - The text to display in the popup.
     * @param displayTime - The duration, in seconds, to display the popup.
     * @param options - The configuration, such as color or position, for the popup.
     */
    showPopupForEveryone(text: string | i18n_utils.LocalizableText, displayTime: number, options?: Partial<PopupOptions
    /**
     * Shows a popup modal to a player.
     * @param player - The player to whom the popup is to displayed.
     * @param text - The text to display in the popup.
     * @param displayTime - The duration, in seconds, to display the popup.
     * @param options - The configuration, such as color or position, for the popup.
     */
    showPopupForPlayer(player: Player, text: string | i18n_utils.LocalizableText, displayTime: number, options?: Partia
    /**
     * Shows a tooltip modal to a specific player
     * @param player - the player this tooltip displays for
     * @param tooltipAnchorLocation - the anchor point that is used to determine the tooltip display location
     * @param tooltipText - the message the tooltip displays
     * @param options - configuration for the tooltip (display line, play sounds, attachment entity, etc)
     */
    showTooltipForPlayer(player: Player, tooltipAnchorLocation: TooltipAnchorLocation, tooltipText: string | i18n_utils
    /**
     * Dismisses any active tooltip for the target player
     * @param player - the player that has their tooltip dismissed
     * @param playSound - determines if a default "close sound" should play when the tooltip is closed
     */
    dismissTooltip(player: Player, playSound?: boolean): void;
}

export declare type PopupOptions = {
    position: Vec3;
    fontSize: number;
    fontColor: Color;
    backgroundColor: Color;
    playSound: boolean;
    showTimer: boolean;
};

export declare const DefaultPopupOptions: PopupOptions;

export declare enum TooltipAnchorLocation {
    /**
     * The tooltip is anchored at the left wrist.
     */
    LEFT_WRIST = "LEFT_WRIST",
    /**
     * The tooltip is anchored at the right wrist.
     */
    RIGHT_WRIST = "RIGHT_WRIST",
    /**
     * The tooltip is anchored at the torso.
     */
```

```
    TORSO = "TORSO"
}
/*

export declare type TooltipOptions = {
    tooltipAnchorOffset?: Vec3;
    displayTooltipLine?: boolean;
    tooltipLineAttachmentProperties?: TooltipLineAttachmentProperties;
    playSound?: boolean;
};

export declare type TooltipLineAttachmentProperties = {
    lineAttachmentEntity?: Entity | PlayerBodyPartType;
    lineAttachmentLocalOffset?: Vec3;
    lineAttachmentRounded?: boolean;
    lineChokeStart?: number;
    lineChokeEnd?: number;
};

DefaultTooltipOptions
```

Custom UI

TODO
Overview - immutable tree (even on ownership transfer?) with bindings. Flexbox; many supported HTML/CSS attributes.

UIComponent Class

Bindings

Technical overview (what *T* is allowed, set, derive, and notes on preventing memory growth - e.g. don't keep deriving). T must be serializable (not throwing via JSON.stringify. For example: bigint is not allowed which means that Entity is not allowed.)

Limits of type, amount, and frequency.

Style

View Types

View

Image

Pressable

Dynamic List

ScrollView

Animated Bindings

Cross Screens - Mobile vs PC vs VR

Camera

Scratch notes

XS only

Local Only
Local Camera

- Spawn point camera options
- Turnkey modes (1st and 3rd person)
- Granular modes? (Fixed, Attach, Orbit, Pan)
- Collision (enable/disable)
- Disabling perspective switch

Performance Optimization

Physics Performance

Colliders, triggers,

Gizmos

- pool FX, sounds,
- limit mirror (1) and dynamic lights (20)

Bridge calls explanation

Draw-call specification

Perfetto hints

Memory

- UIGizmos have an option to enable mipmaps; this will increase visual quality but also increase memory use

List of all desktop editor shortcuts

e.g. alt-click to orbit

Common Problems and Troubleshooting

- stop, reset, play (don't just hit escape)
- leave and come back
- let the instance die

Go to your local directory where all the scripts are and copy all the .ts files that you created
Save the copies in a different location (this is just-in-case backup, so you don't lose any work)
Close the desktop editor
Delete the directory from where you copied the files (like delete the entire world folder)
Then open the desktop editor again, and go to the world
Validate that the world folder has been created again
And last, if needed, bring back the files that you copied with the first step

Glossary

Horizon TypeScript Symbols

AchievementsGizmo
AIAgentGizmo
AimAssistOptions
AnimatedEntity
AnimationCallbackReason
AnimationCallbackReason
AnimationCallbackReasons
assert
Asset
AssetContentData
AttachableEntity
AttachablePlayerAnchor
AudioGizmo
AudibilityMode
AudioOptions
AvatarGripPose
AvatarGripPoseAnimationNames
BaseRaycastHit
BuiltInVariableType
ButtonIcon
ButtonPlacement
clamp
Color
CodeBlockEvents
Comparable
Component
CodeBlockEvent
DefaultFetchAsDataOptions
DefaultFocusedInteractionTapOptions
DefaultFocusedInteractionTrailOptions
DefaultPopupOptions
DefaultSpringOptions
DefaultThrowOptions
DefaultTooltipOptions
degreesToRadians
DisposableObject
DisposableOperation
DisposableOperationRegistration
DynamicLightGizmo
EntityInteractionMode
EntityRaycastHit
EntityStyle
EntityTagMatchOperation
EulerOrder
EventSubscription
FetchAsDataOptions

FocusedInteraction
FocusedInteractionTapOptions
FocusedInteractionTrailOptions
GrabbableEntity
Handedness: force hold, haptics, throwing
HapticSharpness
HapticStrength
HorizonProperty
HorizonSetProperty
ILeaderboards
InteractionInfo
IPersistentStorage
IUI
IWPSellerGizmo
LaunchProjectileOptions
LayerType
LocalEvent
LocalEventData
MaterialAsset
MeshEntity
MonetizationTimeOption
NetworkEvent
NetworkEventData
ParticleFXPlayOptions
ParticleFXStopOptions
ParticleGizmo
PersistentSerializableState
PhysicalEntity
PhysicsForceMode
PlayAnimationOptions
Player
PlayerBodyPart
PlayerBodyPartType
PlayerDeviceType
PlayerHand
PlayerControls
PlayerControlsConnectOptions
PlayerInput
PlayerInputAction
PlayerInputStateChangeCallback
PlayerRaycastHit
PlayerVisibilityMode
PopupOptions
ProjectileLauncherGizmo
PropTypes
Quaternion
radiansToDegrees
RaycastGizmo
RaycastHit
RaycastTargetType
ReadableHorizonProperty
SerializableState
SetMaterialOptions

SetMeshOptions
SetTextureOptions
Space: body part, transform relative to
SpawnController
SpawnControllerBase
SpawnError
SpawnPointGizmo
SpawnState
SpringOptions
StaticRaycastHit
StopAnimationOptions
TextGizmo
TextureAsset
ThrowOptions
TooltipAnchorLocation
TooltipLineAttachmentProperties
TooltipOptions
TrailGizmo
Transform
TriggerGizmo
Vec3
VoipSettingValues
World
WritableHorizonProperty

All Built-In CodeBlockEvents

In the table below:

- 🔈 is a *CodeBlockEvent broadcast on the server*.
- 🏠 is a *CodeBlockEvent broadcast on the device owned by the player in the parameters*.

Built-In CodeBlockEvent	Parameter(s)
🔈 OnAchievementComplete	`player: Player` `scriptId: string`
🔈 OnAssetDespawned	`entity: Entity` `asset: Asset`
🔈 OnAssetSpawnFailed	`asset: Asset`
🔈 OnAssetSpawned	`entity: Entity` `asset: Asset`
OnAttachEnd	`player: Player`
OnAttachStart	`player: Player`
OnAudioCompleted	
OnButton1Down	`player: Player`
OnButton1Up	`player: Player`
OnButton2Down	`player: Player`
OnButton2Up	`player: Player`
🔈 OnCameraPhotoTaken	`player: Player` `isSelfie: boolean`
OnEntityCollision	`collidedWith: Entity` `collisionAt: Vec3, normal: Vec3, relativeVelocity: Vec3, localColliderName: string,`
OnEntityEnterTrigger	`enteredBy: Entity`
OnEntityExitTrigger	`enteredBy: Entity`
OnGrabEnd	`player: Player`
OnGrabStart	`isRightHand: boolean` `player: Player`
OnIndexTriggerDown	`player: Player`
OnIndexTriggerUp	`player: Player`
OnItemConsumeComplete	`player: Player` `item: string, success: boolean`
OnItemConsumeStart	`player: Player` `item: string`
🔈 OnItemPurchaseComplete	`player: Player` `item: string, success: boolean`

Built-In CodeBlockEvent	Parameter(s)
🔈 OnItemPurchaseFailed	`player: Player` `item: string`
🔈 OnItemPurchaseStart	`player: Player` `item: string`
🔈 OnItemPurchaseSucceeded	`player: Player` `item: string`
OnMultiGrabEnd	`player: Player`
OnMultiGrabStart	`player: Player`
OnPassiveInstanceCameraCreated	`sessionId: Player` `cameraMode: string`
OnPlayerCollision	`collidedWith: Player` `collisionAt: Vec3, normal: Vec3, relativeVelocity: Vec3, localColliderName: string,`
🔈 OnPlayerConsumeFailed	`player: Player` `item: string`
🔈 OnPlayerConsumeSucceeded	`player: Player` `item: string`
🔈 OnPlayerEnterAFK	`player: Player`
OnPlayerEnterTrigger	`enteredBy: Player`
🔈 OnPlayerEnterWorld	`player: Player`
🏠 OnPlayerEnteredFocusedInteraction	`player: Player`
🔈 OnPlayerExitAFK	`player: Player`
OnPlayerExitTrigger	`exitedBy: Player`
🔈 OnPlayerExitWorld	`player: Player`
🏠 OnPlayerExitedFocusedInteraction	`player: Player`
OnPlayerSpawnedItem	`player: Player` `item: Entity`
OnProjectileExpired	`position: Vec3` `rotation: Quaternion, velocity: Vec3`
OnProjectileHitObject	`objectHit: Entity` `position: Vec3, normal: Vec3`
OnProjectileHitPlayer	`playerHit: Player` `position: Vec3, normal: Vec3, headshot: boolean`
OnProjectileHitWorld	`position: Vec3` `normal: Vec3`
OnProjectileLaunched	`launcher: Entity`

OPEN QUESTIONS - TODO

TODO: player locomotion speed, jump speed, grounded, apply force (and put a list of all player properties in the player class "overview" section)

NOTE: force-hold can take a number of frames to send the grabEvent (saw 13 frames in a test - which is about 250ms, or 1/4s)

- When do entity.owner vs world.getLocalPlayer() change - it seems that in `transferOwnership` that the former has already changed but not the latter?
- Does simulation=false disable a collision (e.g. can something still hit it or go through a trigger)? The answer should be yes!
- when calling allowPlayerJoin(false), can players join by invite or is the instance actually LOCKED vs Closed?