DRAFT

Note: This content represents the current state of the HSR platform, its applications, and workflows. The platform is under continuing development, and this content may require periodic updating. This content should not be interpreted as official product documentation.

This guide walks through the steps of creating a very simple yet functional world in Meta Horizon Worlds HSR platform.

Learning objectives

In this guide, you walk through the following objectives:

- Explore the user interfaces of the Asset Hub and the Horizon Editor.
- Create a new project.
- Create entities in your project.
- Create templates and add them into your project.
    - Create from pre-built primitives or build from available components.
- Perform basic TypeScripting for:
    - Adding script properties

    - Audio playback
    - Trigger enter/exit
    - Spawning
    - World onUpdate() actions
- Preview your world.

Before you begin

Before you begin this tutorial, you should do the following:

- Acquire permissions to access HSR, including its developer builds.
- Download the **Horizon Editor Standalone (Rift) app** through your Meta Quest Link application.
    - You may also use the Windows installer to install the app.
- Load the Horizon Editor.

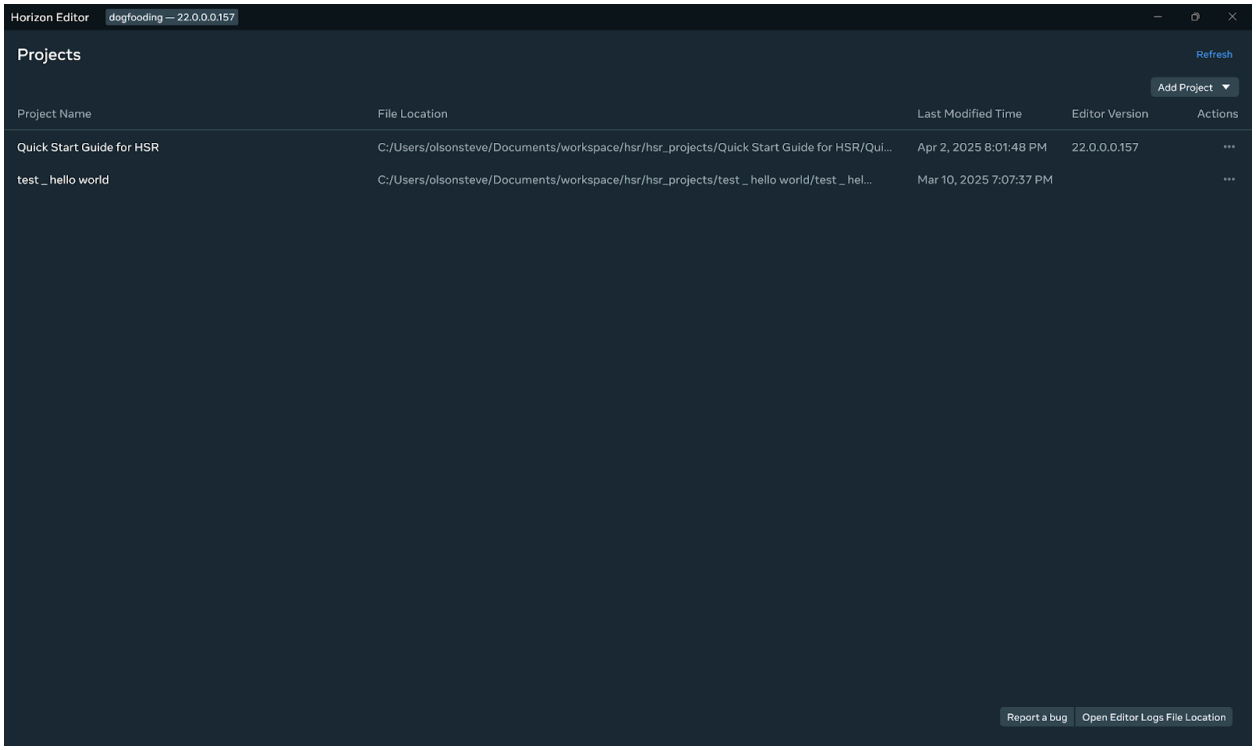
For more information, see Get Started for HSR doc.

Start building

Create your project

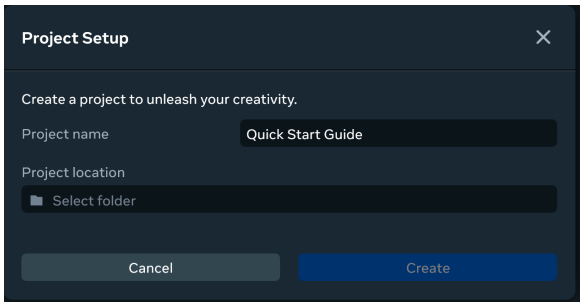
To begin:

1. Open the Horizon Editor through the desktop or through your Meta Quest Link application.
2. The Projects window is displayed, where you access your projects, which can include whole worlds, asset packages, and dependencies.



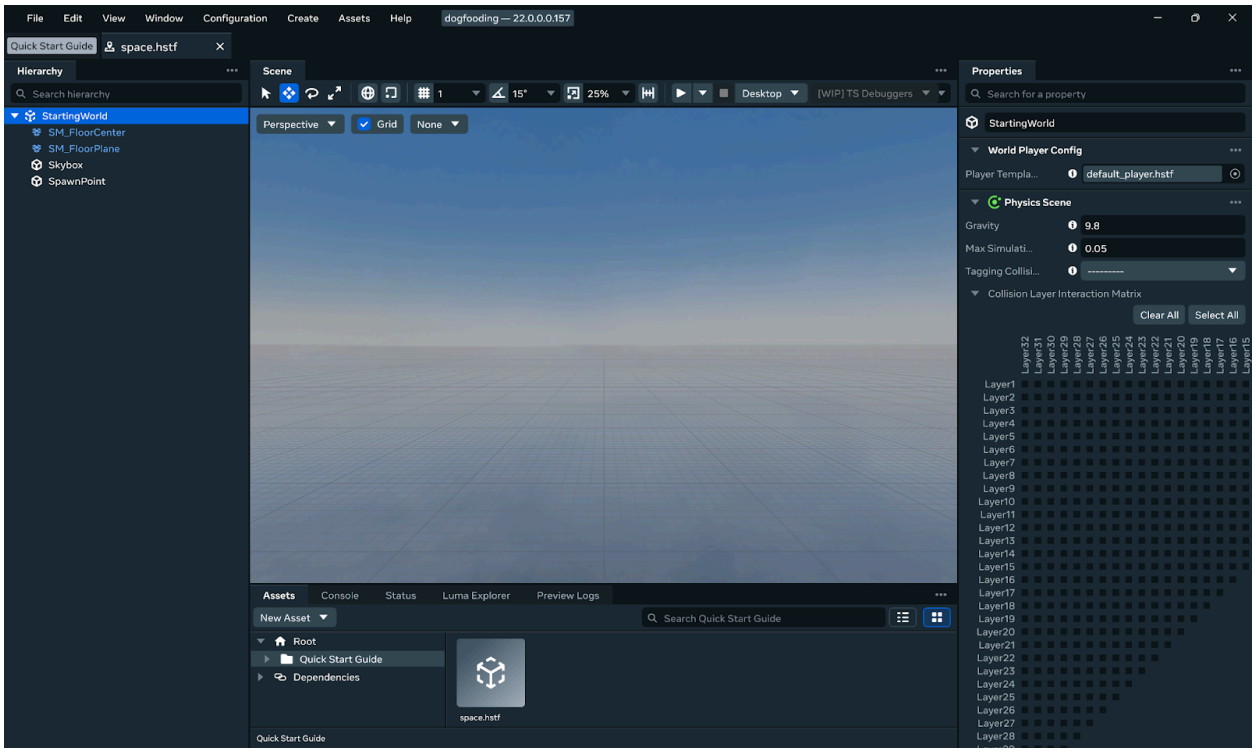
To create a new project:

1. To create a new project, click **Add Project > Create new project**.
2. In the Project Setup window, enter a name for your project, and select the local folder where your project and its files are to be stored:



3. Click **Create**.
4. A folder with the same name as the Project name is created in the selected Project location.
    a. A core set of files is added to the folder.
    b. The project master file is named: `<Project name>.hzproject`.
    c. The space template for your project is: `space.hstf`.

5. The Editor opens displaying your new world.



Camera controls:

You may need to tilt the camera to see the surface. Right-click your mouse button and use the mouse to change the direction of the camera.

Structures of a project:

You have created your first project! Your project is stored as a set of system files and user-created files within a directory on your local desktop.

For more information on the files of your project, see Structures of a Project doc.

Add entity

You can now add an entity to your world. In this example, the process takes two steps:

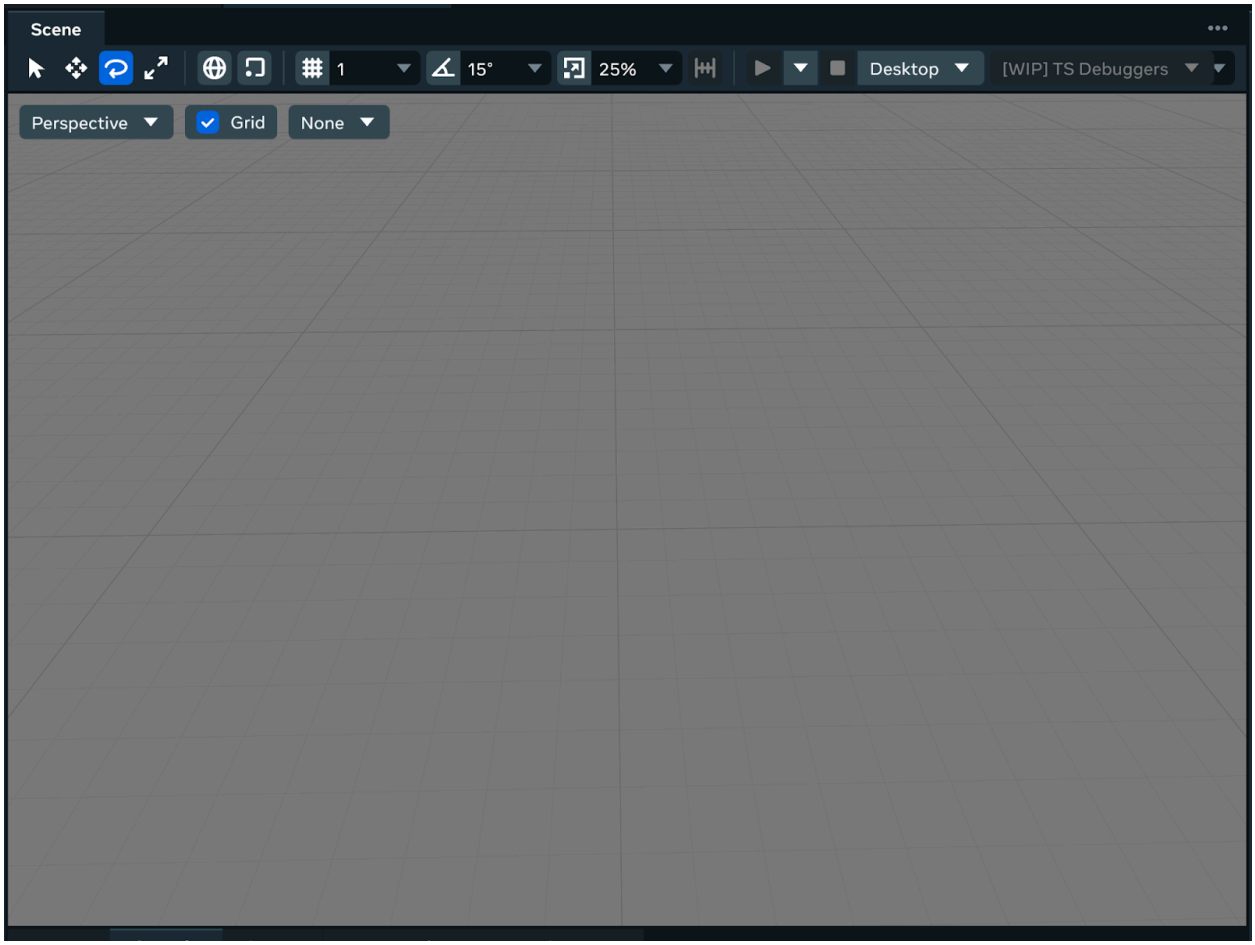
- Create the entity in a separate `.hstf` template file. This template can be used in this world and, if needed, added to other worlds, simply by copying over a couple of files. This workflow has two approaches:
    - Quick method: Add a premade asset (cube) to your template.

    - Component method: Add a mesh component to your template. Then, specify the Cube mesh.
    - **Tip**: You can also create any entity or set of entities to a template for later reuse.
- Add an instance of the template to your world in development.

Create template file

In this example, you create the entity in a separate file from your main project file (`space.hstf`), which makes it easier to share the asset as needed.

1. In the Horizon editor, make sure the `space.hstf` tab is selected.
2. From the menu, select **File > New Template**.
3. The Scene window now displays a blank space:



4. You should also see an `<untitled>` tab above the Hierarchy panel.

5. From the menu, select **File > Save**.
    a. Verify that you are in the project directory for your project. The directory should have the same name as your project: `Quick Start Guide`.
    b. You might want to create a new folder to store your templates: `templates`.
    c. Save the file as: `blueCube.hstf`.
6. The next step is to add a cube to this template.

1. Quick method

In this specific example, you are adding a premade asset to your world. In this case, you can add the entity from a premade primitive asset.

Select **Create menu > Shapes > cube**. The entity is added to the template.

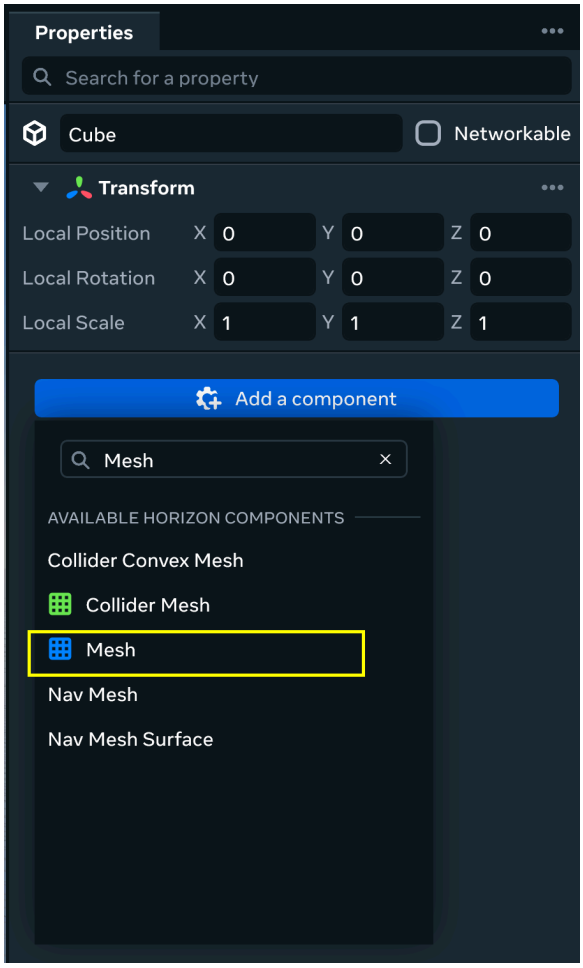
Tip: This method adds multiple components to make the cube behave like a physical object in the world. This may be the simplest method of creation.

2. Add by component method

You may find it more instructive to follow the template file method.

1. Above the Hierarchy panel, verify that `blueCube.hstf` is selected.
2. Right-click the **Root node** in the Hierarchy panel. Select **Add > Entity**.
    a. When you add an entity, the Transform component is automatically included on the entity, which allows you to position, rotate, and scale the entity in the world. However, until it has a visual component to it (a mesh and texture), there is nothing to see in the world.
3. An entry named `New Entity` is added to the bottom of the Hierarchy panel.
4. When you click the entity, its properties are displayed in the right panel.
    a. In the Properties panel, you can change the name of the entity. Rename it to `blueCube`.
    b. Right now, this entity has a transform attached to it. However, it does nothing at present.
5. In the Properties panel, click **Add a component**.

6. In the Search bar, enter `Mesh`:

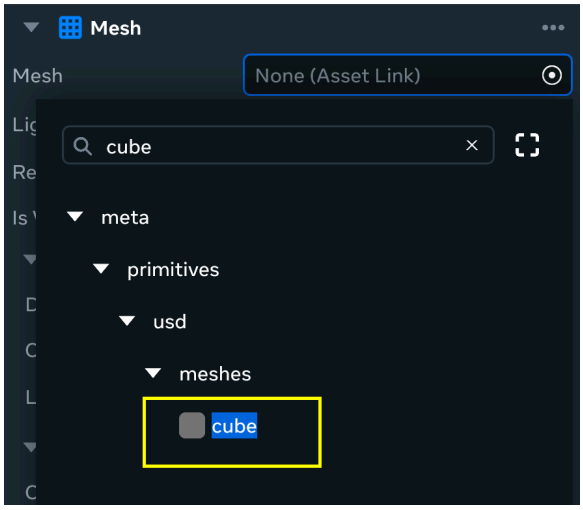


7. Select the highlighted one.

    a. Note that these other components are important for developing collision and navigation, which is outside of our scope at present.

8. You have added a Mesh component container, but it does not yet have a reference to a mesh asset. In the Properties panel, next to the Mesh property, click the **Circle icon**. In

the Search bar, enter: `cube`:



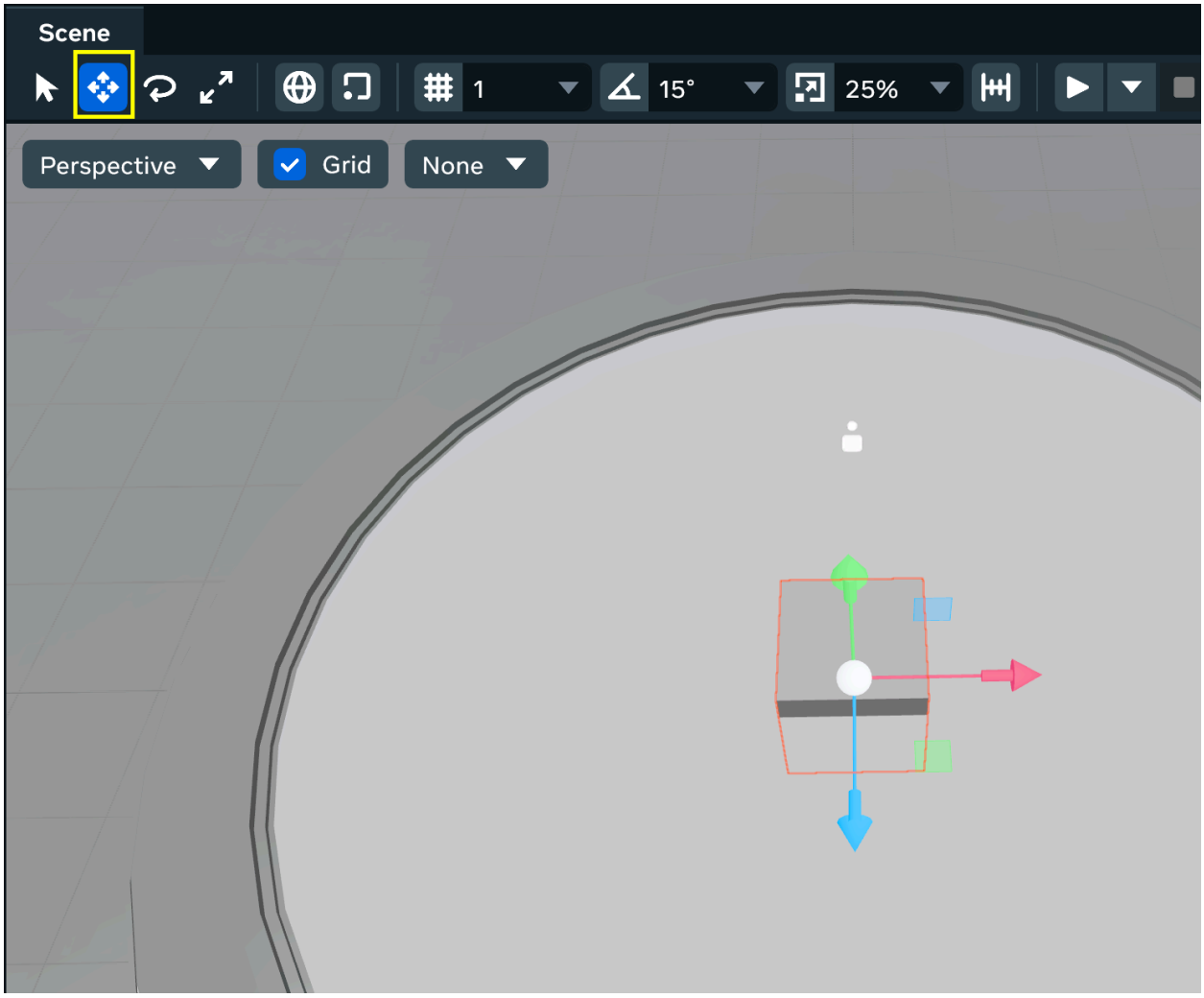
9. Select `cube`. The cube mesh asset is applied to your `Cube` entity.

Configure template properties

Whichever method you used, you can now configure some properties on the cube template.

1. In the Scene window, pivot the camera to point downward. You should now see your cube. You may notice that it appears half-submerged beneath the surface, which is

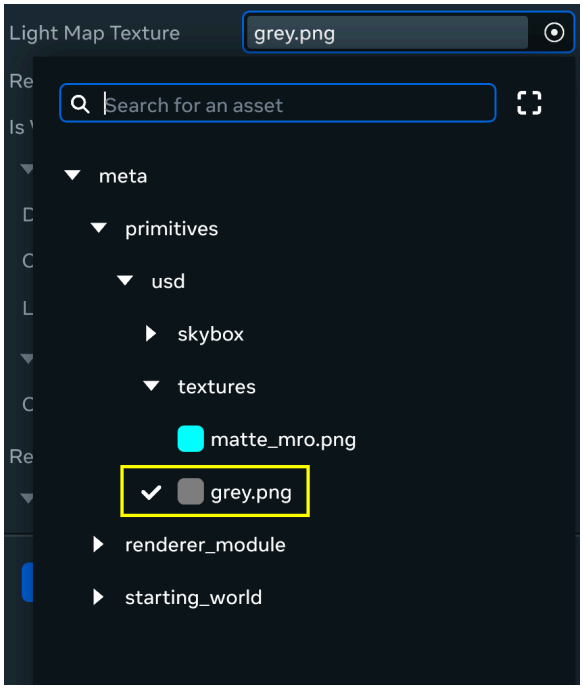
caused by the center of your primitive entity being placed at `(0,0,0)`:



2. To fix this, select the Move tool in the toolbar (highlighted above). Pull on the green arrow until the bottom is above the surface. You can also modify the Local Position properties directly. You might try: `(0,1,0)`.

3. You can now apply some color to your cube's surface. In the Properties panel, click the **Circle icon** next to the Light Texture Map field. Enter `grey`. Select the `grey.png`

texture:



4. The texture of your cube is now grey in color.
5. You can save your template now. Select **File menu > Save**.
    a. For other projects, you may wish to save your templates in separate folders for common access.

Later, you may add in your own textures to the project.

Apply your own texture

Tip: This step is optional.

The above example uses a texture currently available for use in the default project. As an exercise, you should create two textures:

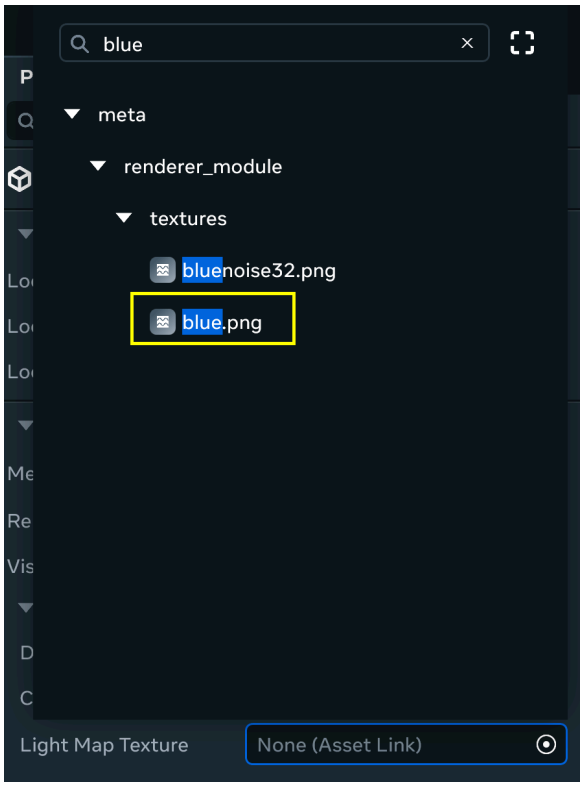
Texture	Dimensions	Filename
Red PNG	512 x 512	`red.png`
Blue PNG	512 x 512	`blue.png`

- In your project directory, create a directory: textures.
- Save these textures into your new textures directory.

- In the Horizon Editor, press F5 to refresh your assets.
    - Note: This step creates the metadata file associated with each .png asset.

To integrate:

1. Create the above files in an external graphics program.
2. Save these files into your project's folder.
    a. Create a sub-folder: textures
    b. Save the files in this location.
3. In your `cube.hstf` template in the Horizon Editor, click the search icon next to Light Map Texture.
4. Search for blue.png:



5. Select the texture.
6. Save your template.
7. You may choose to rename the file: `blueCube.hstf`, which can be done through your desktop directory.

Integrate template

Refresh project

You have saved your template file into the project directory. A separate process refreshes the project assets, which passes newly added assets through the Asset Processor to generate usable assets for the Horizon Editor. This step makes your assets available for use and reference in the Horizon Editor.

- If your project is opened, the project is automatically refreshed to pick up the new assets.
    - If your project is closed, the project is refreshed when you open it.
- If you save the space template file (space.hstf), then the project assets are refreshed automatically.
- At any time, you can refresh assets in a template manually. Press F5.

In this case, the new `blueCube.hstf` is now accessible through the Horizon Editor.

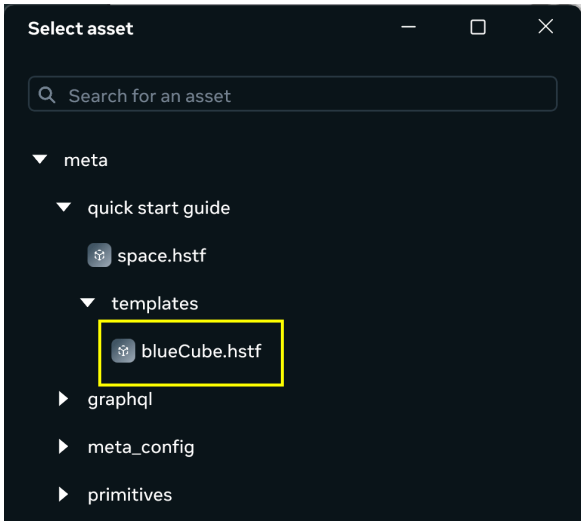
Add instance of template

Click the `space.hstf` tab in the Hierarchy panel. Notice that your new cube template is not present. The template file exists among the project assets, but a reference to it has not been added yet to the project.

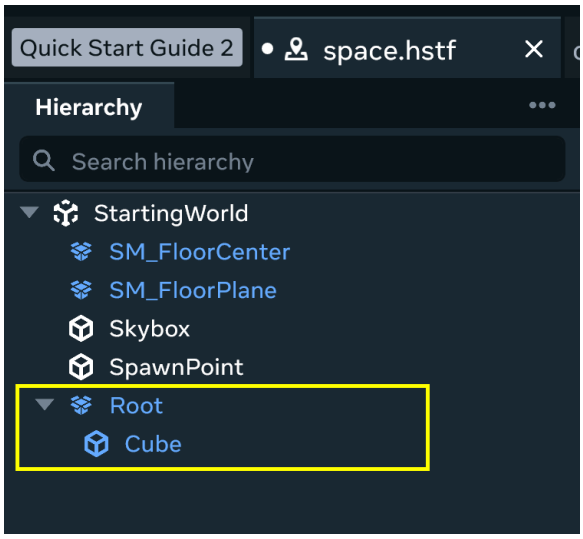
You can now add an instance of this template back to your project.

1. In the Horizon editor, click the `space.hstf` tab.
2. Right-click the **StartingWorld node** in the Hierarchy panel and select **Add > Entity**.
3. You should see a new entry labeled `New Entity`. Right-click this entity and select **Replace with Instance**.

4. In the window, navigate your project assets to select the `blueCube.hstf` file.



5. You should now see that the new entity has been replaced by the hierarchy of nodes from your `blueCube.hstf` file:



6. The cube should appear in the Scene window.

Add scripted entity

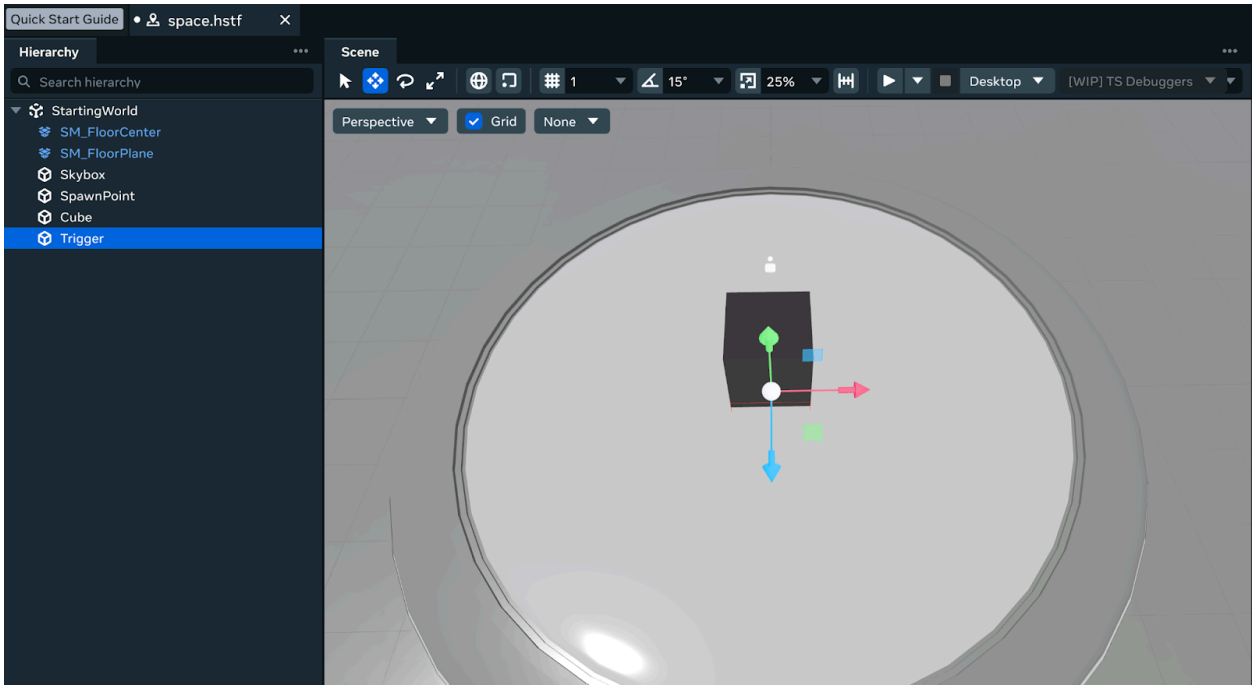
Next, you add an entity with a script attached to it. For this tutorial, you add a Trigger Volume gizmo and then attach a script to it, which responds to the player entering or leaving the trigger.

Add trigger volume

You can now add a trigger volume. A **trigger volume** is a defined space in the world of any size, which triggers TypeScript events when a player enters or exits the volume. In this manner, you can gate programmatic actions based on player movement and location in the world.

In this case, you do not need to create the trigger volume in a separate template file, since its usage is very specific to this project.

1. Select the `space.hstf` tab. Select the root node in the Hierarchy panel.
2. From the menubar, select **Create > Trigger Volume**.
3. A Trigger entity is added to the Hierarchy panel and to the world. It may not be visible:



4. The cube and the trigger volume are the same size, which means that a player cannot enter the trigger volume. To expand its size:
    a. Select the Trigger entity.
    b. In the Properties panel, change the X,Y, and Z values for Local Scale to `5`.
    c. **Important:** Move the trigger volume to a location other than the origin `(0,0,0)`. It may be a temporary issue, but the trigger volume is triggered at the origin even if the SpawnPoint is located elsewhere. Since the Local Scale is `(5,5,5)`, try moving the trigger volume to `(0,0,6)` so that the origin is excluded from it.

Reposition SpawnPoint

With a much larger trigger volume, you should select the SpawnPoint and reposition it to a new location. You could set its Z value to `-8`. The SpawnPoint and the trigger volume are now some distance from each other.

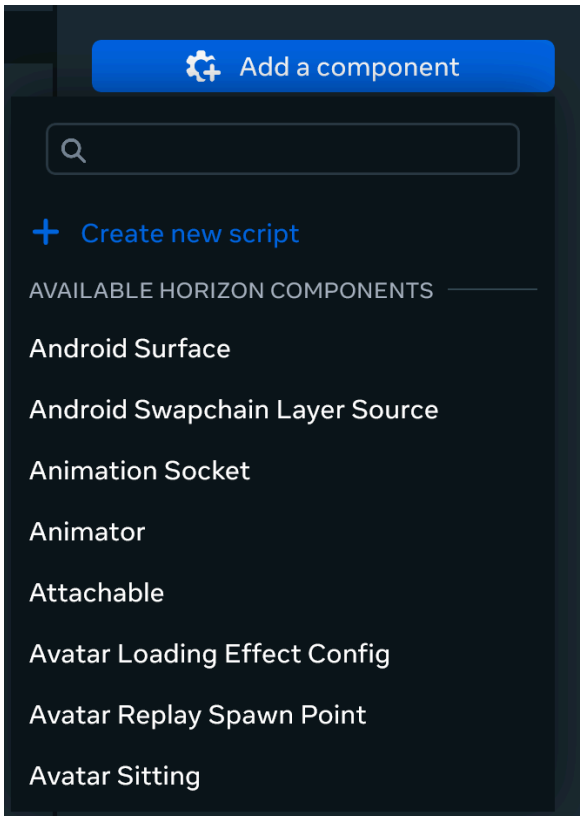
Add script

You now create a script that drives behavior based on players entering or exiting the trigger volume.

Create script

To add a new script:

1. In the Hierarchy panel, select the **Trigger node**.
2. In the right panel, click **Add a component**.
3. Click **+ Create new script**.



4. In your project directory, you might want to create a new folder: `scripts`.

5. Enter a name for your script, such as: `TriggerManager.ts`.

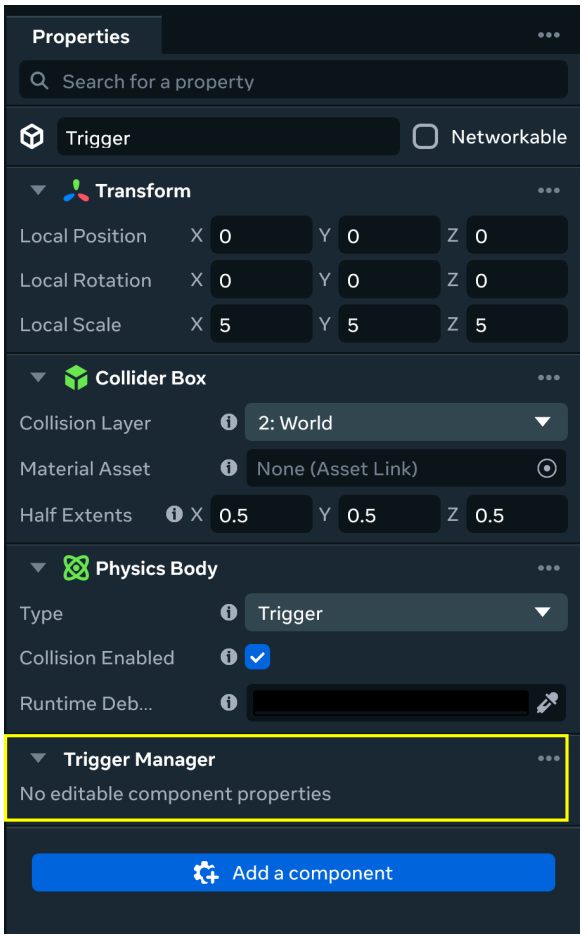
Attach script

The script may not appear in the right panel automatically. You need to attach it to the entity.

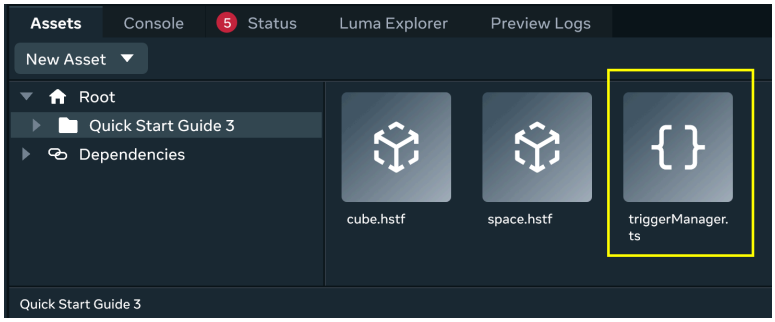
1. Click **Add a component** again.

2. Scroll to the bottom and select **Trigger Manager**. Note that Custom components are listed at the bottom of the list. Select the script entity.

3. The script is now attached to the trigger volume.



In the Horizon editor, you can see it in the **Assets tab** in your project folder:

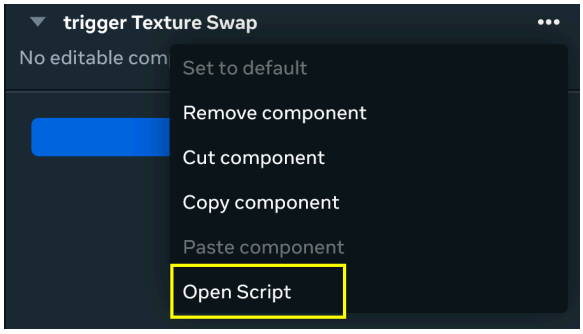


The script has been added to your project and attached to the Trigger entity as a new component.

Edit script

After you have attached the script, you can edit it.

1. In the Hierarchy panel select the trigger volume entity.
2. In the Properties panel, locate the **Trigger Manager** component.
3. From the context menu, select **Open Script**:



4. The script is opened in the external editor.
    a. **Tip**: You can also open a script in your external editor through the Assets tab of the Horizon Editor. Browse your assets and double-click the script.

Here is the script template:

```
JavaScript
/**
 * (c) Meta Platforms, Inc. and affiliates. Confidential and proprietary.
```

```
 */


import {component, Component, OnEntityStartEvent, subscribe,
OnWorldUpdateEvent, OnWorldUpdateEventPayload} from 'meta/platform_api@index';


@component()
export class triggerManager extends Component {
  // Called upon the creation of the Component and before OnUpdateEvent
  @subscribe(OnEntityStartEvent)
  onStart() {
    console.log('onStart');
  }

  // Called once per frame
  @subscribe(World.OnWorldUpdateEvent)
  onUpdate(params: OnWorldUpdateEventPayload) {
  }
}
```

You can see two subscriptions (`@subscribe`) in the template. These subscribe to:

- On entity start (`OnEntityStartEvent`)
    - This method is fired when the entity to which the script is attached is loaded and activated in the world. For any script, this method is typically the main execution area.
- On world update [per-frame] (`OnWorldUpdateEvent`)

    - Note that the per-frame world update event is available through the `WorldService` class.
    - Later in the tutorial, you develop an example for using this event.

Add Trigger event subscriptions

To the above script you can add the following event subscriptions. These are simple listeners for the `onTriggerEnterEvent` and `onTriggerExitEvent` system events.

- An **event** is a custom or system message containing a payload of data.
- A **listener** is a code that awaits the arrival of an event message that has been targeted to the entity hosting the code.

In this case, these are two system events that execute a local function using the payload in `params` as input. These placeholder functions push a log message to the console.

JavaScript

```
@subscribe(OnTriggerEnterEvent)
private onTriggerEnterEvent(params: OnTriggerEnterPayload) {
  console.log("onTriggerEnterEvent() begin");
}

@subscribe(OnTriggerExitEvent)
private onTriggerExitEvent(params: OnTriggerExitPayload) {
  console.log("onTriggerExitEvent() begin");
}
```

Imports:

When the above are added, you should see errors in your TypeScript editor. The above code requires more imports. Please add the following to the top of the script:

JavaScript

```
import { OnTriggerEnterEvent, OnTriggerExitEvent } from 'meta/physics@index';
```

```
import { OnTriggerEnterPayload, OnTriggerExitPayload} from
'meta/physics@index';
```

The above statements import into your script the definitions for the events and their payloads.

Assuming that your script has no errors, you can save this script. Back in the Editor, press `F5` to refresh the world.

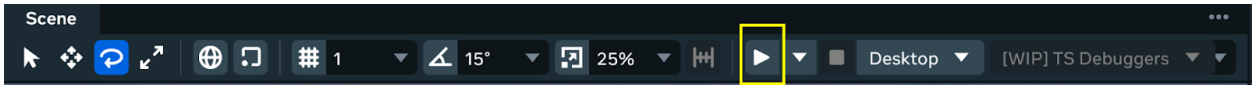
Checkpoint: Let's check out your world!

Checkpoint: Preview your world

You can now preview your world.

Note: The preview controls launch a preview of whichever `.hstf` file is active at present. To preview your entire world, please verify that `space.hstf` has been selected.

To preview click the **Play button** in the toolbar:



When you press the button:

- The files of the project are assembled into a single package.
- The preview opens as a separate window.
- The package is loaded into the preview.

You can now explore your previewed world:

Controls:

- Press the left mouse button and move the mouse to direct your avatar.
- Press `ESC` to open the debug tools.

To test:

When you approach the cube, you pass into the Trigger volume. You should see in the Console the `onTriggerEnterEvent() begin` message.

When you back away, you should see the `onTriggerExitEvent() begin` message.

To exit preview:

To end your preview, press the **Stop button** in the Horizon Editor toolbar.

Note: Do not press the on-screen Quit button or the X button to close the preview window.

Trigger a sound

When someone breaks the Trigger volume, you can configure it to play a sound.

Import a sound file

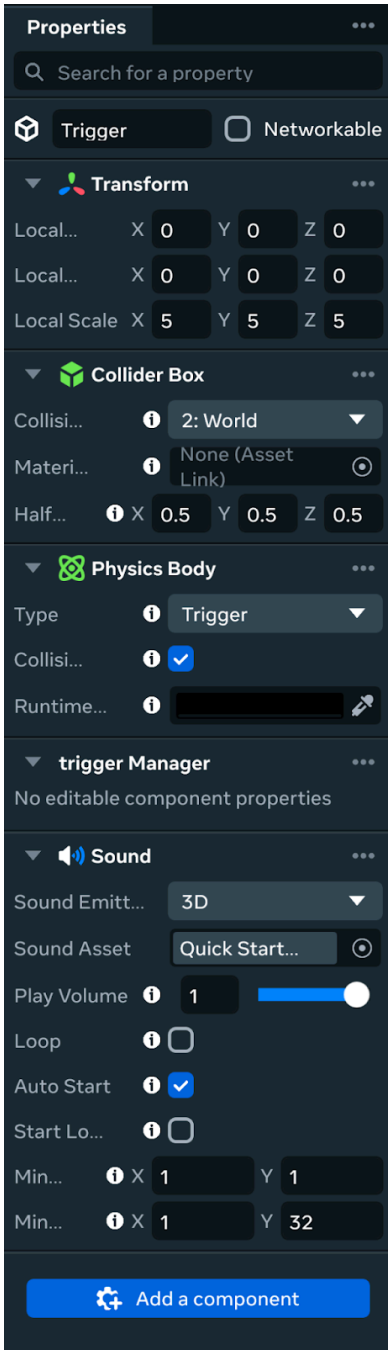
Find a sound and save it at a 48 Khz sample rate `.WAV` format file in your project directory.

Tip: For HUR users, you can use the Generate AI tool to create sounds. Enter a prompt like `rolling thunder`. Play back the options. When you find the one you like, select **Download** from the context menu. Download and save the `.wav` file to your project directory. It should be ready for integration into your HSR Horizon Editor.

Add Sound component

1. Refresh your space template.
2. Select the Trigger entity.
3. Click **Add a component**.
4. Search for: `Sound`. Add it.
5. If you have added a `.wav` file to your directory, click the **Search icon** next to **Sound Asset** in the Sound component. Browse your project folder tree to find and select the sound file.

The properties of your Trigger entity should look like the following:



Checkpoint: Select `Auto Start` on the Sound component. Preview your world. Your sound should play back. Stop the preview and disable `Auto Start`.

Edit trigger script

The sound must be played back when the Trigger volume is entered. In the script, a reference to the SoundComponent is added, which can then be used to reference the `play()` and `stop()` methods exposed on the component.

In the Properties panel for the entity, click the context menu for the `Trigger Manager` component and select **Open script**.

Replace the script with the following:

JavaScript

```
/**
 * (c) Meta Platforms, Inc. and affiliates. Confidential and proprietary.
 */

import {component, Component, OnEntityStartEvent, subscribe,
OnWorldUpdateEvent, OnWorldUpdateEventPayload} from 'meta/platform_api@index';
import { OnTriggerEnterEvent, OnTriggerExitEvent } from 'meta/physics@index';
import { OnTriggerEnterPayload, OnTriggerExitPayload} from
'meta/physics@index';

// added
import { SoundComponent } from 'meta/audio@index';
import type { Maybe } from 'meta/platform_api@index';

@component()
export class triggerManager extends Component {

  // added
```

```
private soundComponent: Maybe<SoundComponent> = null;


@subscribe(OnEntityStartEvent)
onStart() {
  console.log('onStart');
}


@subscribe(OnTriggerEnterEvent)
private onTriggerEnterEvent(params: OnTriggerEnterPayload) {
  console.log("onTriggerEnterEvent() begin");
  // added
  this.soundComponent = this.entity.getComponent(SoundComponent);
  if (this.soundComponent) {
    this.soundComponent.play();
  }
}


@subscribe(OnTriggerExitEvent)
private onTriggerExitEvent(params: OnTriggerExitPayload) {
  console.log("onTriggerExitEvent() begin");
  // added
  this.soundComponent = this.entity.getComponent(SoundComponent);
  if (this.soundComponent) {
```

```
      this.soundComponent.stop();
    }
  }

  // Called once per frame
  @subscribe(OnWorldUpdateEvent)
  onUpdate(params: OnWorldUpdateEventPayload) {
  }
}
```

To the existing script, these items have been `// added`:

- The following declaration adds a variable `soundComponent` of type SoundComponent reference.
    - The `Maybe` type declaration allows the variable to exist without a defined value. You can see below that the initial value is set to `null`.

JavaScript
```
  private soundComponent: Maybe<SoundComponent> = null;
```

- In the `onTriggerEnterEvent()` method, the `soundComponent` variable is assigned to the SoundComponent of the hosting entity:
    - The assignment `this.entity.getComponent(SoundComponent)` works because there is only 1 Sound component in the entity.

JavaScript
```
    this.soundComponent = this.entity.getComponent(SoundComponent);
    if (this.soundComponent) {
      this.soundComponent.play();
```

```
    }
```

- A similar set of steps is included in the `onTriggerExit()` method to `stop()` playback.

Checkpoint

After you save your script and your project, you should preview your world. When you enter the Trigger volume, the connected sound plays back. When you leave the Trigger volume, playback stops.

Add a billboard

Now, you add the text display to your world. This display changes its message depending on whether you have approached the cube (entered the trigger volume).

For this example:

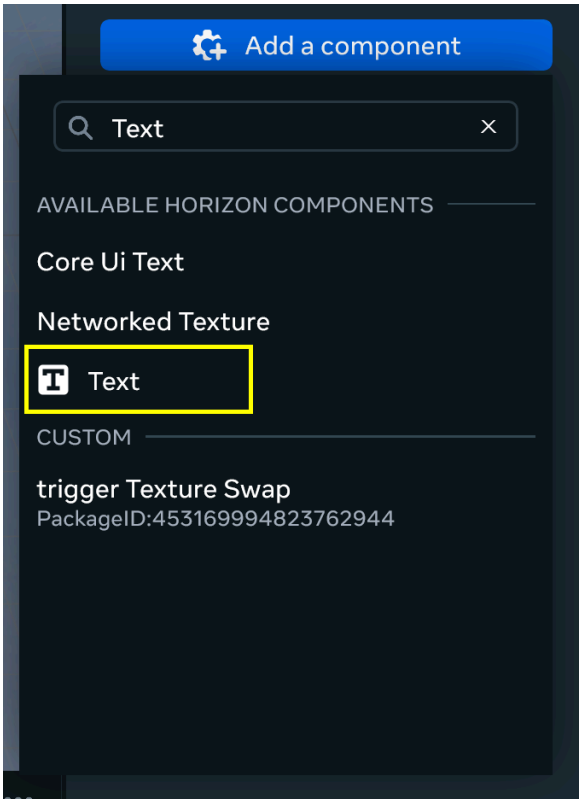
- By default, the message is friendly: `Welcome!`
- When the player enters the trigger volume, the message changes: `Dare you approach the Cube of Doom?`
- When the player exits the trigger volume, the message reverts to: `Welcome!`

Create billboard entity

You can quickly create the billboard entity.

1. Right-click `StartingWorld` in the Hierarchy panel and select **Add > Entity**.
2. Rename the entity to: `Billboard`
3. Add the text component:
    a. Click **Add a component**.

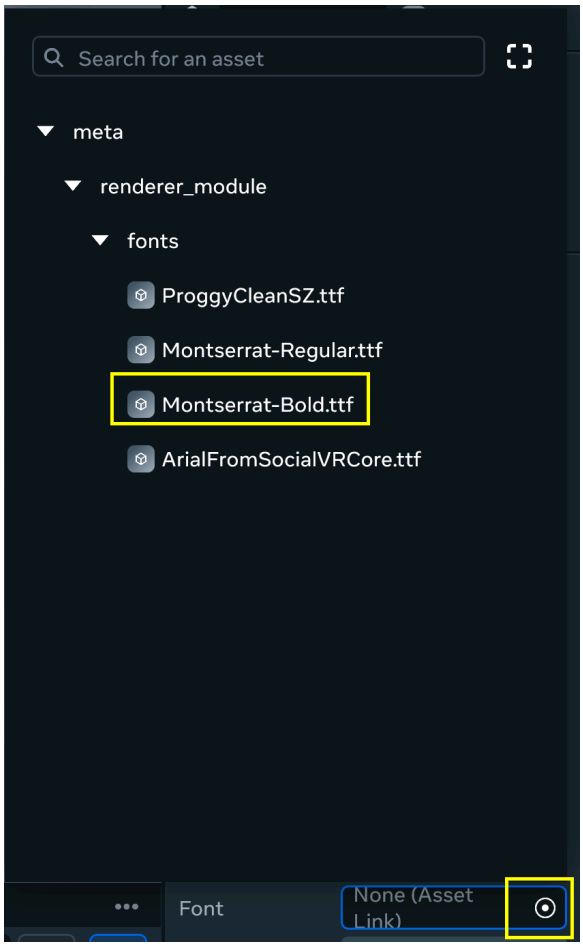
b. Search for `Text`. Select it:



4. Reposition the entity in the world so that it appears over the cube. Try setting Local Position to: `(0,3,6)`.

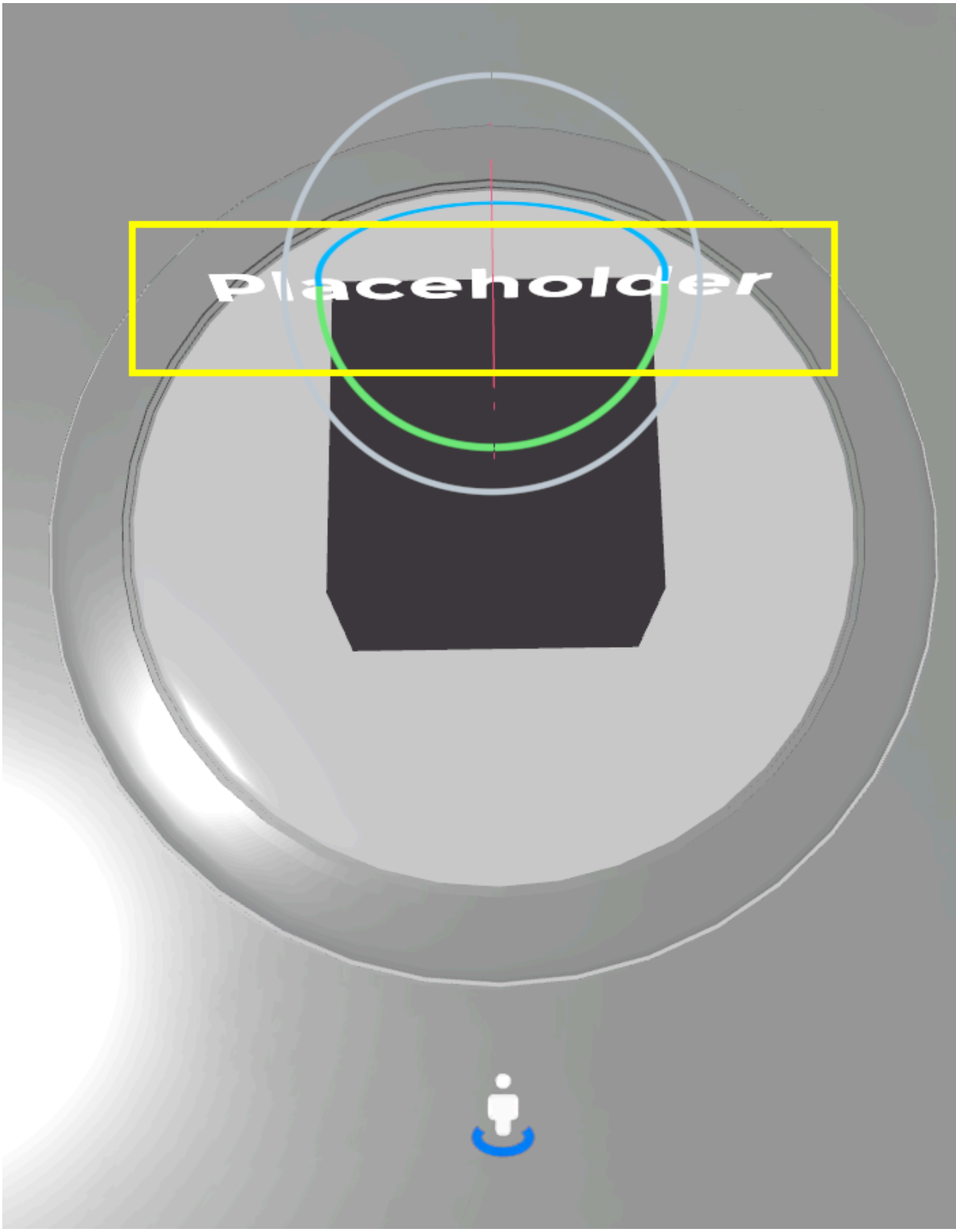
5. You may need to change the rotation of the Billboard entity. Try `(0,180,0)`.

6. The text value (`Placeholder`) does not appear in the Scene window because no font has been associated with the entity. Click the icon next to **Font**:



   a. Navigate and select `Montserrat-Bold.ttf`.

b. The `Placeholder` value should now be visible in the Scene window:



7. **Important:** At the top of the Properties panel, select the **Networkable** checkbox.

a. By default, all entities are local to the client.

b. In this case, however, updates to the billboard should be visible across all clients. Select this checkbox to ensure that all clients receive updates to the message on the billboard.

8. You can modify other properties:

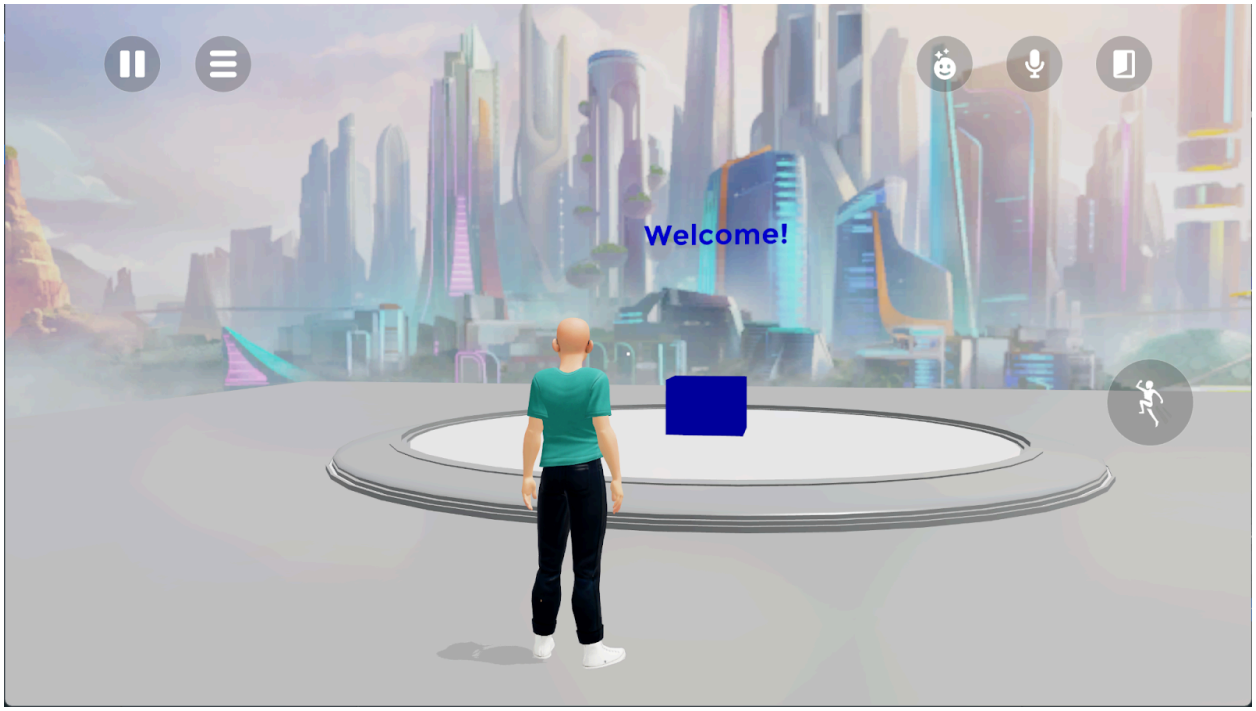
    a. Choose a different color.

        i. **Tip:** Use the Color Picker to select the color that matches the texture of the cube.

    b. Recommended font size: `0.4`

    c. You might want to change the default message from `Placeholder` to something like `Welcome!`.

Checkpoint: If you press the **Play button** to preview your world, the text element should now appear over the cube.



Add script property for billboard entity

To manipulate the billboard entity from the `TriggerManager` script, you must create a script property to hold a reference to the entity.

To your `TriggerManager` script, add the following `@property` entry just below the `class` declaration:

```
JavaScript

@component()

export class triggerManager extends Component {

  @property() billboardEntity: Entity | null = null;
```

The above declaration adds a property to the Trigger Manager component on the trigger volume entity. Currently, it is empty (`null`). Note that `Entity | null` defines the data type for the property; adding `| null` permits a null value, which would otherwise generate a TypeScript error.

Imports:

However, when you paste in the line of code, you may see TypeScript errors for `@property` and `Entity`:

These objects are undefined because they have not been imported to your script. Please add the following to the top of the file:

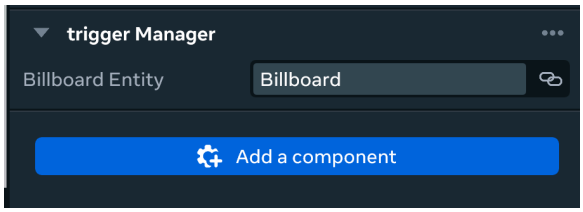
```
JavaScript

import { property } from 'meta/platform_api@index';

import type { Entity, IEntity } from 'meta/platform_api@index';
```

The errors should disappear.

Add billboard entity as script property

Now that you've created the `BillboardEntity` script property, you should populate it with the Billboard entity from your project.

1. Select the **Trigger Volume** entity in the Hierarchy panel.
2. Under the Trigger Manager component, you should see the empty **Billboard Entity field**.
3. From the Hierarchy panel, click and drag the **Billboard node** into the empty field under Trigger Manager.
4. Your screen should look like the following:



The Trigger Manager script now contains a reference to the Billboard entity.

Change billboard message and color

From the `TriggerManager` script, you can apply a new message and a color change directly to the Text component of the Billboard entity.

Imports:

Add the following imports to the top of the file:

```javascript
import { TextComponent } from 'meta/renderer@index';

import { Color } from 'meta/platform_api@index';
```

Local values:

Note: For best results, you should store the default message and color as constants in a script attached to the Billboard entity and exported. To keep it simple, they are added to the `TriggerManager` script. Avoid retrieving the values at startup, since multiple players may be entering the world in different states.

Add these variables just below the `@property` declaration inside the `class` line:

```JavaScript
private defaultBillboardText: string = "Welcome!";

private defaultBillboardColor: Color = new Color (0,0,155); // blue.png

private textComponent: Maybe<TextComponent> | undefined =
this.billboardEntity?.getComponents(TextComponent)[0];
```

Notes:

- `defaultBillboardColor` has a preliminary assignment of the same color as the font color, which is the same color as `blue.png`. Replace this value with your own RGB value.
- `textComponent` is a reference to the `TextComponent` component of the `billboardEntity`. Note that this value is not resolvable if `this.billboardEntity` is undefined, so you must check for that later.

Modify `onTriggerEnterEvent`:

To the `onTriggerEnterEvent()` function, add the following:

```JavaScript
    if (this.billboardEntity) {

      if (this.textComponent) {

        this.textComponent.text = "Dare you approach the Cube of Doom?";

        this.textComponent.color = Color.red;

      } else {

        console.error("textComponent is null!")

      }

    } else {

      console.error("BillboardEntity is null!")

    }
```

Checkpoint: You can now preview your world. Press `F5` to refresh your project, and save your project file. Then, press the **Play button** to begin the preview.

When you approach the cube, the sound should play, and the message and color should be updated.

Tip: The new message and color are buried in code. You might experiment with externalizing these value as:

1. Constants at the top level of your script, or
2. Script properties to be set by non-engineers

Revert billboard message and color

To change the color back to the default values you saved, add the following code to `onTriggerExitEvent()`:

JavaScript

```
    if (this.billboardEntity) {
      if (this.textComponent) {
        this.textComponent.text = this.defaultBillboardText;
        this.textComponent.color = this.defaultBillboardColor;
      } else {
        console.error(scriptName + " textComponent is null!")
      }
    } else {
      console.error(scriptName + " billboard is null!")
    }
```

Checkpoint: Save and refresh your project. Then, preview it.

You should be able to switch back and forth between messages and colors by approaching and moving back from the blue cube.

Spawn entities

In world building, **spawning** refers to the process of adding entities to the world at runtime. Entities can be spawned in and spawned out. Spawning is an important component of building a performant world. Without spawning:

- Load times grow.
- Assets must be loaded even if they are not used until the end of the world/level.
- Device memory is consumed for entities that are not in use.
- Entities in the world have a performance hit, even if they are not in use or near any players.

Spawning is an important technique to learn. In the following example, the blue cube is a static entity that has been placed in the world. Instead, when the trigger volume is entered, the blue cube transforms into a larger red cube to match the red message and sound that has already been added.

This effect is achieved through spawning.

Create redCube template

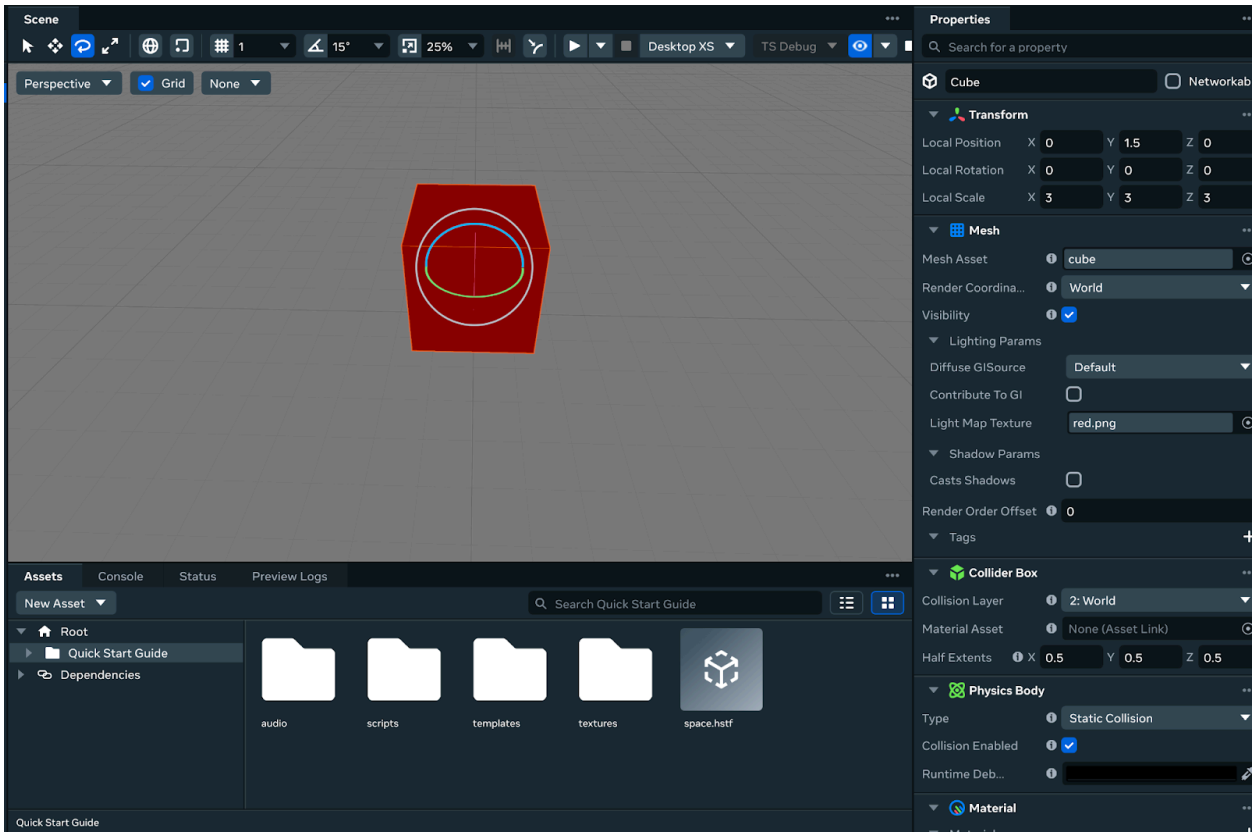
To replace the blue cube with a different entity, you can create another template containing a cube, which is both larger and red, to render a more threatening entity.

1. In the Horizon Editor, select **File > New Template**.
2. To the **Root node** of the template, add a **Transform** component. You can leave its transformation properties as defaults. Adding to the root node enables you to transform the entire template.
3. Add a cube either through the quick method or the template method.
    a. Whichever method you used previously, you might try the other.
    b. **Tip**: Use **Create > Shapes > Cube** to generate a cube, which includes other components for Navigation and Collision. While these components are not used in this tutorial, you can review them for your later projects.
    c. See previous sections.
4. Change cube properties:
    a. **Local Position**: Set Y value to `1.5`.

b. **Local Scale**: Set value to `(3,3,3)`.

c. **Light Map Texture**: Click the **Search icon**. Search for the second `.png` texture that you created. If you cannot find it, verify that you have saved the `.png` files within your project directory.

5. Save the template in the templates directory in your project directory. Filename: `redCube.hstf`.



You now have the two assets to spawn:

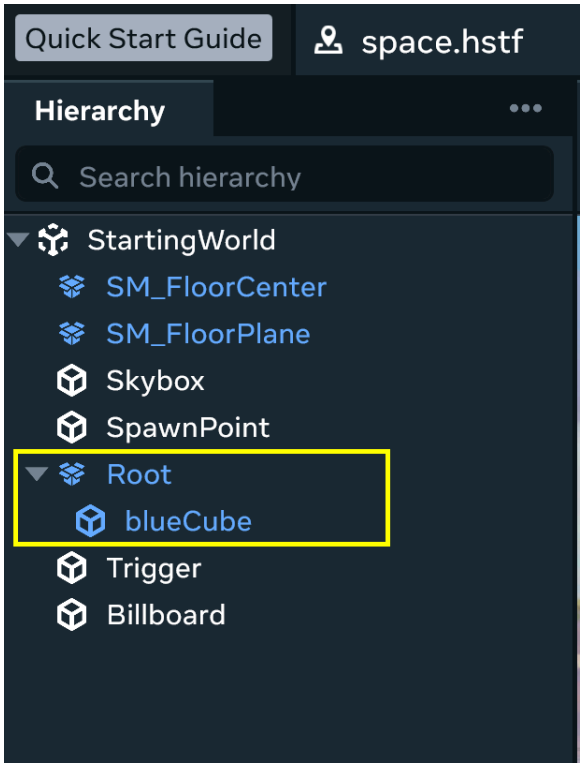
- `blueCube` - default blue cube is to be spawned in at `onStart()` and when `onTriggerExitEvent()` fires
- `redCube` - template to be spawned in when `onTriggerEnterEvent()` fires.

When one cube is spawned in, the other must be removed using the `destroy()` method.

Delete the existing Cube entity

Before you spawn in the entities, you must delete the entity in your Hierarchy panel for the cube that is in the world.

1. Click the `space.hstf` tab.
2. In the Hierarchy panel, locate the entity for the blue cube. It's likely to be:



3. Right-click the **Root node** and select **Delete selection**.
4. The Root node and underlying blueCube node are deleted.

Set up properties for spawning

In the `TriggerManager` script, you must create new properties for each of your templates. The `BillboardEntity` property was of `Entity` type. In this case, they are of `TemplateAsset` type, which means that they must be selected from the assets available to your project.

- Objects of `Entity` type are in the Hierarchy panel and are part of the world.
- Objects of `TemplateAsset` type are template files that can be imported into the world.

Import:

Add the following import to the top of your file:

```JavaScript
import type { TemplateAsset } from 'meta/platform_api@index';

import { Vec3 } from 'meta/platform_api@index';
```

`Vec3` is used later to position the spawned entities.

Properties:

Add these just below the `BillboardEntity` property:

```JavaScript
    @property() blueCube: Maybe<TemplateAsset> | null = null;

    @property() redCube: Maybe<TemplateAsset> | null = null;
```

Local variables:

The above properties must be captured in local variables. Add the following below the `@property` references:

```JavaScript
  private blueCubeEntity: Entity | null = null;

  private redCubeEntity: Entity | null = null;

  private spawnPosition: Vec3 = new Vec3(0, 0, 0);
```

Note the `spawnPosition` variable of `Vec3` type, which is used to place the position of spawned entities.

Set script properties:

After creating the script properties, you must populate them with references to your templates.

1. In the Hierarchy panel, select the trigger volume entity.
2. In the Properties panel, locate the **Trigger Manager component**. You should see two new properties: **Blue Cube** and **Red Cube**.
3. You can populate them using either of the following methods:

a. From the Assets tab, drag the `blueCube.hstf` file into the Blue Cube property. Do the same for the `redCube.hstf` file.

b. Click the **Search icon**. Search for blue or red depending on the property from which you are searching.

Spawn blueCube in onStart()

At this point, you've created the objects in your script and connected the script to the template files associated with each entity to spawn.

Imports:

The following code requires the following imports to be added:

```
JavaScript
import { WorldService, NetworkMode, TransformComponent, } from
'meta/platform_api@index';
```

`onStart()` code:

Since the `blueCube` template was removed as an entity from the world, you must spawn it in at start. To the `onStart()` method, add the following code:

```
JavaScript
    if (this.blueCube) {

WorldService.spawnTemplate({templateAsset: this.blueCube!, networkMode:
NetworkMode.Networked, position: this.spawnPosition})

  .then((entity: IEntity) => {

    const transform: Maybe<TransformComponent> =
entity?.getComponent(TransformComponent)

    if (transform) {

      transform!.worldPosition = this.spawnPosition;

    } else {

      console.error(scriptName + " transform is null!")
```

```
    }
    this.blueCubeEntity = entity;
  })
  .catch((e) => {
    console.error(scriptName + " Failed to spawn template " + e)
  })
}
```

Notes:

- Through the `WorldService`, the `spawnTemplate` method is called to spawn in the template referenced in the `blueCube` property.
  - `NetworkMode` defines if the spawned asset is networked. Since this asset should be applicable to all clients, it is enabled.
  - `position` parameter defines where the entity is spawned, which is assigned the value from the `spawnPosition` variable.
- The method returns a Promise, which is a structure for handling asynchronous methods and can be used to:
  - Determine success or failure of the spawning
  - Capture the returned asset into a variable for reference
- Two methods:
  - `then(entity: IEntity)` - Method is executed on success, with the variable `entity` containing the reference to the spawned entity.
    - The variable `this.blueCubeEntity` is assigned the `entity` object. This is used later to destroy the entity when the red cube is spawned.
  - `catch()` - Method is executed on failure, with the variable `e` capturing any returned error information.

Checkpoint: Save and refresh your project. When you click the **Play button**, the blue cube should spawn in at startup.

Spawn in redCube when trigger volume is entered

In the `onTriggerEnterEvent()` function, you insert the following code:

JavaScript

```
    if (this.blueCubeEntity && this.redCube) {

      this.blueCubeEntity.destroy();

      WorldService.spawnTemplate({templateAsset: this.redCube!, networkMode:
NetworkMode.Networked, position: this.spawnPosition})

        .then((entity: IEntity) => {

          const transform: Maybe<TransformComponent> =
entity?.getComponent(TransformComponent)

          if (transform) {

            transform!.localScale = new Vec3(1, 1, 1);

            transform!.worldPosition = this.spawnPosition;

            this.redCubeEntity = entity;

          } else {

            console.error(scriptName + " transform is null!")

          }

        })

        .catch((e) => {

          console.error(scriptName + " Failed to spawn template " + e)

        })
```

Notes:

- Here, the code is gated by a check to see if the `blueCubeEntity` has been created and `redCube` property has been specified.
- The `blueCubeEntity` is removed using the `destroy()` method. The object is not set to null, since it may be needed again if the player re-approaches the cube.

Checkpoint: If you want to check, Preview the world and approach the blue cube. It should switch to the red cube.

Spawn back in blueCube on trigger exit

In the `onTriggerExitEvent()` function, you insert the following code to spawn in the `blueCube` and to destroy the `redCube`:

JavaScript

```
    if (this.redCubeEntity && this.blueCube) {

      this.redCubeEntity.destroy();

      WorldService.spawnTemplate({templateAsset: this.blueCube!, networkMode:
NetworkMode.Networked, position: this.spawnPosition})

        .then((entity: IEntity) => {

          const transform: Maybe<TransformComponent> =
entity?.getComponent(TransformComponent)

          if (transform) {

            transform!.worldPosition = this.spawnPosition;

            this.blueCubeEntity = entity;

          } else {

            console.error(scriptName + " transform is null!")

          }

        })

        .catch((e) => {

          console.error(scriptName + " Failed to spawn template " + e)

        })

    }
```

Checkpoint: In Preview, you should be able to spawn in the red cube on entering the trigger volume, and spawn back in the blue cube on exiting it.

World Update

When you create a new script, you may have noticed the `OnWorldUpdateEvent` event subscription:

```
JavaScript
  @subscribe(OnWorldUpdateEvent)

  onUpdate(params: OnWorldUpdateEventPayload) {}
```

The `OnWorldUpdateEvent` is fired by the server every frame that is rendered for the client. When you create a subscription to it, you can update behaviors in your script each frame.

Important: Avoid performing updates or running extensive code every frame, which can be a performance bottleneck. Common issues include using code that requires RPCs on the server, which are time-consuming and impact framerate.

deltaTime

Instead of firing every frame, you can create simple code to ensure that the event fires every second instead.

Replace the above placeholder with the following code:

```
JavaScript
  private intervalTime : number = 0.0;

  @subscribe(OnWorldUpdateEvent)

  onUpdate(params: OnWorldUpdateEventPayload) {

    this.intervalTime += params.deltaTime;

    if (this.intervalTime < 1.0) {

      return; // only run code once per second

    } else {

      console.log("onUpdate() 1 second interval");
```

```
      this.intervalTime = 0.0;
    }
  }
```

deltaTime: The event payload is captured to the `params` parameter, which contains the `deltaTime` system variable. Each frame, the value of `deltaTime` is updated to indicate the elapsed time in seconds since the previous frame. Each updated value is typically a small fraction of a second.

In the above code, `deltaTime` is added to `intervalTime`. When `intervalTime` is greater than `1.0`, a consistent message is pushed to console, and the `intervalTime` is reset to `0.0`.

Checkpoint: Save your project. When you preview it, you should see the console message appearing in the console. If you do nothing, the count of the message increases every second.

The Growing Cube

What do you want to do every second in your world?

In the following example, when the red cube is being shown, the cube is scaled slightly larger each second. This simple effect makes the cube look more menacing with each passing second. You can replace all of the `onWorldUpdateEvent` code with the following:

JavaScript

```
  private intervalTime : number = 0.0;
  @subscribe(OnWorldUpdateEvent)
  onUpdate(params: OnWorldUpdateEventPayload) {
//    return; // disable for now
    this.intervalTime += params.deltaTime;
    if (this.intervalTime < 1.0) {
```

```
      return; // only run code once per second

    } else {

      // increase size of redCube if present

      if (this.redCubeEntity != null) {

        const transformRedCube: Maybe<TransformComponent> =
this.redCubeEntity.getComponent(TransformComponent);

        if (transformRedCube) {

          transformRedCube.localScale = new Vec3(transformRedCube.localScale.x
+ 0.1, transformRedCube.localScale.y + 0.1, transformRedCube.localScale.z +
0.1);

        } else {

          console.error(scriptName + " transform is null!")

        }

      } else {

        console.error(scriptName + " redCubeEntity is null")

      }

      // console.log("onUpdate() 1 second interval");

      this.intervalTime = 0.0;

    }

 }
```

Beware the red cube menace!

Checkpoint: That completes the tutorial. Nice work!