Table of Contents

Overview

FVS-Unity-SDK is a dedicated Unity development SDK designed for FVS. It is essentially a Unity package that encapsulates common dashboard 3D component APIs.

When developing 3D scenes, Unity developers can create C# scripts and use the APIs to fulfill business requirements as needed.

After building the Unity project, the generated build output can be imported into FVS for display, enhancing its 3D presentation capabilities.

Before building the project, FVS-Unity-SDK automatically generates a scene configuration (in JSON format).

Below is the configuration for scene 1 in the FVS-Unity-SDK built-in demo (we will explain what each field does later).


```
{
 "scene": {
      "value": {
           "current": "scene1",
           "scenes": [{
                "name": "scene1",
                "displayName": "Scene 1"
           }, {
                "name": "scene2",
                "displayName": "Scene 2"
           }]
      }
 },
 "data": {
      "value": [{
           "gameObject": "FVSContainer",
           "fields": [{
                "name": "x",
                "displayName": "X Coordinate"
           }, {
                "name": "y",
                "displayName": "Y Coordinate"
           }],
           "displayName": "Position Distribution",
           "customFields": true
```

```
    }]
},
"style": {
    "value": [{
        "gameObject": "FVSContainer",
        "displayName": "Style Setting",
        "properties": [{
            "name": "cubeColor",
            "displayName": "Cube Color",
            "getter": "GetCubeColor",
            "setter": "SetCubeColor",
            "type": "Color",
            "value": "#FFFFFF"
        }, {
            "name": "cubePosition",
            "displayName": "Cube Position",
            "getter": "GetPosition",
            "setter": "SetPosition",
            "type": "Vector3",
            "value": [0, 20, 0]
        }, {
            "name": "cubeSize",
            "displayName": "Cube Size",
            "getter": "GetCubeSize",
            "setter": "SetCubeSize",
            "type": "Number",
            "value": 10,
            "min": 1,
            "max": 20,
            "step": 0.1
        }, {
            "name": "text",
            "displayName": "Custom Text",
            "getter": "GetText",
            "setter": "SetText",
            "type": "Text",
```

```
                "value": "Hello, FVS-Unity."
            }, {
                "name": "visible",
                "displayName": "Show Cube",
                "getter": "GetVisible",
                "setter": "SetVisible",
                "type": "Select",
                "value": "true",
                "options": [{
                    "name": "Visible",
                    "value": "true"
                }, {
                    "name": "Invisible",
                    "value": "false"
                }]
            }]
        }]
    },
    "action": {
        "value": [{
            "name": "RotateCube",
            "displayName": "Rotate Cube"
        }]
    },
    "event": {
        "value": [{
            "name": "click",
            "displayName": "Click Cube",
            "parameters": [{
                "key": "name",
                "displayName": "Object Name"
            }]
        }]
    }
}
```

When the packaged resource file is imported into the Unity component in FVS, the component connects to the Unity scene, retrieves and parses the configuration file, and communicates with the Unity scene based on the configuration file content — for example, modifying object styles (color, size) within the scene, passing data into the scene, intercepting mouse events in the scene, and invoking methods provided within the scene.

Let's start with installation and learn how to embed Unity into FVS.

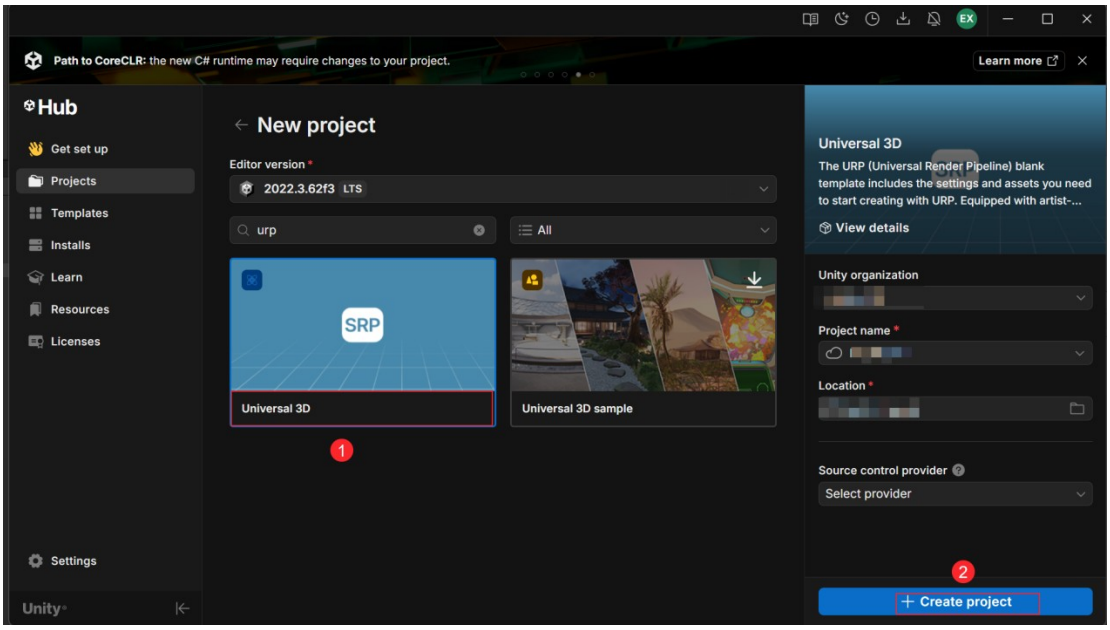
Installation and Building

Creating a Project

First, create a new Unity project in Unity Hub.

This tutorial is intended for experienced Unity developers. If you are new to Unity or have not used it before, we recommend installing Unity and learning the basics before returning to this course.

Here, select **Universal 3D** to create a URP-based 3D project.



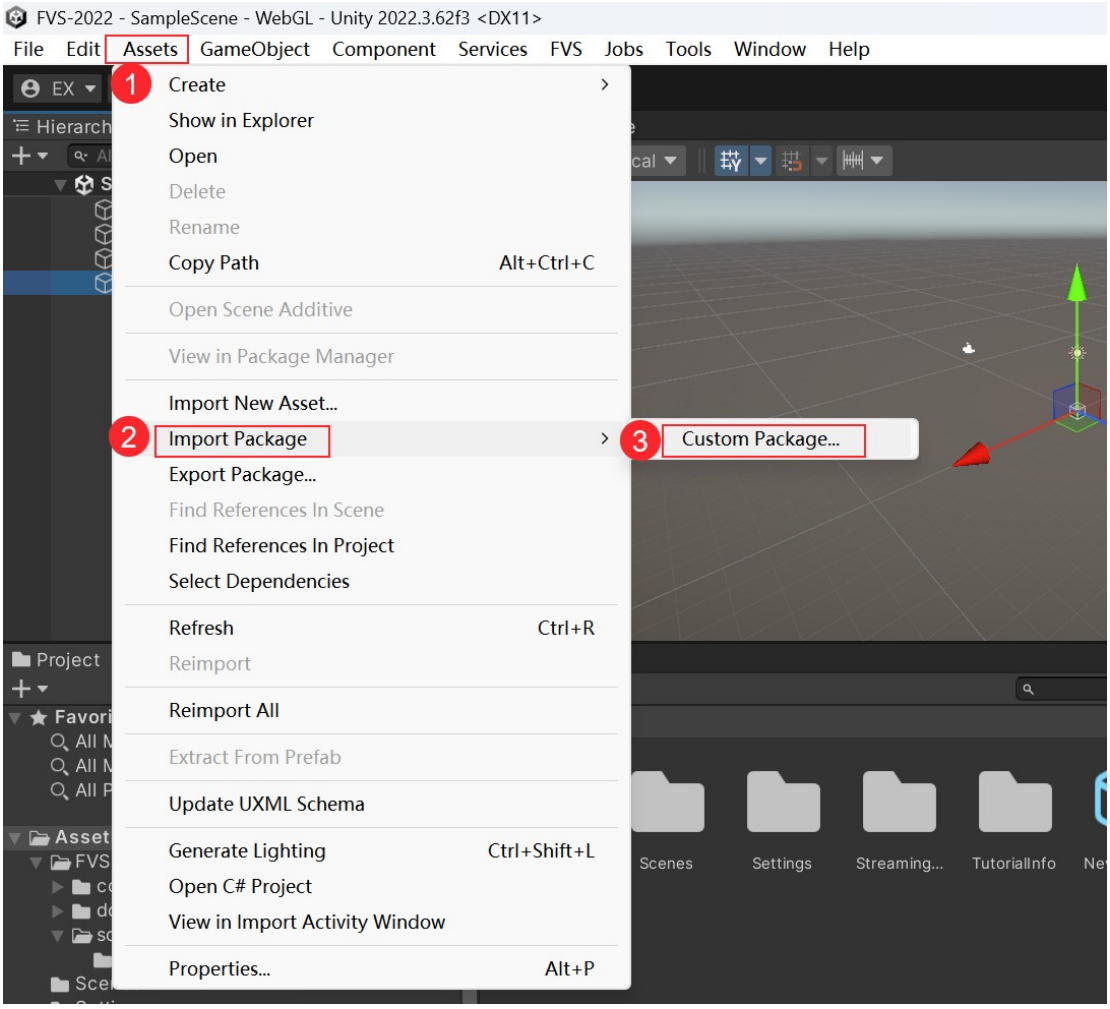
URP stands for Universal Render Pipeline. Since our Unity scene will ultimately be built for the Web platform, URP offers excellent compatibility. With High Definition Render Pipeline (HDRP), some high-quality effects would be compromised when built for the Web platform.

Installing FVS-Unity-SDK

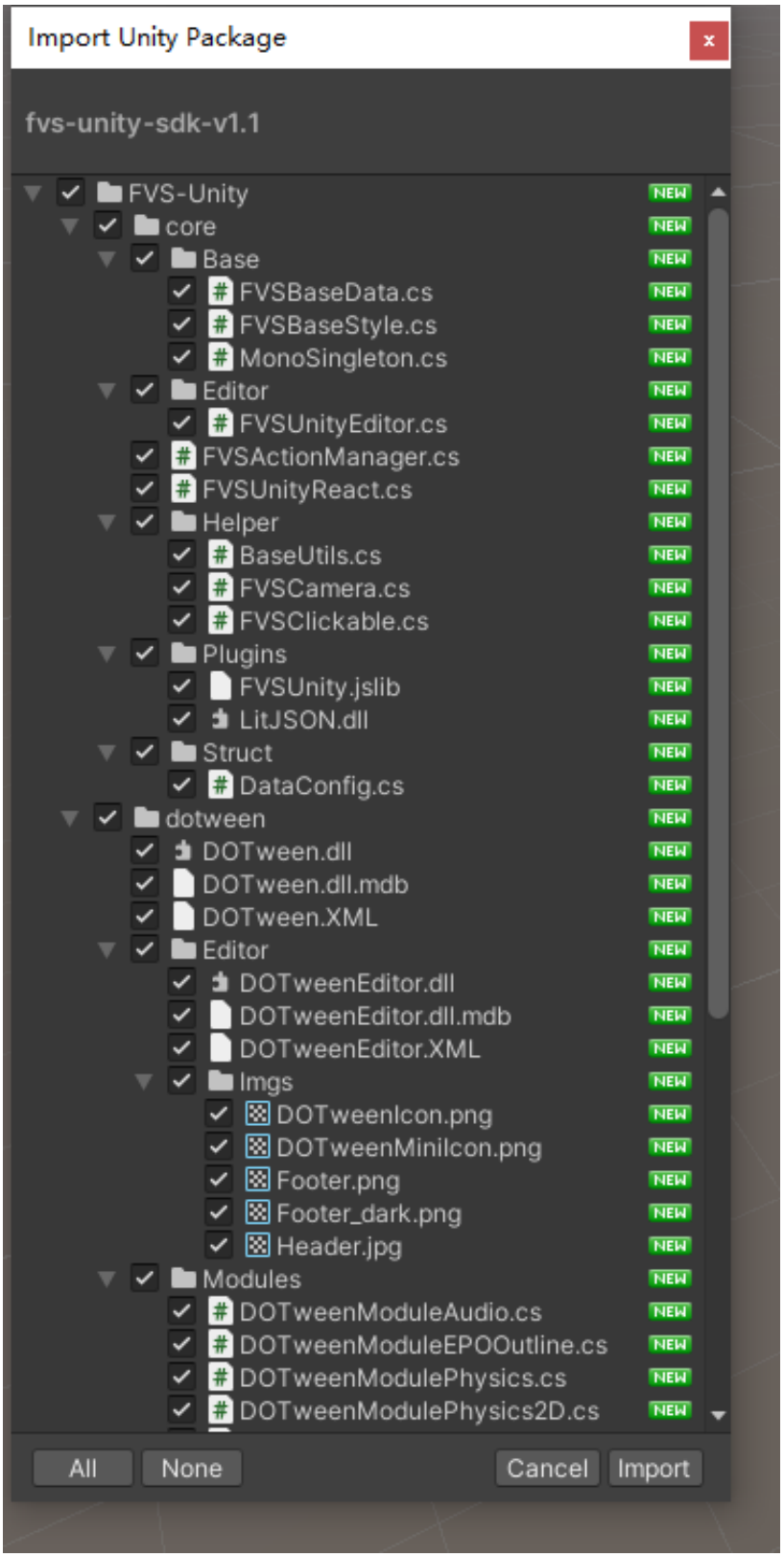
First, we need to download and import the FVS-Unity-SDK package. The download link is below:

https://offline-document-en.obs.cn-north-4.myhuaweicloud.com/EN/Document/fvs-unity-sdk.1.1.9.unitypackage

After the download completes, go to the Unity menu bar and click **Assets** > **Import Package** > **Custom Package**,

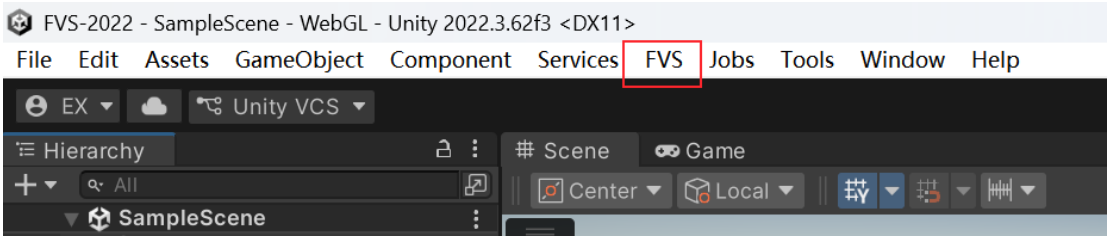


then select the SDK package you just downloaded.



Select all items and click **Import**.

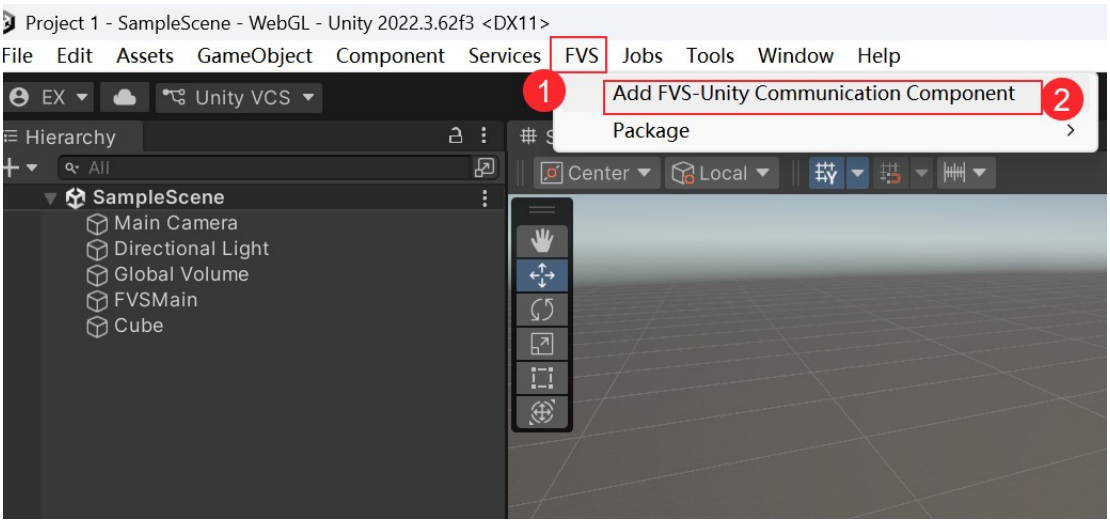
Once script compilation is complete, a new FVS menu item appears in the menu bar, indicating the SDK was imported successfully.



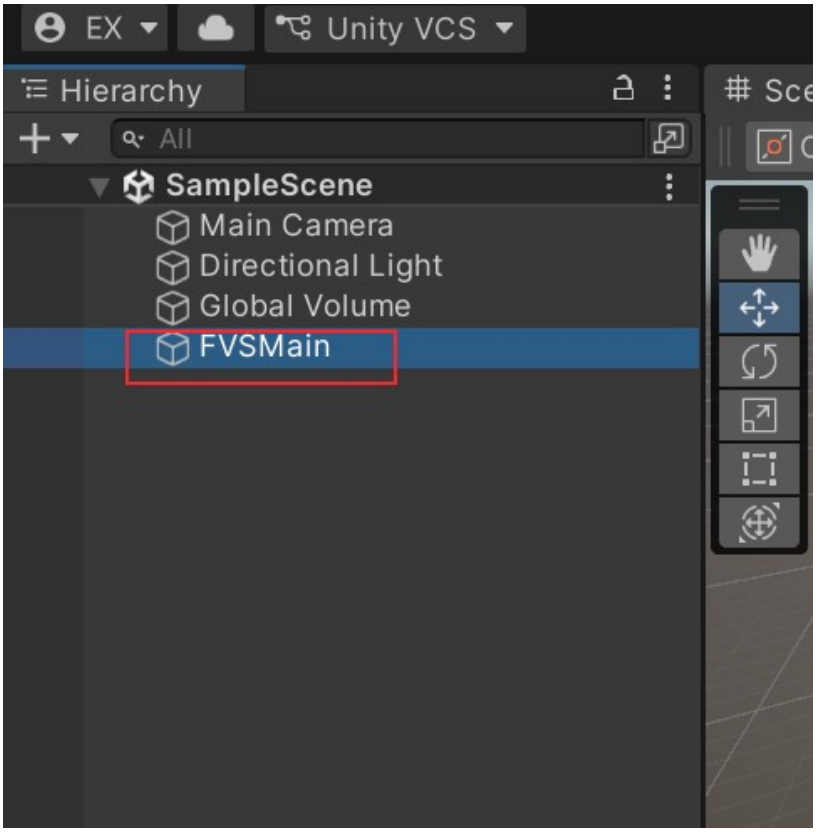
Adding the Communication Component

After importing the SDK, we need to add a communication component to the scene.

Choose **FVS** > **Add FVS-Unity Communication Component** in the menu bar, as shown in the following figure.



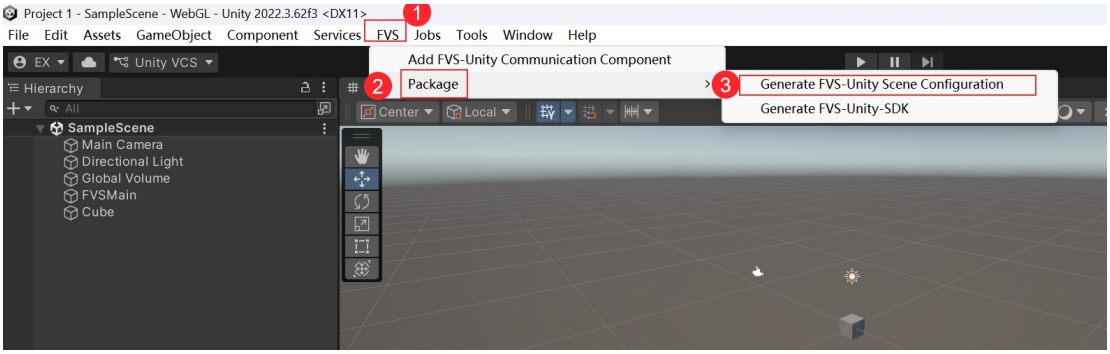
After the component is added, an empty GameObject named **FVSMain** will appear in the hierarchy panel (used for communication with Unity components).



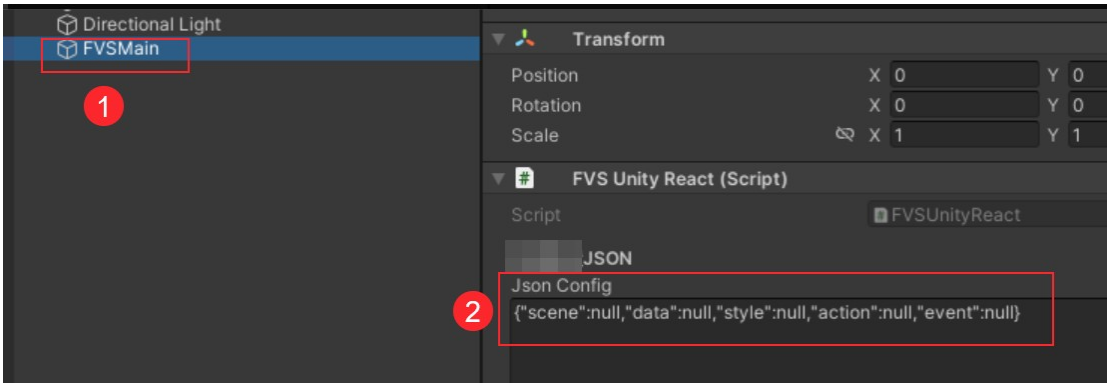
Building

Once the FVSMain communication component has been added, the connection is ready to be established. At this point, we can build the current Unity project and use it in the Unity component of FVS.

Before building, we need to generate the scene configuration file. Click **FVS** > **Package** > **Generate FVS Unity Scene Configuration** in the menu bar.



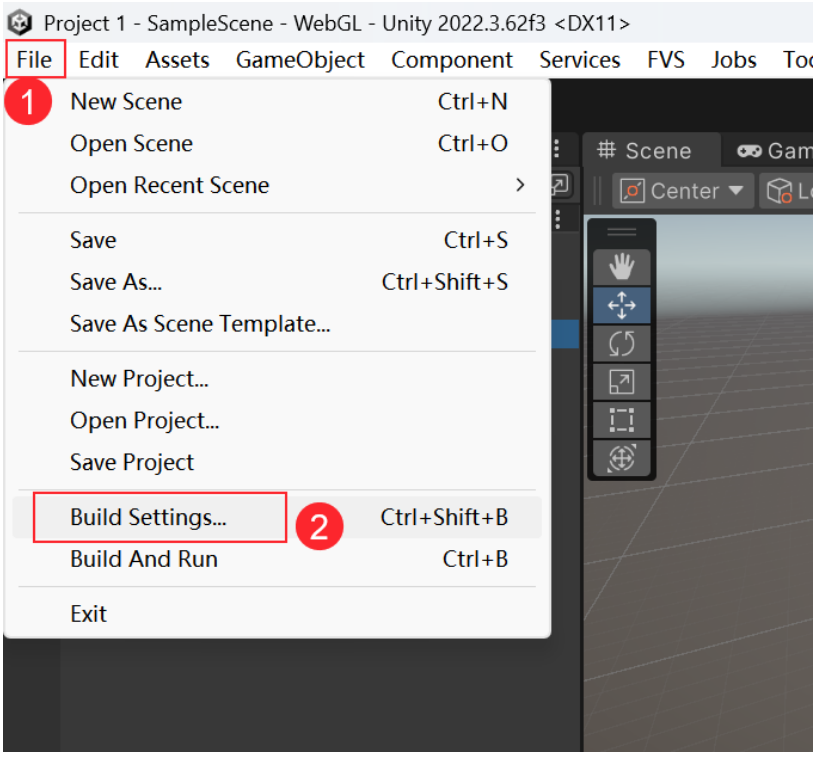
Then select FVSMain, and you will see additional content in its configuration JSON.



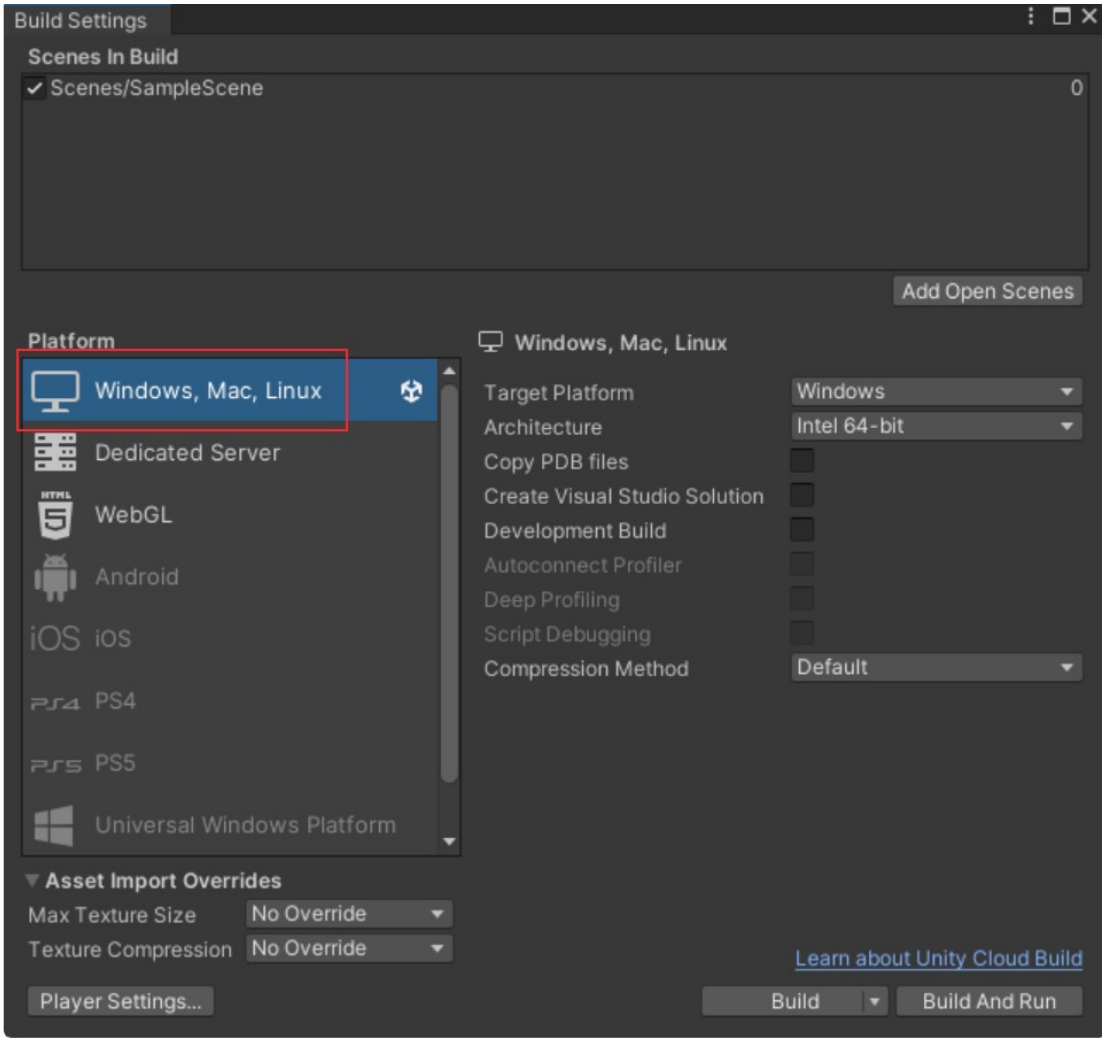
The content is currently empty because we have not added any interactive features yet.

Then save the scene by pressing Ctrl+S, or choose **File** > **Save** from the menu bar.

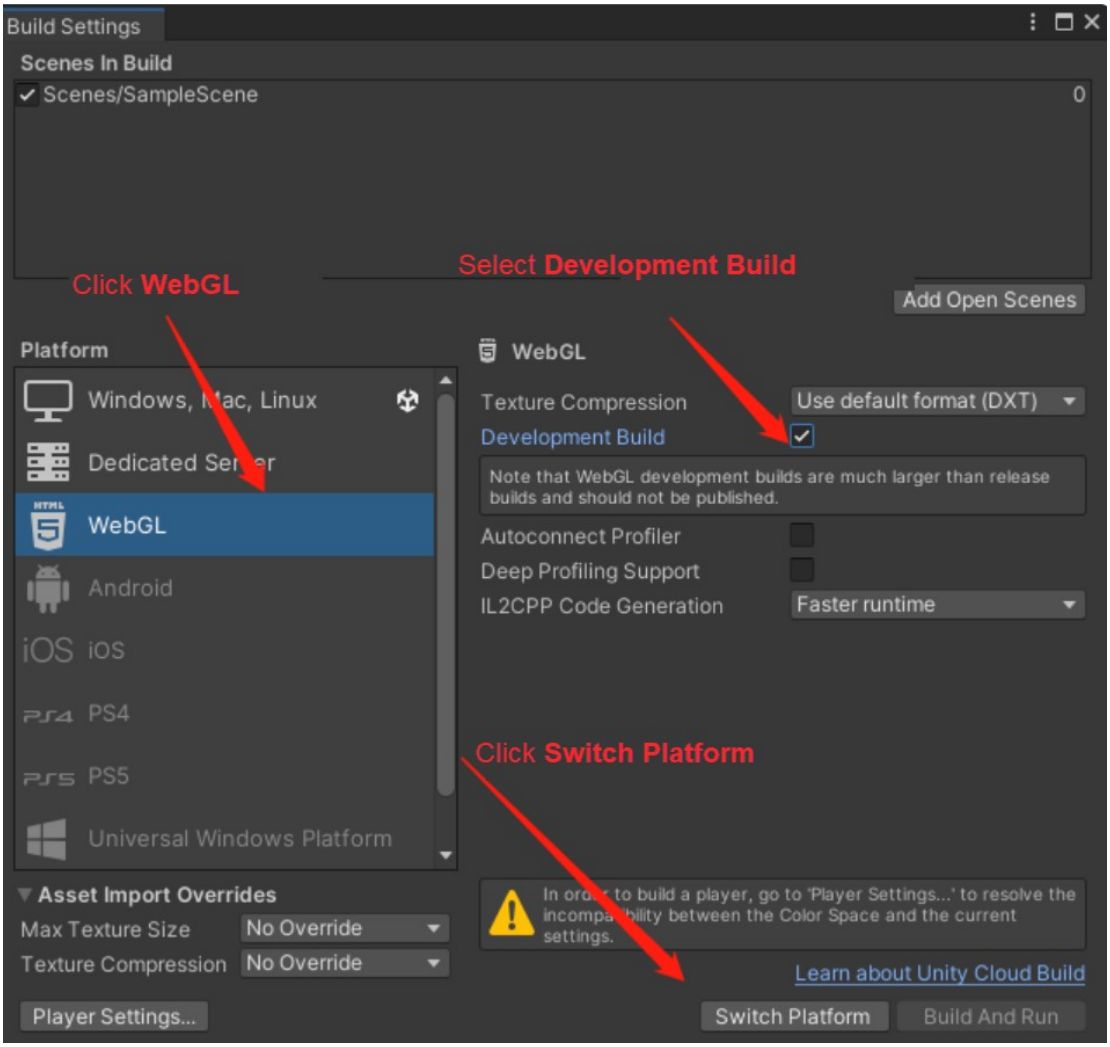
Next, we can start building the project. Choose **File** > **Build Settings** from the menu bar.



By default, Unity builds for the desktop platform.

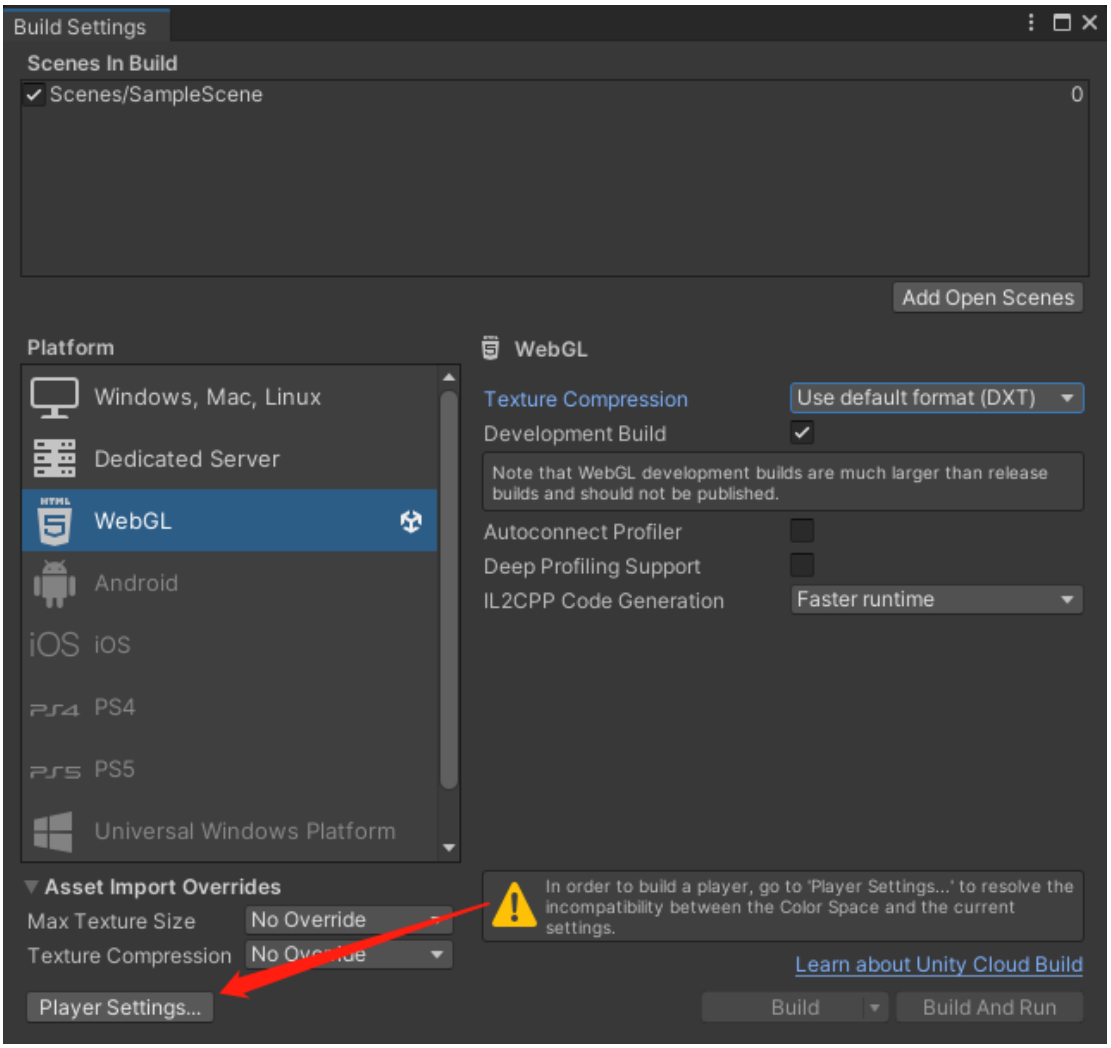


Here we need to switch to the WebGL platform.

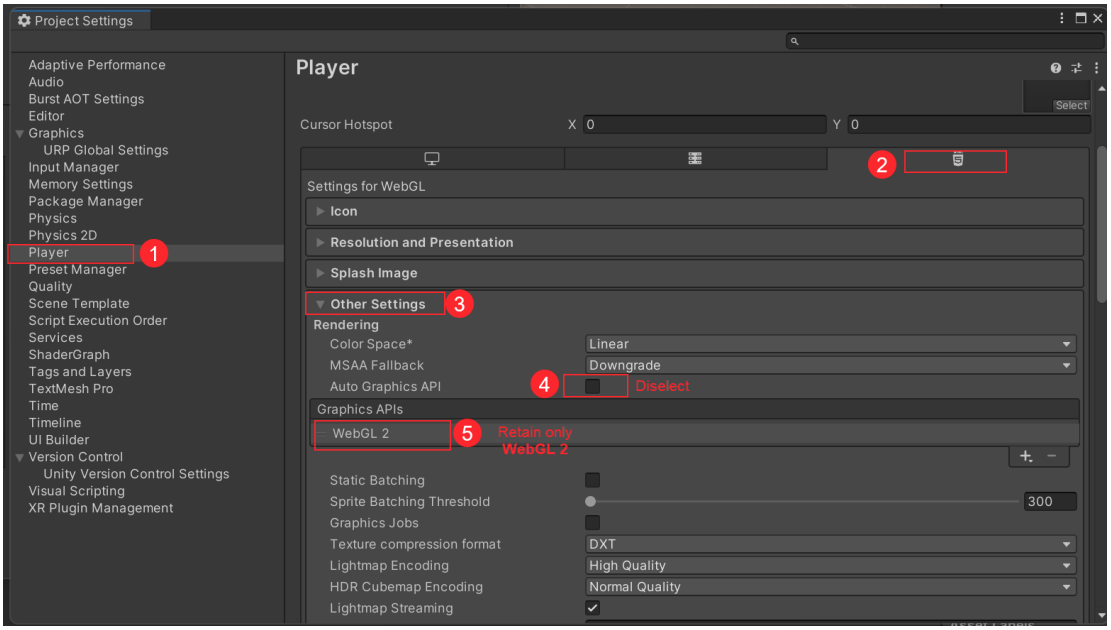


For now, the Unity component of FVS only supports importing projects built with **Development Build**. Non-development builds will be supported in the future.

After a brief script compilation, we find that although we have switched to the WebGL platform, the **Build** button is still not clickable, because of a warning.

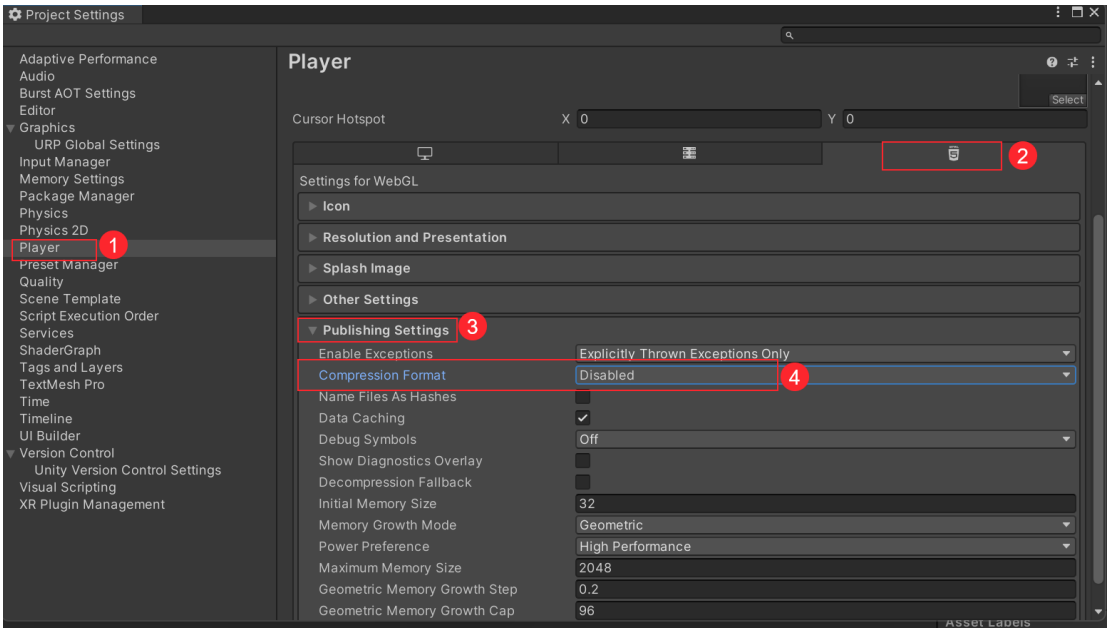


This is due to a color space compatibility issue in the current project. Click **Player Settings** in the lower-left corner, click the **WebGL settings** tab, and expand the **Other Settings** foldout.

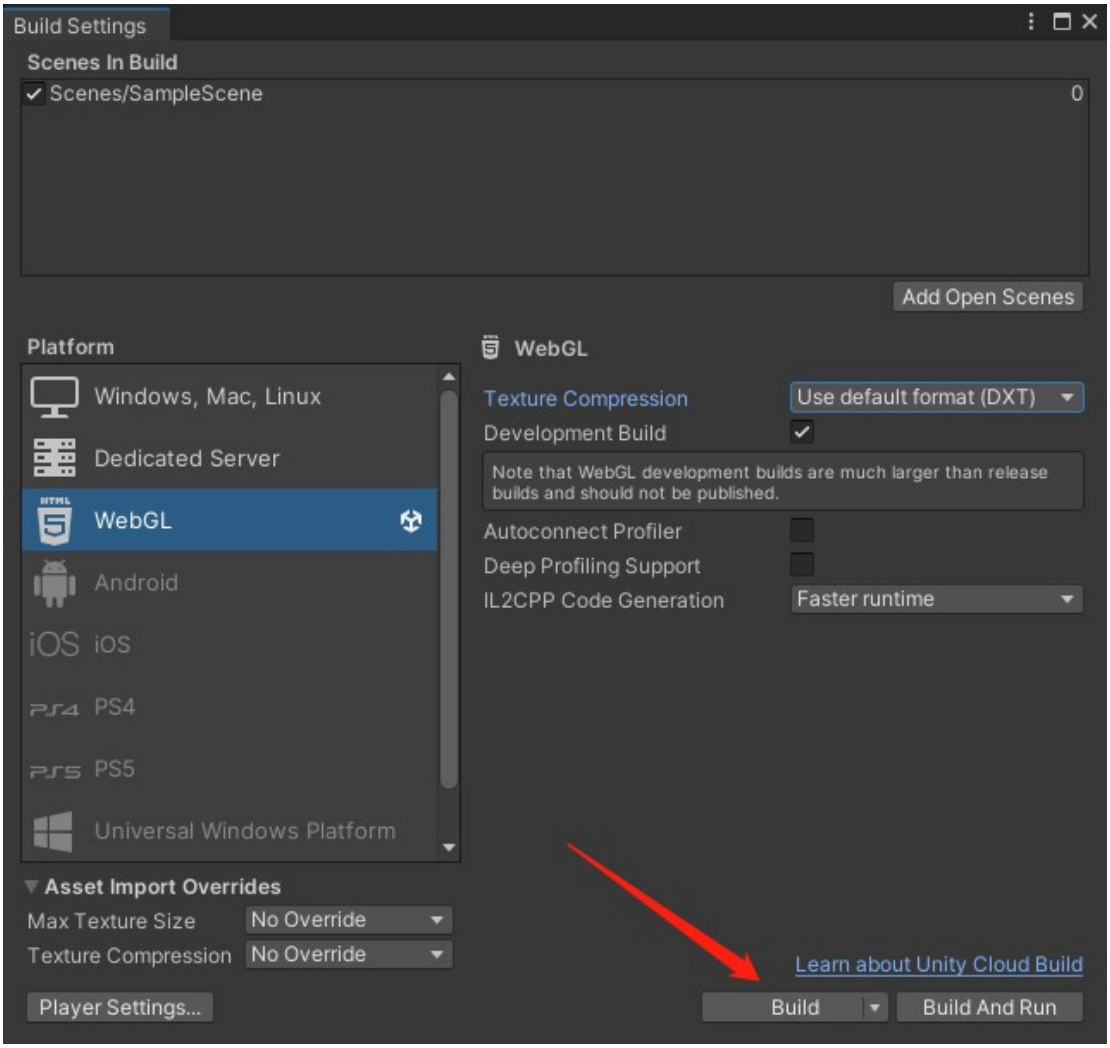


Follow the instructions shown in red in the image, and the warning will then disappear.

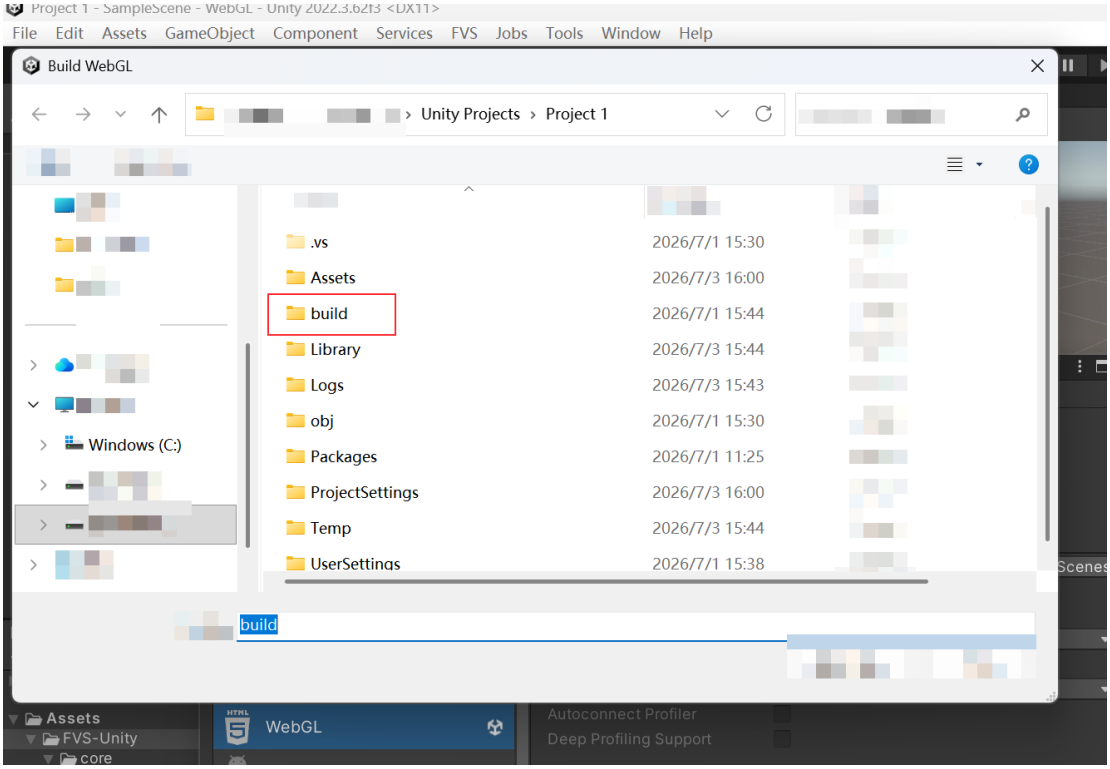
As shown in the following figure, in the expanded **Publishing Settings** foldout, we need to set **Compression Format** to **Disabled**, because the current Unity component of FVS does not support compressed Unity builds.



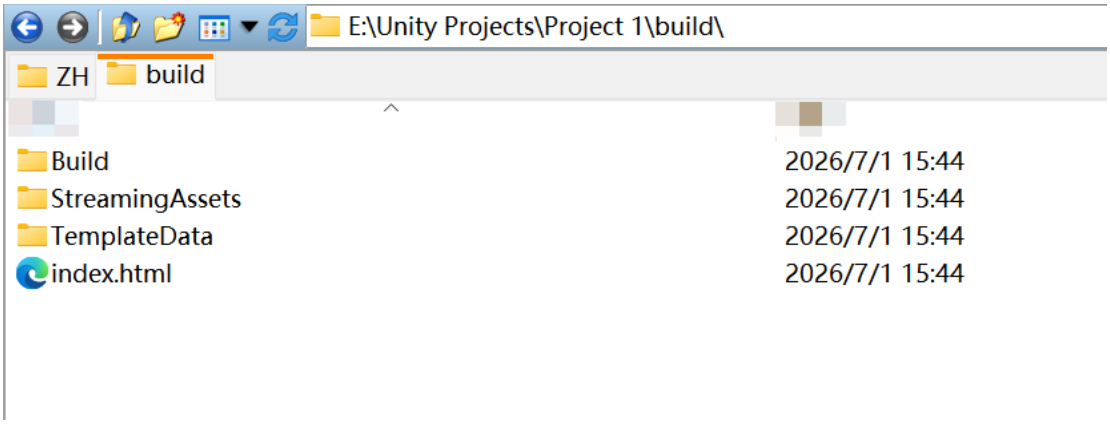
Finally, click **Build** to start the build process.



A file selection panel will appear, letting you choose the build output location. You can create a new folder named **build** under the current project and select it.

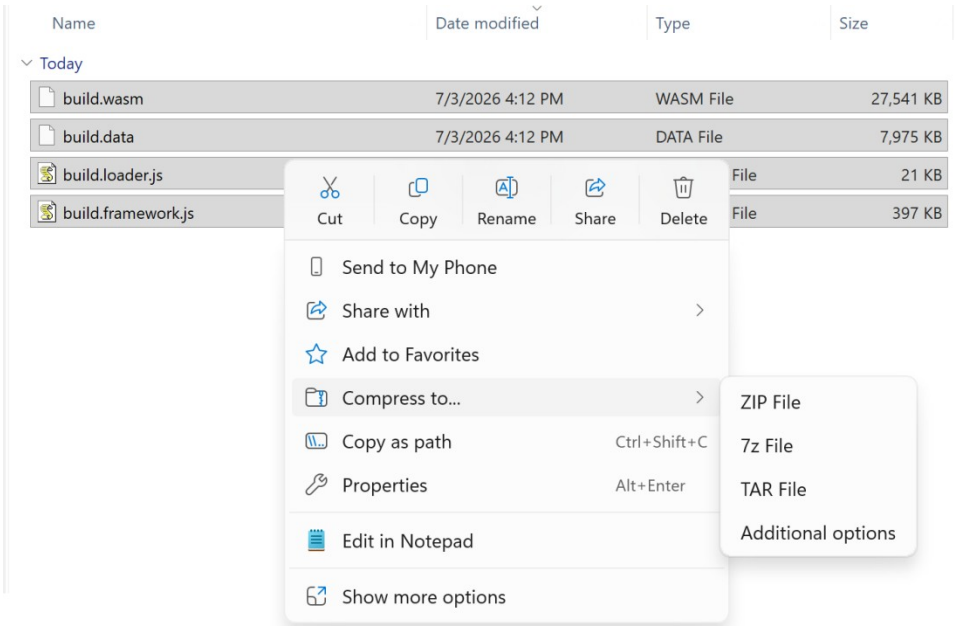


After the building process completes, a number of files will appear in the **build** folder.

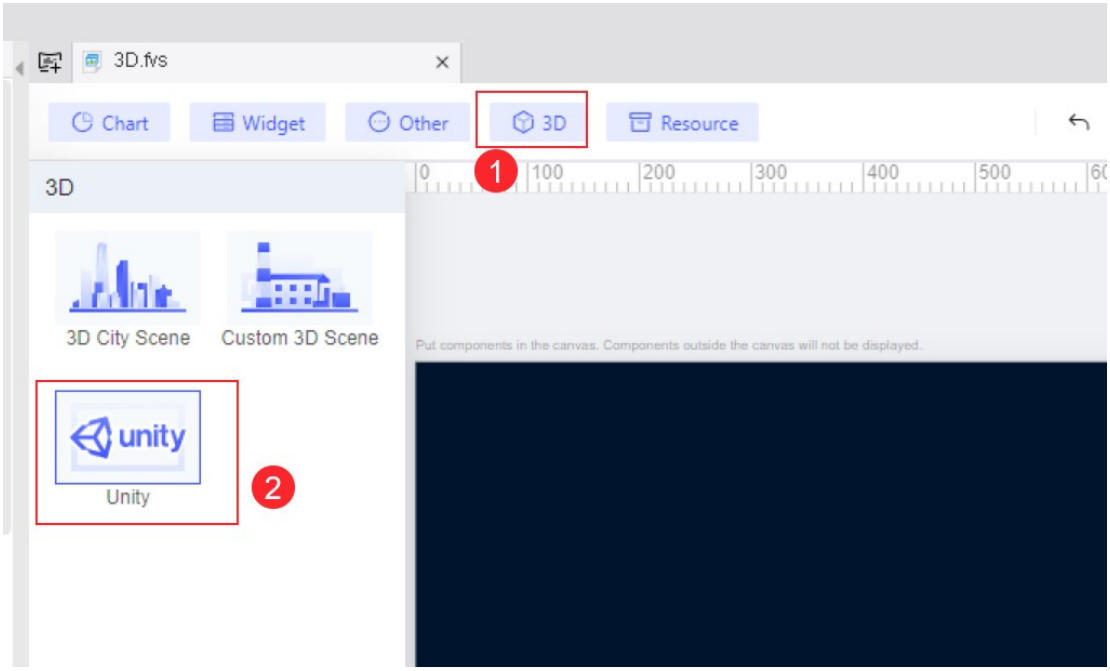


We only need the Unity assets inside the **Build** folder (automatically generated).

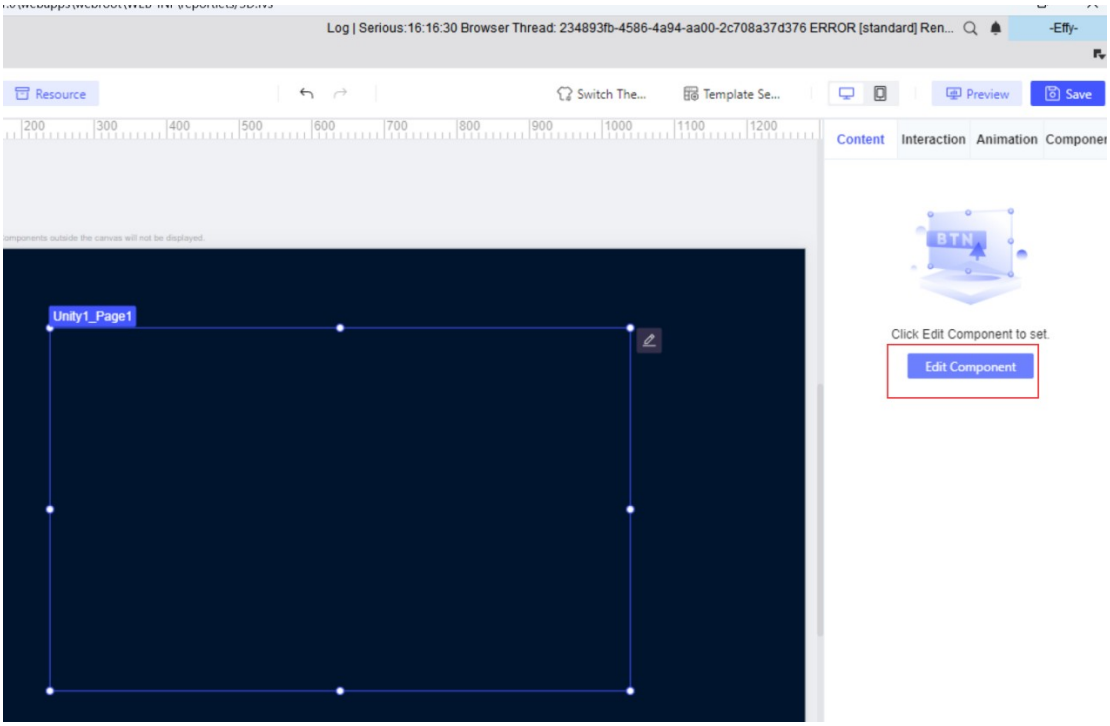
Go into the **Build** folder, select all four files, and compress them to a ZIP file.



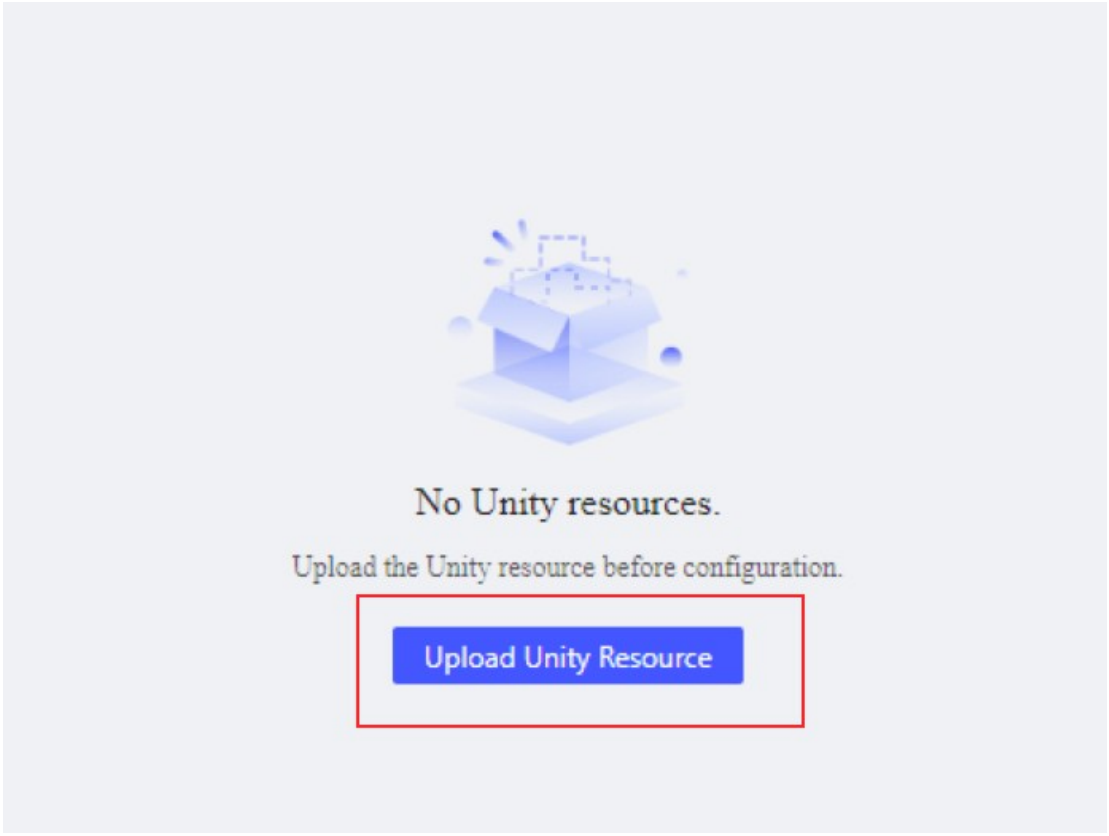
Then, create a new FVS template in the FineReport designer. Click **3D** and then **Unity**.



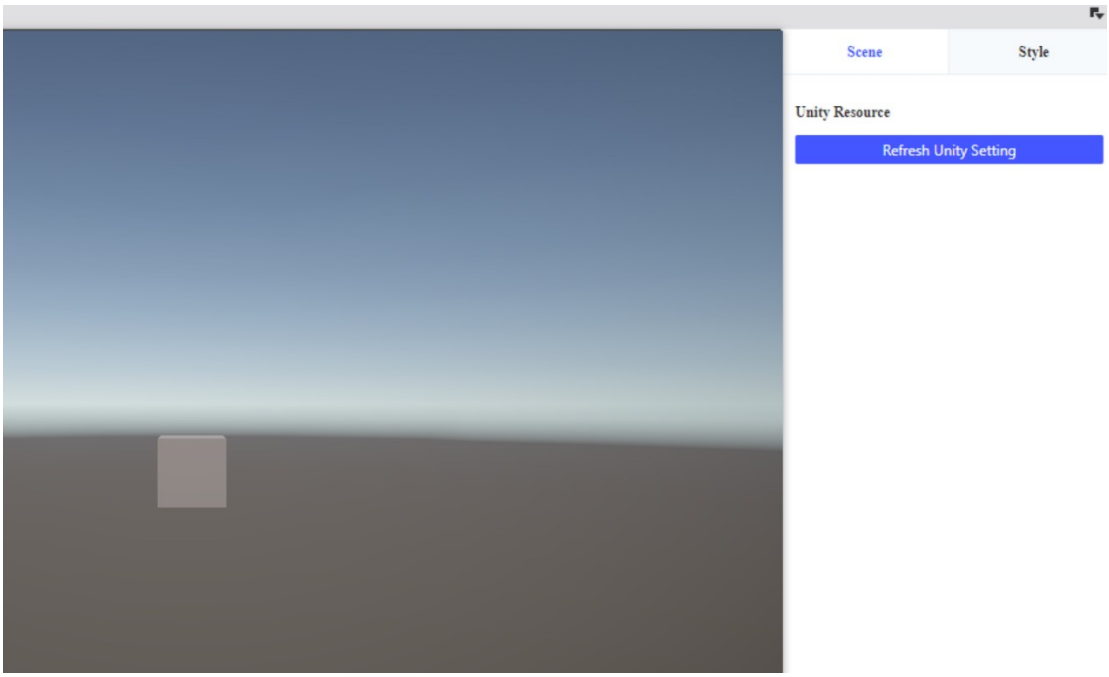
Click **Edit Component** to enter the editing page.



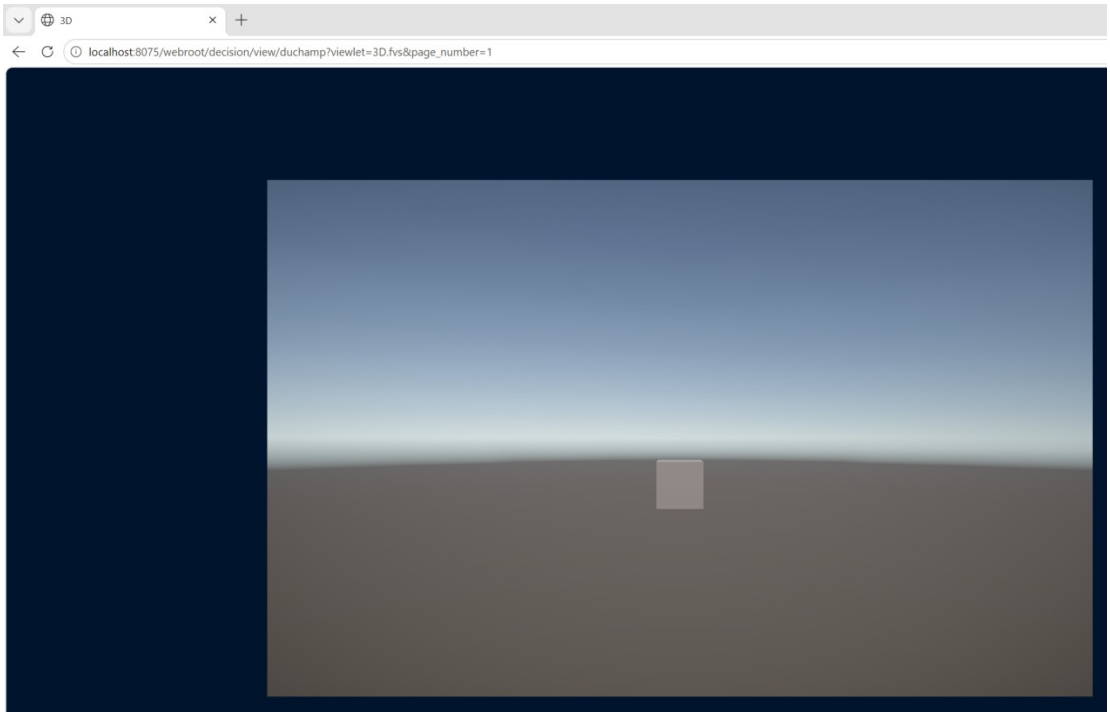
You can see an **Upload Unity Resource** button. Click it and select the ZIP file you just created.



After a short loading process, the built Unity scene is successfully loaded.



Click **Back to Visualization Dashboard**, and you can save and preview the template with the Unity scene.



Summary

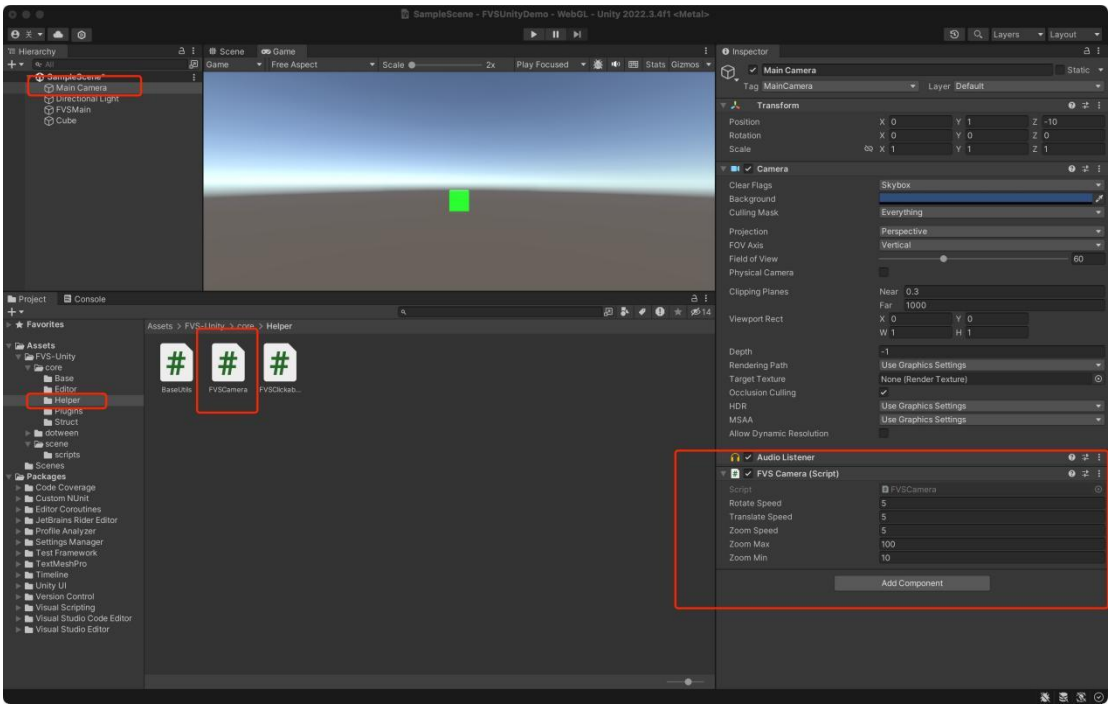
This document covers the basic installation and usage of FVS-Unity-SDK,

as well as how to generate the configuration files and package them for importing into FVS.

Of course, simply displaying a Unity scene is only the beginning. In other documents, we will go into detail about other common features the SDK provides for dashboard 3D scenes.

Adjusting the View

To adjust and save the view of a Unity scene in FVS, attach the **FVSCamera** script from FVS-Unity-SDK to the **Main Camera** in the scene.



The FVSCamera script is located in **Assets/FVS-Unity/core/Helper.**

Changing the FVS Camera property values in the **Inspector** lets you control mouse sensitivity and zoom range when adjusting the view.

After building the Unity project, you can import the build to the **Unity** component of FVS. You can then use the mouse to adjust the scene view in the **Unity** component.

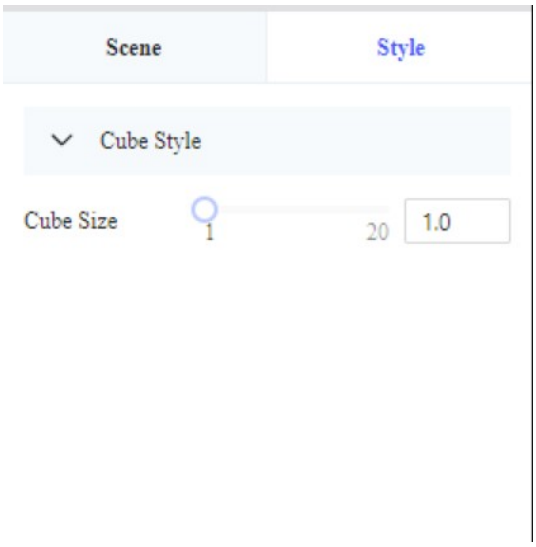
Custom Style

Why Do You Need Custom Styles?

In many cases, you may want to adjust the style of objects in a scene — such as color, size, position, and so on.

These properties are typically determined during Unity development. Once the project is built for the Web platform, users only receive a compressed file — they have no way to customize styles on their own.

To address this, FVS-Unity-SDK provides a custom styling API. By following the specified coding conventions, you can export resource files that, when imported into the FVS Unity component, will automatically generate a style panel with live style bindings. For example, the SDK's built-in default demo (scene 1) generates the following style panel after being built and imported into the Unity component.

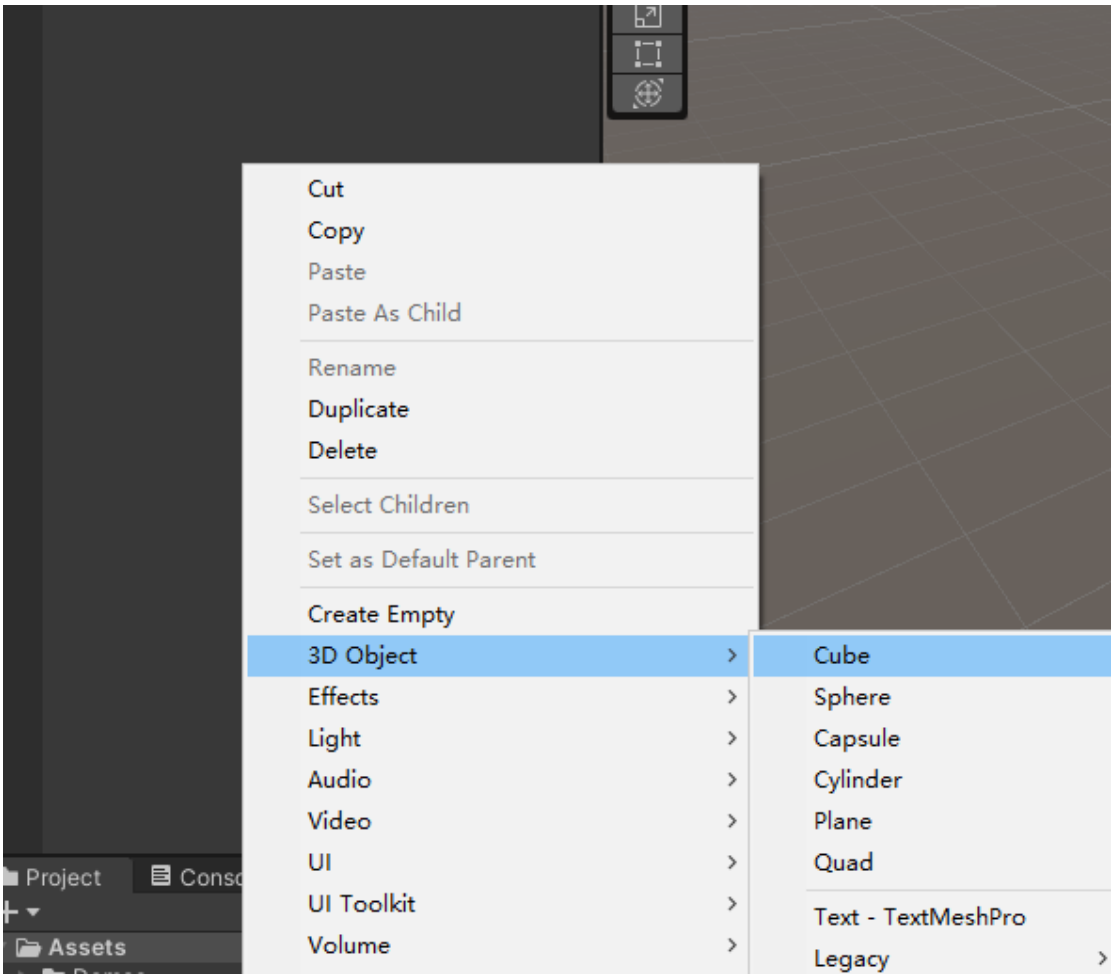


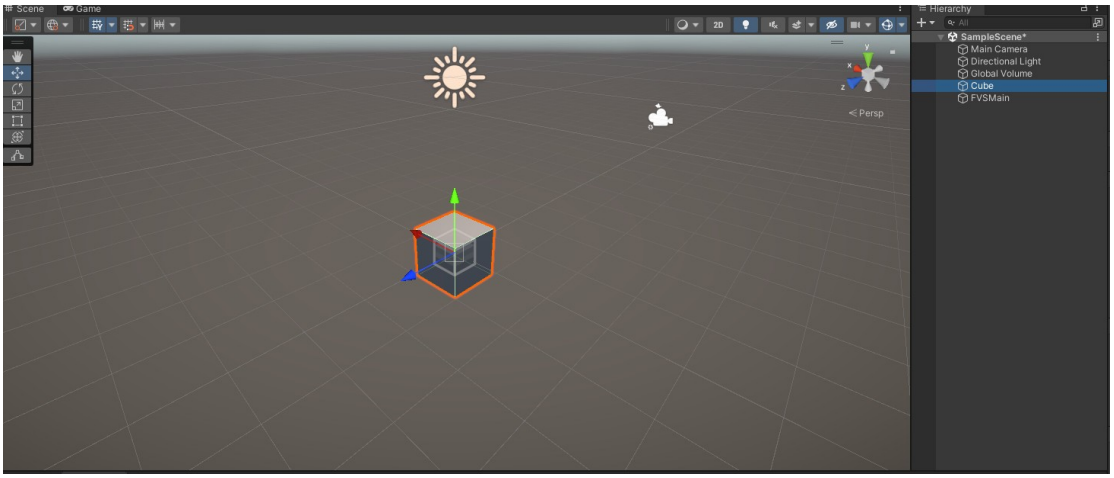
At this point, you can freely adjust styles through the panel, see the scene update in real time, and save your style changes for preview.

Implementing this feature is fairly straightforward. Let's walk through creating a simple custom style — controlling the size of a Cube.

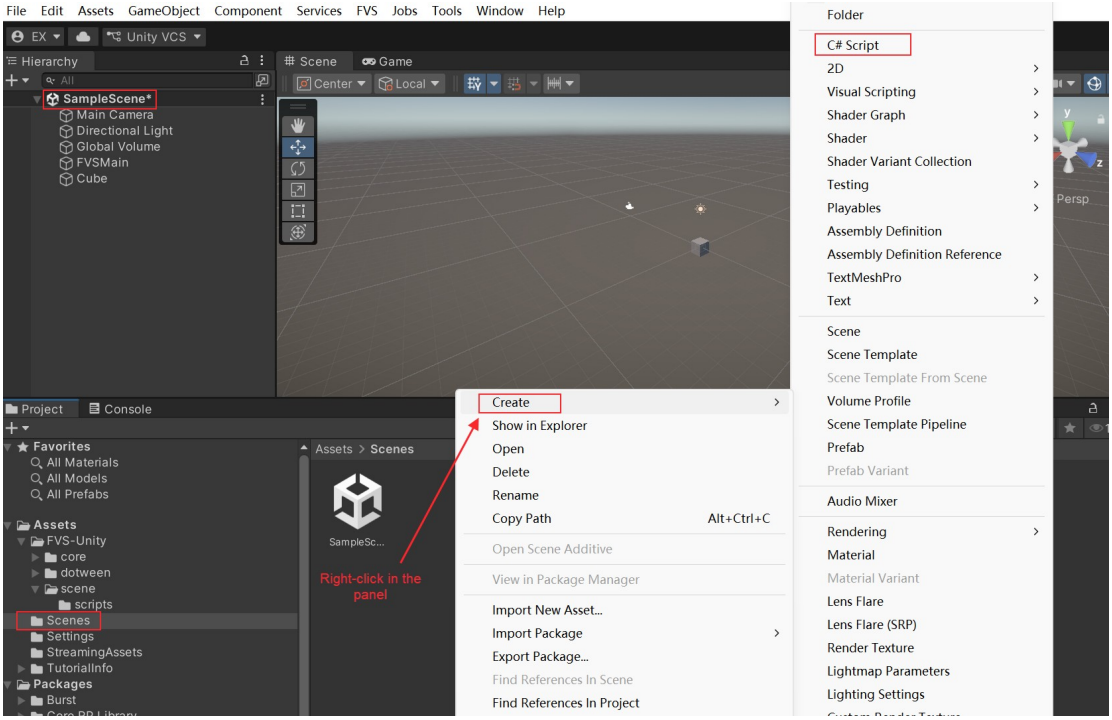
Style Development: Custom Size

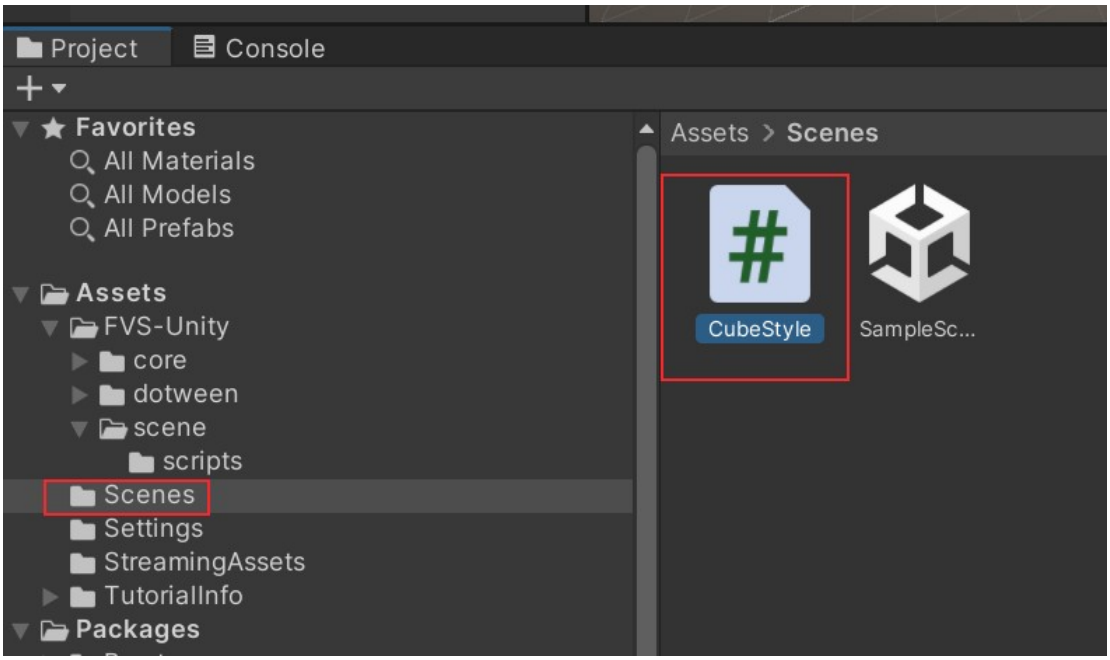
First, right-click in the **Hierarchy** panel, and choose **3D Object** > **Cube** in the shortcut menu to add a Cube.





Next, create a C# script and rename it **CubeStyle** to control the Cube's size.





Double-click **CubeStyle** to open Microsoft Visual Studio.

Write the following code:

```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using FVS;



public class CubeStyle : FVSBaseStyle

{

    // Start is called before the first frame update
```

```
    [FVSGetConfig("cubeSize", "Cube Size")]
    public NumberType GetCubeSize()
    {
        float scale = transform.localScale.x;
        return new NumberType(scale, 1, 20, 0.1f);
    }

    [FVSSetConfig("cubeSize")]
    public void SetCubeSize(string data)
    {
        float size = float.Parse(data);
        transform.localScale = new Vector3(size, size, size);
    }
}
```

Let's briefly go over what this code does.

Let's start with the namespace at the top.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using FVS;
```

The first three lines are standard Unity built-in objects. The fourth line imports FVS, which gives you access to the objects and methods provided by FVS-Unity-SDK.

Then the following is declared:

```
public class CubeStyle : FVSBaseStyle
{


}
```

FVSBaseStyle is a built-in SDK abstract class that inherits from **MonoBehaviour**, indicating this is a base style configuration class.

Having **CubeStyle** inherit from **FVSBaseStyle**, you can then register custom style-related methods.

Next, you added two methods to **CubeStyle**: **GetCubeSize** and **SetCubeSize**. As the names suggest, one retrieves the Cube's size and the other sets the size.

```
[FVSGetConfig("cubeSize", "Cube Size")]
public NumberType GetCubeSize()
{
```

```
    float scale = transform.localScale.x;
    return new NumberType(scale, 1, 20, 0.1f);
}

[FVSSetConfig("cubeSize")]
public void SetCubeSize(string data)
{
    float size = float.Parse(data);
    transform.localScale = new Vector3(size, size, size);
}
```

The Get and Set methods each have an attribute annotation — namely, **FVSGetConfig** and **FVSSetConfig**. These two annotations are used to bind additional information to the Get and Set methods, as described below:

1. **FVSGetConfig** takes two parameters: one is the property key, and the other is the property's **displayName**. The former is used for storage, while the latter serves as the title displayed when the panel is generated.
2. **FVSSetConfig** only needs a single key (note that it must match the key used in **FVSGetConfig**).

With these two annotations, the SDK can identify the property key, **displayName**, and the Set and Get methods for a custom style — which is essential when saving and updating styles.

Moreover, the GetCubeSize method returns **NumberType**.

This indicates that the **cubeSize** property is a numeric type. When configuring detailed style settings, the Unity component in FVS will generate a numeric widget for the **cubeSize** property.

new NumberType(scale, 1, 20, 0.1f) means the default value of the numeric widget is **scale**, the minimum value is **1**, the maximum value is **20**, and the step size is **0.1**. If you don't need a numeric slider widget, simply omit the last three parameters.

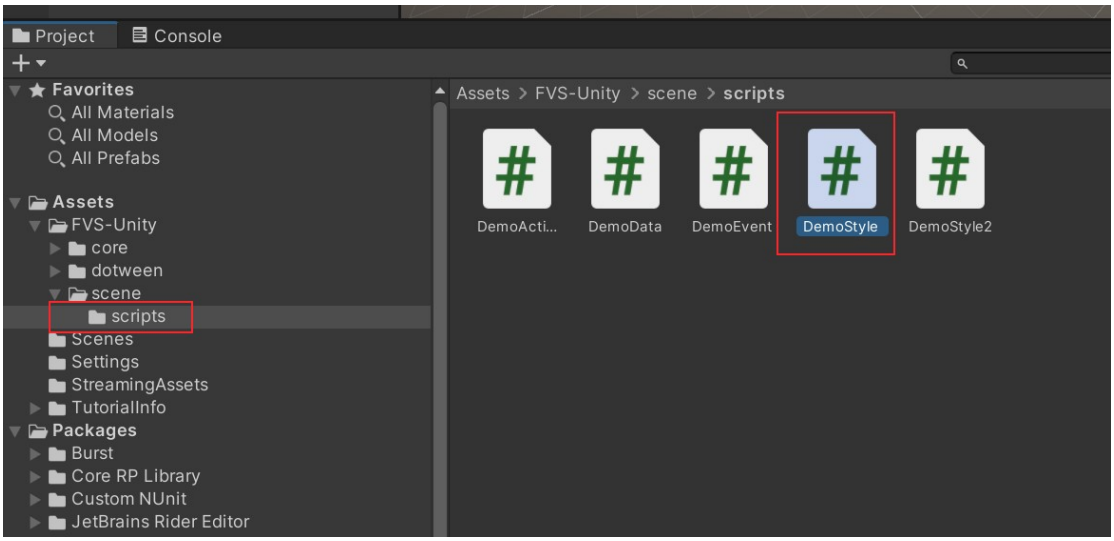
The SetCubeSize method accepts a string by default, so you need to first convert it to a numeric type using **float.Parse** before using it.

In addition to **NumberType**, the SDK also supports other style types, such as color, 3D vector, text, and drop-down boxes.

1. **ColorType**: The Get method works with a Unity color value, while the Set method takes a hex color string. A utility method is required to convert the hex string into a Unity color before use.

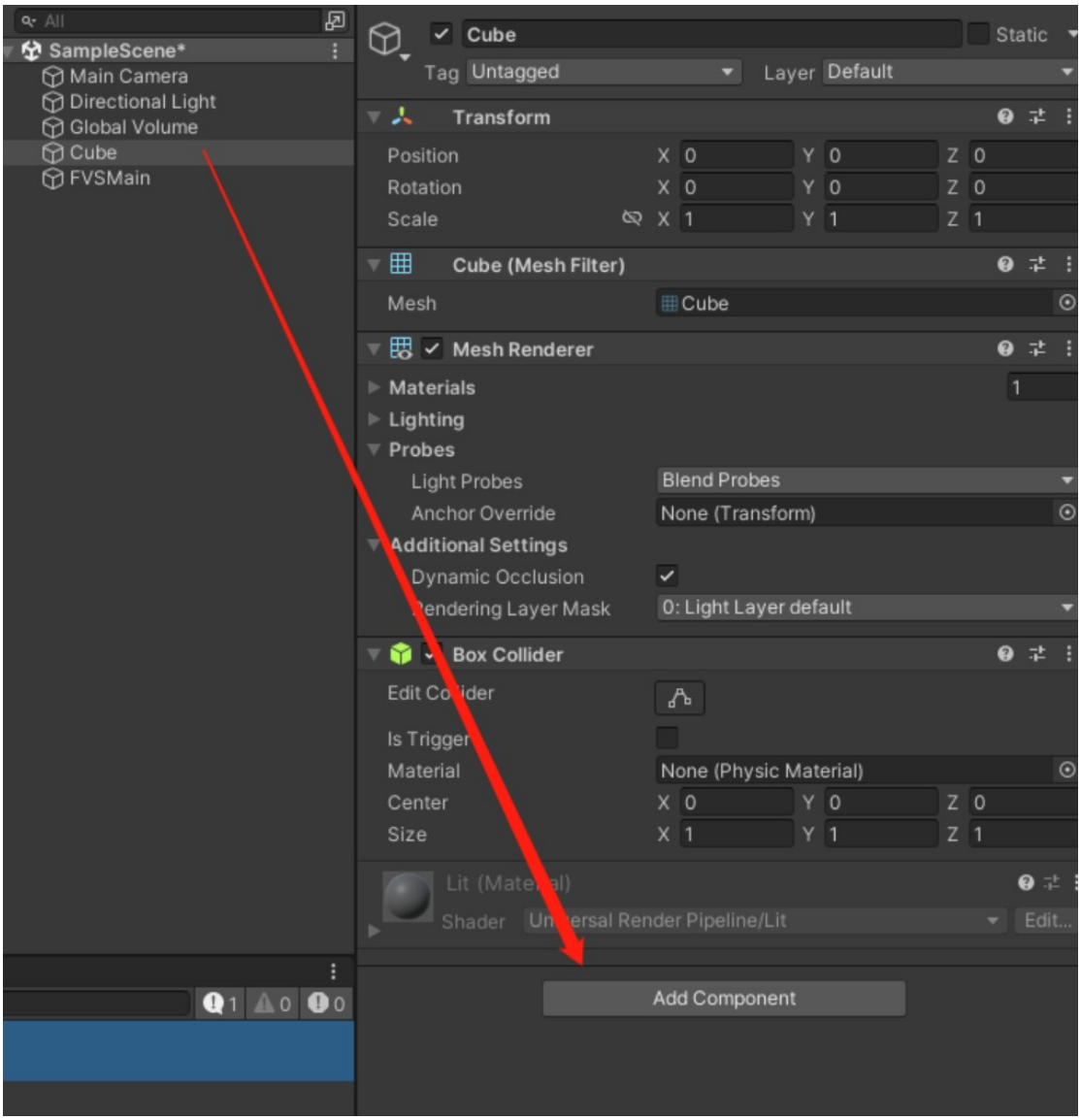
2. **Vector3Type**: is initialized with three numeric values. The Set method takes an array string, which can be converted into a numeric array using LitJSON’s built-in methods.
3. **TextType**: is initialized with a string.
4. **SelectType**: is initialized with an initial string value and a list of options.

For specific code, you can refer to the SDK's built-in style setting script **DemoStyle**.

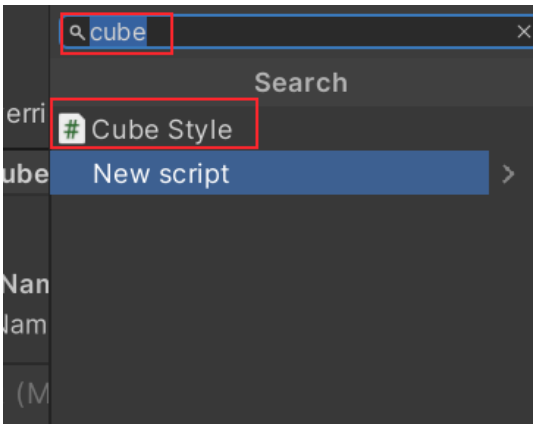


At this point, you have completed a basic numeric-type custom style script (for Cube size). Next, attach it to the Cube.

Select the Cube in the **Hierarchy** panel, and then click **Add Component** in the **Inspector** panel.

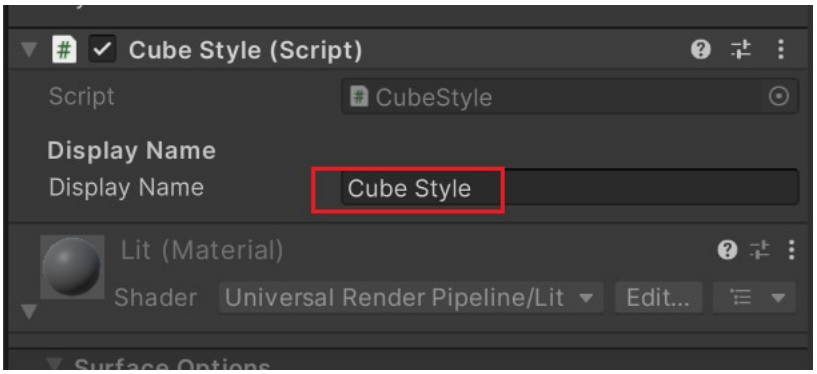


Then type **cube** to search for and add the C# script **CubeStyle** you just wrote.



After adding it, you'll see a display name input field on the **CubeStyle** script. This is used to define the title of our style group — each custom style script is a style group.

You can input **Cube Style**.

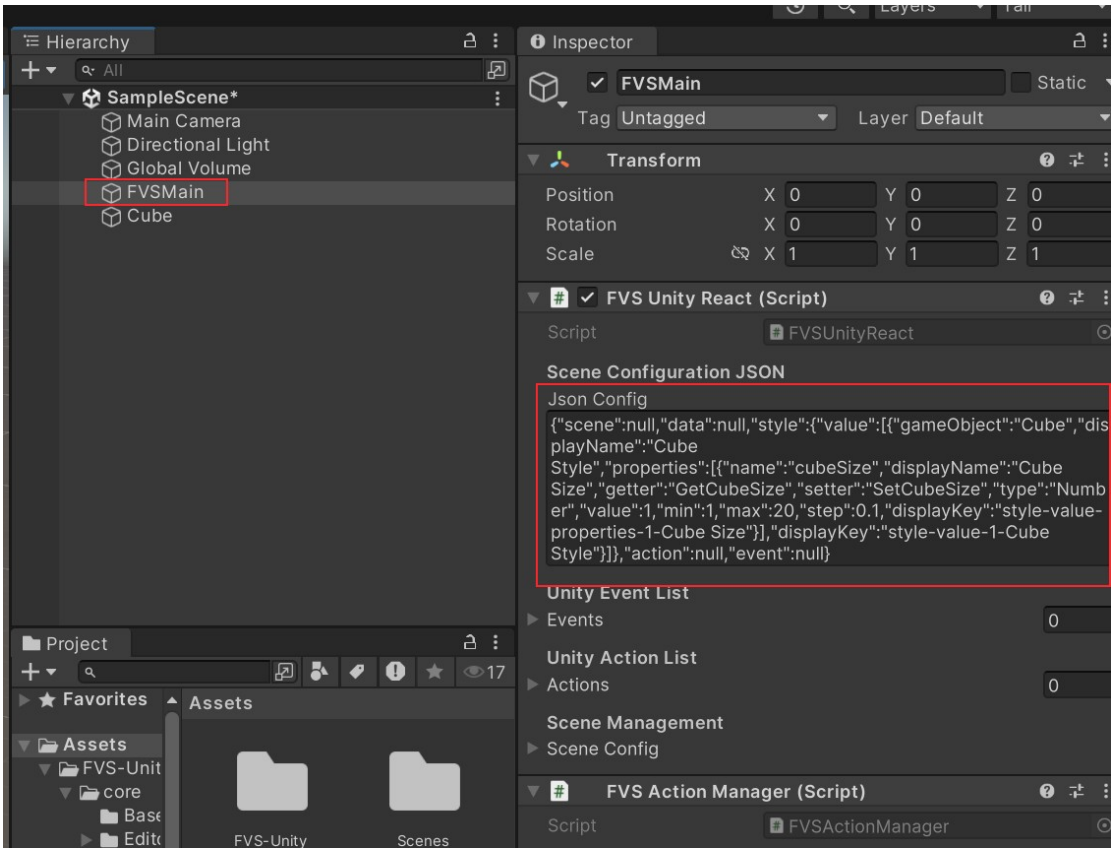


Building & Test

Now that you have developed the custom size style control script, you can build the Unity project containing the script and test it in FVS.

Before building, you still need to generate the scene configuration first — you can refer to the chapter "Installation and Building."

After the building process is complete, you'll notice that the JSON configuration on FVSMain has changed — it now includes style configuration information.



Let's copy this JSON and format it — the result looks like this:

```
{
 "scene": null,
 "data": null,
 "style": {
        "value": [{
```

```
            "gameObject": "Cube",
            "displayName": "Cube Style",
            "properties": [{
                "name": "cubeSize",
                "displayName": "Cube Size",
                "getter": "GetCubeSize",
                "setter": "SetCubeSize",
                "type": "Number",
                "value": 1,
                "min": 1,
                "max": 20,
                "step": 0.1
            }]
        }]
    },
    "action": null,
    "event": null
}
```


The property named **style** stores the basic information for the cube size style you created earlier.

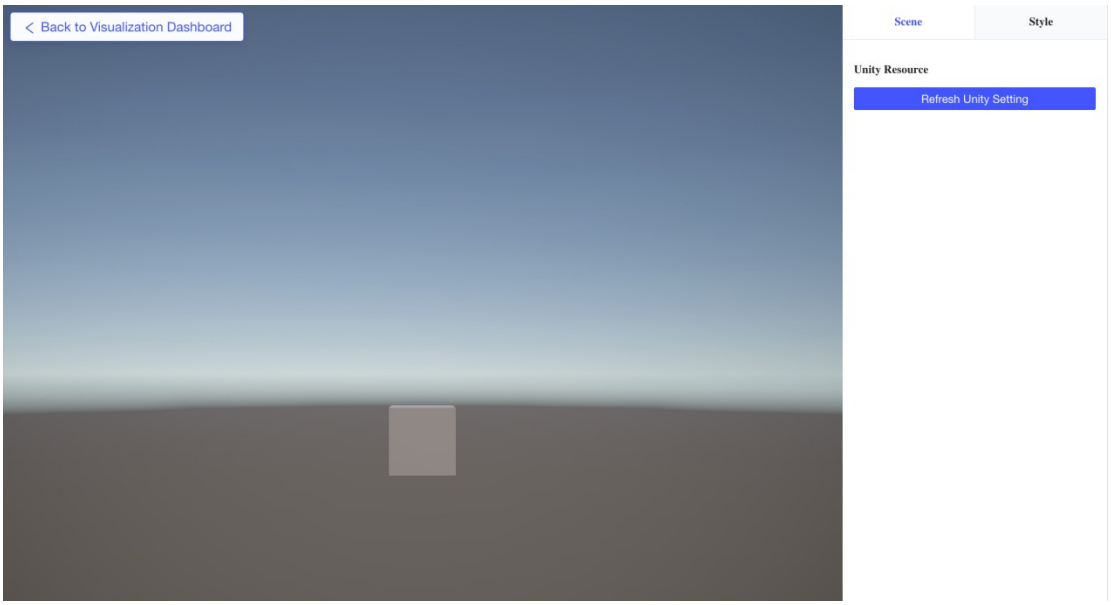
- **gameObject**: GameObject to which the style script is attached
- **displayName**: Style group name — **Cube Style** that you entered in the **Display Name** field of the **Cube Style** script
- **properties**: Detailed style group. In this example, you only set one group **cubeSize**, and you can see its corresponding style details.

For styles like color and text, the compiled configuration looks similar.

Next, save the scene and start building. After the building process is complete, add the build to a compressed file, as mentioned in the "Installation and Building" chapter.

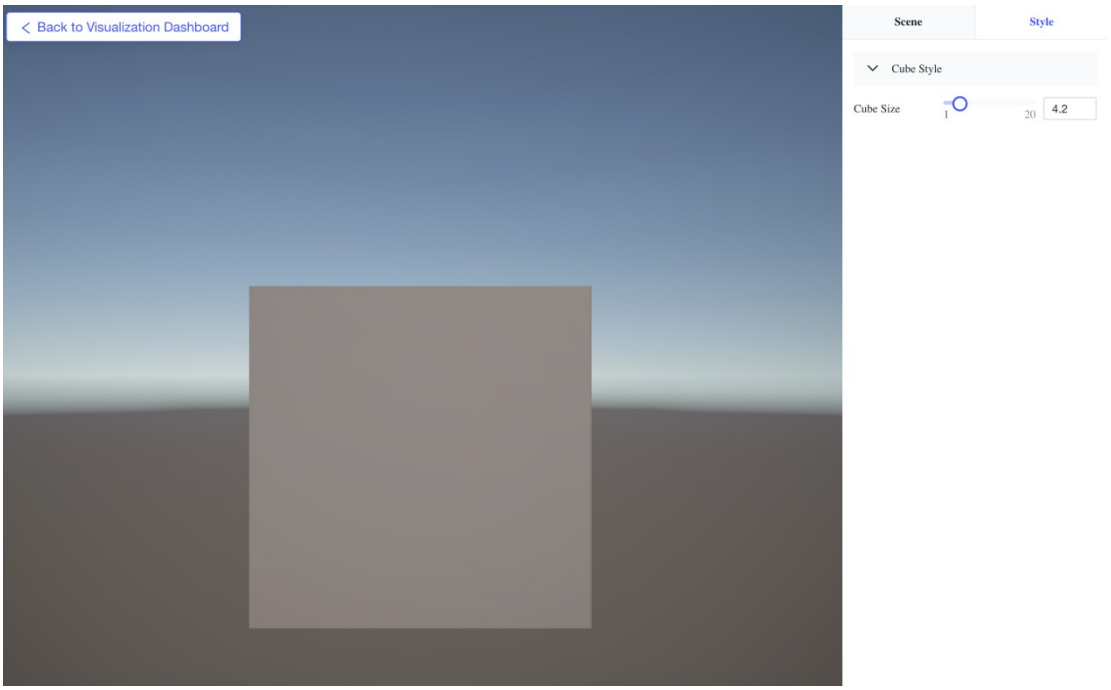
Then return to FVS, and enter the Unity component editing page — in the lower-right corner you'll find a **Replace Unity** button.

Click it and select the new build file. After a short loading animation, you'll see the new scene.



You'll also notice that the configuration panel on the right has a new **Style** tab. Switch to the tab, and you'll see the **Cube Size** widget you just created.

Drag the slider, and you'll see the Cube's size in the scene change accordingly.



The WYSIWYG custom style feature is now complete.

Connecting Dynamic Data

Differences Between Dynamic Data and Custom Styles

In the previous chapter, you learned how to implement custom styles through the FVS-Unity-SDK. In this chapter, you will learn how to connect dynamic data to a Unity scene in FVS.

Although custom styles are highly capable, they still require manual adjustment each time.

You may want certain content in the scene to update automatically based on rules. This is when you need to connect dynamic data.

Next, you will connect data to a Unity scene to control a Cube's position and achieve simple data-driven control.

Developing the Data Integration Script

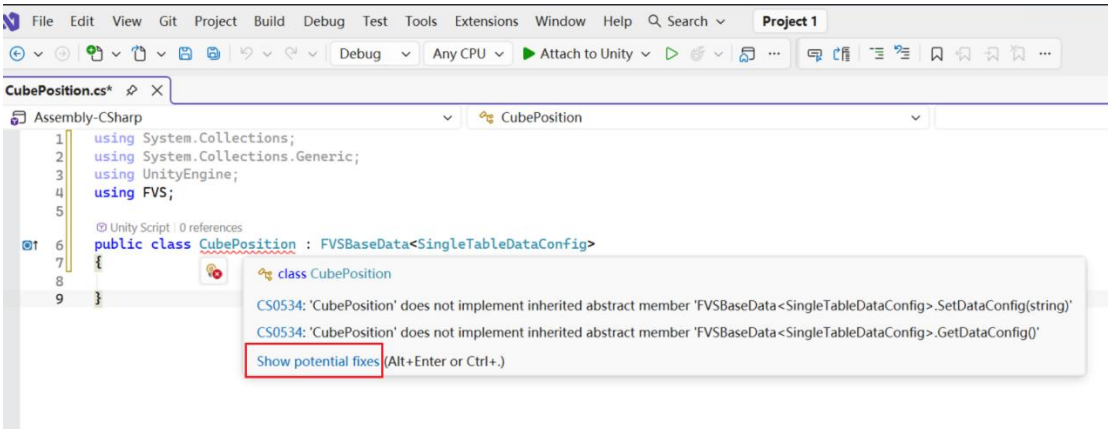
First, create a new script named **CubePosition.cs** in the same directory as **CubeStyle.cs**.

Double-click **CubePosition.cs** and enter the following content:

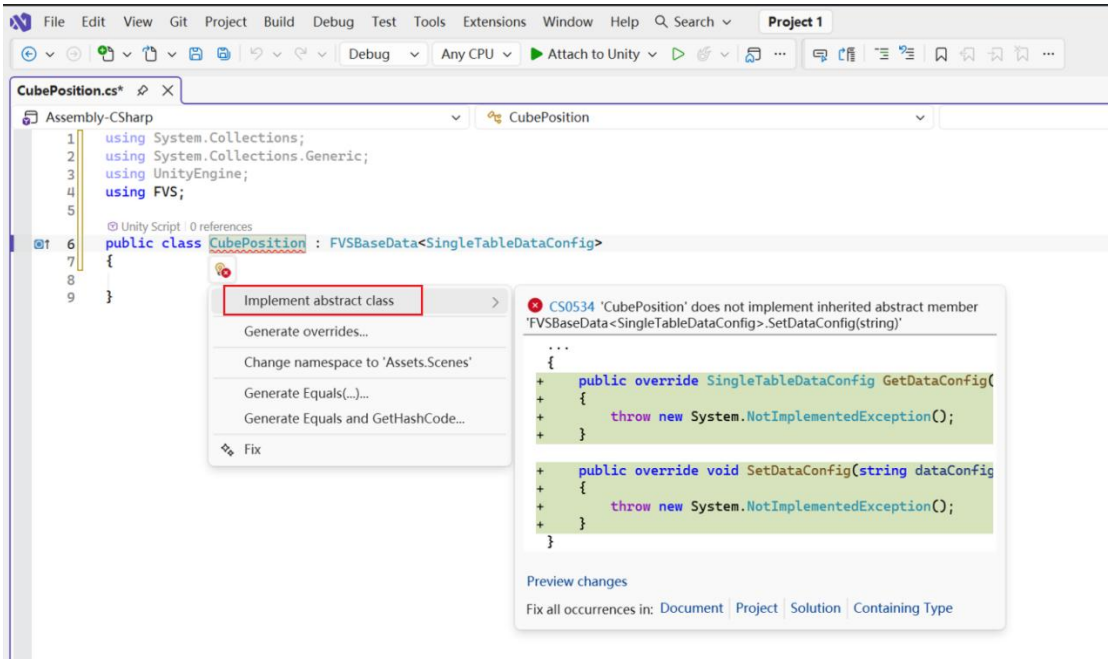
```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using FVS;



public class CubePosition : FVSBaseData<SingleTableDataConfig>

{

}
```

This code means that CubePosition inherits from FVSBaseData, indicating that it is a data configuration class.

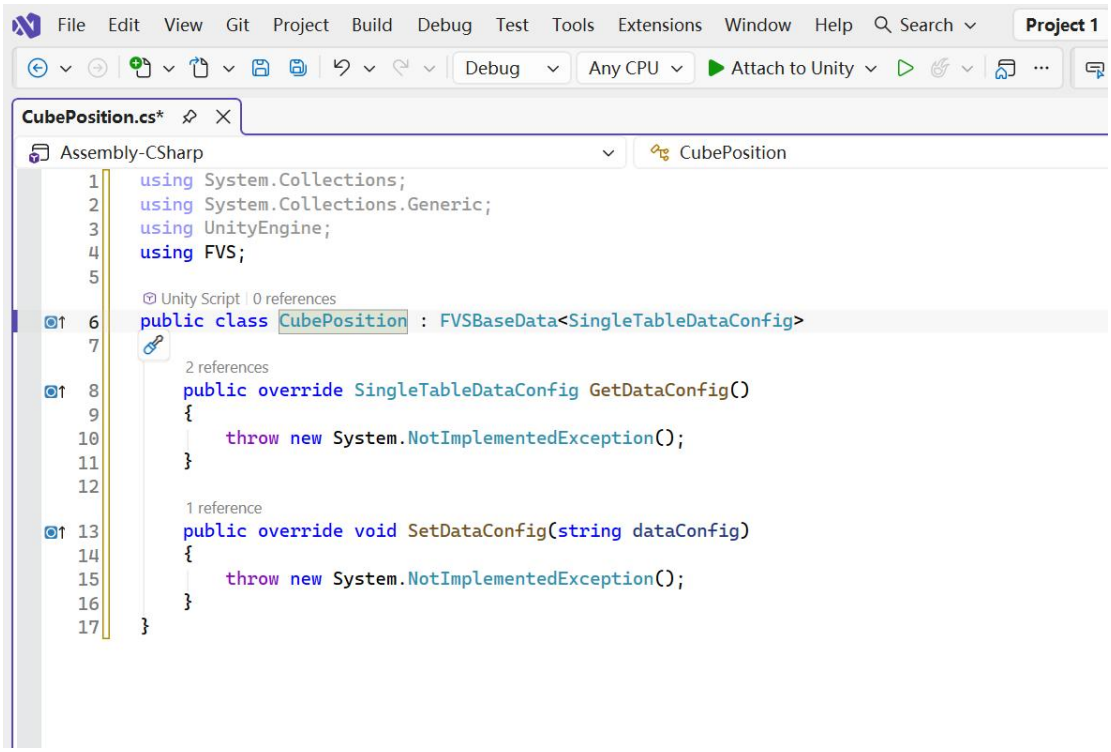
After you entered the code, Visual Studio will display a warning, as shown in the following figure. Click **Show potential fixes**.



Then click **Implement abstract class**.



The editor will auto-complete the code.

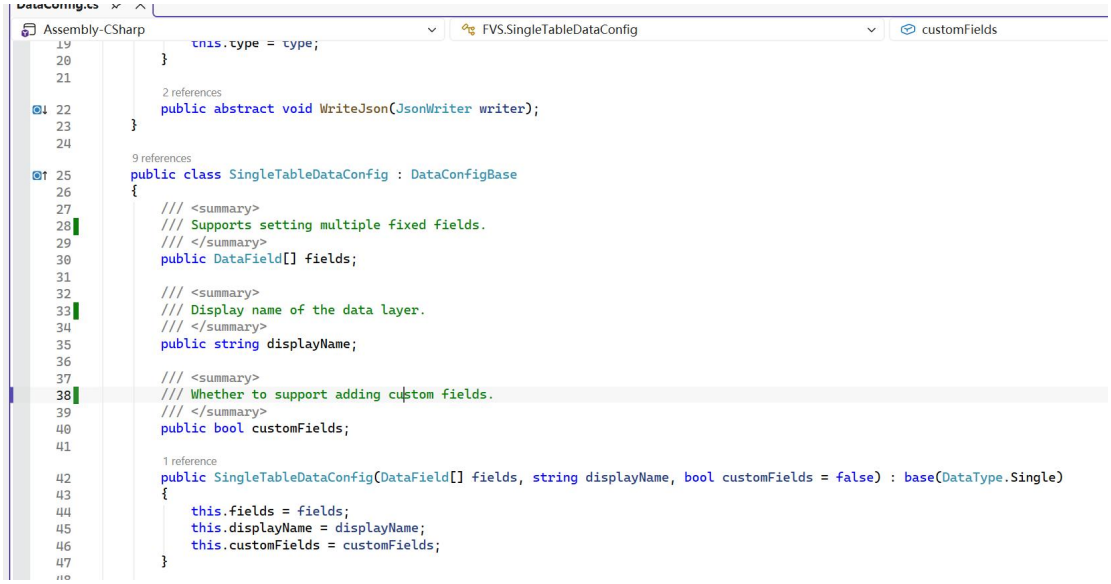


Of course, this only generates the structure. You still need to implement the actual logic yourself.

Before writing the logic, you should first understand what these two methods mean.

The first method, GetDataConfig, is used to define the structure of the required data. It returns a SingleTableDataConfig, which is a single table data configuration.

You can navigate into the SingleTableDataConfig class to view its structure and constructor, as shown in the following figure.



Before using data, you need to specify the data configuration structure for the FVS-Unity-SDK, such as the number of fields and the display name of the data configuration.

Here you only need the Cube's position, which consists of three fields: x, y, and z. You can construct the corresponding data table configuration inside GetDataConfig. The code is as follows:

```
public override SingleTableDataConfig GetDataConfig()

{

   DataField[] fields = new DataField[]

   {

      new DataField("x", "X Coordinate"),

      new DataField("y", "Y Coordinate"),

      new DataField("z", "Z Coordinate"),

   };

   return new SingleTableDataConfig(fields, "Position");

}
```

After configuring the data structure, the next method is SetDataConfig, which is used to receive data sent from FVS to Unity.

The **string dataConfig** parameter is an array of data objects serialized as a string, and its structure corresponds to the GetDataConfig method above.

That is:

```
[

 {

   x: 100,
```

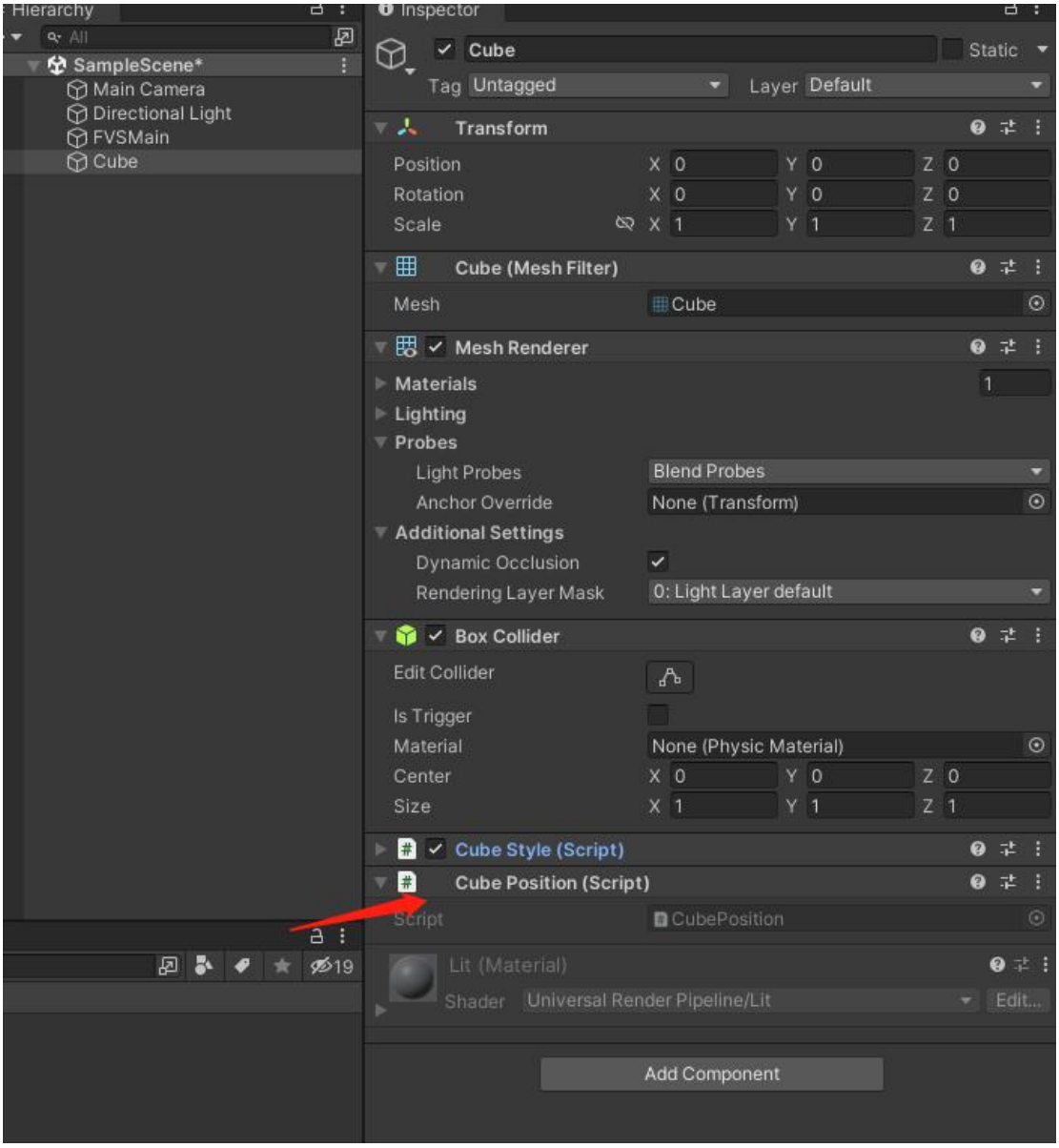
```
    y: 200,
    z: 300
  },
  ...
]
```

You first need to convert it into a normal object structure. The code is as follows:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using FVS;
// Convert data using LitJson.
using LitJson;
public class CubePosition : FVSBaseData<SingleTableDataConfig>
{
    public override SingleTableDataConfig GetDataConfig()
    {
        DataField[] fields = new DataField[]
        {
            new DataField("x", "X Coordinate"),
            new DataField("y", "Y Coordinate"),
            new DataField("z", "Z Coordinate"),
        };
        return new SingleTableDataConfig(fields, "Position");
    }
    public override void SetDataConfig(string dataConfig)
    {
        Datum[] data = JsonMapper.ToObject<Datum[]>(dataConfig);
        // Retrieve only the first record here.
        Datum first = data[0];
        transform.position = new Vector3(first.x, first.y, first.z);
    }
    public class Datum : DatumBase
    {
        public float x { set; get; }
        public float y { set; get; }
```

```
        public float z { set; get; }

    }

}
```

Now that the script for receiving dynamic data updates to the Cube's position is complete, you also need to attach it to the Cube GameObject.



Building & Test

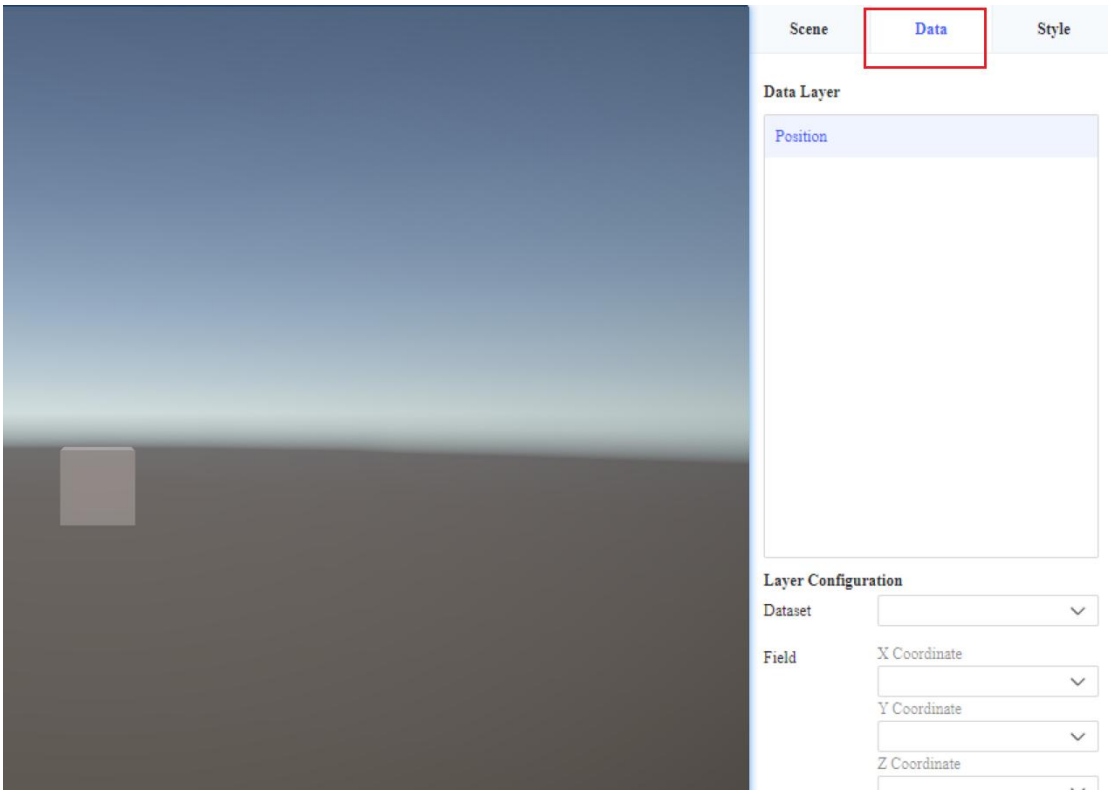
Build the scene configuration again. You will see that the JSON content of **FVSMain** has changed to the following:

```
{
 "scene": null,
 "data": {
     "value": [{
         "gameObject": "Cube",
         "fields": [{
             "name": "x",
             "displayName": "X Coordinate"
```

```
          }, {
               "name": "y",
               "displayName": "Y Coordinate"
          }, {
               "name": "z",
               "displayName": "Z Coordinate"
          }],
          "displayName": "Position",
          "customFields": false
     }]
 },
 "style": {
     "value": [{
          "gameObject": "Cube",
          "displayName": "Cube Style",
          "properties": [{
               "name": "cubeSize",
               "displayName": "Cube Size",
               "getter": "GetCubeSize",
               "setter": "SetCubeSize",
               "type": "Number",
               "value": 1,
               "min": 1,
               "max": 20,
               "step": 0.1
          }]
     }]
 },
 "action": null,
 "event": null
}
```

The detailed data settings have been added. Then you can save the scene, build the project and then replace the existing resources in FVS.

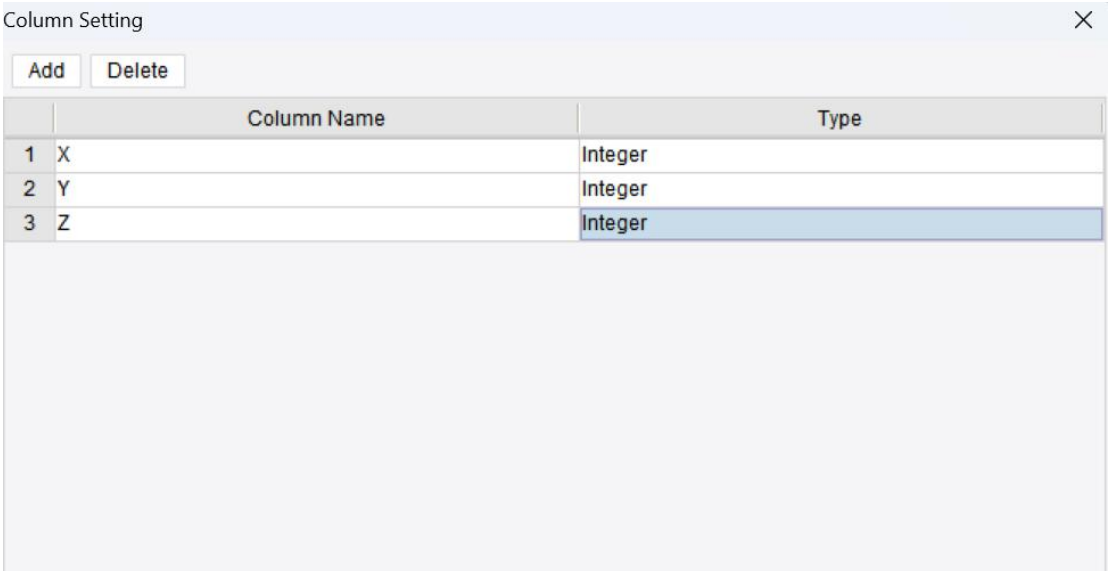
You will see a new **Data** tab page in the right panel.

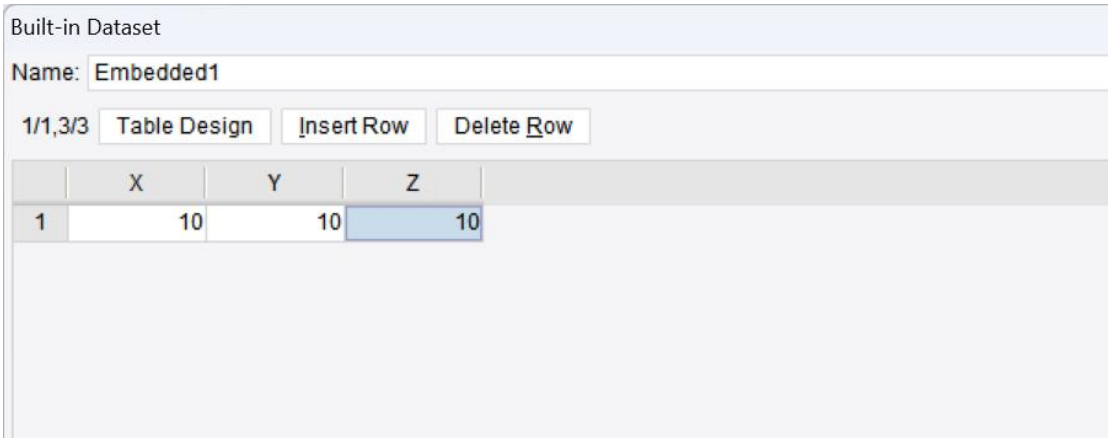


You defined only one data configuration before, so there is only one data layer. In practice, you can define multiple data layers.

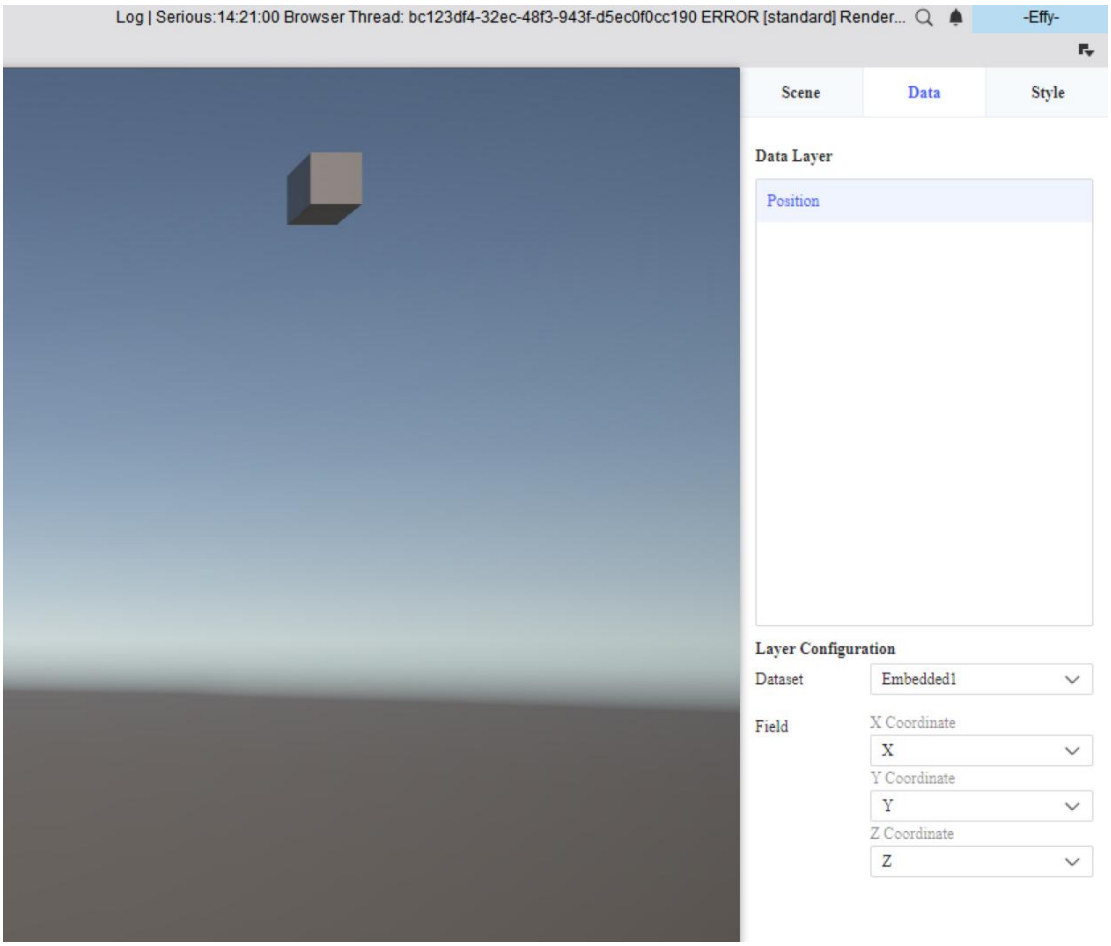
Note that only one data script can be attached to a single GameObject.

In FVS, you can create a built-in dataset for a quick test, as shown in the following figure.





As shown in the following figure, in the **Dataset** drop-down box, select the dataset you just created. In the **Field** area, select the corresponding fields from the X, Y, and Z coordinate drop-down boxes. You will see the Cube's position change accordingly.



This section briefly demonstrated data integration. In practice, data integration can be used to implement dynamic data labels, dynamic crowd distribution, and other features — the principles are the same. The details are not covered here.

Unity Event Pass

In addition to customizing style data, you may want the 3D components in a dashboard to interact with other components.

For example, clicking an element could trigger linkage with other components or navigate to another page.

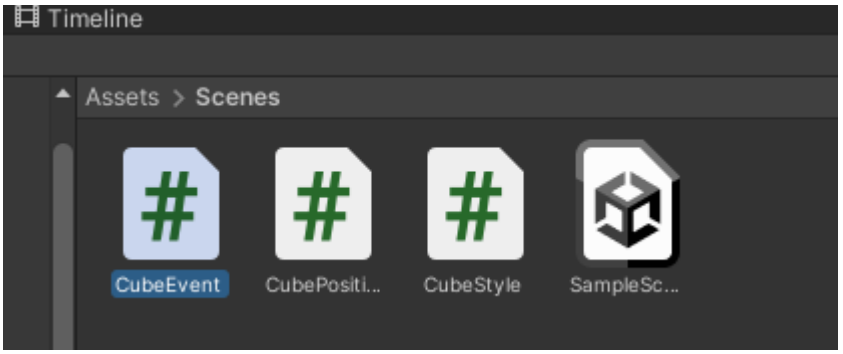
These requirements can also be easily fulfilled using `FVS-Unity-SDK`.

The following takes `Cube` as an example. You can bind a click event to `Cube` so that when `Cube` is clicked in an FVS template, the click event can be passed to FVS to trigger interactions.

Writing the Event Script

Similarly, you first need to create a `C#` script to bind a click event to `Cube` and pass the event to `FVS`.

Create a `C#` script named `CubeEvent`.



Double-click the script to enter the editing page, and enter the following code:

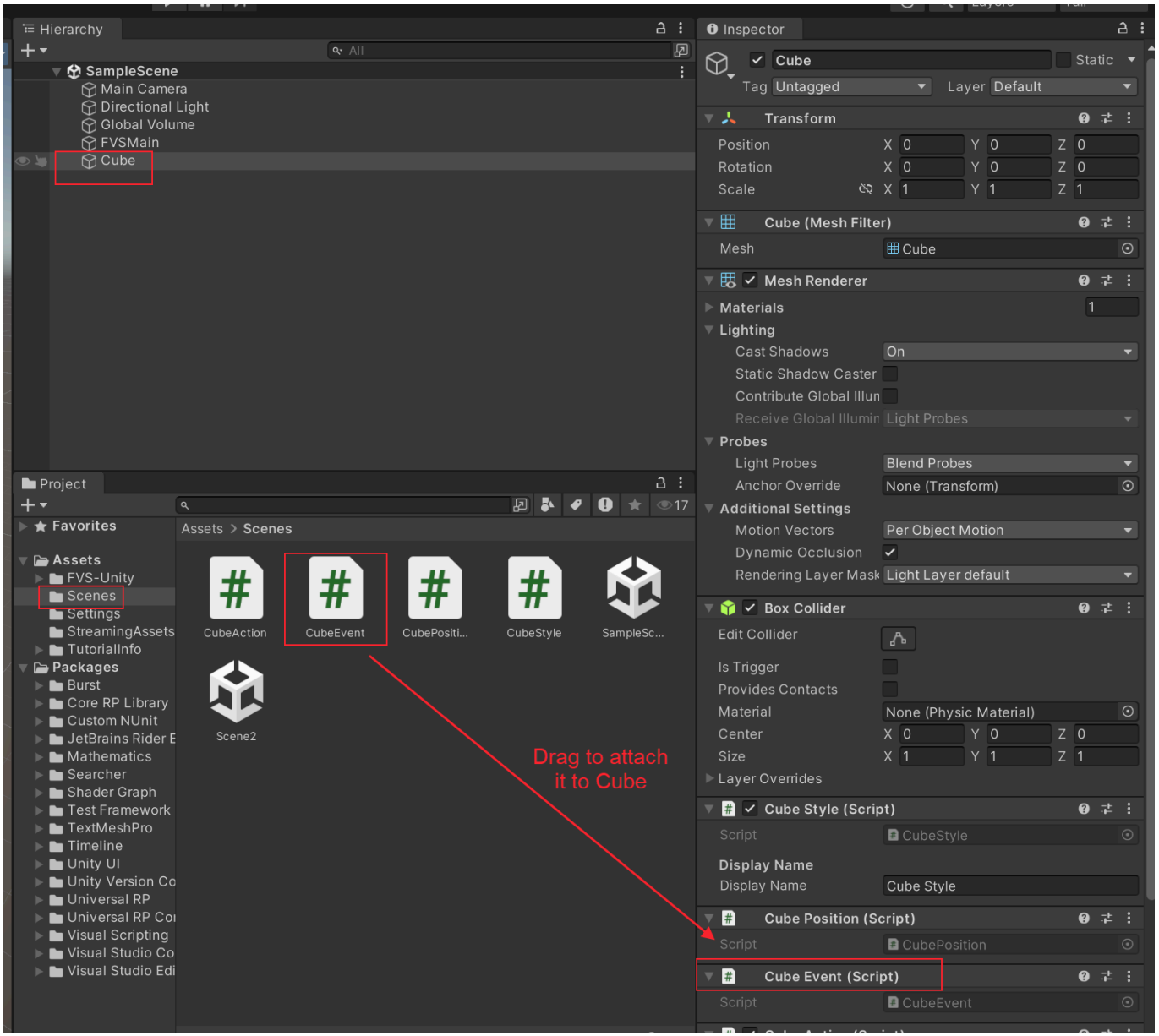
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using FVS;

public class CubeEvent : MonoBehaviour
{
    private void OnMouseUpAsButton()
    {
        // Pass the event through **FVSUnityReact** upon click.
        Debug.Log("clicked");
        FVSUnityReact.Instance.SendUnityEvent("cubeClick", "{\"name\": \"cube\"}");
    }
}
```

In this code, we added a `mouseup` response method that calls the `FVSUnityReact` singleton's `SendUnityEvent` method to pass a Unity event to `FVS`.

The two parameters in this method are:

1. **eventName**: The name of the event.
2. **eventData**: The serialized event object string, which is used to pass parameters.

At this point, a simple script for passing `Unity` events is complete. You also need to attach it to `Cube`.

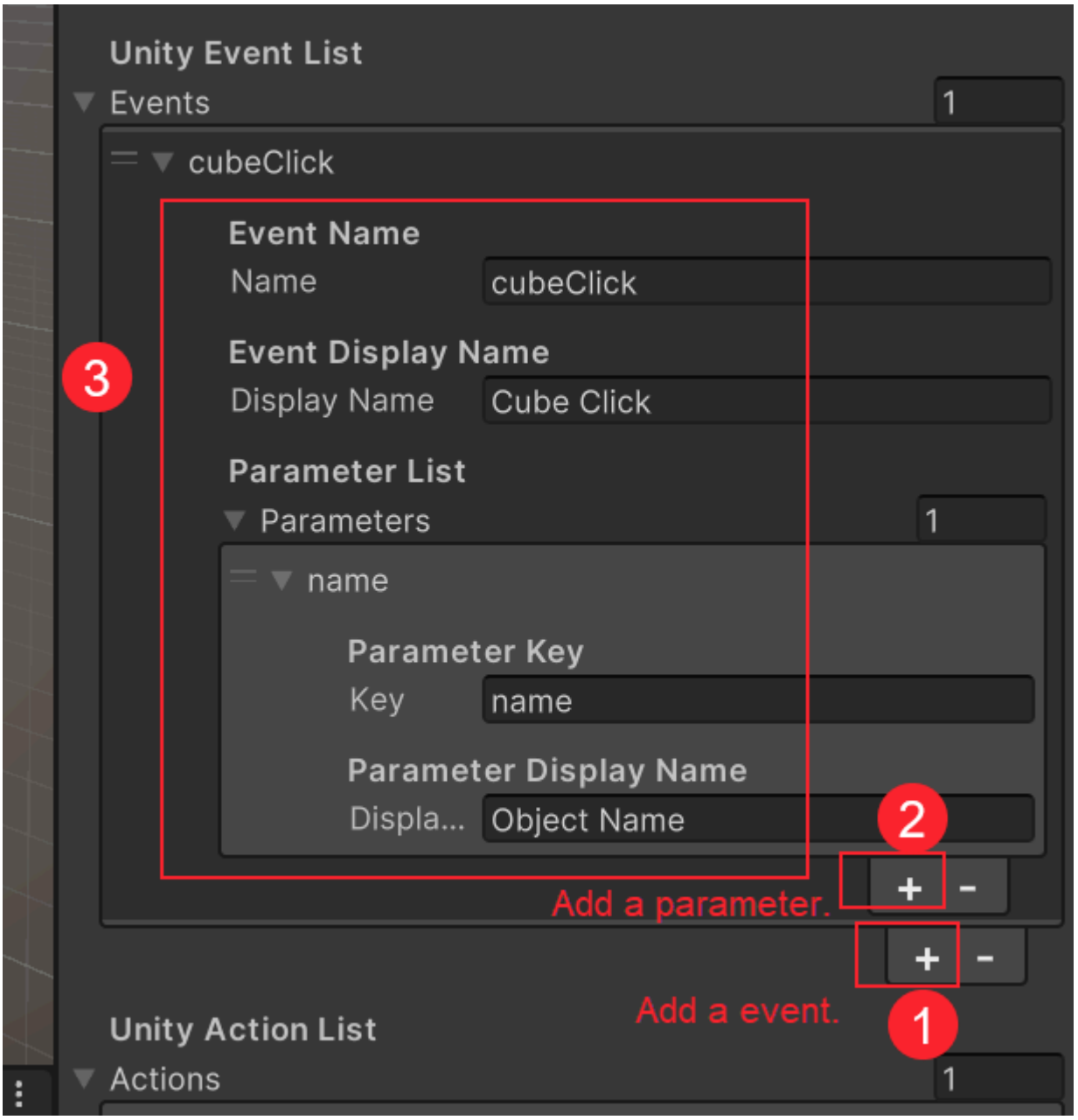


Building & Test

Next, let's build and test whether the event passing works correctly.

Unlike `custom styles` and `dynamic data`, events require some additional configuration before building (because events are triggered during script execution and are difficult to capture at the editing stage).

In the Hierarchy panel, choose `FVSMain` to view the Unity event list configuration. It is empty by default. Click the plus sign below to add an event, then fill in the event name, display name, and parameters in order.



The event name here is the first parameter in `SendUnityEvent` above. The event display name is the name shown on the UI panel, which we will see shortly.

Similarly, the parameter list records the meaning of each parameter. In this case, you only want to pass the parameter `name`, so the parameter list contains only one entry.

After completing the configurations in **Unity Event List**, you can choose **FVS** > **Package** > **Generate FVS-Unity Scene Configuration** from the main menu to update the scene configurations. After the generation is complete, the scene configuration JSON is:

```
{
    "scene": null,
    "data": {
        "value": [{
            "gameObject": "Cube",
            "fields": [{
                "name": "x",
                "displayName": "X Coordinate"
            }, {
                "name": "y",
```

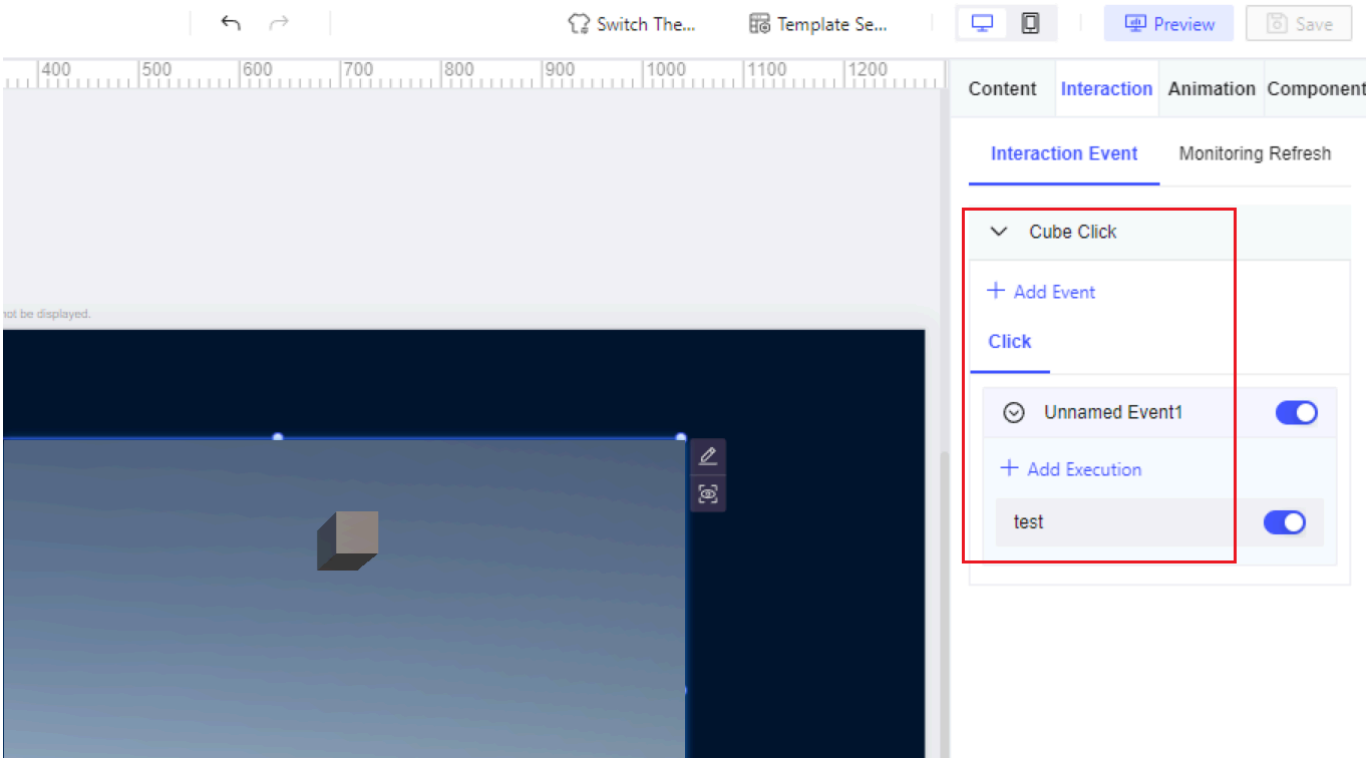
```
                "displayName": "Y Coordinate"
            }, {
                "name": "z",
                "displayName": "Z Coordinate"
            }],
            "displayName": "Position",
            "customFields": false
        }]
    },
    "style": {
        "value": [{
            "gameObject": "Cube",
            "displayName": "Cube",
            "properties": [{
                "name": "cubeSize",
                "displayName": "Cube Size",
                "getter": "GetCubeSize",
                "setter": "SetCubeSize",
                "type": "Number",
                "value": 1,
                "min": 1,
                "max": 20,
                "step": 0.1
            }]
        }]
    },
    "action": null,
    "event": {
        "value": [{
            "name": "cubeClick",
            "displayName": "Cube Click",
            "parameters": [{
                "key": "name",
                "displayName": "Object Name"
            }]
        }]
    }
}
```

Save the scene, choose **File** > **Build Settings** from the main menu, click **Build** in the **Build Settings** window, and compress the generated files to a ZIP file, as described in the chapter "Installation and Building".

Return to `FVS`, go to the `Unity` component editing page, and click **Replace Unity** to import the ZIP file mentioned earlier.

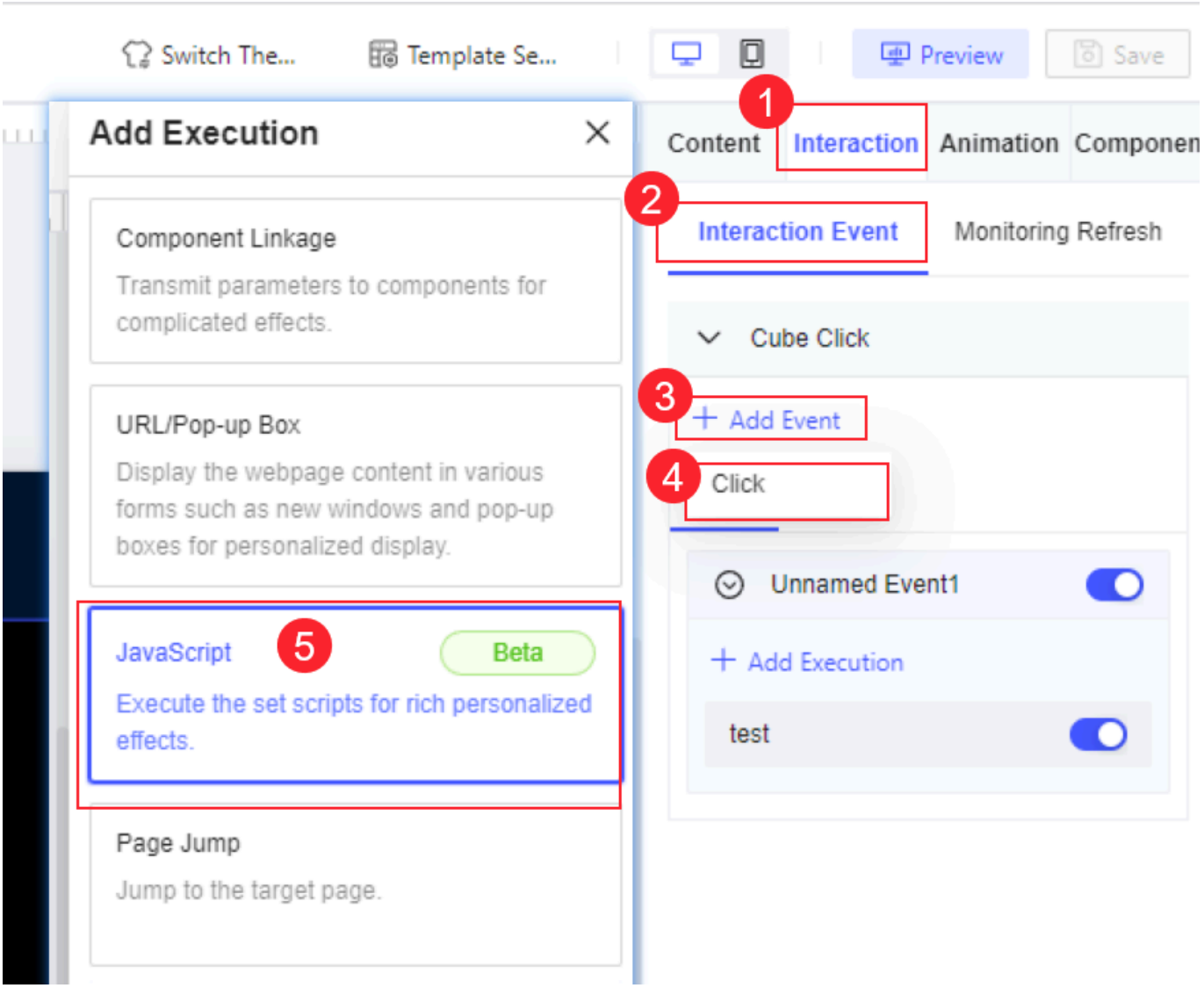
After importing the ZIP file, the event configuration panel is not displayed on the Unity component editing page. Since event interactions need to be configured in the FVS template, click **Back to Visualization Dashboard** first to return to the template editing page.````

Select the `Unity` component and click the **Interaction** tab in the component configuration panel. You will see a click event.

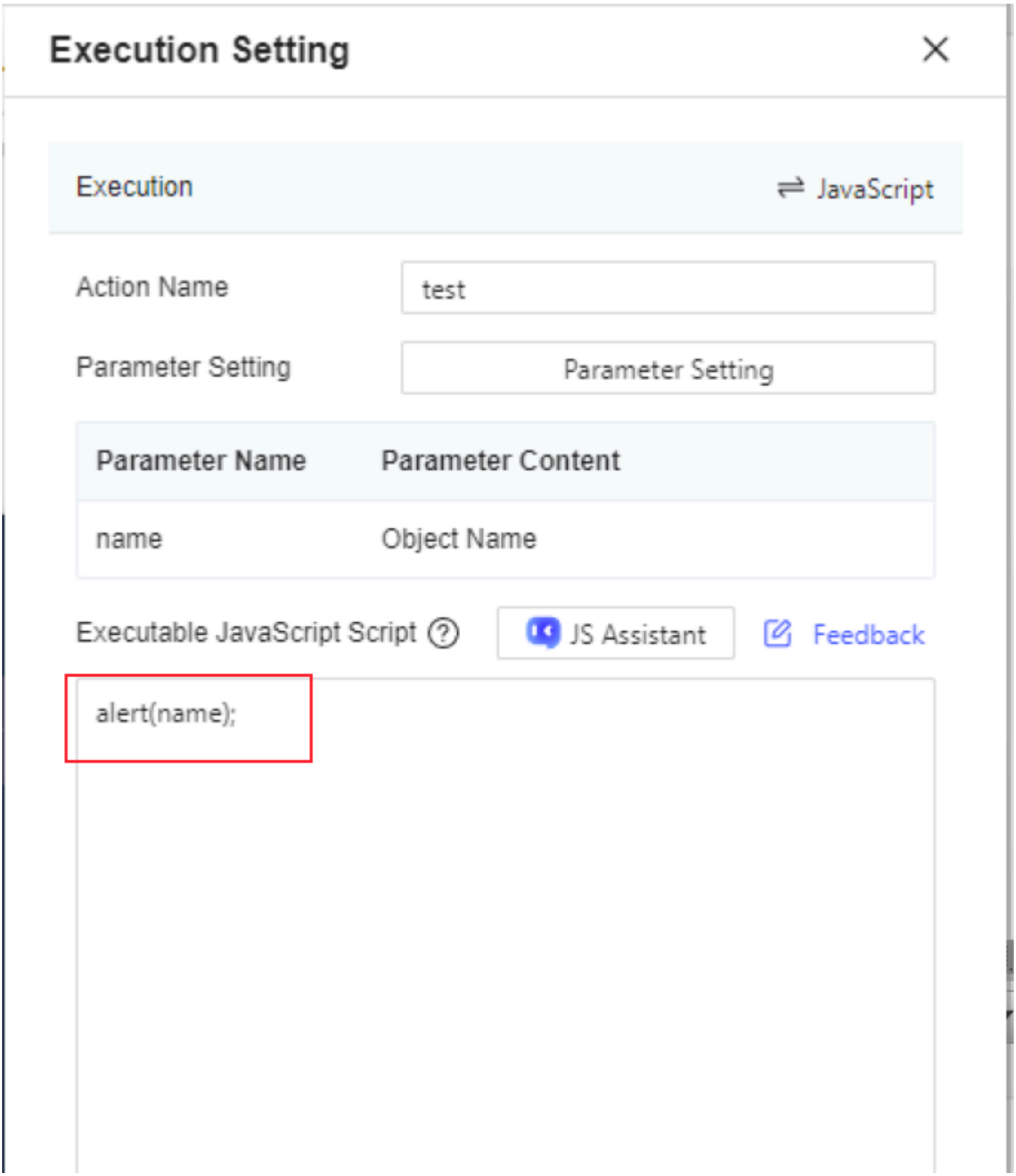


Once the event is registered through the `FVS-Unity-SDK` in Unity, it becomes available in FVS. In FVS, click **Add Event** and then **Click**.

In the displayed window, click **JavaScript**.

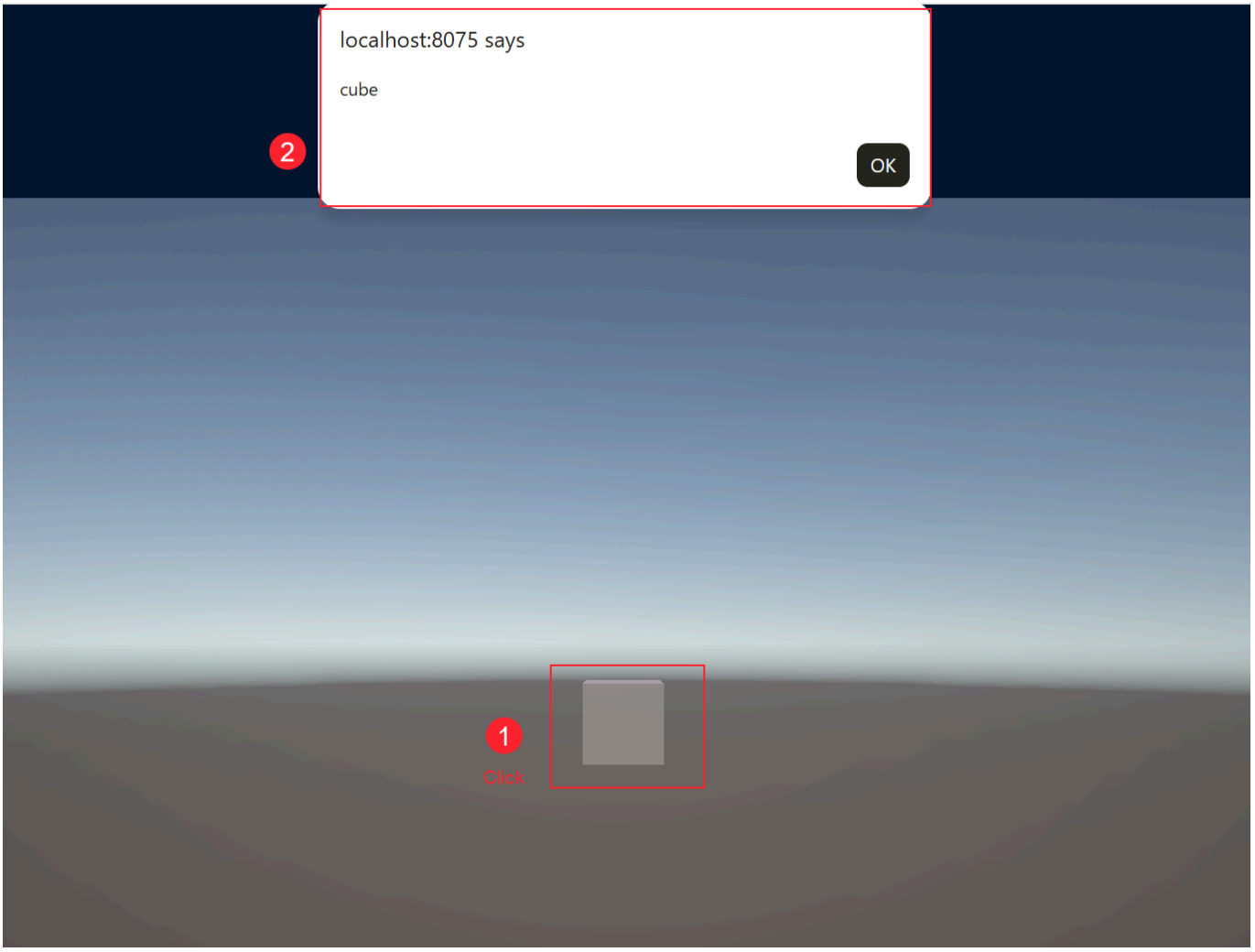


You will then enter the custom JS settings panel, which is nearly identical to that of other default JS settings panel.



After confirming the settings, save the template, and click **Preview**. Click the cube, and you can see a pop-up box, as shown in the following figure.

p?viewlet=%25E4%25BA%258B%25E4%25BB%25B6%25E4%25BC%25A0%25E9%2580%2592.fvs&page_number=1



In practice, you can generate models based on dynamic data, or bind dynamic data to a model.

You can pass events from Unity to FVS through the FVS-Unity-SDK.

Then configure component interactions to handle Unity events, such as navigating to another page, passing parameters between components, or executing custom JavaScript.

Unity Action Control

In the previous chapter about passing Unity events, you learned how to pass Unity events to FVS through the FVS-Unity-SDK, enabling interaction between Unity and the entire FVS template.

In that process, events are sent from Unity to FVS through the SDK, with Unity acting as the initiator of the interaction.

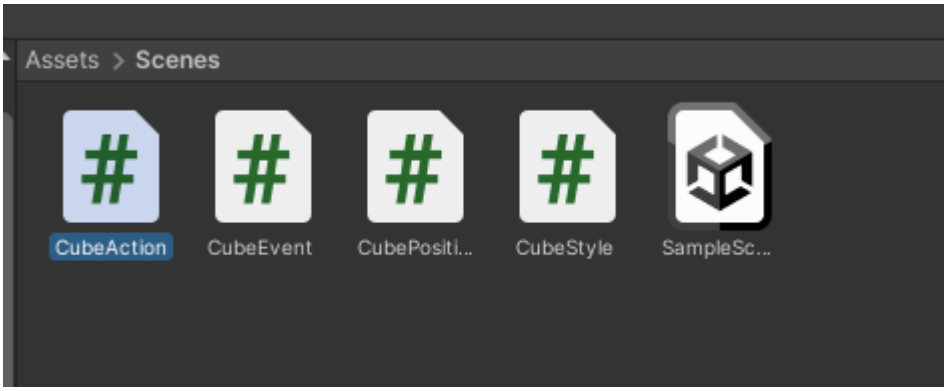
In practice, you may encounter the opposite interaction pattern, where events originate from FVS and Unity acts as the executor. For example, clicking a chart component to pass parameters to Unity, or using JavaScript to change elements within a Unity scene.

In this chapter, you will learn how to use the SDK to control Unity objects from FVS.

> Here is a hypothetical requirement: in FVS, you want to show or hide a Cube in the Unity scene by clicking another component. Let's see how to implement this requirement.

Creating an Action Control Script

Create a new C# script for behavior control and name it **CubeAction**.



Double-click the script to enter the editing page, and enter the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using FVS;

public class CubeAction : MonoBehaviour
{
    // Start is called before the first frame update
    void OnEnable()
    {
        FVSActionManager.Bind("SetCubeVisible", SetVisible);
    }

    // Update is called once per frame
    void SetVisible(string[] data)
    {
        bool visible = data[0] == "true";
        GetComponent<Renderer>().enabled = visible;
    }
}
```

In this code, a `SetVisible` method is defined to receive data passed from FVS.

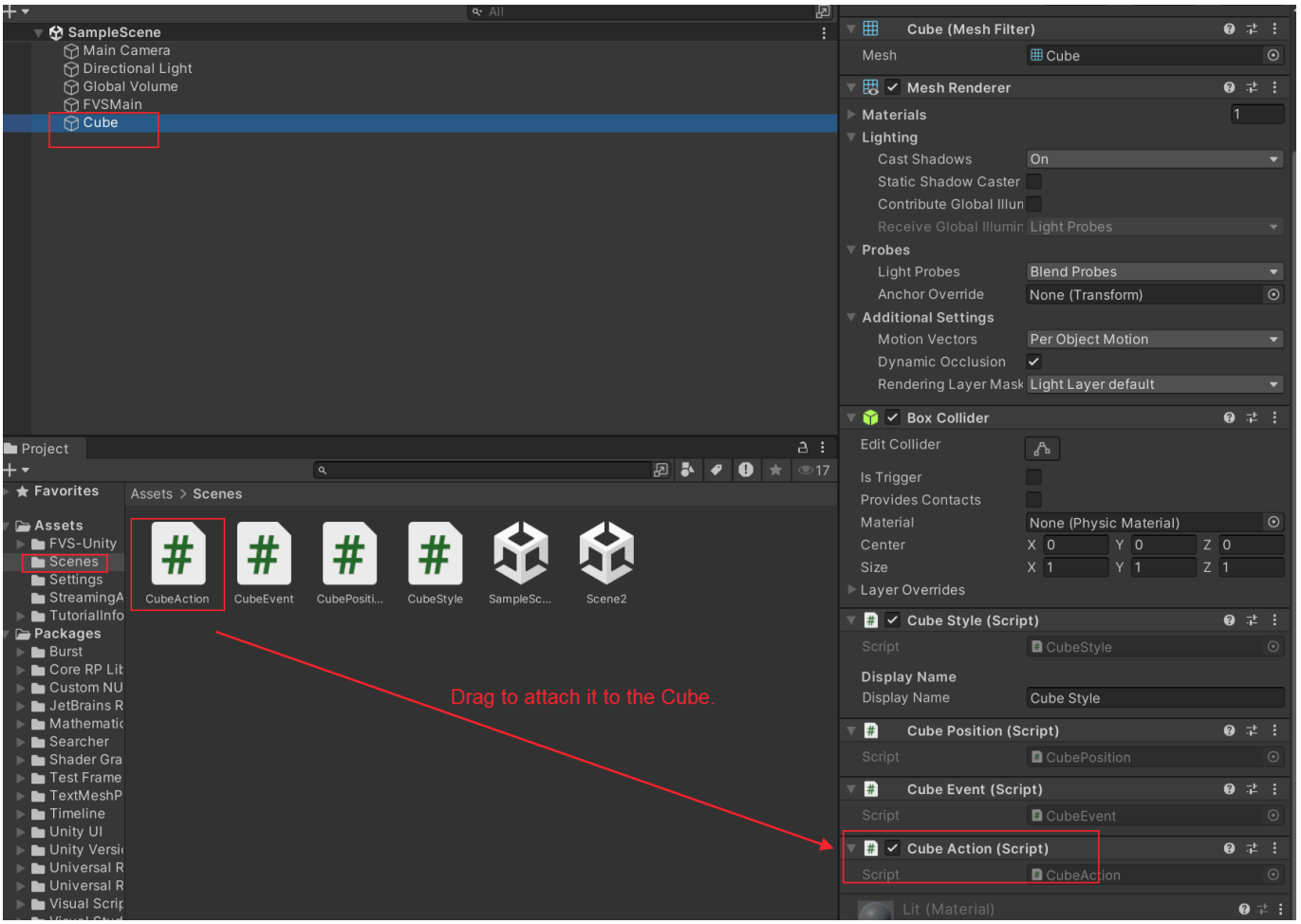
> Since communication between FVS and Unity only supports string data transmission, the parameter of this method is defined as a string array **string[]**.
>
> The parameter passing method is consistent with the way Unity receives data from FVS in the previous chapter "Unity-to-FVS Event Passing."

It checks whether the first string in the array is `true`. If so, it makes the current component visible; otherwise, it hides the current component.

In `GameObject`'s `OnEnable` method, `SetVisible` is bound to the `FVSActionManager`.

The first parameter, `SetCubeVisible`, is the `name` (identifier) of this Unity behavior.

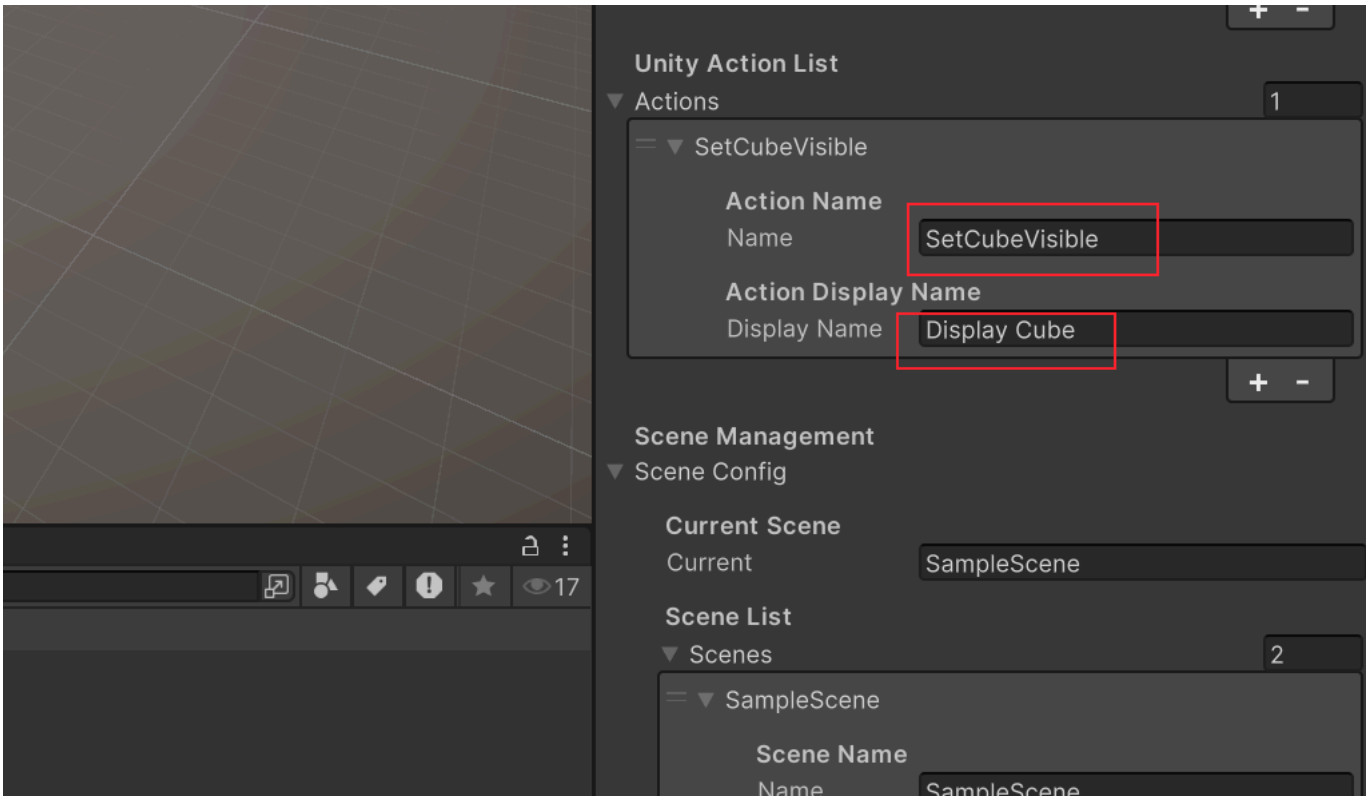
At last, select `Cube` and attach this script to it.



Building & Test

Same as with event passing, you need to clarify some information for `UnityAction` before building the project.

Select `FVSMain` and add a record in `Unity Action list`, as shown in the following figure.



Then, you can choose **FVS** > **Package** > **Generate FVS-Unity Scene Configuration** from the main menu to update the scene configurations. After the generation is complete, your scene configuration will include `Action`-related information.

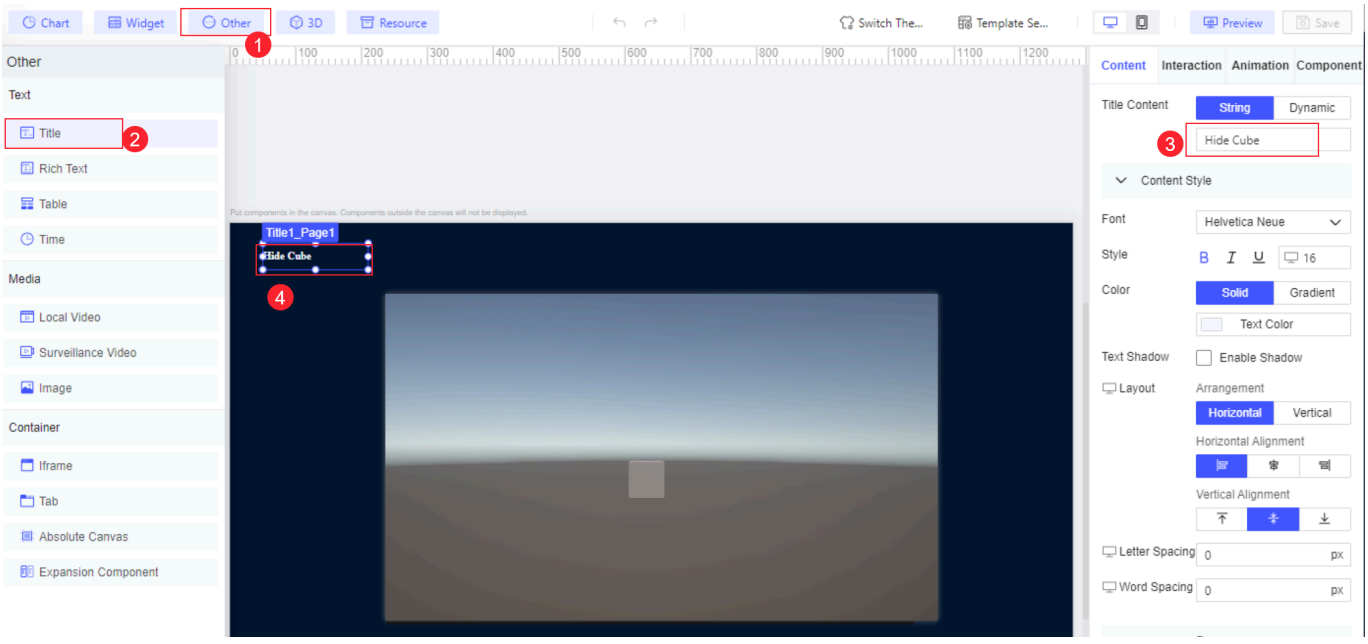
```
{
    "scene": null,
    "data": {
        "value": [{
            "gameObject": "Cube",
            "fields": [{
                "name": "x",
                "displayName": "X Coordinate"
            }, {
                "name": "y",
                "displayName": "Y Coordinate"
            }, {
                "name": "z",
                "displayName": "Z Coordinate"
            }],
            "displayName": "Position",
            "customFields": false
        }]
    },
    "style": {
        "value": [{
            "gameObject": "Cube",
            "displayName": "Cube",
            "properties": [{
                "name": "cubeSize",
                "displayName": "Cube Size",
                "getter": "GetCubeSize",
                "setter": "SetCubeSize",
```

```
                "type": "Number",
                "value": 1,
                "min": 1,
                "max": 20,
                "step": 0.1
            }]
        }]
    },
    "action": {
        "value": [{
            "name": "SetCubeVisible",
            "displayName": "Show Cube",
        }]
    },
    "event": {
        "value": [{
            "name": "cubeClick",
            "displayName": "Cube Click",
            "parameters": [{
                "key": "name",
                "displayName": "Object Name"
            }]
        }]
    }
}
```

The next steps are saving the scene and building the project. After that, return to FVS, enter the Unity component editing page, replace the resource file, and click **Back to Visualization Dashboard**.

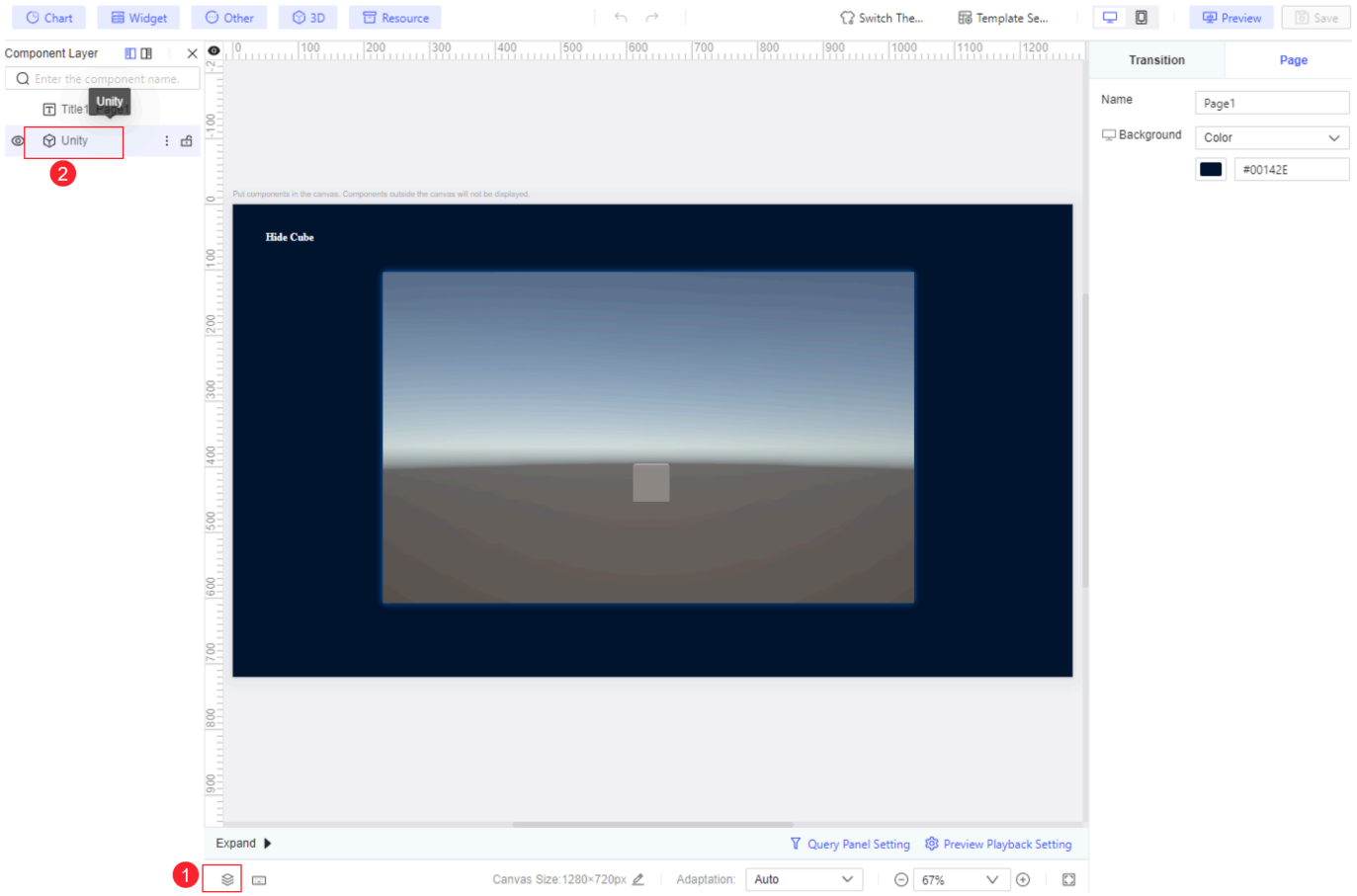
In the component panel, choose `Other > Title` and drag a title component onto the canvas.

Set the title content to `Hide Cube`.

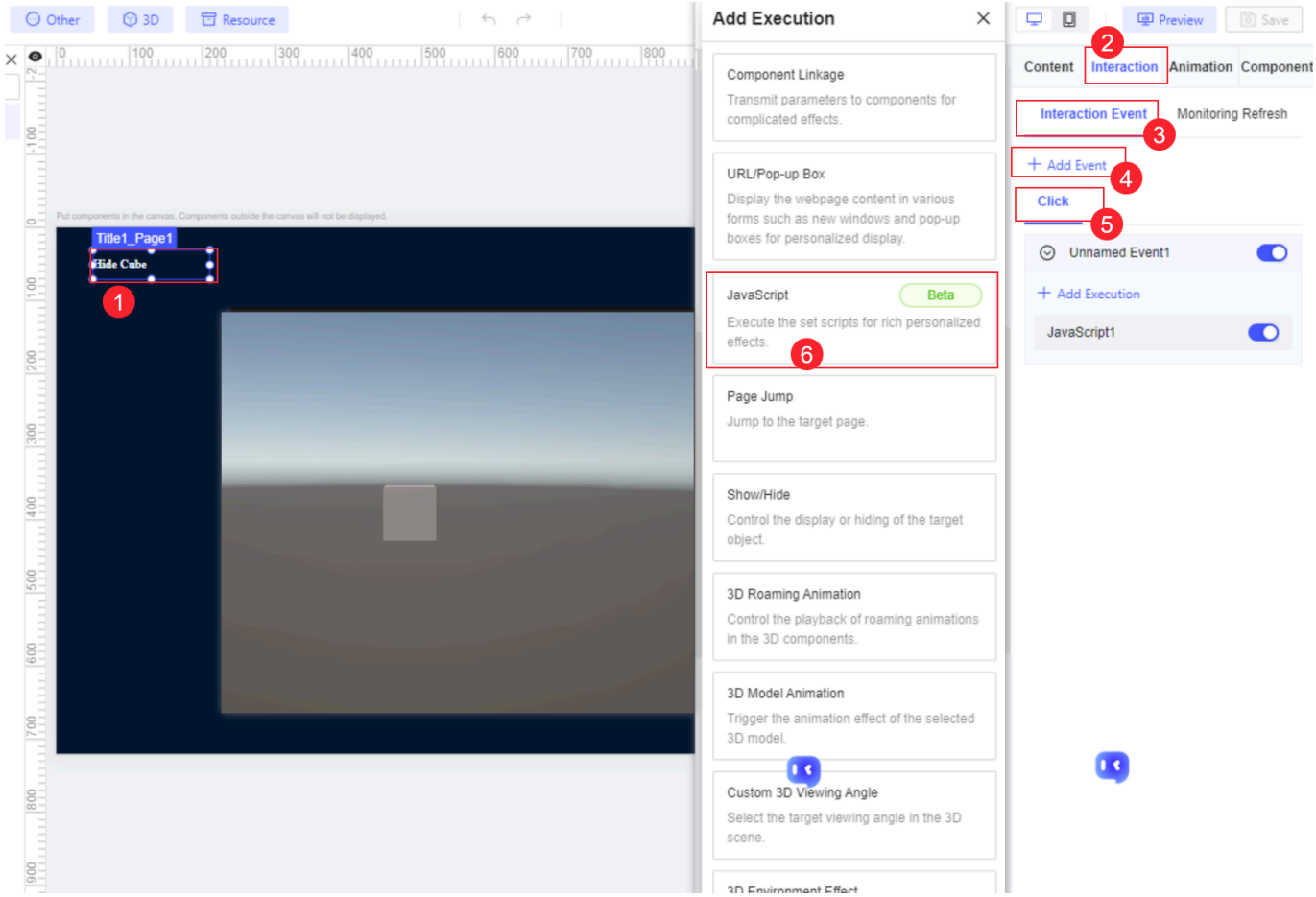


Next, bind a click event to the component and configure it to call the `SetCubeVisible` action defined in Unity when the event is triggered.

To implement this, first assign a name to the component. Click the  button in the lower-left corner of the canvas to display the **Component Layer** panel, select the Unity component, and rename it to **Unity**.

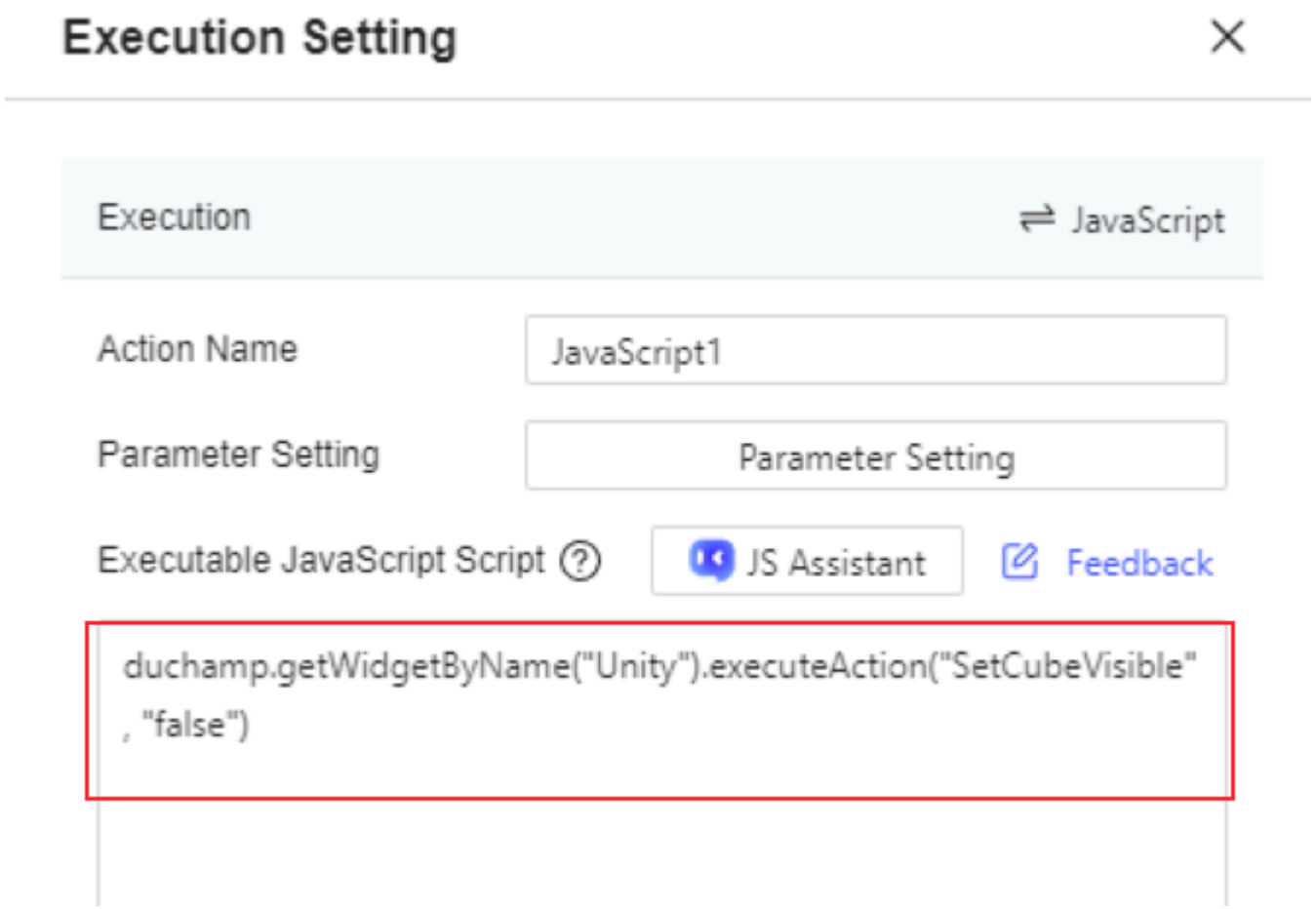


Then return to the canvas, select the title component, click the `Interaction` tab in the right panel, and click **Add Event** and then **Click**. In the displayed panel, click `JavaScript`.



The event name can be customized. Since no event name or parameters need to be configured, directly write the JavaScript code.

```
duchamp.getWidgetByName("Unity").executeAction("SetCubeVisible", "false")
```

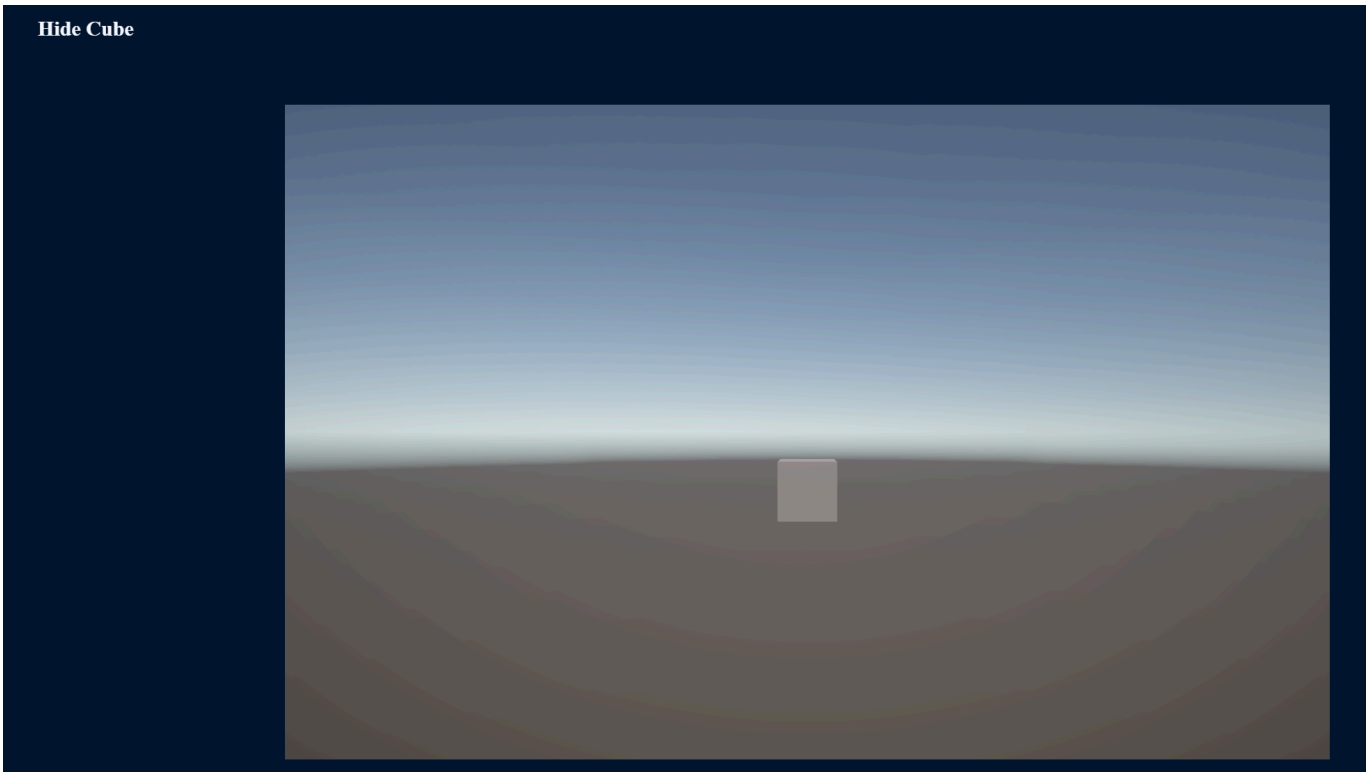


Then click **OK**.

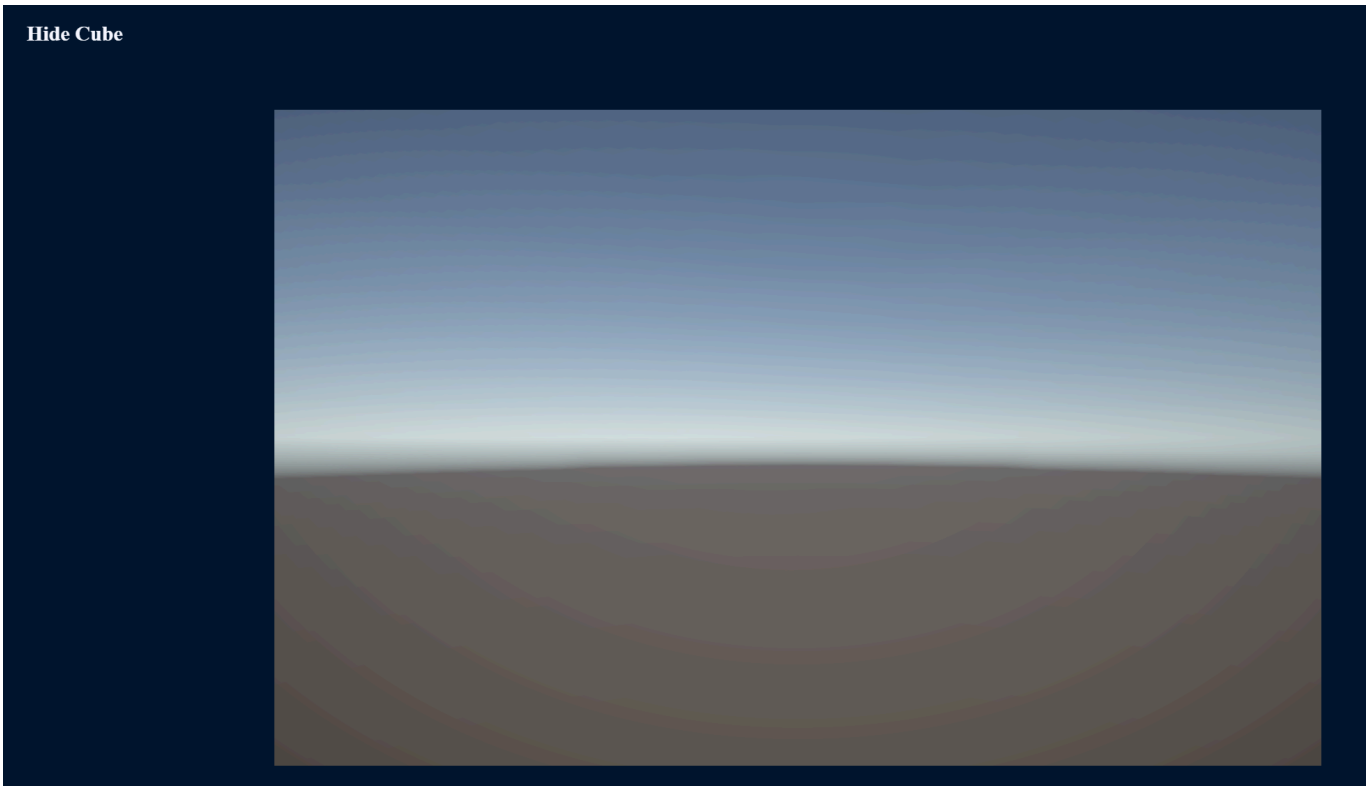
The following section describes the functionality of this code.

> First, it calls the `duchamp.getWidgetByName("Unity")` method to obtain the Unity component by its name. The "Unity" parameter passed here is the name assigned to the Unity component earlier.``
>
> Then it uses the `executeAction` method of the Unity component to call the `SetCubeVisible` action registered in Unity. This method requires two parameters. The first parameter is the `Action` name, which is `SetCubeVisible` registered through `FVSActionManager`. The second parameter is the data passed to `Action`, which is passed to Unity as the input parameter of the `SetVisible` method. Since the Unity component automatically converts multiple parameters into a string array, you do not need to manually pass an array here. Simply pass the string "false".

Next step is to click **Save** and then **Preview**.



When you click the `Hide Cube` button, the Cube is hidden. The complete FVS-to-Unity control workflow has now been implemented.



Multiple Scenes

When developing 3D scenes with Unity, you may encounter requirements that involve multiple scenes.

Generally, you can combine multiple scenes into a single scene and control them by toggling visibility.

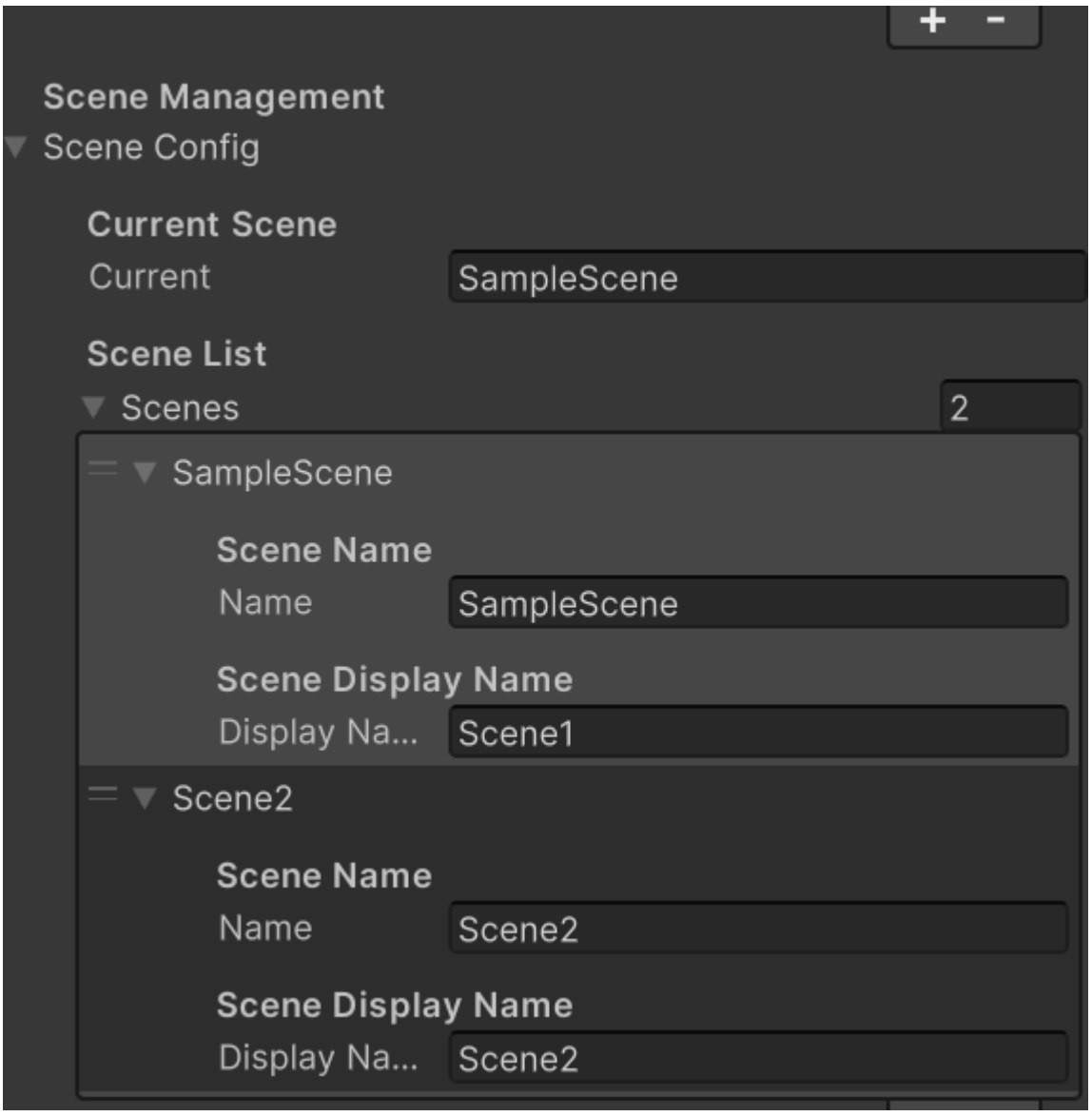
However, when the scene content becomes more complex, managing all the scenes within a single scene can become difficult. To address this issue, you can split the content into multiple scenes and configure them using the FVS-Unity-SDK. Then, you can switch between scenes through the component panel in FVS, which is very helpful for multi-page design.

Next, you will learn how to create multiple Unity scenes and use the SDK to update **Scene Configuration JSON** in Unity. After importing the Unity resources into FVS, you can switch between different Unity scenes through FVS pages.

Creating Multiple Scenes

First, configure the multi-scene settings in **Scene1**. Select `FVSMain`, on the `FVS Unity React (Script)` panel, you will see a scene management configuration list (empty by default — you need to configure it when using multiple scenes).

Configure the settings as shown in the following figure.



> This configuration means there are two scenes:

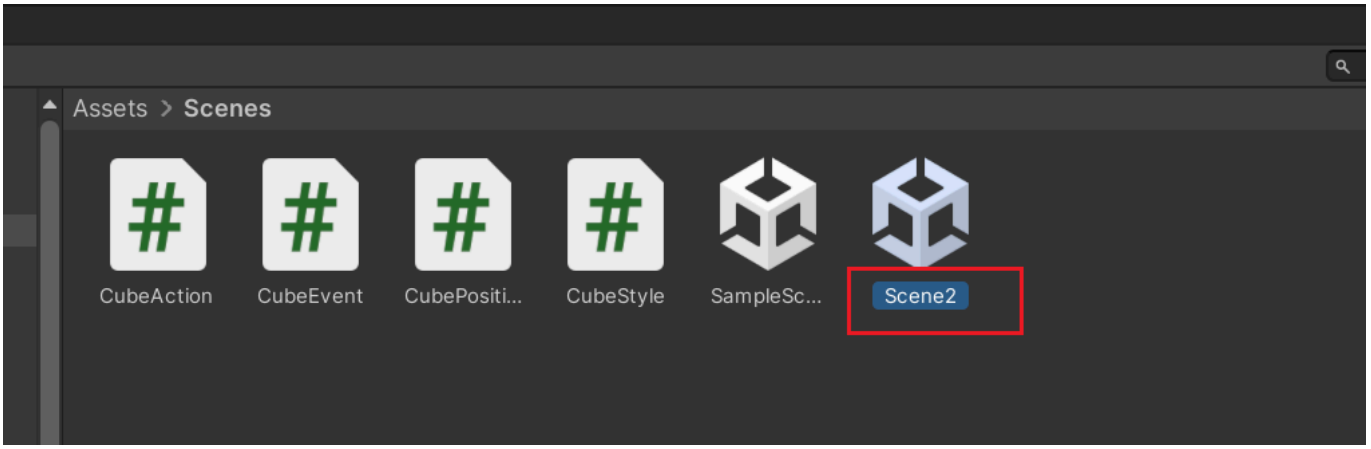
> 1. Scene 1 has the file name `SampleScene` and the display name **Scene 1**.
> 2. Scene 2 has the file name `Scene2` and the display name **Scene 2**.
>
> The current scene is **Scene2** (which must match the scene name in the scene list).

Then, you can choose **FVS** > **Package** > **Generate FVS-Unity Scene Configuration** from the main menu to update the scene configurations.


```
{
    "scene": {
        "value": {
            "current": "SampleScene",
            "scenes": [{
                "name": "SampleScene",
                "displayName": "Scene 1"
            }, {
                "name": "Scene2",
                "displayName": "Scene 2"
            }]
        }
    },
    "data": {
        "value": [{
            "gameObject": "Cube",
            "fields": [{
                "name": "x",
                "displayName": "X Coordinate"
            }, {
                "name": "y",
                "displayName": "Y Coordinate"
            }, {
                "name": "z",
                "displayName": "Z Coordinate"
            }],
            "displayName": "Position",
            "customFields": false
        }]
    },
    "style": {
        "value": [{
            "gameObject": "Cube",
            "displayName": "Cube",
            "properties": [{
                "name": "cubeSize",
                "displayName": "Cube Size",
                "getter": "GetCubeSize",
                "setter": "SetCubeSize",
                "type": "Number",
                "value": 1,
                "min": 1,
                "max": 20,
                "step": 0.1
            }]
        }]
```

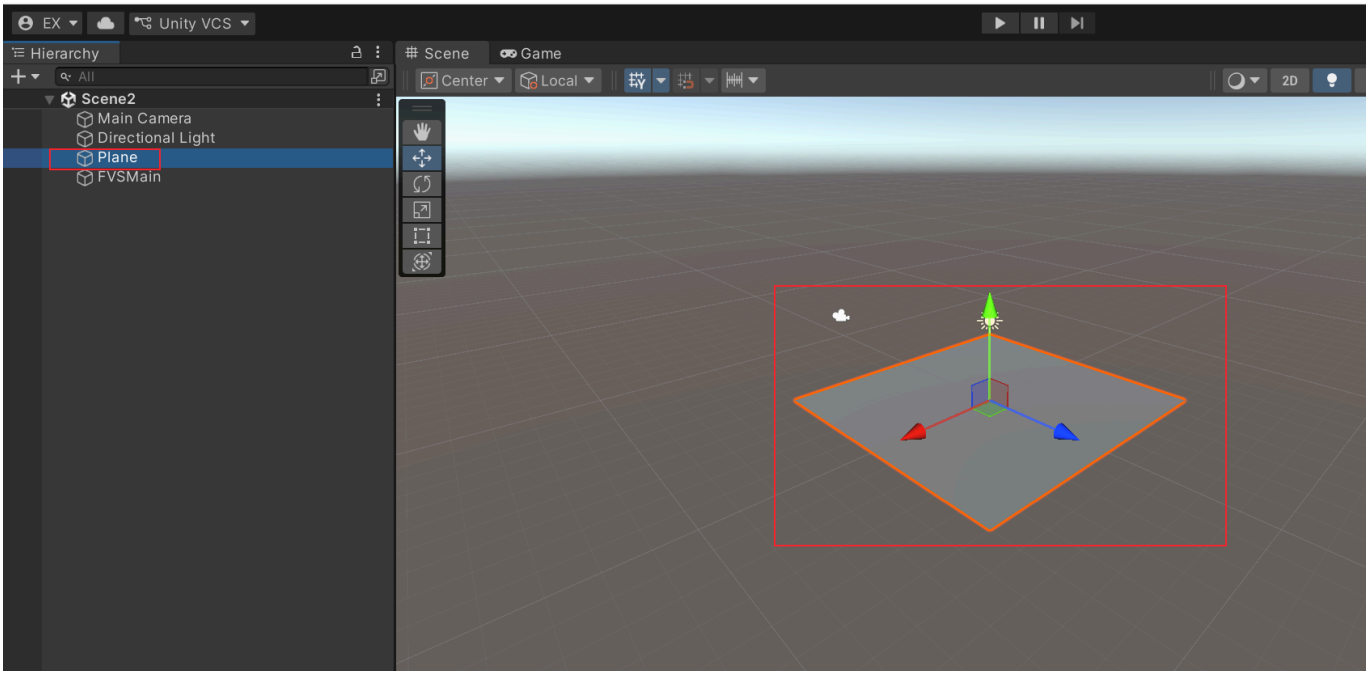
```
    },
    "action": {
        "value": [{
            "name": "SetCubeVisible",
            "displayName": "Show Cube",
        }]
    },
    "event": {
        "value": [{
            "name": "cubeClick",
            "displayName": "Cube Click",
            "parameters": [{
                "key": "name",
                "displayName": "Object Name"
            }]
        }]
    }
}
```

The configuration file now contains the scene-related settings. Save **Scene1**, then create a second scene. The second scene name must match the name in the **Scene Management** list: `Scene2`.



Double-click to open `Scene2`. By default, it is an empty scene.

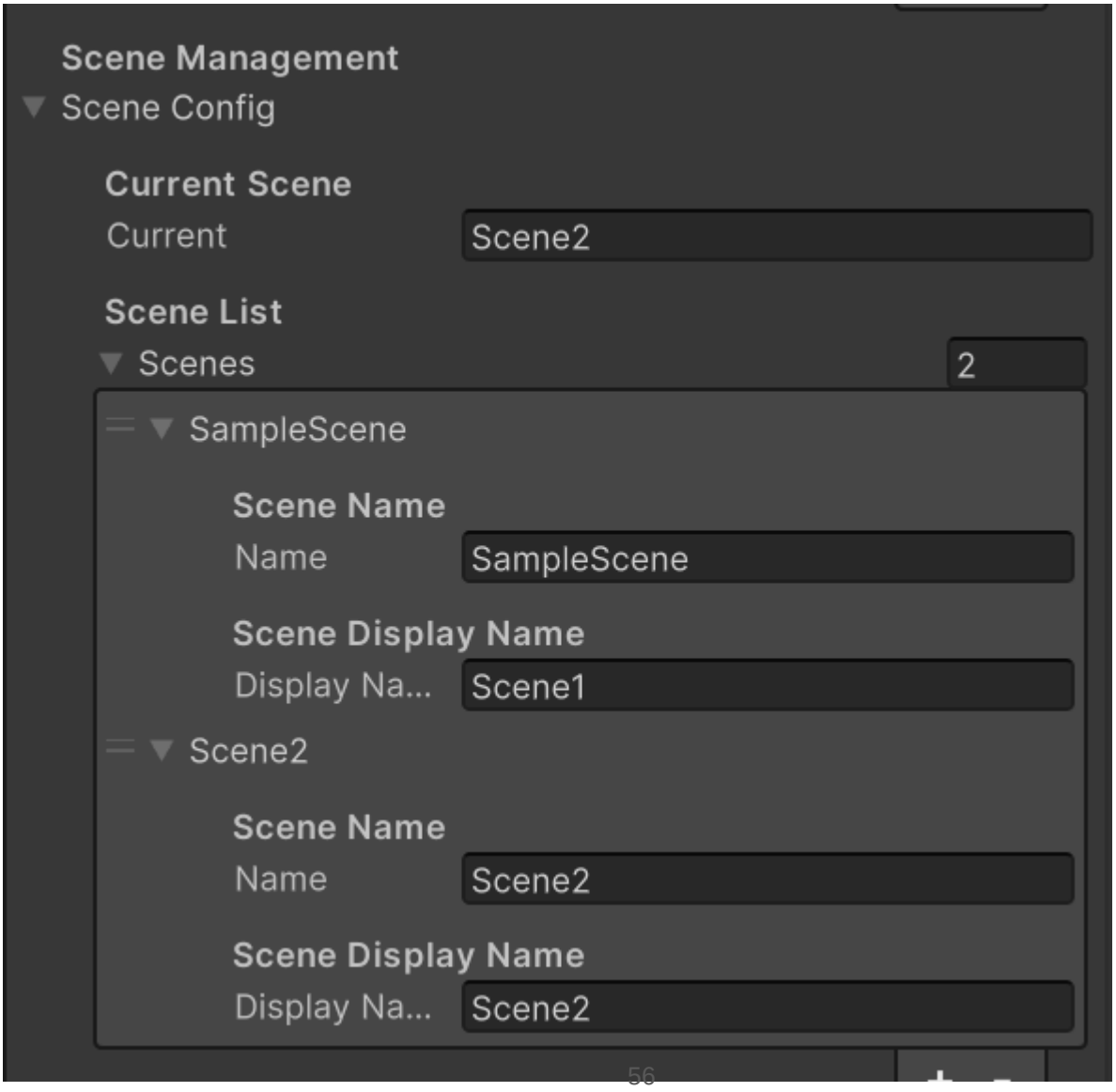
To distinguish it from the first scene, add a Plane to it.



Next, follow the workflow described in the preceding chapters: First, add the FVS-Unity communication component.

Since the second scene is used only for test, you do not need to create scripts for custom styles or data.

You only need to configure scene management for **Scene2** as well. Expand **Scenes** under **Scene Management** > **Scene Config** > **Scene List**, and enter the information as shown in the following figure.



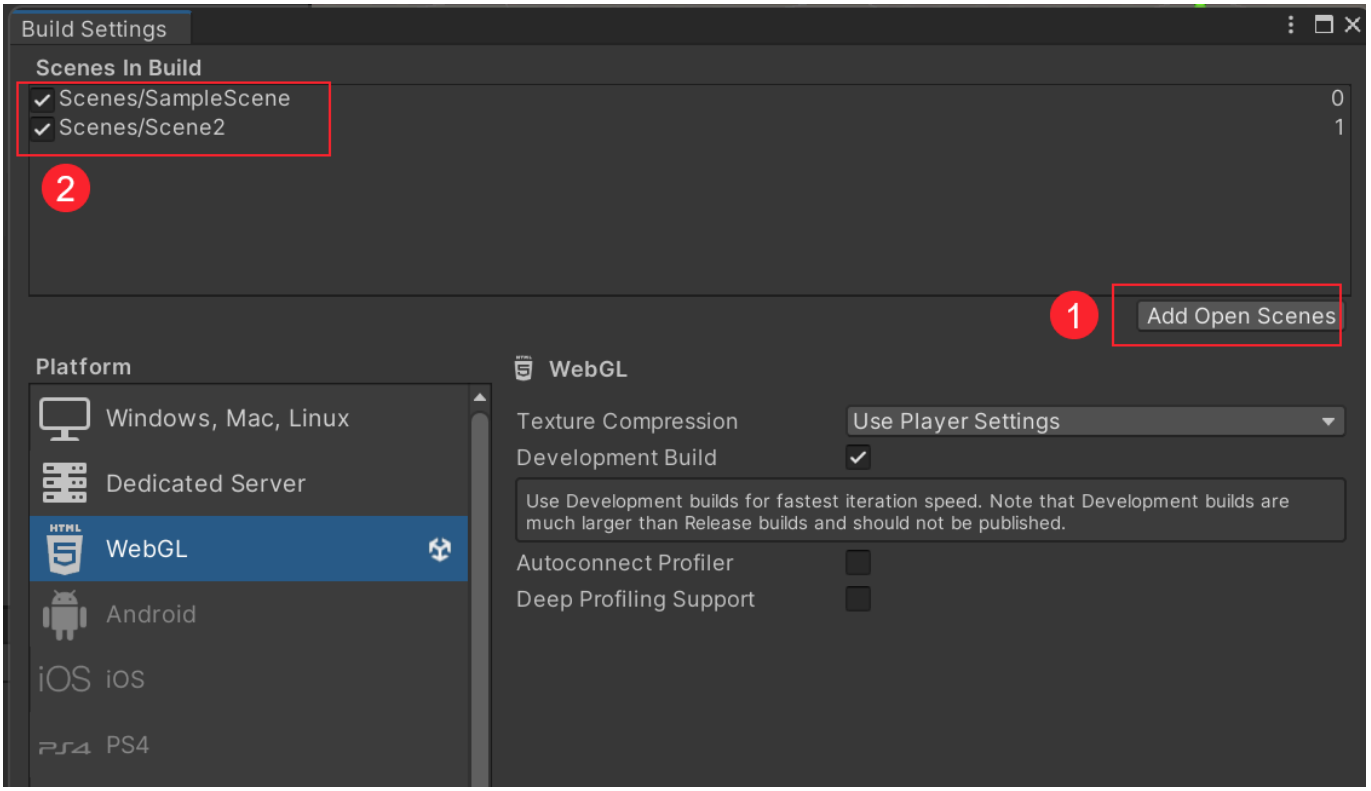
Then you can update Scene Configuration JSON. The updated configuration JSON is as follows:

```
{
    "scene": {
        "value": {
            "current": "Scene2",
            "scenes": [{
                "name": "SampleScene",
                "displayName": "Scene 1"
            }, {
                "name": "Scene2",
                "displayName": "Scene 2"
            }]
        }
    },
    "data": null,
    "style": null,
    "action": null,
    "event": null
}
```

Save **Scene2**, then build and test the project.

Building and Test

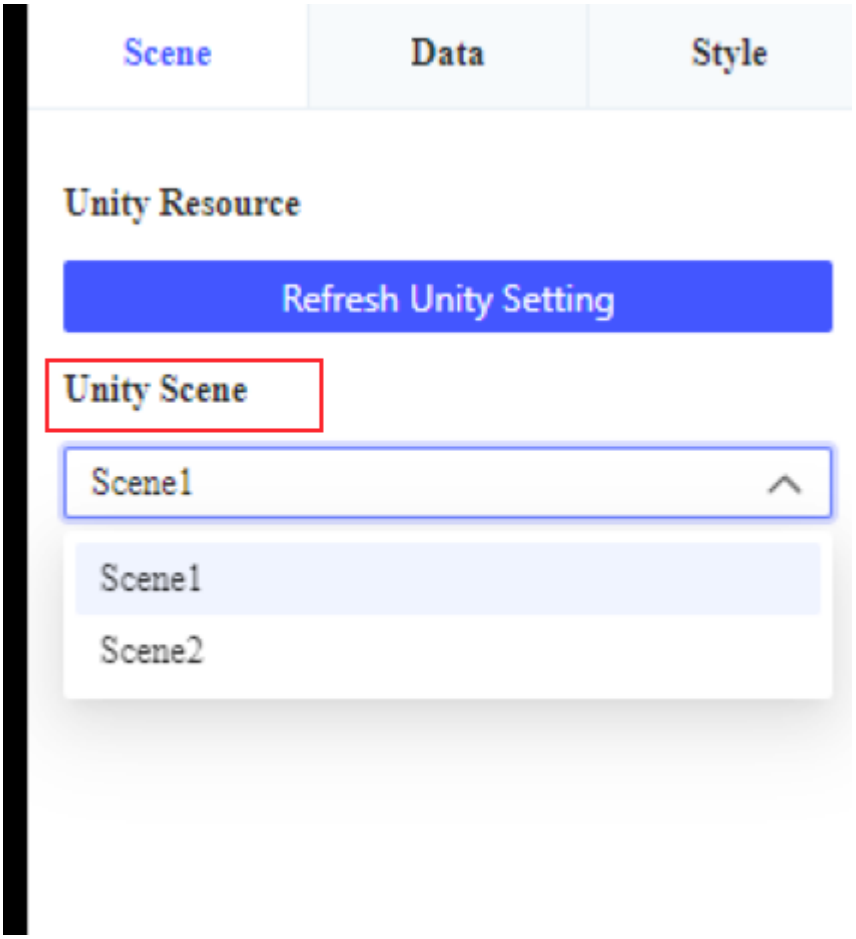
Click `File > Build Settings` from the main menu. By default, only the first scene is listed in **Scenes In Build**. You can click `Add Open Scenes` to add the current scene to the build list.



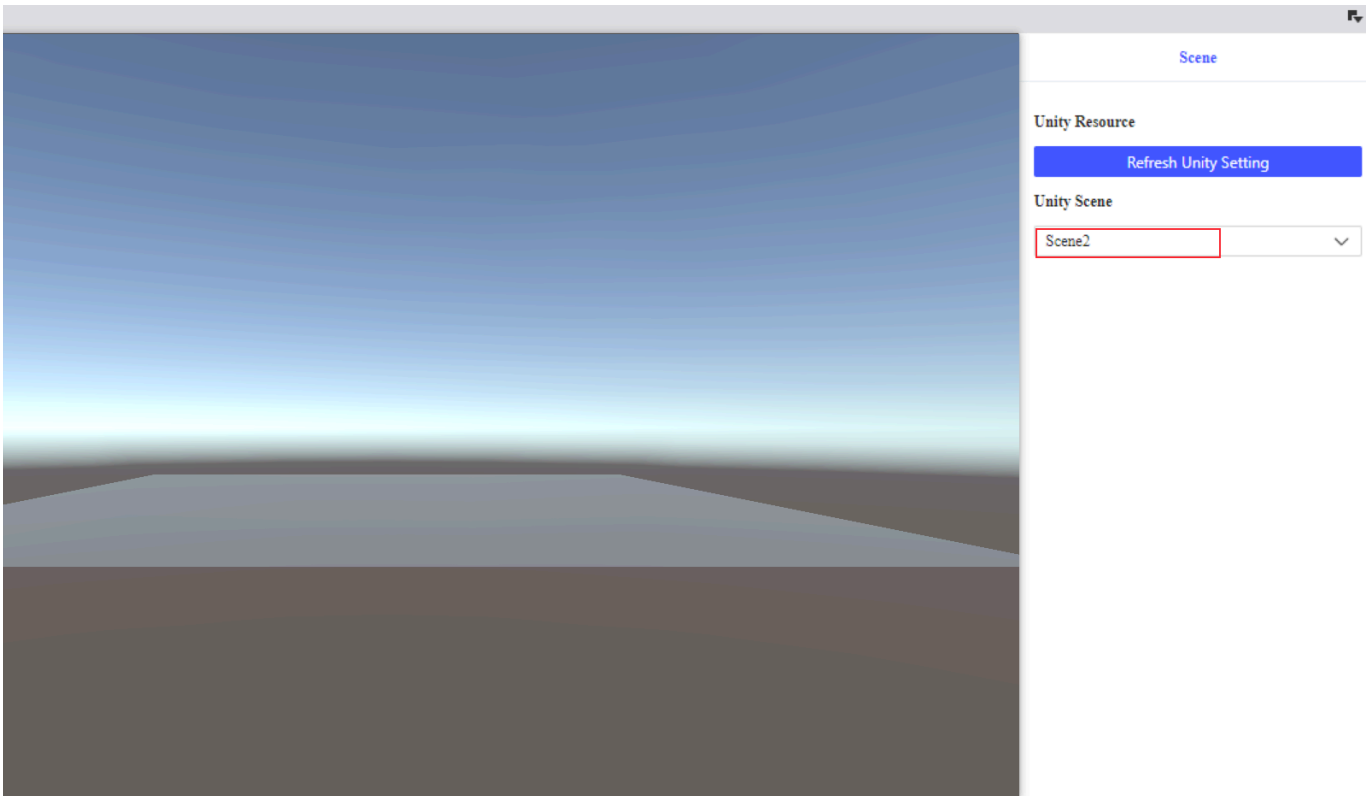
> You can drag to adjust the default scene order. This example keep the default settings.

Then click `Build` to start the building process. After building, return to FVS, enter the Unity component editing page, and replace the resources.

In the **Scene** tab, you will find a new **Unity Scene** option.



By default, **Scene1** is selected. You can select **Scene2** from the drop-down list.



With multiple Unity scenes configured, you can switch between different scenes on different FVS pages through the UI panel.

Internationalization

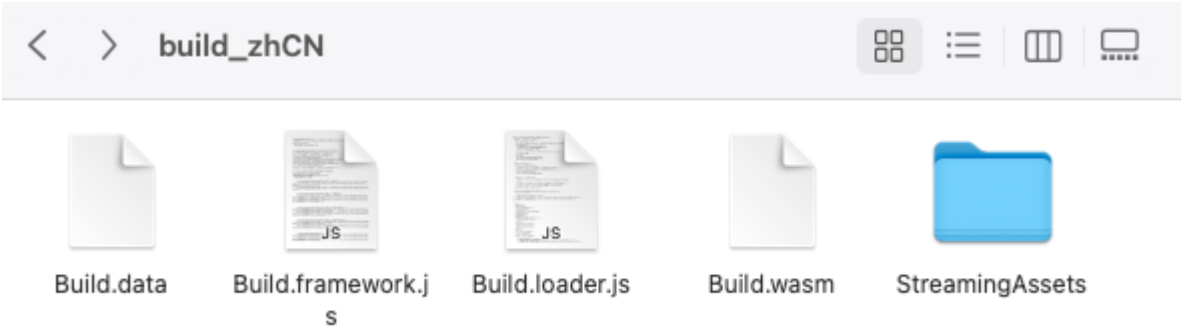
In an FVS-Unity project, internationalization involves three areas: the frontend page, scene configuration information (stored in JSON format), and Unity project itself. In the latter two cases, Unity developers can implement internationalization based on project requirements.

Scene Configuration Information

As mentioned in preceding chapters, before building, clicking **Generate FVS-Unity Scene Configuration** from the SDK menu will automatically generate the scene configuration information. After building, the SDK generates a **Localization.json** file based on the original **displayName** values in the scene configuration information. This file is generated in the **StreamingAssets** directory at the same level as the **Build** directory along with other project resources.

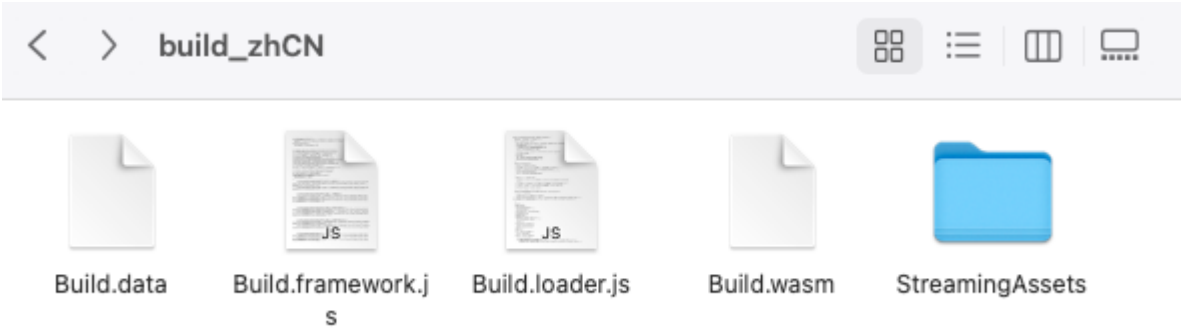
If Unity developers need to support internationalization, they can modify the **displayName** values in the **Localization.json** file based on actual requirements.

When creating the package (ZIP file) for import into FVS, you need the **Localization.json** file together with the four Unity resource files generated in the Build directory. Therefore, you need to move the **StreamingAssets** folder (which includes the **Localization.json** file) to the **Build** directory that contains the four Unity resource files (*build*.data, *build*.framework.js, *build*.loader.js, and *build*.wasm), and package them together.



Unity Project

When implementing internationalization in a Unity project, Unity developers often introduce the Localization Library (https://docs.unity3d.com/Packages/com.unity.localization@1.4/manual/ComponentLocalizers.html) or other internationalization plugins. Make sure that any internationalization-related resource files generated after building are present in the StreamingAssets directory.

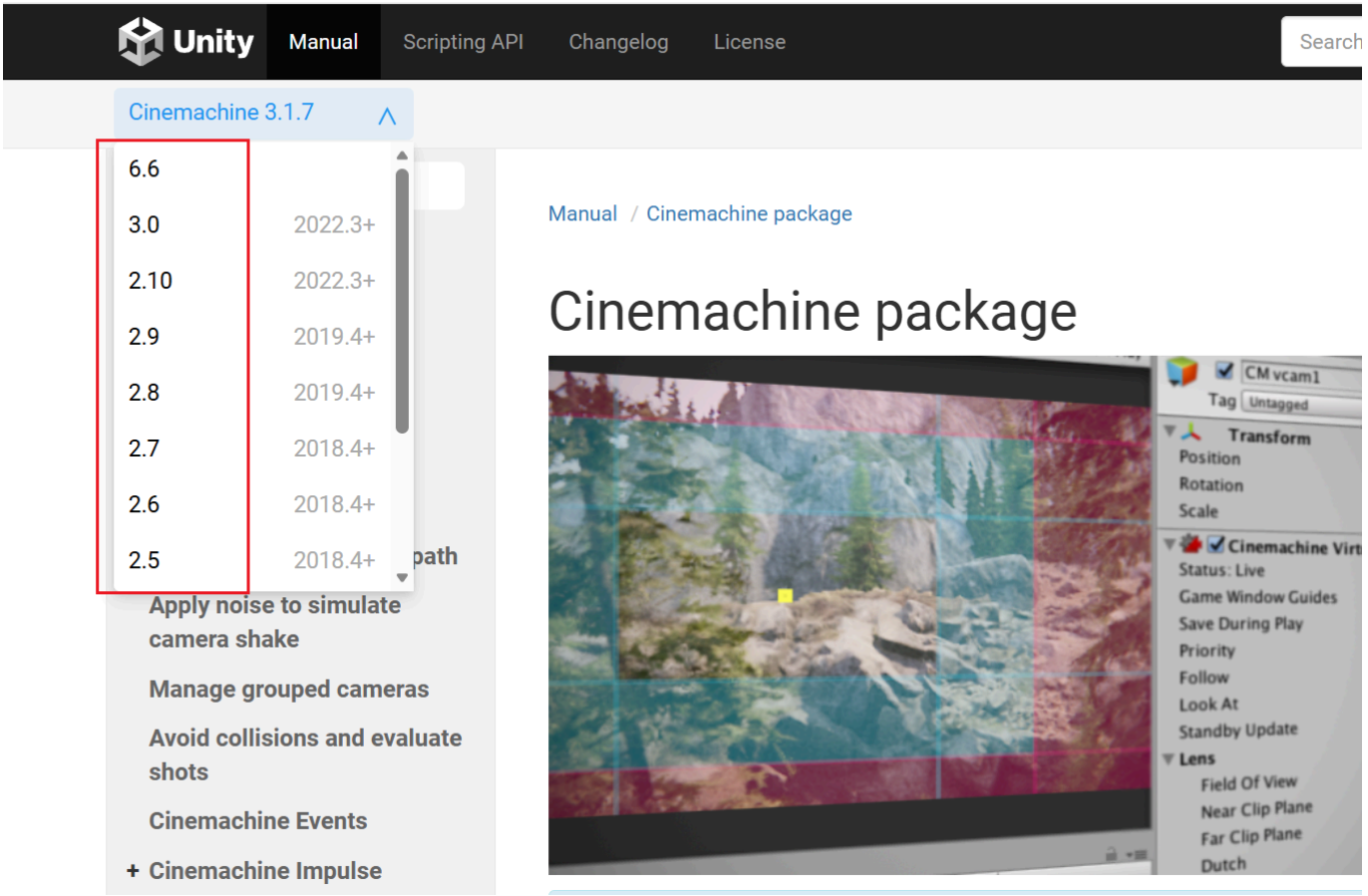


Camera Roaming (Unity Cinemachine)

Camera roaming can be implemented using Unity's official Cinemachine package. For setup and usage instructions, refer to the Unity official documentation:

https://docs.unity3d.com/Packages/com.unity.cinemachine@3.1/manual/index.html

You can select the Cinemachine version you want to view from the drop-down list.



FAQ

1. How Do I Use the Unity Component After Purchasing It?

> The Unity component is a container primarily used to integrate `Unity WebGL` resources into FVS.
>
> The Unity WebGL resources must be developed by experienced Unity developers using the FVS-Unity-SDK based on actual business requirements.

2. What Is the FVS-Unity Project Workflow?

> First, you need to understand why Unity is chosen for development.
>
> The cost of implementing a `Unity` project is invariably higher than using a 3D city or custom model component. The reason for choosing the `Unity` component is typically that the existing 3D components cannot deliver the desired effects, or you need special functionality that they cannot provide. In such cases, if your budget allows, `Unity` is a good choice.
>
> Once you have decided to use `Unity` for development, you should first outline the dashboard scene requirements. These requirements fall into two categories.
>
> The first category covers the 3D scene content and effects, such as:
>
> - The overall style and visual effects.
> - The number of scenes and the content of each scene.
>
> The second category covers the interactions between FVS and the 3D scenes, such as:
>
> - The data to be integrated (such as labels).
> - The objects that users can interact with by clicking, and whether they need to be linked with charts in an FVS template.
> - Whether buttons are required to trigger behaviors in the 3D scene, such as camera roaming or scene switching.
>
> Once these requirements are defined, identify which features should be implemented entirely in Unity and which require SDK integration. For data, events, and behaviors, define the corresponding data structures, method names, parameters, and other required elements.
>
> Next, develop the Unity scene. During development, you can periodically build the project and import it into FVS for testing, such as verifying that data integration works correctly and that events can be received as expected.

3. What Skills Are Required for Unity Development?

> The FVS team has a Unity implementation partner that can assist customers with Unity project development.
>
> Of course, if the customer has in-house developers who are familiar with Unity, they can develop 3D projects based on the FVS-Unity-SDK.
>
> Other technical staff can also learn Unity on their own — there are plenty of tutorials available online, and AI assistants can help answer questions during the learning process.
>
> Generally speaking, an understanding of the URP render pipeline, the Unity development workflow, and some C# coding skills is sufficient.

4. What Are the Future Plans for the Unity Component?

> Common issues encountered during actual project development will be organized and developed into solutions for easier use going forward.