ThingJS进阶

第5章
CHAPTER 5

本章将重点介绍 ThingJS 数字孪生应用开发的进阶部分，包括组件、插件、预制件的开发方法，以及场景层级控制、数据对接和界面展示。

5.1 组件

5.1.1 组件的定义

组件(Component)是一种对象功能的扩展方式。组件是对象的组成部分，提供物体的生命周期方法，对象和组件的关系如图 5-1 所示。

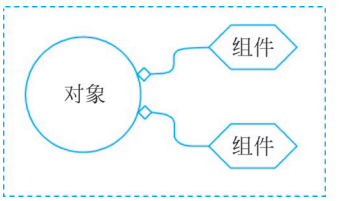

图 5-1 对象和组件的关系

5.1.2 组件的作用和生命周期

使用组件开发，可以大大地减少代码中的重复部分，提高代码的质量和效率。简单地讲，组件的生命周期是指从组件创建到组件销毁的过程，组件生命周期如图 5-2 所示。

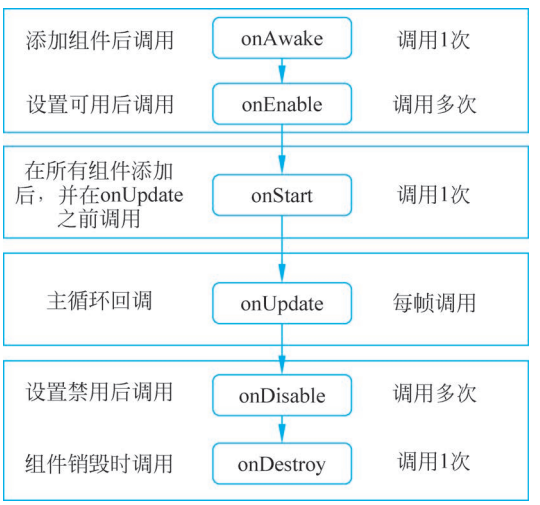

图 5-2 组件生命周期

5.1.3 组件开发

下面通过一个具体案例来介绍如何编写代码,以便实现一个简单的自定义组件。

【例 5-1】 创建一个可以让对象旋转的自定义组件,先将组件添加到立方体上,再通过添加按钮来实现禁用、启用、卸载该组件的功能,代码如下:

```
//教材源代码/examples/component/ComponentRotator.html

//创建自定义组件 MyRotator
class MyRotator extends THING.Component {
    onAwake(params) {
        //获取旋转的速度
        this.speed = params.speed
    }

    onUpdate(deltaTime) {
        //让对象旋转
        this.object.rotateY(this.speed * deltaTime)
    }
}

//初始化程序
const app = new THING.App();

//创建立方体
const box = new THING.Box(3, 3, 3);

//给立方体添加组件
box.addComponent(MyRotator,'rotator',{speed: 10 })

//添加禁用组件按钮
new THING.widget.Button('禁用组件',function () {
    box.rotator.enable = false
});
//添加启用组件按钮
new THING.widget.Button('启用组件',function () {
    if( !box.rotator){
        console.log('The Rotator Component has been removed.')
        return
    }
    box.rotator.enable = true
});
//添加卸载组件按钮
new THING.widget.Button('卸载组件',function () {
    box.removeComponent('rotator')
});
```

注意：deltaTime 是当前帧距上一帧之间的时间，可用于设置动画播放在时间上的准确性。

编写完代码后，保存为 ComponentRotator.html 文件。启动 HTTP 服务，预览运行效果，如图 5-3 所示。可以看到立方体可以逆时针旋转，当单击禁用组件按钮时，立方体停止旋转；当单击启用组件按钮时，立方体恢复旋转；当单击卸载组件按钮时，立方体停止旋转。卸载组件后，单击启用组件无效。

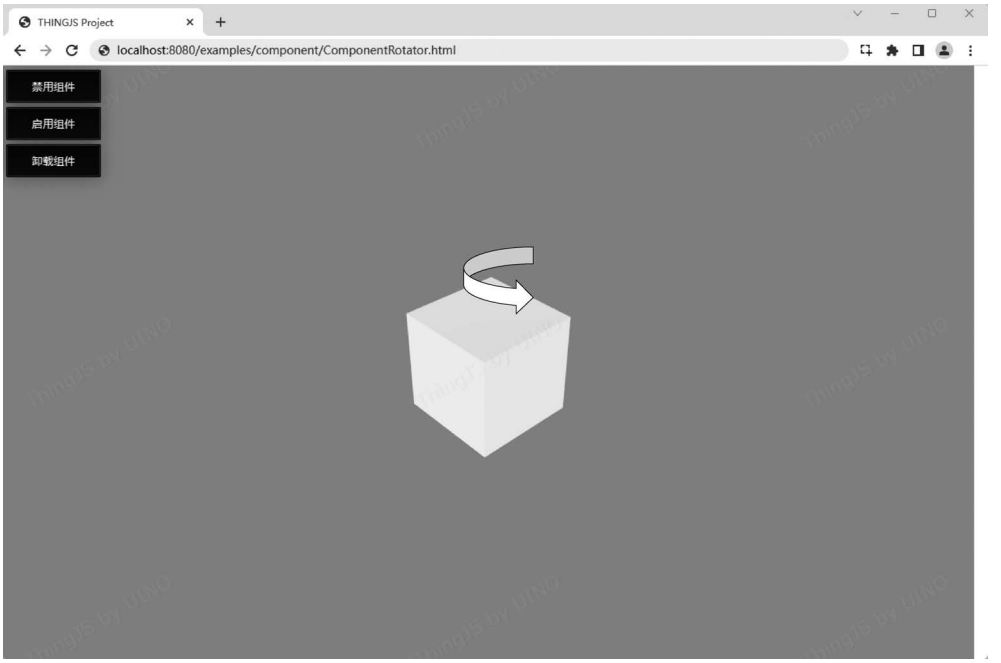

图 5-3 自定义组件的运行效果

在上面的例子中，当使用 addComponent 给立方体添加了自定义组件时，需要传入参数，这里传入的参数分别为自定义组件类名 MyRotator、自定义组件的名字 rotator、自定义组件的参数（旋转速度 speed）等，代码如下：

```
obj.addComponent(MyRotator,'rotator',{speed: 10 });
```

如果需要给查询到的所有对象添加组件，则可以通过下面的方法实现，代码如下：

```
var objs = app.query( * );
objs.addComponent(MyRotator,{speed: 10 });
```

另外，还可以导出自定义的组件的常用属性或方法，并将其直接作用在对象上，代码如下：

```
class MyRotator extends THING.Component {
    //需要导出的属性
    static exportProperties = [
        'speed'
    ]
    //需要导出的方法
    static exportFunctions = [
        'setSpeed'
    ]

    speed = 10;
    setSpeed(value) {
        this.speed = value;
    }
}

const box = new THING.Box();
box.addComponent(MyRotator);
box.speed = 50;            //直接访问成员
box.setSpeed(100);         //直接调用方法
```

通过这个例子,让大家对组件的开发和使用有了一定的认识。

5.2 预制件

5.2.1 预制件介绍

预制件是一个预先制定好的资源,是具有一定属性、行为、效果的对象模板。预制件被用于创建大量重复的对象,例如在一个场景中有多个叉车对象,每个叉车对象都有移动、搬运、装载等行为,那么就可以制作一个叉车预制件以供重复使用。如果只创建一个对象,也可以使用预制件。

5.2.2 预制件开发

在 4.1.3 节中,介绍了用 Blender 导出场景文件(.gltf 格式的文件),在预制件开发中,同样需要用到场景文件。首先打开 Blender,在场景中添加一个立方体,然后给立方体添加自定义属性。

先添加一个 type 属性,属性类型选择字符串型,属性值为 Box。再添加一个 components 属性,属性类型选择 Python,属性值为[{"type": "PrefabRotator","name": "rotator"}],此时立方体就具有了 Box 类型和 PrefabRotator 预制件属性,如图 5-4 所示。

接着选择场景,新建一个自定义属性 type,属性类型选择字符串型,属性值为 prefab,添加预制件属性的过程如图 5-5 所示。

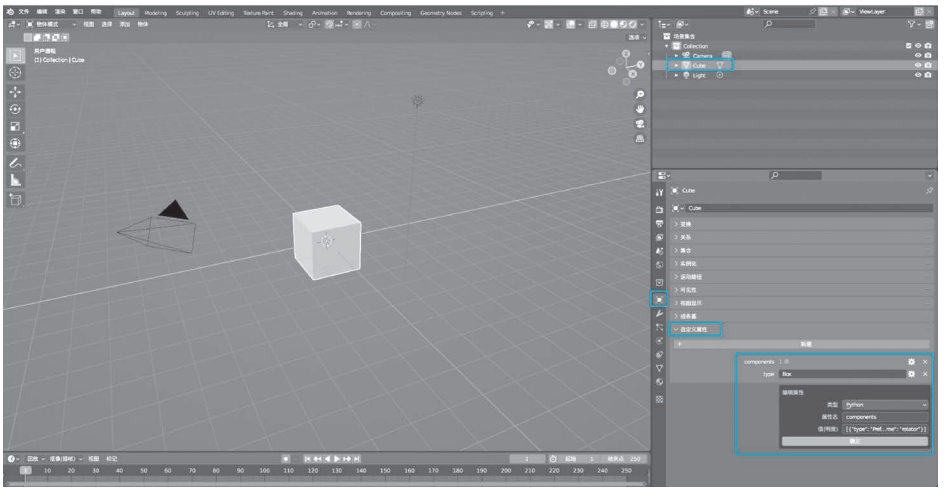

图 5-4　给立方体添加属性

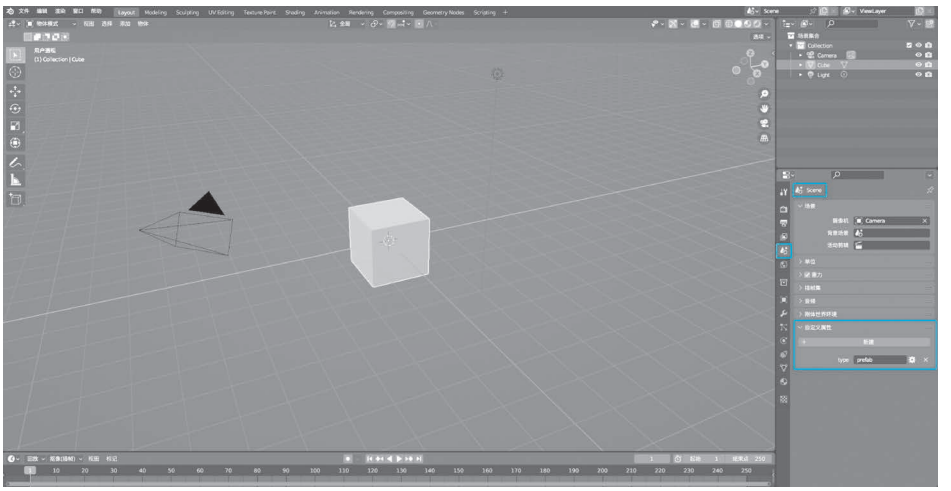

图 5-5　添加预制件属性的过程

按前面介绍的方法导出.gltf 文件，导出时记得要勾选自定义属性，将导出的文件命名为 prefab.gltf，然后打开 prefab.gltf 文件，加入 extensionsUsed 和 extensions 两个配置项，代码如下：

```
"extensionsUsed": [
    "TJS_component"
],
"extensions": {
    "TJS_component": {
        "files": [
            "./PrefabRotator.js"
```

```
            ]
        }
    },
```

extensionsUsed 是指额外的拓展引用，类型为数组，这里只需填写 TJS_component，此名称不可更改。extensions 是指具体的拓展文件配置，类型为对象。这里填写和上面一致的 TJS_component 即可，然后添加 files，类型为数组，将所需拓展的文件路径填写在此。那么这里使用的 PrefabRotator.js 文件应该如何编写呢？下面通过一个案例来介绍如何编写 PrefabRotator.js。

【例 5-2】 编写一个让对象旋转的预制件.js 文件，代码如下：

```
//教材源代码/examples/prefab/PrefabRotator.js

class PrefabRotator extends THING.Component {
    constructor() {
        super()

        //定义 props 属性
        this.props = {
            speed: {
                type: 'number',
                value: 10
            }
        }
    }

    //添加一个 setter 来设置这个值
    set speed(value) {
        this.props.speed.value = value
    }
    //添加一个 getter 来获取这个值
    get speed(){
        return this.props.speed.value;
    }

    onUpdate(deltaTime) {
        this.object.rotateY(this.props.speed.value * deltaTime)
    }

    //除了可以用 setter 设置组件中的值外，还可以使用一个函数来修改值，从而修改对象的行为
    rotateSpeed(speed) {
        this.props.speed.value = speed
    }
}

//将组件注册到 ThingJS
THING.Utils.registerClass( 'PrefabRotator', PrefabRotator);
```

编写完代码后，保存为 PrefabRotator.js 文件，预制件就完成了。

【例 5-3】 预制件完成后应如何加载预制件呢？下面通过编写 HTML 文件来加载预制件，新建一个 PrefabRotator.html 文件，在＜script＞标签中编写加载预制件的代码，代码如下：

```
//教材源代码/examples/prefab/PrefabRotator.html

//初始化程序
const app = new THING.App();

//加载预制件
const prefab = new THING.Entity({url:'./prefab.gltf'});

//加载完成时，获取类型为 Box 的对象，在控制台打印旋转速度
prefab.waitForComplete().then(() => {
    const box = prefab.query(".Box")[0];
    console.log(box.rotator.speed);
})
```

保存 PrefabRotator.html 文件，启动 HTTP 服务，预览预制件的运行效果，如图 5-6 所示。

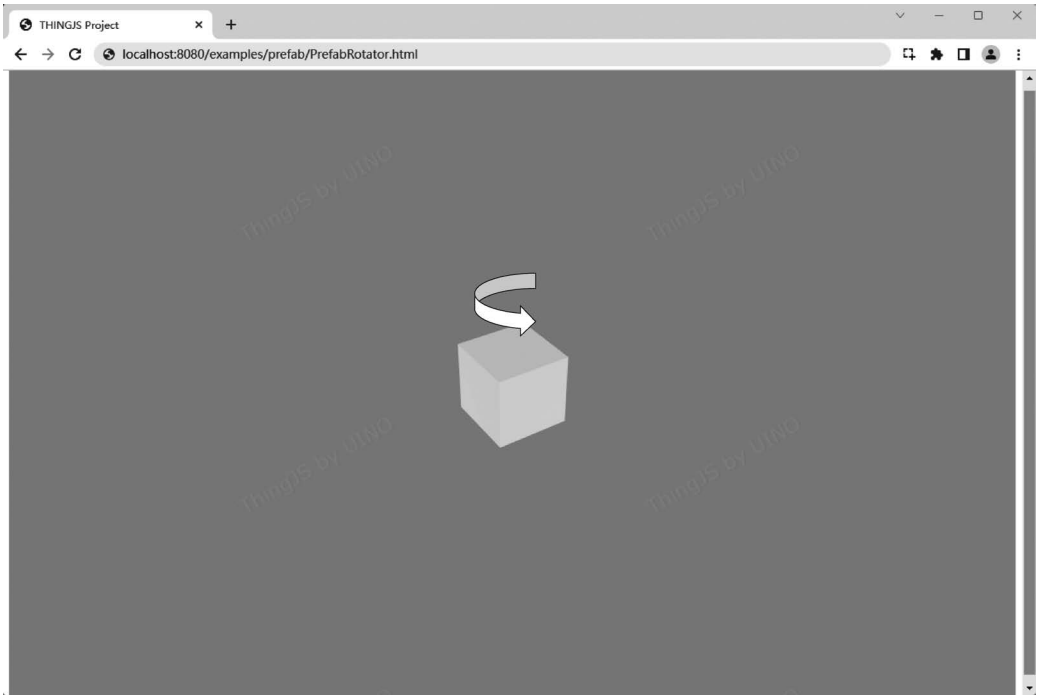

图 5-6 预览创建的预制件 PrefabRotator 的效果

5.3 插件

5.3.1 插件介绍

插件是一种系统功能的扩展方式，可以在插件中开发自定义脚本、使用组件、引用模型、图片等资源，对一个综合的业务功能进行封装和复用。插件有生命周期，插件的生命周期如图 5-7 所示。需要重点关注以下生命周期方法。

(1) onInstall()：插件安装时执行的函数，一些参数的命名和注册与赋值都应该在此周期内执行。

(2) onUpdate()：插件更新时执行的函数，这时可以修改一些参数，例如对象的大小和颜色等。

(3) onUninstall()：插件卸载时执行的函数，可以用来清理一些不再使用的数据，以便保证程序高效运作。

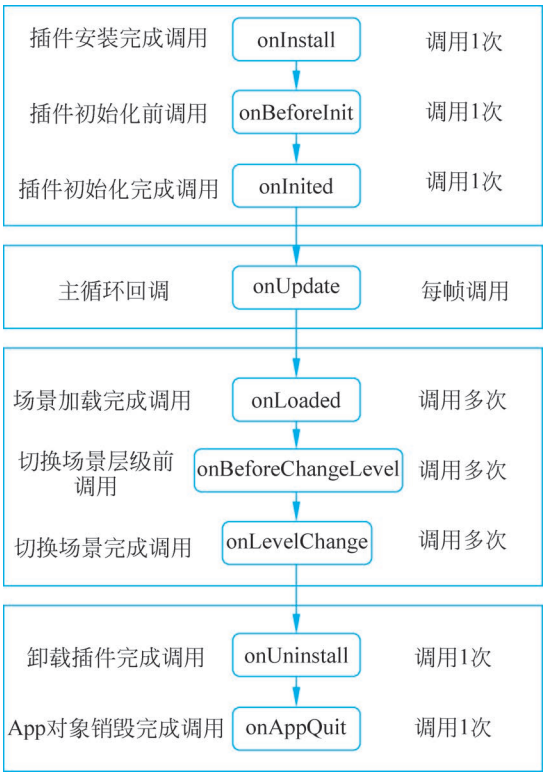

图 5-7 插件的生命周期

5.3.2 插件开发

【例 5-4】 编写一个让立方体旋转的插件，代码如下：

```
//教材源代码/examples/plugin/PluginRotator.html

class MyRotator extends THING.BasePlugin{
    constructor() {
        super()
        this.box = null
    }

    //自定义函数,用来让对象旋转
    rotate(speed) {
        this.box.rotateTo([0,360,0],{
            loopType: THING.LoopType.Repeat,
            time: speed * 1000
        })
    }

    //插件的生命周期函数,在插件安装时被调用
    onInstall() {
        this.box = new THING.Box(2, 2, 2)
    }

    //插件的生命周期函数,在插件卸载时被调用
    onUninstall() {
        this.box.destroy()
    }
}
//注册 ThingJS App
const app = new THING.App();

app.install(new MyRotator(),'MyRotator')

//获取插件并调用插件的自定义函数
const plugin = app.plugins['MyRotator']
plugin.rotate( 10)
```

编写完代码后，保存为 PluginRotator.html 文件。启动 HTTP 服务，预览插件的运行效果，如图 5-8 所示。

当不需要插件时，可以通过 uninstall 卸载插件，代码如下：

```
//卸载插件
app.uninstall('MyRotator')
```

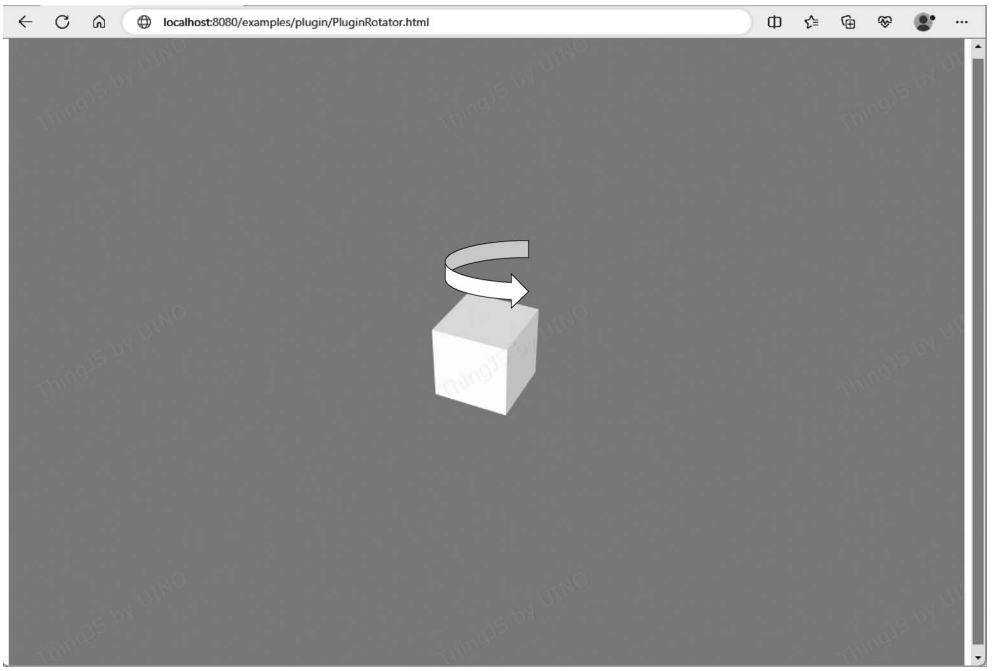

图 5-8　预览创建插件 MyRotator 的效果

5.4　场景和层级

5.4.1　场景的概念和加载的意义

场景(Scene)是具有一定关系的对象组成的集合,一般指三维场景,场景空间承载的对象既可以是室外的山体地形、建筑、道路、各种植被、景观,室内的楼层、房间、设备等,又可以是一些对象组成的一个园区场景,通常通过三维建模软件实现,例如 Blender、3ds Max、Maya 等软件工具。

加载场景是对场景文件中的数据按照空间结构进行对象化处理的过程。在开发过程中,加载场景后,就可以相应地去获取和控制场景中的对象了。

5.4.2　层级和层级切换

场景层级(Scene Level),简称层级,指系统当前正在关注的对象,也被称为这个场景所处的层级,例如楼层层级是指视角处在建筑的某个楼层内,此时,建筑的其他楼层及其他建筑都被隐藏或虚化,只关注当前楼层内的对象。下面通过一个案例来介绍如何实现层级的切换。

【例 5-5】　编写代码实现层级的切换。初始化程序并加载场景,通过 app.on 注册全局

加载事件 THING.EventType.Load，添加按钮，使用 queryByTag 查询到建筑，并且通过 app.levelManager.change 切换到建筑内部；添加按钮，通过 app.levelManager.back 退出建筑，代码如下：

```
//教材源代码/examples/SceneLevel.html

//初始化程序并加载场景
const app = new THING.App({
    url: '../assets/scenes/scene1/scene1.gltf',
})
app.on( THING.EventType.Load, (ev) => {
    new THING.widget.Button('进入建筑',() => {
        const building = app.queryByTags('Building')[0];
        app.levelManager.change( building);
    });
    new THING.widget.Button('退出建筑',() => {
        app.levelManager.back();
    });
})
```

保存 SceneLevel.html 文件，启动 HTTP 服务后预览运行效果，当单击“进入建筑”按钮时，摄像头拉近，展示建筑内部楼层，如图 5-9 所示。当单击“退出建筑”按钮时，回到初始视角。

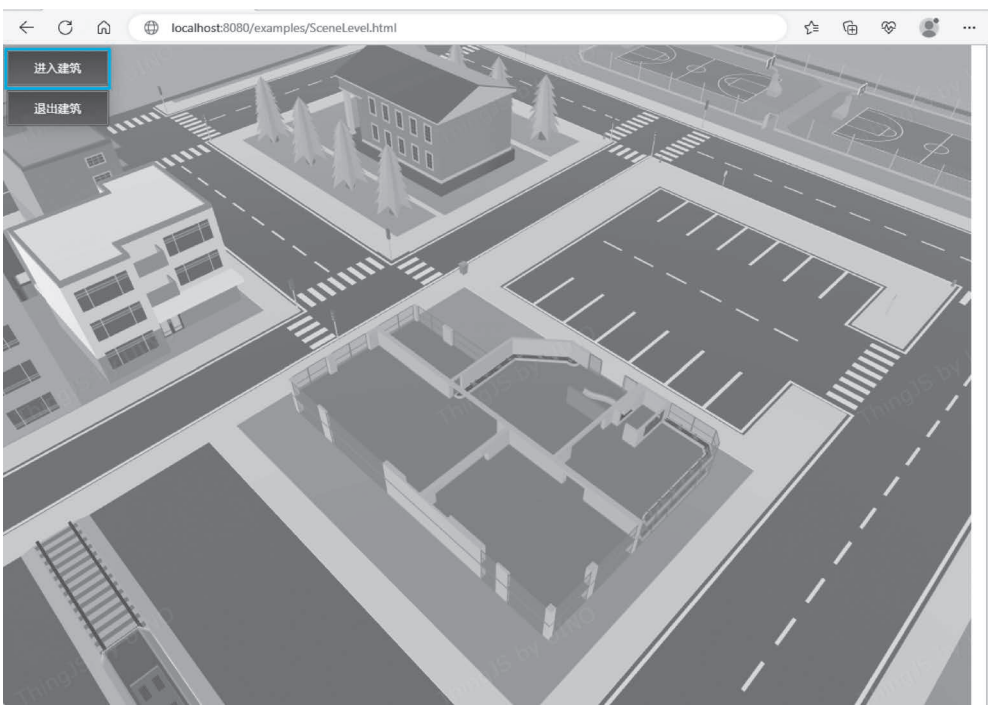

图 5-9 将层级切换到建筑内部

5.5 数据对接

本节主要介绍与第三方物联网系统进行通信(数据传输)的4种常用对接方式及其数据交换原理,它们分别为AJAX、JSONP、WebSocket和MQTT。

5.5.1 数据对接介绍

(1) AJAX(Asynchronous JavaScript and XML):是一种异步的JavaScript与XML技术,使用AJAX技术网页应用能够快速地将增量更新呈现在用户界面上,而不需要重载整个页面,但是AJAX会受到同源策略限制,需要注意跨域问题。数据交换流程为浏览器使用JavaScript借助XMLHttpRequest对象以非阻塞的方式向服务器发送HTTP请求;在得到服务器返回的数据后对页面部分区域进行刷新。

(2) JSONP(JSON with Padding):是JSON的一种使用模式,可以让网页从别的域名(网站)获取资料,即跨域读取数据。JSONP的基本原理就是利用< script >标签没有跨域限制的特点,创建一个包含回调函数的< script >标签,通过< script >标签向服务器请求数据;服务器收到请求后,将数据放在指定回调函数的参数中返回浏览器。

(3) WebSocket:是HTML5提供的一种在单个TCP连接上进行全双工通信的协议(双向通信协议),同时WebSocket允许跨域,其本质是先通过HTTP/HTTPS进行握手后创建一个用于交换数据的TCP连接,服务器端与客户端通过此TCP连接进行数据的双向实时传输,直到有一方主动发送关闭连接请求或出现网络错误才会关闭连接。

(4) MQTT(Message Queuing Telemetry Transport):是一个轻量级协议,该协议构建于TCP/IP上,提供有序、无损、双向连接,MQTT允许跨域。MQTT协议采用的是典型的发布者/订阅者模式,MQTT服务器也称为消息代理(Broker)。客户端既可以为发布者(Publish),也可以为订阅者(Subscribe)。不同客户端之间通过订阅消息和发布消息进行数据交互。

5.5.2 数据对接接口

1. AJAX

AJAX的数据获取是通过XMLHttpRequest对象来实现的,XMLHttpRequest是浏览器提供的JavaScript对象,浏览器可以基于该对象请求到服务器上的数据资源,代码如下:

```
var xhr = new XMLHttpRequest()
xhr.open('GET','https://3dmmd.cn/getMonitorDataById?id=1605')
xhr.send(null)
xhr.onreadystatechange = function () {
    if ( xhr.readyState === 4 && xhr.status === 200) {
        console.log( xhr.responseText)
    }
}
```

(1) open(type,url,async)：初始化请求接口，该接口用于初始化请求所需的信息，不会真正发送该请求。第1个参数为请求的类型，可以为GET、POST等；第2个参数为请求的URL，如果需要参数，则可以直接增加到URL后面；第3个参数为是否需要发送异步请求，默认值为true。

(2) send(data)：发送请求接口，该接口用于执行真正的发送请求指令。参数data表示发送的请求主体数据，如果不需要主体数据，则可填写null。

(3) onreadystatechange：readystate变更事件，当readystate变化时会触发该事件的回调。

(4) readystate：readystate属性表明当前请求的处理状态，请求状态有UNSENT：XMLHttpRequest对象已经被创建，但是尚未调用open方法；OPENED：open方法已经被调用；HEADERS_RECEIVED：send方法已经被调用；LOADING：数据接收中；DONE：请求结束。

(5) status：status属性可以理解为请求在某个处理状态下的状态；status的种类比较多，其中200表示此次请求成功。

在开发环境中通常会引入jQuery库，jQuery库封装了很多前端的实用接口，因此可以直接使用JQuery封装的AJAX方法进行数据对接，对接的封装格式的代码如下：

```
$.ajax( {
    type: "get",
    url: "https://3dmmd.cn/getMonitorDataById",
    data: {"id":1605 },
    dataType: "json",
    success: function (d) {
        console.log(d.data)
    }
});
```

其中，type属性是HTTP请求的方法，通常使用GET方法请求，url是数据来源的URL，data是请求所需的参数；dataType是返回的数据类型，success则是数据返回后的回调函数。

2. JSONP

JSONP是通过<script>标签及其src属性上带有的回调函数的URL来实现的，服务器在接收到请求后，将数据放到回调函数callback的参数中返回浏览器，返回格式的代码如下：

```
function callback(result)
{
    console.log(result);
}
<script type = "text/javascript"
src = "http://www.yiwuku.com/myService.aspx?jsonp = callback">
</script>
```

JQuery 的 AJAX 请求对 JSONP 也进行了封装，因此可以直接使用相关方法请求 JSONP 数据，请求格式的代码如下：

```
$.ajax({
    type: "get",
    url: "https://3dmmd.cn/monitoringData",
    data: {"id": 1605 },
    dataType: "jsonp",
    jsonpCallback: "callback",
    success: function (d) {
        console.log(d.data)
    }
});
```

dataType：返回的数据类型，注意这里必须设置为 JSONP 方式；jsonpCallback：返回数据中的回调函数名，其他参数同 AJAX 数据对接方式。

3. WebSocket

由于并非所有浏览器都支持 WebSocket，所以在使用 WebSocket 之前，需要通过 window.WebSocket 来判断当前浏览器的支持情况。当 window.WebSocket == true 时，可使用如下接口，进行数据对接。

1) WebSocket(url,[protocol])

WebSocket 的构造函数，用于创建一个 WebSocket 实例。第 1 个参数 url 用于指定连接的 URL。第 2 个参数 protocol 是可选的，用于指定可接受的子协议。

2) send(msg)

WebSocket 发送消息的接口，该接口用于主动关闭连接。该接口只有一个参数，该参数为一个 String、ArrayBuffer 或者 Blob 类型的数据，用于表示发送的信息，该信息会被发送到服务器端。

3) close(code)

WebSocket 的关闭连接接口，参数为一个可选的关闭状态号，常见的关闭状态号如下。

(1) 1000：表示正常关闭(默认值)。

(2) 1001：表示离开，例如服务器出现故障，浏览器离开了打开连接的页面。

(3) 1002：表示协议错误。

(4) 1003：表示由于接收到不允许的数据类型而断开连接。

4) onXXX 事件

WebSocket 的监听事件接口，可以设置为一个回调函数来处理相关业务逻辑；具体支持为 onopen、onmessage、onerror、onclose 等，分别对应连接打开、收到消息、建立与连接过程中发生错误和连接关闭事件。

4. MQTT

目前支持 MQTT 协议的 JS 有很多，比较推荐的是 mqtt.js，其功能完善，并且支持的平

台较多。mqtt.js 的相关 API 如下。

(1) mqtt.connect([url],options)：MQTT 连接接口，连接到指定的 MQTT Broker，并返回一个 Client 对象。第 1 个参数用于传入一个 URL 值，该 URL 指向 Broker 代理。第 2 个参数为一个连接的可选配置信息，具体支持的参数有 keepalive、clientId、connectTimeout 等。

(2) Client.publish(topic,message,[options],[callback])：Client 对象发布消息接口，用于 Client 对象向某个 topic 发布消息。第 1 个参数为发送的 topic；第 2 个参数为发送的消息；第 3 个参数为发布消息的可选配置信息，具体支持 Qos、Remain 等；第 4 个参数为发布消息后的回调函数，当发布成功时函数无参数，当发布失败时回调函数有 error 参数。

(3) Client.subscribe(topic/topic array/topic object,[options],[callback])：Client 对象订阅消息接口，用于 Client 对象向某个或者某些 topic 订阅消息。第 1 个参数为订阅的 topic 或 topic 数组；第 2 个参数为订阅消息的可选配置信息；第 3 个参数为订阅消息后的回调函数，当订阅成功时函数无参数，当订阅失败时回调函数有 error 参数。

(4) Client.unsubscribe(topic/topic array,[options],[callback])：Client 对象取消订阅消息接口，用于 Client 对象取消某个或者某些 topic 的订阅消息。第 1 个参数为取消订阅的 topic 或 topic 数组；第 2 个参数为取消订阅消息的可选配置信息；第 3 个参数为取消订阅消息后的回调函数，当取消成功时函数无参数，当取消失败时回调函数有 error 参数。

(5) Client.end([force],[options],[callback])：Client 对象关闭接口，用于关闭当前客户端。第 1 个参数为是否立即关闭客户端，true 表示立即关闭，false 表示需要等待断开链接的消息被接收后关闭客户端；第 2 个参数为关闭客户端时的可选配置信息，具体支持 ReasonCode 等；第 3 个参数为关闭客户端时的回调函数。

(6) Client.on(key,callback)：Client 对象的监听事件接口，用于监听一个或多个常用的事件。第 1 个参数为字符串类型的事件类型，具体支持 connect、disconnect、reconnect、message、error、end 等。

5.5.3 数据对接案例

【例 5-6】 使用 WebSocket 进行数据对接，实现当按下开启按钮时，每隔 5s 读取信息，并在控制台打印消息；当按下“关闭”按钮时，返回“WebSocket 关闭”，实现代码如下：

```
//教材源代码/examples/DataWebSocket.html

let webSocket = null;
let startReading = function () {
    if (!webSocket) {
        webSocket = new WebSocket('wss://3dmmd.cn/wss');
        webSocket.onopen = function () {
            console.log("WebSocket 服务器连接成功");
        };
        webSocket.onmessage = function(evt) {
```

```
                var data = evt.data;
                console.log(data);
            };
            webSocket.onclose = function (evt) {
                console.log("WebSocket 关闭");
                webSocket = null;
            };
        }
    }
    //关闭连接
    let stopReading = function () {
        if(webSocket) {
            webSocket.close();
            webSocket = null;
        }
    }

    //初始化程序
    const app = new THING.App();
    //创建立方体
    const box = new THING.Box(5, 5, 5);

    new THING.widget.Button('开启读取', function () {
        startReading();
    });

    new THING.widget.Button('关闭读取', function () {
        stopReading();
    });
```

在给出的 WebSocket 对接案例中，提供了两个按钮，分别是“开启读取”按钮和“关闭读取”按钮。

当单击“开启读取”按钮时会触发执行 updateData 函数，该函数会判断 webSocket 是否存在，若不存在，则客户端通过构造 WebSocket 对象打开一个 URL 为 wss://3dmmd.cn/wss 的链接，同时定义 webSocket 对象的 onopen、onmessage 和 onclose 事件响应函数。

当连接建立时，webSocket 对象会触发 Open 事件，调用 onopen 函数，在浏览器的控制台打印“WebSocket 服务器连接成功”连接消息。该连接设定每隔 5s 向客户端推送一次数据，因此连接建立之后，webSocket 对象会每隔 5s 触发一次 Message 事件，同时每隔 5s 调用一次 onmessage 函数，在浏览器的控制台每隔 5s 打印一次 data 内容。

当单击“关闭读取”按钮时会触发执行 stopUpdate 函数，webSocket 对象调用 close 函数关闭链接并置空。当链接关闭时，webSocket 对象触发 Close 事件，调用 onclose 函数，在浏览器的控制台打印“WebSocket 关闭”消息。

启动 HTTP 服务后预览运行效果，按 F12 键调出开发者工具，选择控制台，当按下“开启读取”按钮时，控制台打印消息，每隔 5s 读取信息；当按下“关闭读取”按钮时，返回

“WebSocket 关闭”，如图 5-10 所示。

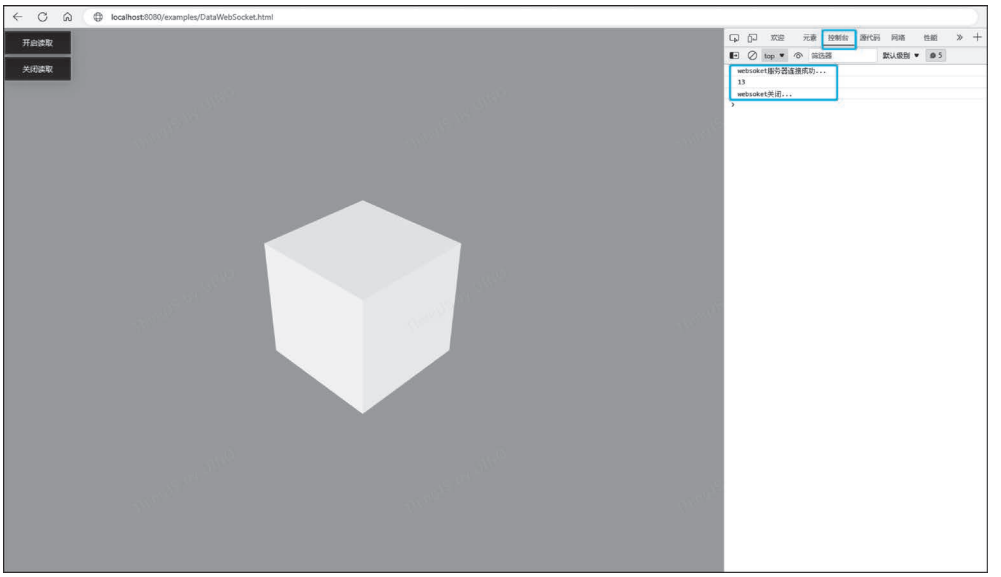

图 5-10 WebSocket 数据对接

5.6 界面展示

5.6.1 Marker

第 4 章介绍了对象标记，给立方体添加了 Marker 对象。添加 Marker 也是一种常用的界面展示方式。将 Marker 作为子对象添加到指定对象上，设置 Marker 的相对位置，使其随对象一同移动。Marker 默认为受距离远近影响，呈现近大远小的三维效果，也会在三维空间中实现前后遮挡。这里，介绍一种绘制 canvas 并添加 Marker 的方法。

【例 5-7】 绘制 canvas，将绘制完成的 canvas 转换为图片对象并添加到 Marker 上，代码如下：

```
//教材源代码/examples/MarkerCanvas.html

//初始化程序
let app = new THING.App();

//创建立方体
let box = new THING.Box();

//将 canvas 转换为图片对象
let image = new THING.ImageTexture( {resource: createTextCanvas('88') })
```

```
let marker = new THING.Marker({
    name: "marker",
    parent: box,
    localPosition: [0, 3, 0],
    scale: [2, 2, 2],
    style: {
        image: image
    }
})

//绘制 canvas
function createTextCanvas(text) {
    const canvas = document.createElement("canvas");
    canvas.width = 64;
    canvas.height = 64;

    const ctx = canvas.getContext("2d");
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.fillStyle = "rgb(32, 32, 256)";
    ctx.beginPath();
    ctx.arc(32, 32, 30, 0, Math.PI * 2);
    ctx.fill();

    ctx.strokeStyle = "rgb(255, 255, 255)";
    ctx.lineWidth = 4;
    ctx.beginPath();
    ctx.arc(32, 32, 30, 0, Math.PI * 2);
    ctx.stroke();

    ctx.fillStyle = "rgb(255, 255, 255)";
    ctx.font = "32px sans-serif";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText(text, 33, 36);
    return canvas;
}
```

绘制 canvas 并转换为图片 Marker 对象的运行效果如图 5-11 所示。

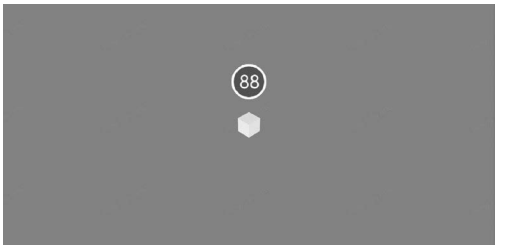

图 5-11　绘制 canvas 并转换为图片 Marker 对象

5.6.2 WebView

【例 5-8】 创建 WebView 对象，将 ThingJS 网页加载到三维场景中，设置页面属性，设置摄像头位置和目标位置，代码如下：

```
//教材源代码/examples/WebView.html

//初始化程序
let app = new THING.App();

//创建页面
let webView = new THING.WebView({
    type: 'WebView',
    url: 'https://www.thingjs.com',
    position: [0, 0.5, 5],
    domScale:0.01, //网页缩放系数
    domWidth: 1920, //页面高度,单位为 px
    domHeight: 1080 //页面高度,单位为 px
});

//设置页面不可拾取交互
webView.pickable = false;

//设置摄像头
app.camera.position = [0,0,20];
app.camera.target = [0,0,0];
```

运行效果如图 5-12 所示。

图 5-12 加载页面

5.6.3 ECharts

ECharts 是一个使用 JavaScript 实现的开源可视化库,提供了常规的折线图、柱状图、散点图、饼图等多种图表,并且支持图表与图表之间进行混合使用。提供交互丰富,可高度个性化定制的数据可视化图表。通过 ECharts 图表可以更直观地查看场景中的数据情况。在 HTML 页面中通过< script >标签引用 echarts.js 文件,代码如下:

```
< script src = "../src/echarts.min.js"></script >
```

【例 5-9】 使用 ECharts 创建全年平均温度、降水量、蒸发量变化图表,代码如下:

```
//教材源代码/examples/Echarts.html

const app = new THING.App();
//创建需要的 DOM 节点,DOM 节点为需要在场景中显示的节点
//背景颜色
let bottomBackground = document.createElement('div');
//标题
let bottomFont = document.createElement('div');
//图表
let bottomDom = document.createElement('div');
//背景样式左上角对齐
let backgroundStyle = 'top: 0px; position: absolute; left: 0px; height: 400px; width: 600px;
background: rgba(41,57,75,0.74);';
//字体样式
let fontStyle = 'position: absolute;top:0px;right:0px;color:rgba( 113,252,244,1);height:
78px;width:600px;line-height: 45px;text-align: center;top: 20px;';
//图表 DIV 样式
let chartsStyle = 'position: absolute;top:80px;right:0px;width:600px;height:300px;';

//设置样式
bottomBackground.setAttribute('style', backgroundStyle);
bottomFont.setAttribute('style', fontStyle);
bottomDom.setAttribute('style', chartsStyle);
//标题文字
bottomFont.innerHTML = '温度降水量平均变化图';
//通过调用 window.echarts 获取 ECharts 对象.通过 init 方法创建图表实例,传入的参数为需要
//ECharts 图表的 DOM 节点,返回的是图表实例
let bottomCharts = window.echarts.init(bottomDom)
//配置图表的属性,图表的各项属性 options 代表的含义可以在 ECharts 官网中查询
let echartOptions = {
"tooltip": {
  "trigger": "axis",
  "axisPointer": {
      "type": "cross",
      "crossStyle": {
          "color": "#999"
```

```
        }
    }
},
"legend": {
    "textStyle": {
        "color": "auto"
    },
    "data": [
        "蒸发量",
        "降水量",
        "平均温度"
    ]
},
"xAxis": [
    {
        "axisLabel": {
            "textStyle": {
                "color": "#fff"
            }
        },
        "type": "category",
        "data": [
            "1月",
            "2月",
            "3月",
            "4月",
            "5月",
            "6月",
            "7月",
            "8月",
            "9月",
            "10月",
            "11月",
            "12月"
        ],
        "axisPointer": {
            "type": "shadow"
        }
    }
],
"yAxis": [
    {
        "type": "value",
        "name": "水量",
        "min": 0,
        "max": 250,
        "interval": 50,
        "splitLine": {
            "lineStyle": {
```

```
                "type": "dotted"
            },
            "show": true
        },
        "nameTextStyle": {
            "color": "#fff"
        },
        "axisLabel": {
            "textStyle": {
                "color": "#fff"
            },
            "formatter": "{value} ml"
        }
    },
    {
        "splitLine": {
            "lineStyle": {
                "type": "dotted"
            },
            "show": true
        },
        "type": "value",
        "name": "温度",
        "min": 0,
        "max": 25,
        "interval": 5,
        "nameTextStyle": {
            "color": "#fff"
        },
        "axisLabel": {
            "textStyle": {
                "color": "#fff"
            },
            "formatter": "{value} °C"
        }
    }
],
"series": [
    {
        "name": "蒸发量",
        "type": "bar",
        "data": [
            2,
            4.9,
            7,
            23.2,
            25.6,
            76.7,
            135.6,
```

```
            162.2,
            32.6,
            20,
            6.4,
            3.3
        ]
    },
    {
        "name": "降水量",
        "type": "bar",
        "data": [
            2.6,
            5.9,
            9,
            26.4,
            28.7,
            70.7,
            175.6,
            182.2,
            48.7,
            18.8,
            6,
            2.3
        ]
    },
    {
        "name": "平均温度",
        "type": "line",
        "yAxisIndex": 1,
        "data": [
            2,
            2.2,
            3.3,
            4.5,
            6.3,
            10.2,
            20.3,
            23.4,
            23,
            16.5,
            12,
            6.2
        ]
    }
],
"color": [
    "#2b908f",
    "#90ee7e",
    "#f45b5b",
```

```
    "#7798BF",
    "#aaeeee",
    "#ff0066",
    "#eeaaee",
    "#55BF3B",
    "#DF5353",
    "#7798BF",
    "#aaeeee"
  ]
  }
  //调用 setOptions 方法将配置好的 options 传入图表
  bottomCharts.setOption(echartOptions);
  //将节点放到页面根节点下
  bottomBackground.appendChild(bottomFont);
  bottomBackground.appendChild(bottomDom);
  document.querySelector('#div3d').appendChild( bottomBackground);
```

创建 ECharts 图表,运行效果如图 5-13 所示。

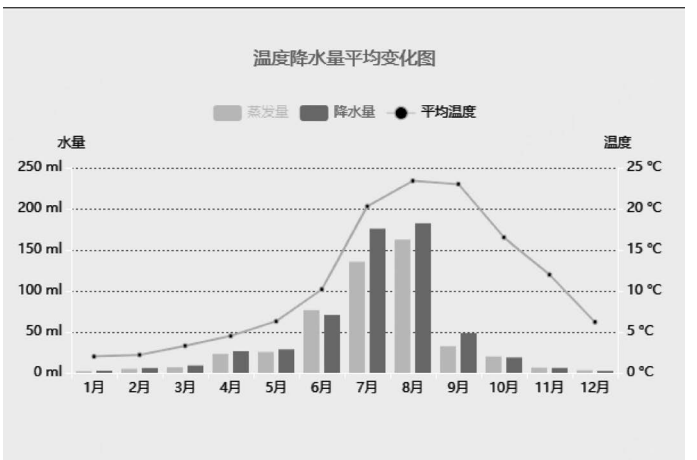

图 5-13 创建 ECharts 图表

5.6.4 Widget

Widget 是一个支持动态数据绑定的轻量级界面库。可以通过 THING.widget 界面库创建 Button 按钮、Banner 通栏及 Panel 面板,其中 Panel 面板中可以添加滑动条、双向按钮、单选框、复选框、文字框等其他组件,通过修改组件值来达到动态修改场景中的对象属性的效果。HTML 页面中通过< script >标签引用 widget.js 文件,代码如下:

```
< script src = "../src/thing.widget.min.js"></script >
```

【例 5-10】 使用 Widget 库创建按钮,代码如下:

```
//教材源代码/examples/WidgetButton.html

let app = new THING.App();
//创建按钮
new THING.widget.Button('WidgetButton',() =>{
  //单击按钮执行回调方法
  console.log('WidgetButton')
})
```

运行后会在浏览器窗口的左上角添加按钮，如图 5-14 所示。

图 5-14 创建按钮

【例 5-11】 使用 Widget 库创建面板，代码如下：

```
//教材源代码/examples/WidgetPanel.html

const app = new THING.App();
//创建 Panel 面板
let panel = new THING.widget.Panel( {
    //设置面板样式
    template: 'default',
    //角标样式
    cornerType: "none",
    //设置面板宽度
    width: "300px",
    //是否有标题
    hasTitle: true,
    //设置标题名称
    titleText: "我是标题",
    //面板是否允许有关闭按钮
    closeIcon: true,
    //面板是否支持拖曳功能
    dragable: true,
    //面板是否支持收起功能
    retractable: true,
    //设置透明度
    opacity: 0.9,
    //设置层级
    zIndex: 99
});
//定义面板数据
let dataObj = {
    pressure: "0.14MPa",
    temperature: "21°C",
    checkbox: { 设备 1: false, 设备 2: false, 设备 3: true, 设备 4: true },
    radio: "摄像机 01",
    open1: true,
    height: 10,
    maxSize: 1.0,
```

```
        iframe: "https://www.thingjs.com",
        progress: 1,
        img: "https://www.thingjs.com/guide/image/new/logo2x.png",
        button1: false,
        button2:true
};
//向 Panel 面板中添加组件
let press = panel.addString(dataObj, 'pressure').caption('水压').isChangeValue(true);
let height = panel.addNumberSlider(dataObj, 'height').caption('高度').step(10).min(0).max
(100).isChangeValue(true).on('change',function(value){
     dataObj.height = value;
});
let open1 = panel.addBoolean(dataObj, 'open1').caption('开关 01');
let radio = panel.addRadio(dataObj, 'radio', ['摄像机 01', '摄像机 02']);
let check = panel.addCheckbox(dataObj, 'checkbox').caption( { "设备 2": "设备 2( rename)" });
let iframe = panel.addIframe(dataObj, 'iframe').caption('视屏');
let img = panel.addIframe(dataObj, 'img').caption('图片');

let button1 = panel.addImageBoolean(dataObj, 'button1').caption('仓库编号').url('https://
www.thingjs.com/static/images/sliohouse/warehouse_code.png');
//可以通过 font 标签设置 caption 颜色
let button2 = panel.addImageBoolean(dataObj, 'button2').caption( '<font color = "red">温度检
测</font>').url('https://www.thingjs.com/static/images/sliohouse/temperature.png');
```

创建面板,运行效果如图 5-15 所示。

图 5-15 创建面板

【例 5-12】 使用 Widget 库创建 Tab 面板，代码如下：

```
//教材源代码/examples/WidgetTable.html

const app = new THING.App();
let panel = THING.widget.Panel( {
    template: "default",
    hasTitle: true,
    titleText: "粮仓信息",
    closeIcon: true,
    dragable: true,
    retractable: true,
    width: "380px"
});

//定义面板数据
let dataObj = {
    '基本信息': {
        '品种': "小麦",
        '库存数量': "6100",
        '保管员': "张三",
        '入库时间': "19:02",
        '用电量': "100",
        '单仓核算': "无"
    },
    '粮情信息': {
        '仓房温度': "26",
        '粮食温度': "22"
    },
    '报警信息': {
        '温度': "22",
        '火灾': "无",
        '虫害': "无"
    },
};
panel.addTab(dataObj);
```

创建 Tab 面板，运行效果如图 5-16 所示。

图 5-16 创建 Tab 面板

【例 5-13】 通过 Widget 库中的 Banner 组件创建一个通栏,代码如下:

```
//教材源代码/examples/WidgetBanner.html

const app = new THING.App();
let banner_left = new THING.widget.Banner( {
    //通栏类型: top 为上通栏(默认), left 为左通栏
    column: 'left'
});

//引入图片文件
let baseURL = "https://www.thingjs.com/static/images/sliohouse/";
//数据对象,用于为通栏中的按钮绑定数据
let dataObj = {
    orientation: false,
    cerealsReserve: false,
    video: true,
    cloud: true
};

//向左侧通栏中添加按钮
let img5 = banner_left.addImageBoolean(dataObj, 'orientation').caption('人车定位').imgUrl
(baseURL + 'orientation.png');
let img6 = banner_left.addImageBoolean(dataObj, 'cerealsReserve').caption('粮食储存').
imgUrl( baseURL + 'cereals_reserves.png');
let img7 = banner_left.addImageBoolean(dataObj,'video').caption('视频监控').imgUrl(baseURL +
'video.png');
let img8 = banner_left.addImageBoolean(dataObj,'cloud').caption('温度云图').imgUrl(baseURL +
'cloud.png');

//为按钮绑定事件
img5.on('change', function (value) {
    //当按钮值改变时触发
    console.log(value)
})

//根据页面调整布局
$('.ThingJS_wrap').css('position','absolute').css('top','0px')
```

创建通栏,运行效果如图 5-17 所示。

图 5-17 创建通栏

5.6.5 CSS 组件

CSS 组件提供了将 HTML/CSS 元素添加到三维场景的能力，包括 CSS2DComponent 和 CSS3DComponent。

在渲染方式上，CSS3DComponent 支持设置界面的渲染类型包括精灵渲染方式和平面渲染方式，CSS2DComponent 只支持精灵渲染方式。在性能方面上，CSS2DComponent 的性能更高。

【例 5-14】 使用 CSS2DComponent 组件给立方体添加一个 HTML 界面，代码如下：

```
//教材源代码/examples/CSS2D.html

//创建 HTML 面板
const sign =
    `<div class = "sign" id = "board" style = "width: 162px; position: absolute">
        <img src = '../assets/images/camera.png' />
    </div>`;

//初始化程序
const app = new THING.App();

//将 HTML 面板添加到 div3d 标签中
$('#div3d').append($(sign));

//创建立方体
const box = new THING.Box();

//创建 CSS2D 组件
let component = new THING.DOM.CSS2DComponent();

//给立方体注册 CSS2D 组件
box.addComponent(component,'cameraSign');

//设置 DOM 元素
box.cameraSign.domElement = document.getElementById('board');

//设置偏移量
box.cameraSign.offset = [0, 3, 0];
```

这里，在 HTML 文件的< head >标签中，通过引用 jquery.min.js 文件来添加 JQuery 库，代码如下：

```
<script src = "../src/jquery.min.js"></script>
```

保存 CSS2D.html 文件，启动服务后预览效果如图 5-18 所示。

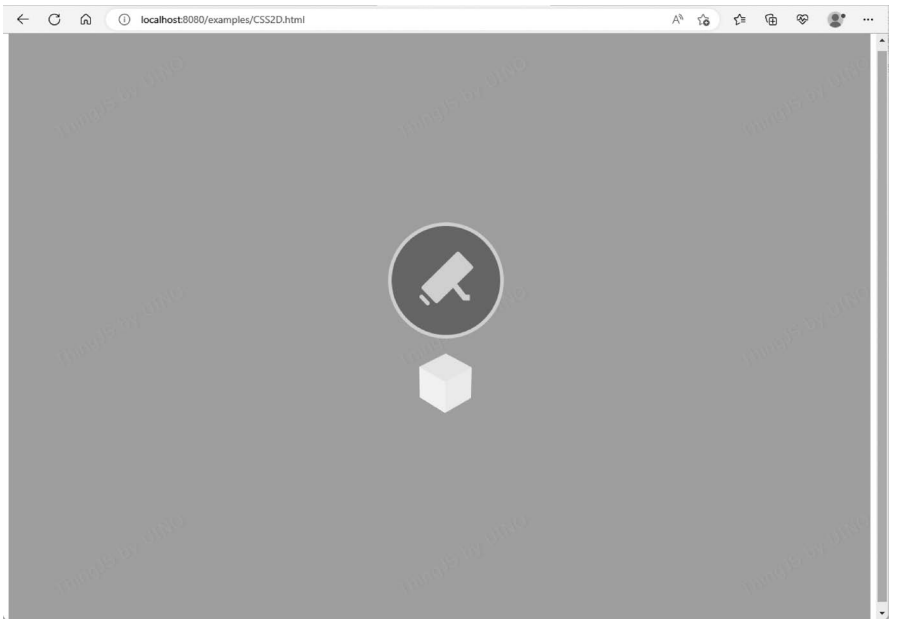

图 5-18　使用 CSS2D 创建 HTML 界面

【例 5-15】　使用 CSS3DComponent 组件给立方体添加一个 HTML 界面,代码如下:

```
//教材源代码/examples/CSS3D.html

//创建 HTML 面板
const htmlElementStr = `
    <style>
        .video_monitor_camera {
            font-size: 12px;
            color: #fff;
            position: relative;
            width: 162px;
            height: 162px;
        }
        .video_monitor_camera .bottom {
            position: absolute;
            top: 160px;
            left: 27px;
        }
    </style>
    <div class="video_monitor_camera" id="board">
        <img class="camera" src='../assets/images/camera.png'/>
        <img class="bottom" src='../assets/images/bottom.png'/>
    </div>`;

//初始化程序
```

```
const app = new THING.App();

//将 HTML 面板添加到 div3d 标签中
$('#div3d').append($(htmlElementStr));

//设置摄像头位置和目标位置
app.camera.target = [0, 6, -3];
app.camera.position = [10, 10, 40];

//创建立方体
let box = new THING.Box(3, 3, 3);

//给立方体添加 CSS3D 组件
box.addComponent(THING.DOM.CSS3DComponent, 'cameraSign');

//设置 DOM 元素
box.cameraSign.domElement = document.getElementById('board');

//设置轴心点
box.cameraSign.pivot = [0.5, -0.5];

//设置渲染类型
box.cameraSign.renderType = THING.RenderType.Plane;
```

保存 CSS3D.html 文件，启动服务后，使用 CSS3D 创建 HTML 界面，预览效果如图 5-19 所示。

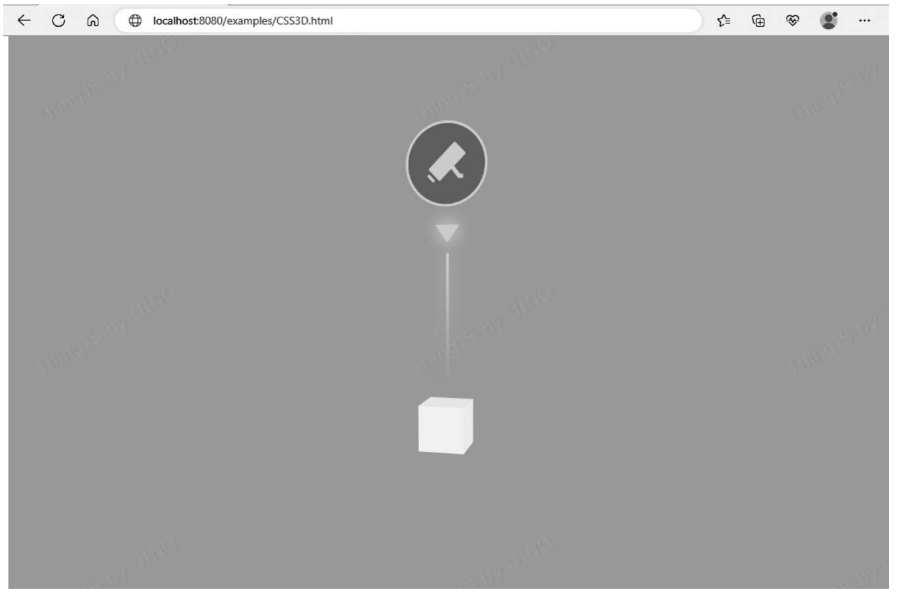

图 5-19 使用 CSS3D 创建 HTML 界面

5.7 人员定位场景案例

5.7.1 创建项目结构

新建一个文件夹,在文件夹内新建以下目录,用于存放对应资源。

(1) assets:存放场景及模型资源。

(2) css:存放样式资源。

(3) images:存放图片资源。

(4) src:存放 ThingJS 文件,以及项目入口 index.js 等项目脚本文件。

(5) index.html:项目主页面文件。

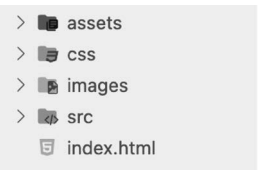

图 5-20 项目的目录结构

项目的目录结构如图 5-20 所示。

在 index.html 文件的< head >标签内引入 thing.js、css/index.css、src/index.js 文件。在< body >标签内创建 id 为 div3d 的 div 标签,用于挂载 App 容器。完整的 index.html 文件的代码如下:

```
//教材源代码/project - examples/personControl/index.html

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>人员定位</title>
    <link rel="icon" href="./images/title.png" />
    <script src="./src/thing.js"></script>
    <link rel="stylesheet" href="./css/index.css" />
</head>

<body>
    <div id="div3d"></div>
</body>
<script src="./src/index.js"></script>

</html>
```

5.7.2 加载场景

接下来开始在 src/index.js 文件里编写本案例的主要代码。

使用 THING.App( )初始化三维 App 容器,并加载园区场景。声明全局变量 campus

来记录园区对象，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

let campus = null;

const app = new THING.App( {
    url: '../assets/campus/ThingJSUINOScene.gltf',
    complete: (e) => {
        //1. 记录园区对象
        campus = e.object;
    },
});
```

5.7.3 创建人员对象

场景加载成功后，根据人员名称、所在位置、人物模型地址，通过 THING.Entity 创建人物孪生体对象。

第 1 步，首先设置人员数据、人员数据格式及内容，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

//注意:每个人物数据里的位置信息 position 来源于点位采集.点位采集方法:在场景的控制台打
//印:app.on('click',e=>console.info(e.pickedPosition))

//人员数据
const personData = [
    {
        name: '悠悠',
        modelUrl: '../assets/man/普通男性.gltf',
        position: [6.213384959090007, 0.1, -177.5293234032834],
    },
    {
        name: '小白',
        modelUrl: '../assets/man/普通男性.gltf',
        position: [-189.44220394798003, 0.1, 12.000833392713702],
    },
    {
        name: '伊斯',
        modelUrl: '../assets/woman/实施工程师女.gltf',
        position: [54.04290252443742, 0.1, -87.88724099236363],
    },
];
```

第 2 步，接下来声明一个函数，命名为 createPerson。在此方法中通过 THING.Entity 创建人物孪生体对象实例。在创建时，设置人物 userData 下的 type 类型，以便对人员执行批量获取操作，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 创建人员孪生体对象
 * @param {String} item.name 人员姓名
 * @param {String} item.modelUrl 人员模型地址
 * @param {Array<Number>} item.position 人员位置
 */
function createPerson(item) {
    const {name, modelUrl, position } = item;
    return new THING.Entity( {
        name,
        url: modelUrl,
        position,
        scale: [5, 5, 5],
        parent: campus,
        userData: {
            type: '人物',
        },
        complete: (e) => {},
    });
}
```

第 3 步,在场景加载完毕后根据数据 personData 创建人员对象,代码如下:

```
//教材源代码/project-examples/personControl/src/index.js

...
const app = new THING.App({
    url: '../assets/campus/ThingJSUINOScene.gltf',
    complete: (e) => {
        ...
        //2. 创建人员模型
        personData.forEach((item) => createPerson(item));
    },
});
...
```

运行 index.html 文件,打开浏览器页面,拉近视角可以看到场景中出现了人员对象,运行效果如图 5-21 所示。

图 5-21 创建人员对象

5.7.4 创建人员标记

为了更直观地显示人员分布情况，可以给人员顶部创建标记，在标记中显示人员名称。这里使用 THING.DOM.CSS3DComponent 组件，给人员注册图文类型标记。

第 1 步，首先声明一个函数，命名为 createMarkerDom，创建标记当中使用的 DOM 元素，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 创建人员标记 DOM 元素
 * @param {String} name 人员姓名
 */
function createMarkerDom(name) {
    const div = document.createElement('div');
    div.className = 'person-marker';
    div.innerHTML = `<div style='cursor:pointer'><span>${name}</span></div>`;
    return div;
}
```

第 2 步，接下来声明函数 createCssMarker，将上述 DOM 元素添加到 App 容器中，在使用孪生体对象的 addComponent 方法注册 CSS3DComponent 组件之后，将 DOM 元素挂载到该组件上，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 给孪生体对象创建 CSS3D 类型标记
 * @param {Object} obj 孪生体对象
 */
function createCssMarker(obj) {
    //1. 获取 DOM 元素，添加到 App 容器中
    const domElement = createMarkerDom( obj.name);
app.container.append(domElement);

    //2. 通过注册 css3d 组件添加 obj 的标记，并命名为 person_marker
    obj.addComponent(THING.DOM.CSS3DComponent, 'person_marker');
const css = obj.person_marker;

    //3. 设置标记的位置，绑定 DOM 元素
    css.pivot = [0.5, -1.3];
    css.domElement = domElement;
}
```

第 3 步，在人员创建函数 createPerson 的回调函数里进行调用，代码如下：

```
//教材源代码/project - examples/personControl/src/index.js

function createPerson(item) {
    const { name, modelUrl, position } = item;
    return new THING.Entity({
        ...
        complete: (e) => {
            //1. 获取创建完成的人员对象
            const person = e.object;
            //2. 给人员创建标记
            createCssMarker(person);
        },
    });
}
```

运行 index.html 文件，打开浏览器页面，可以看到场景中的人员顶部都生成了一个标记，标记中包含图片及人员名称，创建人员标记的效果如图 5-22 所示。

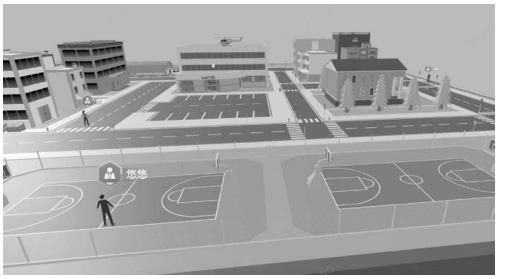

图 5-22 创建人员标记的效果

5.7.5 定位事件

接下来对人员及其标记绑定定位事件，当使用左键单击人员、人员标记时，执行摄影机飞行将视角拉近；当双击右键时，取消定位，视角还原为场景的默认视角。

1. 人员定位

首先声明一个摄像机飞行事件，代码如下：

```
//教材源代码/project - examples/personControl/src/index.js

/**
 * @description 摄像机飞行
 * @param {Object} target 目标孪生体对象
 * @param {Function} cb 飞行结束后的回调事件
 */
function cameraFly(target, cb) {
    app.camera.flyTo({
        target,
```

```
            time: 1000,
            distance: 20,
            complete: () => {
                cb && cb();
            },
        });
    }
```

声明函数 bindSingleClick，给对象绑定左键单击事件并在人员创建完毕后调用。当单击人员对象时，执行摄像机飞行事件，代码如下：

```
//教材源代码/project - examples/personControl/src/index.js

/**
 * @description 对象绑定左键单击事件
 * @param {Object} obj 孪生体对象
 */
function bindSingleClick(obj) {
    obj.on(THING.EventType.Click,(e) => {
        if (e.button === 0) {
            cameraFly(obj);
        }
    });
}

//在人员创建完毕的回调函数里调用 bindSingleClick 方法
function createPerson(item) {
    const {name, modelUrl, position} = item;
    return new THING.Entity({
        ...
        complete: (e) => {
            //1. 获取创建完成的人员对象
            const person = e.object;
            //2. 给人员创建标记
            createCssMarker(person);
            //3. 人员绑定左键单击事件
            bindSingleClick(person);
        },
    });
}
```

运行 index.html 文件，打开浏览器页面，单击一个人员对象，可以观察到视角切换到了该人员对象附近。

2. 标记定位

在创建人员标记时，创建了一个与标记绑定的 DOM 元素。现在给这个 DOM 元素绑定单击事件，实现标记的定位事件，代码如下：

```
function createCssMarker(obj) {
    ...
    css.domElement = domElement;
    //4. 给 DOM 元素绑定单击事件
    domElement.onclick = () => cameraFly(obj);
}
```

刷新页面之后单击其中的一个人员标记,视角切换到该人员附近,操作效果与左键单击人员模型一致。

3. 视角还原

首先,在场景加载完毕时,将园区调整到一个满意的视角,在浏览器控制台通过 app.camera.target 及 app.camera.position 获取摄影机的默认视角并保存,运行效果如图 5-23 所示。

```
> app.camera.target
<- (3) [3.769242704351535, -1.8620023541182378, -50.11113316347249]
> app.camera.position
<- (3) [-77.22022681144554, 94.32731469894514, -309.9243192954776]
```

图 5-23 控制台打印摄像机视角

在 src/index.js 文件的顶部声明变量 defaultView,保存以上数据,代码如下:

```
//设置场景默认视角
const defaultView = {
    target: [-6.609600761047991, 0.39999018625129523, -93.9594556614312],
    position: [-38.074039116635944, 102.11487831643213, -314.7837660292081],
};
```

声明函数 backToDefaultView,用于执行回到园区默认视角的操作,代码如下:

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 回到园区默认视角
 * @param {Function} cb 回到默认视角之后的回调事件
 */
function backToDefaultView(cb) {
    app.camera.flyTo({
        ...defaultView,
        complete: () => cb && cb(),
    });
}
```

接下来注册一个右键双击事件,并在场景加载完毕后调用。当在场景中执行鼠标右键

双击时，回到园区默认视角，代码如下：

```
...
const app = new THING.App( {
    url: '../assets/campus/ThingJSUINOScene.gltf',
    complete: (e) => {
        ...
        //3. 右键双击事件注册
        dblclick();
    },
});
...

/**
 * @description 注册鼠标右键双击事件
 */
function dblclick( ) {
    app.on( THING.EventType.DBLClick, (e) => {
        if ( e.button === 2) {
            backToDefaultView();
        }
    });
}
```

刷新页面，当使用鼠标左键单击人员对象或者人员标记时，视角切换到被单击的人员附近；当在场景中执行鼠标右键双击时，视角回到园区默认视角。

5.7.6 人员行走

为了使场景更加丰富，可以让人员根据路径点位进行移动。在本案例中给每个人员提供了一些路径点位，通过定时随机推送点位数据，让人员行走起来。

1. 设置数据

首先声明一个全局变量 routeDatas，保存每个人员的点位数据，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

//人员路径点位数据
const routeDatas = {
    悠悠: [
        [6.213384959090007, 0.1, -177.5293234032834],
        [-21.2088889321722, 0.1, -177.04905444624734],
        [-48.0122922044485, 0.1, -181.5357741614862],
        [-80.50517022882623, 0.1, -178.748029345187],
        [-104.18977999523601, 0.1, -180.0202804955114],
    ],
    小白: [
        [-189.44220394798003, 0.1, 12.000833392713702],
```

```
            [ - 158.1897925428529, 0.1, 12.187102614381331],
            [ - 117.99871799845272, 0.1, 12.832986204660415],
            [ - 69.24975074784928, 0.1, 12.248780539643946],
            [ - 48.7666450450301, 0.1, 12.519957234189548],
        ],
        伊斯: [
            [54.04290252443742, 0.1, - 87.88724099236363],
            [53.13752963890114, 0.1, - 28.47973963860879],
            [52.88835514710519, 0.1, - 5.994313014155281],
            [52.810582253523265, 0.1, 24.556240243422266],
            [53.061559269011994, 0.1, 50.18269783789641],
        ],
    };
```

接下来声明一种方法 setData,随机获取每个人员的点位,代码如下:

```
    //教材源代码/project - examples/personControl/src/index.js

    /**
    * @description 获取每个人员的随机点位
    */
    function setData() {
        return Object.keys(routeDatas).reduce( (prev, name) => {
            const routes = routeDatas[name];
            const index = Math.floor(Math.random() * 8);
            const targetPosition = routes[index];
            prev[name] = targetPosition || routes[0];
            return prev;
        }, {});
    }
```

获取人员的随机点位,声明全局变量,targetPoint 用于存储每个人员的下一个点位数据,historyData 用于存储每个人员的历史路径数据,代码如下:

```
    //教材源代码/project - examples/personControl/src/index.js

    let historyData = new Map();     //存储每个人员的历史点位数据
    let targetPoint = null;          //存储每个人员的下一个位置数据
    /**
    * @description 获取人员的随机点位,存储在全局变量里
    */
    function setRandomData() {
        //1. 获取每个人员的随机点位
        targetPoint = setData();
        Object.keys(targetPoint).forEach((name) => {
            //2. 获取每个人员对应的点位历史数据
            const posDatas = historyData.get(name);
```

```
        //3. 某个人员对应的历史点位数据如果存在,则继续存储;如果不存在,则先新建一个集
        //合,再存储
        const historyPathData = posDatas ? posDatas : new Set();
        historyPathData.add(targetPoint[name]);
        historyData.set(name, historyPathData);
    });
}
```

2. 路径移动

声明函数 walkByRoute,用于执行对象沿路径行走的相关逻辑。在 ThingJS 中,通过调用孪生体对象的 movePath 方法,可以让对象沿着路径移动。在本案例中要让人员行走起来,还需要调用人员的行走动画。完整的 walkByRoute 函数的代码如下:

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 对象沿着路径行走
 * @param {Object} obj 孪生体对象
 * @param {Array<Array>} path 路径数据
 */
function walkByRoute(obj,path) {
    //1. 计算从当前点位走到目标点位的用时
    const distance = calculateDistance(path);
    const time = ((distance/1) * 1000) / 5;
    //2. 调用人员行走动画
    obj.playAnimation({name: '走', loopType: THING.LoopType.Repeat});
    //3. 执行人员沿着路径行走
    obj.movePath(path, {
        time,
        next: (ev) => {
            //获取相对下一个目标点位的旋转值
            const quaternion = THING.Math.getQuatFromTarget(
                ev.from,
                ev.to,
                [0,1,0]
            );
            //在 1s 内将物体转向到目标点位
            ev.object.lerp.to( {
                to: {
                    quaternion,
                },
                time: 1000,
            });
        },
        complete: (e) => {
            obj.stopAnimation('走');
        },
```

```
    });
}

/**
 * @description 计算空间两点之间的距离
 * @param {Array<Array>} 长度为 2 的点位数组
 */
function calculateDistance(path) {
    const [x, y, z] = path[0];
    const [x1, y1, z1] = path[1];
    return Math.trunc(Math.hypot(x - x1, y - y1, z - z1));
}
```

声明函数 execWalk,先对获取的人员点位数据进行处理,然后调用 walkByRoute 方法,控制场景中的所有人员从其当前所在位置走到下一个位置,代码如下:

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 人员行走至下一个点位
 */
function execWalk() {
    Object.keys(targetPoint).forEach((personName) => {
        //1. 根据人员名称获取人员孪生体对象
        const person = app.query(personName)[0];
        //2. 获取开始点位:当前人员的位置
        const startPoint = person.position;
        //3. 获取结束点位:定时器随机推送的人员位置
        const endPoint = targetPoint[personName];
        //4. 组成路径数据
        const path = [startPoint, endPoint];
        //5. 执行人员沿路径行走
        walkByRoute(person, path);
    });
}
```

3. 数据推送

最后声明函数 pushData,每隔 10s 进行一次数据推送,执行人员从当前点位行走至下一个点位,并在场景加载完毕后调用,代码如下:

```
//教材源代码/project-examples/personControl/src/index.js

...
const app = new THING.App( {
    url: '../assets/campus/ThingJSUINOScene.gltf',
    complete: (e) => {
        ...
```

```
        //4. 定时推送数据
        pushData();
    },
});

let timer = null //定时器

/**
 * @description 定时推送人员点位数据
 */
function pushData() {
    setRandomData();
    execWalk();
    timer && clearInterval(timer);
    timer = setInterval(() => {
        setRandomData();
        execWalk();
    }, 10000);
}
```

刷新页面，可以观察到场景中的人员对象行走起来了，运行效果如图 5-24 所示。

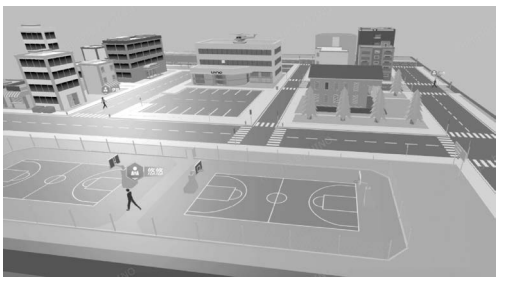

图 5-24 人员在场景里行走

5.7.7 视角跟随

在人员定位结束后，将摄影机锁定在当前人员对象上，可以实现视角跟随的效果。

1. 跟随/停止跟随

在 ThingJS 中，可以通过 app.on 给孪生体对象注册 update 事件，在每帧更新时执行事件回调方法，实现视角跟随，可以通过 app.off 卸载停止视角跟随，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 视角跟随
 * @param {Object} person 人员孪生体对象
 */
function followPerson(person) {
```

```
    //1. 给对象注册 update 事件
    app.on(
        'update',
        () => {
            //2. 在事件回调函数里更改摄影机观察对象、观察位置并绑定观察目标
            app.camera.position = person.selfToWorld( [0, 5, -10]);
            app.camera.target = person.position;
            app.camera.object = person;
        },
        //3. 指定事件 tag,以便对该事件进行卸载
        'camerafollowPerson'
    );
}

/**
 * @description 停止视角跟随事件
 */
function stopFollow() {
    app.off( 'update', 'camerafollowPerson');
}
```

注意：在 ThingJS 中只有在注册事件时指定了 tag，才能通过 app.off 卸载。

2. 优化定位

在 5.7.5 节里，在鼠标单击孪生体对象和标记之后，执行 cameraFly 事件进行定位飞行。

现在引入视角跟随事件，对定位过程进行优化。首先，声明 locate 方法，在定位飞行结束时调用视角跟随功能，并替换掉 bindSingleClick 和 createCssMarker 里对 cameraFly 的调用。在对不同人员切换定位时，先回到园区默认视角，再执行定位操作，代码如下：

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 定位及视角跟随
 * @param {String} name 人员名称
 */
function locate(name) {
    const person = app.query(name)[0];
    //1. 如果当前摄像机被锁定,则先解锁
    stopFollow();
    //2. 先回到园区默认视角再定位
    backToDefaultView(() => {
        cameraFly(person,() => followPerson(person));
    });
}
```

```
function bindSingleClick(obj) {
    obj.on(THING.EventType.Click, (e) => {
        if (e.button === 0) {
            //cameraFly(obj);
            locate(obj.name);
        }
    });
}

function createCssMarker(obj) {
...

    //domElement.onclick = () => cameraFly(obj);
    domElement.onclick = () => locate(obj.name);
}
```

3. 结束跟随

右击，停止定位操作，退出视角跟随，再回到园区默认视角，代码如下：

```
function dblclick() {
    app.on( THING.EventType.DBLClick,(e) => {
        if (e.button === 2) {
            stopFollow();
            backToDefaultView();
        }
    });
}
```

刷新页面，可实现单击页面中的任意一个人员进行定位；当视角被拉近到目标附近之后，摄影机被锁定在目标对象上；如果目标正在行走，则视角将一路跟随，直到右击后退出跟随。

5.7.8 二三维交互

要实现二三维交互，首先需要在页面上创建一个简单的列表，通过列表对场景中的人员进行定位，并可以查看人员的历史轨迹数据，下面介绍它的实现过程。

第 1 步，创建人员定位列表。首先在 index.html 文件中添加创建人员定位列表，代码如下：

```
//教材源代码/project-examples/personControl/index.html

<!DOCTYPE html>
<html lang="en">

<head>
    ...
```

```
</head>

<body>
    <div id="div3d"></div>
    <div class="list-wrap">
        <div class="title-info">
            <span class="name">姓名</span>
            <span class="locate">定位</span>
            <span class="history">历史轨迹</span>
        </div>
        <div class="person-info">
            <span class="name">悠悠</span>
            <span onclick="locate('悠悠')">
                    <img alt="locate" src="https://static.3dmomoda.com/textures/
21110314gabslfaadqbtrv2fumzllnro.png"></img>
            </span>
            <span class="history" onclick="checkHistory(this,'悠悠')">查看</span>
        </div>
        <div class="person-info">
            <span class="name">小白</span>
            <span onclick="locate('小白')">
                    <img alt="locate" src="https://static.3dmomoda.com/textures/
21110314gabslfaadqbtrv2fumzllnro.png"></img>
            </span>
            <span class="history" onclick="checkHistory(this,'小白')">查看</span>
        </div>
        <div class="person-info">
            <span class="name">伊斯</span>
            <span onclick="locate('伊斯')">
                    <img alt="locate" src="https://static.3dmomoda.com/textures/
21110314gabslfaadqbtrv2fumzllnro.png"></img>
            </span>

            <span class="history" onclick="checkHistory(this,'伊斯')">查看</span>
        </div>
    </div>
</body>
<script src="./src/index.js"></script>

</html>
```

人员定位列表的效果如图 5-25 所示。

第 2 步,创建轨迹。在 5.7.6 节里,声明了全局变量 historyData,用于存储每个人员的历史轨迹数据。根据这些数据,可以通过 ThingJS 提供的 THING.RouteLine 方法创建轨迹路线,代码如下:

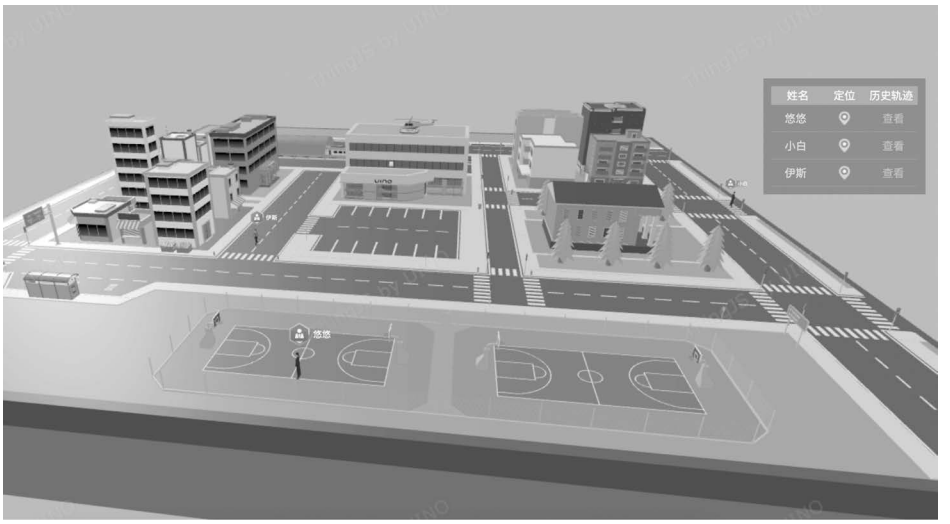

图 5-25 人员定位列表

```
//教材源代码/project - examples/personControl/src/index.js

/**
 * @description 创建轨迹线
 * @param {String} name 人员名称
 * @param {Array<Array>} 路径数据
 */
function createRoute(name,data) {
    //1. 判断数据是否合法
    if ( !data || data.length < 2) return;
    //2. 如果名为`${name}_route`的轨迹线已经存在,则先销毁,避免重复创建
    const route = app.query(`${name}_route`)[0];
    if (route) route.destroy();
    //3. 获取轨迹线的图片资源
    const routeImage = new THING.ImageTexture(
        'https://static.3dmomoda.com/textures/diy_offline_260560_1629883386732.png'
    );
    //4.创建 RouteLine 类型路径轨迹线
    return new THING.RouteLine( {
        name: `${name}_route`,
        selfPoints: data,
        parent: campus,
        width: 2,
        arrow: false,
        cornerRadius: 3,
        cornerSplit: 4,
        style: {
            image: routeImage,
```

```
            },
        });
    }

    /**
     * @description 销毁指定轨迹线
     * @param {String} name 人员名称
     */
    function destroyRoute(name) {
        const route = app.query(`${name}_route`)[0];
        route?.destroy( );
    }
```

第 3 步,查看轨迹。在 index.html 文件里,历史轨迹的“查看”一项绑定了单击事件 checkHistory,在单击事件内部执行查看轨迹的逻辑。首先从存储的全局变量 historyPath 里,根据人员名称获取当前人员的历史点位数据,根据页面操作将“查看轨迹”一栏的文本更新为“查看”或者“取消”,同时对应创建或者销毁轨迹线,代码如下:

```
//教材源代码/project-examples/personControl/src/index.js

/**
 * @description 查看历史轨迹
 * @param {HTMLElement} dom 单击的 DOM 节点
 * @param {String} name 人员名称
 */
function checkHistory(dom,name) {
    //1. 获取存储的历史路径数据
    const historyPath = [...historyData.get(name)];
    //2. 根据面板操作选择创建或者销毁历史路径
    const text = dom.innerHTML;
    text === '查看'? createRoute(name, historyPath) : destroyRoute(name);
    //3. 更新面板数据
    dom.innerHTML = text === '查看'? '取消' : '查看';
}
```

本章小结

本章首先介绍了组件、插件、预制件的概念,以及开发方法和使用方法,然后对场景和场景层级控制进行了讲解。接着介绍了数据对接的基本概念、数据对接的接口和对接方法,此外还介绍了几种界面的开发方法。最后,通过人员定位的综合案例对本章知识点进行了巩固。

本章习题

编程题

（1）编写一个预制件，让小叉车按既定路线行驶。

（2）在查看轨迹线的状态下，如果推送的数据发生变化，则自动刷新轨迹线。

（3）结合“建筑监控案例”一节的内容，在建筑内部创建一个人员，实现本节内容所示的人员定位流程。