

成

绩



**专业综合设计**

题目名称**酒店自助点餐系统**

学生学院 计算机学院

专业班级 13级软件工程（1）班

学 号 3113006215

学生姓名 陈永坤

指导教师 路 璐

201 7 年 1 月 2 日

**目录**

[1 绪论 1](#_Toc481433710)

[1.1 研究背景 1](#_Toc481433711)

[1.2 人体运动捕捉 1](#_Toc481433712)

[1.2.1 运动捕捉设备 1](#_Toc481433713)

[1.2.2 基于Leap Motion的手势捕捉 1](#_Toc481433714)

[1.3 论文研究意义 1](#_Toc481433715)

[1.4 论文主要内容 1](#_Toc481433716)

[2 Leap Motion 2](#_Toc481433717)

[2.1 硬件设备 2](#_Toc481433718)

[2.2 多角成像技术 2](#_Toc481433719)

[2.3 特征点匹配 4](#_Toc481433720)

[2.4 坐标系统 4](#_Toc481433721)

[2.5 运动数据追踪 4](#_Toc481433722)

[2.5.1 手部模型 5](#_Toc481433723)

[3 Unity 8](#_Toc481433724)

[3.1 Unity开发引擎 8](#_Toc481433725)

[3.2 Visual Studio集成开发环境 9](#_Toc481433726)

[4 Leap Motion应用设计 10](#_Toc481433727)

[4.1 主题设计 10](#_Toc481433728)

[4.2 基本功能设计 10](#_Toc481433729)

[4.2.1 菜单界面 10](#_Toc481433730)

[4.2.2 主题场景 10](#_Toc481433731)

[4.2.3 双主题模式 11](#_Toc481433732)

[4.2.4 镜头转换方式 12](#_Toc481433733)

[4.3 构建3D场景 12](#_Toc481433734)

[4.3.1 Leap Motion对象配置 12](#_Toc481433735)

[4.3.2 客厅场景搭建 13](#_Toc481433736)

[4.3.3 走廊场景搭建 14](#_Toc481433737)

[4.3.4 浴室场景搭建 14](#_Toc481433738)

[4.3.5 厨房场景搭建 15](#_Toc481433739)

[4.3.6 光照烘培 16](#_Toc481433740)

[4.4 界面设计 16](#_Toc481433741)

[4.4.1 UI效果要求 16](#_Toc481433742)

[4.4.2 画布属性 18](#_Toc481433743)

[4.4.3 菜单界面 18](#_Toc481433744)

[4.4.4 场景界面 19](#_Toc481433745)

[5 详细设计与编码 21](#_Toc481433746)

[5.1 集成Leap Motion 21](#_Toc481433747)

[5.1.1 Leap Motion插件资源 21](#_Toc481433748)

[5.1.2 主要API 22](#_Toc481433749)

[5.2 代码整体框架搭建 24](#_Toc481433750)

[5.2.1 MVC框架简介 24](#_Toc481433751)

[5.2.2 观察者模式 25](#_Toc481433752)

[5.2.3 框架实现思路 26](#_Toc481433753)

[5.3 全局管理器 27](#_Toc481433754)

[5.3.1 单例模式 27](#_Toc481433755)

[5.3.2 单例管理器 29](#_Toc481433756)

[5.4 应用场景模块实现 30](#_Toc481433757)

[5.4.1 资源管理 30](#_Toc481433758)

[5.4.2 UI界面 32](#_Toc481433759)

[5.4.3 镜头控制 33](#_Toc481433760)

[5.4.4 主场景 35](#_Toc481433761)

[5.5 资源打包与发布 46](#_Toc481433762)

[6 手势交互操作与测试 46](#_Toc481433763)

[6.1 Leap Motion驱动参数调整 46](#_Toc481433764)

[6.2 交互测试和结果分析 46](#_Toc481433765)

[参考文献 47](#_Toc481433766)

# 绪论

## 研究背景

## 人体运动捕捉

### 运动捕捉设备

### 基于Leap Motion的手势捕捉

## 论文研究意义

## 论文主要内容

# Leap Motion

Leap Motion是一款由美国Leap公司所出的专用于跟踪、捕获人体手部运动数据的体感控制器，它提供一种能够识别手势动作并进行手势交互的解决方案。[基于Leap Motion 的三维识别方法]

## 硬件设备

Leap Motion是一个基于双目视觉的小型手势识别设备，配合一条专用的USB数据线，将其连接到PC机或虚拟现实头盔，即可实时反馈传感器所捕获并构建的手部数据帧到处理器。



图2.1 Leap Motion设备

## 多角成像技术

Leap Motion采用基于双目视觉的多角成像技术，在设备上安装有3个LED红外光源和2个灰阶摄像头传感器，可以一次性感知所测物体的所有像素，并使用预置算法进行处理。[基于Leap Motion的数字手势识别]

当两个摄像头对当前环境进行拍摄，会得到两张具有不同视角的照片，再根据摄像头的各项已知参数和相对位置，以及相同物体在画面中的不同位置，即可计算出物体实际距离摄像头的景深深度。[带你了解世界最先进的手势识别技术 -- 微软，凌感，Leap...]

景深的具体计算方法采用的是“三角测量法”，即利用三角形相似原理计算出单点目标在双摄像头下的景深z，如图2.2所示。

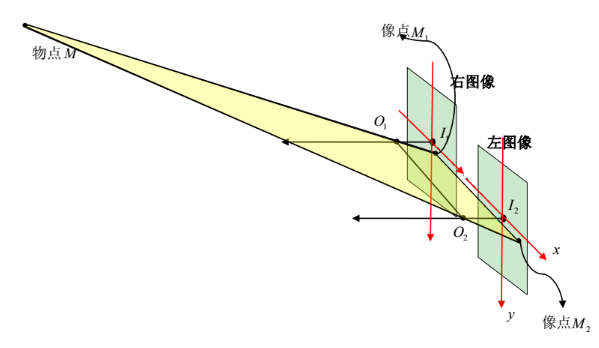


图2.2 三角测量法

假设目标在处于M位置，M到左右摄像头光心O1、O2连线的距离即为所需景深z，而光心后的左右成像面中心点I1、I2连线为基线距离，可由摄像头参数求得。根据三角形相似，有：

其中已知量为焦距f（即O1I1、O2I2），光心距离d（即O1O2、I1I2）；可测得量为M在左右成像面的点M1、M2与成像面中心I1、I2的水平偏移量a和b。代入上面的式子，化简可得：

根据上式，即可求得单点景深z。得到景深，再次根据三角形相似计算可得到目标点的水平位置x和竖直位置y。

## 特征点匹配

Leap Motion在形成深度图像时，仅采用若干个特征点进行手型匹配，在获得这些特征点的信息后，利用IK算法即可推算出整个手部模型。由于避免了匹配手部所以点，所以Leap Motion的处理时间能够减小到10ms以内。[深度：Leap Motion手势识别大揭秘]

## 坐标系统

Leap Motion传感器以其中心为原点建立一个右手笛卡尔坐标系，X轴与传感器长边平行，指向右方，Y轴垂直屏幕指向上方，Z轴背离屏幕指向操作者的方向。若将传感器与头戴设备连接，则Y轴指向前方，Z轴垂直向上。

Leap Motion传感器的探测视野在屏幕上方形成一个倒四棱锥空间结构，视角约为150°，有效范围约为0.3米到0.6米，检测精度可达到毫米级。

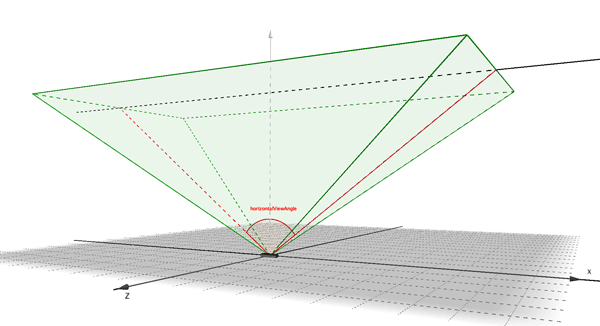


图2.3 Leap Motion探测视野

## 运动数据追踪

Leap Motion所追踪的具体对象为手、手指、手势以及工具，设备定期发送所采集到的目标对象的方向、位置等信息，每份这样的信息都被保存到帧（frame）当中，并为每个被其检测到的对象分配唯一ID，只要设备能够探测到对象，ID便保持不变。假如设备失去目标，而当目标再次出现时，将重新分配新的ID。

Leap Motion视野中的实体发生位移、旋转等变化，都会引起帧信息的变化，通过将当前帧与之前帧的数据进行对比，形成运动信息。

### 手部模型

手部模型提供被检测手的ID、位置等其他信息，包括连接这只手的手指和手臂等等。

即使在部分手探测不到的情况下，Leap Motion也能结合可见部分、内部手部模型以及先前的运动信息来预判该不可见部分接下来最有可能的跟踪数据。[基于LeapMotion与Unity3D的体感游戏“Survial&Shoot”的开发]

1. 手

手对象包含用于描述一只手物理特征的各种属性。

表2.1 手对象属性表

|  |  |
| --- | --- |
| Palm Position | 手掌中心的坐标 |
| Palm Velocity | 手掌运动的速度 |
| Palm Normal | 垂直于手掌平面的向量 |
| Direction | 从掌心指向手指的向量 |
| Sphere Center | 适应手掌弧面的一个球的球心 |
| Sphere Radius | 适应手掌弧面的一个球的半径 |

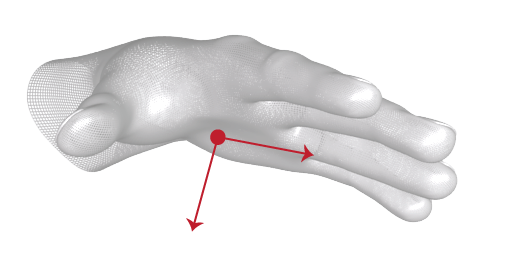


图2.4 手掌向量示意图

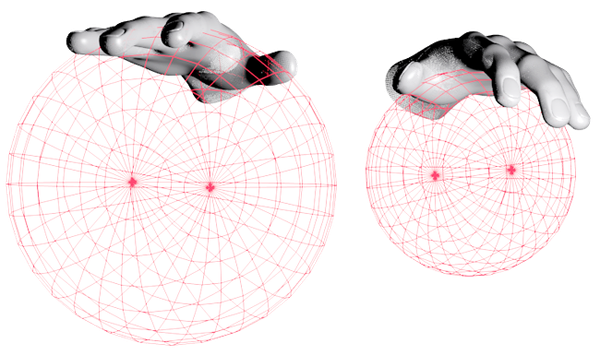


图2.5 手掌弧面球体示意图

1. 手指

Leap Motion可识别手上每根手指的信息。手指被控制器分为拇指、食指、中指、无名指和小拇指，如果某根手指无法被检测到，Leap Motion会根据最近观察数据与手掌的解剖模型推测出该手指的特征。

1. 工具

Leap Motion除了可以检测手指外，也可以检测手持的工具，工具被认为是比手指更细、更长而且更直的物件。



图2.6 工具示意图

手指与工具具有一些相同的属性，Leap Motion将它们统一称为端点对象（Pointable Object）。

表2.2 端点对象属性表

|  |  |
| --- | --- |
| Length | 可见长度 |
| Width | 平均宽度 |
| Direction | 与物体指向相同的单位向量 |
| Tip Position | Leap Motion坐标系下的位置 |
| Tip Velocity | 运动速率 |



图2.7 端点对象示意图

# Unity

## Unity开发引擎

Unity是由丹麦Unity Technologies公司所开发的一款完整的跨平台游戏开发引擎，可以制作出3D或2D的优秀游戏，其编辑器可运行在Windows和Mac OS X下，并支持包括PC，IOS，Android，Web，XBOX等多个平台发布。

Unity提供易于上手的操作界面，内置功能强大的UI系统，并支持物理引擎、多人网络连线、地形处理和角色动作编辑等各项功能，可大幅度降低游戏的开发门槛，缩短游戏的开发时间，并达到降低制作成本的目的，目前已有80%的手机游戏使用了Unity引擎作为开发工具。

Unity不仅只限于游戏行业，在3D设计、工程模拟和虚拟现实等方面也有着广泛的使用。国内常用Unity开发虚拟现实三维展示甚至是军事演练模拟等项目。



图3.1 Unity5 Logo

## Visual Studio集成开发环境

Microsoft Visual Studio（以下简称VS）是美国微软公司所推出的一套完整的开发工具集，其包含了软件生命周期所需要的大部分工具，如集成开发环境、代码管理工具和UML工具等，其发布出来的产品适用于绝大部分平台，包括Windows、Windows Phone、Android、IOS、Web。

Visual Studio是当前Windows平台下最流行、最强大的集成开发环境，并可通过安装扩展插件进行更多功能的定制。目前，VS通过自带扩展便可与Unity引擎无缝对接，并支持断点，成为取代MonoDevelop编辑器的强大利器。



图3.2 Visual Studio 2017 Logo

# Leap Motion应用设计

## 主题设计

为了较大程度地利用Leap Motion及其提供的插件包资源，将开发一个3D场景模拟的桌面应用，用于体验Leap Motion所带来的手势交互。基于现有资源和网络资源，经反复选材，本应用的主题最终定为室内家居模拟控制。

为了增加该主题的趣味性，本应用将在原有的“家居控制”模式（以下简称“常规模式”）上进行功能的扩展或替换，重点开发出一个更加有趣的“惊悚鬼屋”模式（以下简称“鬼屋模式”）。

## 基本功能设计

### 菜单界面

运行应用后应先进入菜单界面，在菜单界面上有多个基本功能选项供用户操作。

1. 操作指引：为用户提供直观的图文介绍，作为使用本应用的操作教程。
2. 模式选择：可供用户选择进入常规模式或者鬼屋模式。
3. 退出程序：会进一步询问用户是否退出应用，防止误操作。



图4.1 菜单界面功能设计

### 主题场景

为契合“家居”“鬼屋”等关键字，本应用将搭建若干个室内场景，场景内布置若干可供交互的家居和电器等物品。

1. 客厅：从菜单界面进入后的第一个场景，布置“电视”，作为该场景的主要交互对象。
2. 浴室：从走廊进行跳转进入，布置“浴缸”，作为该场景的主要交互对象。
3. 厨房：从走廊进行跳转进入，布置“顶灯”、“炉灶”和“冰箱”，作为该场景的主要交互对象。
4. 走廊：从客厅进行跳转进入，作为连接“客厅”、“浴室”和“厨房”的中间通道。



图4.2 应用场景衔接图

### 双主题模式

用户选择常规模式或鬼屋模式，可通过同样的控制操作，获得不同的视听体验，具体体现在以下几个方面。

1. 灯光：常规模式下，场景均选用暖色调的环境灯光，而鬼屋模式下，当触发了事件时，改用以绿色为主色调的灯光来烘托出一种可怕的气氛。
2. 素材：鬼屋模式下选用一些切合氛围的音频、视频和模型素材，如“尖叫声”、“鬼火”或“骷髅头”等。
3. 指引：常规模式下，各主要场景内在首次进行某种操作时会有相应功能的操作指引提示。

### 镜头转换方式

由于Leap Motion的交互系统有很大一部分是建立在接触式交互之上的，同时又因为Unity以米为单位，而Leap Motion的移动是以厘米为单位，故场景中一个微小的镜头移动对Leap Motion而言都是相当大的变化。

为了方便控制，在一个静态场景，最好采用固定视角的方式来设置摄像机，即镜头只在预设的若干个点进行移动和旋转。这样，在搭建场景的过程中，便能忽略场景的空间尺寸，各个模型之间的相对距离可以离得较远，只需在最后选取合适的点作为摄像机的路径点即可。

## 构建3D场景

### Leap Motion对象配置

新建Unity场景，作为应用初始化的入口场景。将下载好的Leap Motion核心包和其他资源包一并导入项目中，并在Hierarchy面板添加摄像机。

在主摄像机下新建一个物体作为Leap Motion对象作为其子物体，这样Leap Motion便会跟随摄像机一起移动和旋转。在该物体上还应挂载三个组件：LeapHandController、LeapServiceProvider和HandPool。具体参数设置如图。

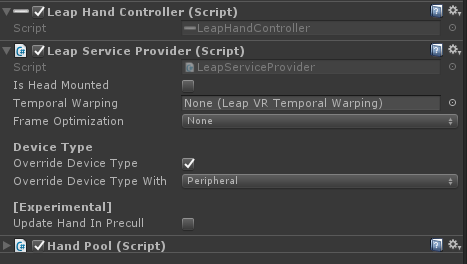


图4.3 Leap Motion对象参数配置

在Leap Motion物体下再新建一个空物体，作为手部模型的父对象，并从资源包中选择合适的手部模型.prefab预制体，拉取到父对象之下。同时，在Leap Motion物体的HandPool脚本上进行引用。

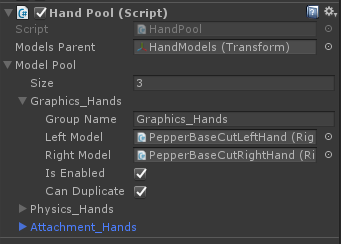
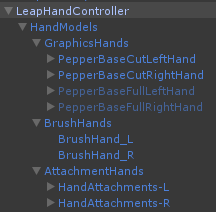


图4.4 Hand Pool对象参数配置

从资源包中分别再找出名为LeapEventSystem和InteractionManager的预制体，挂载在主摄像机下面，即完成了Leap Motion对象的搭建。

### 客厅场景搭建

新建一个空物体，命名为LivingRoom，此后添加添加的物体都作为该空物体的子物体。

在场景中添加若干各Quad（四边形）作为地板和墙壁的面片，并为其附加材质球。按照一定布局添加沙发、茶几、空调、电视以及灯光等物体，最终效果如图。

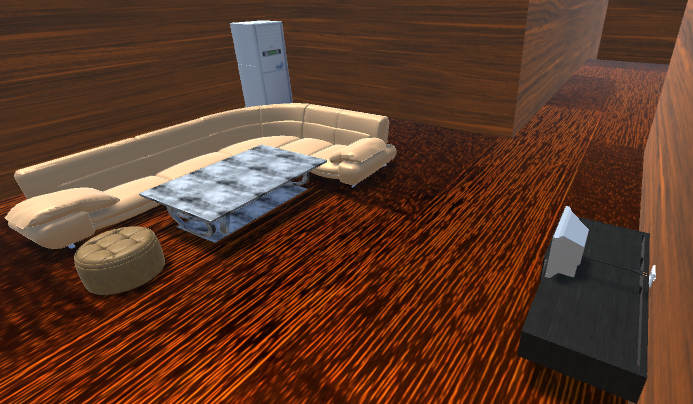


图4.5 客厅场景预览图

### 走廊场景搭建

新建一个空物体，命名为Hallway，此后添加添加的物体都作为该空物体的子物体。

在场景中添加地板和墙壁，参考“4.3.2客厅场景搭建”。走廊的布局较为简单，最终效果如图。

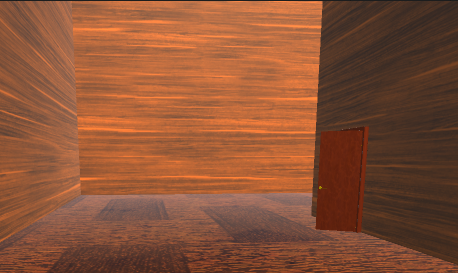


图4.6 走廊场景预览图

### 浴室场景搭建

新建一个空物体，命名为Bathroom，此后添加添加的物体都作为该空物体的子物体。

在场景中添加地板和墙壁，参考“4.3.2客厅场景搭建”。在墙角处放置一个浴缸模型，添加水体并隐藏，最终效果如图。

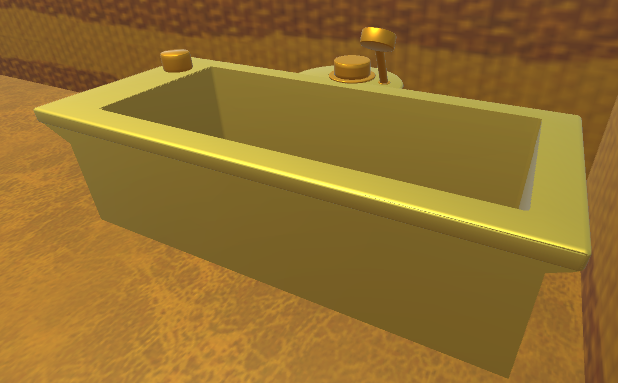


图4.7 浴室场景预览图

### 厨房场景搭建

新建一个空物体，命名为Kitchen，此后添加添加的物体都作为该空物体的子物体。

在场景中添加地板和墙壁，参考“4.3.2客厅场景搭建”。按照一定布局添加餐桌、顶灯、炉灶和冰箱等物体，最终效果如图。



图4.8 厨房场景预览图

### 光照烘培

将以上所有场景添加为预制体，然后选择“Window/Lighting”打卡光照设置窗口，选中“Lightmaps”选项卡，取消Auto选项的勾选，点击“Build”按钮手动生成光照贴图。

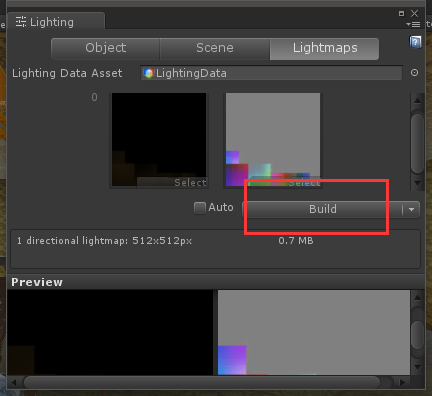


图4.9 灯光烘焙设置

## 界面设计

程序的界面使用UGUI来实现。UGUI是Unity官方推出的的新UI系统，支持可视化开发，同时也支持3DUI的开发，是大名鼎鼎的NGUI（商业游戏中用得最多的UI插件）的作者的又一杰作，并且也是开源的系统。相比于NGUI，使用UGUI进行界面开发，将会更方便、更快捷。

### UI效果要求

各个界面采用的UI元素统一风格，使用同一套贴图素材，并且UI所选用的素材具有一定的立体效果，如按钮素材的边界上有一层下凹效果。

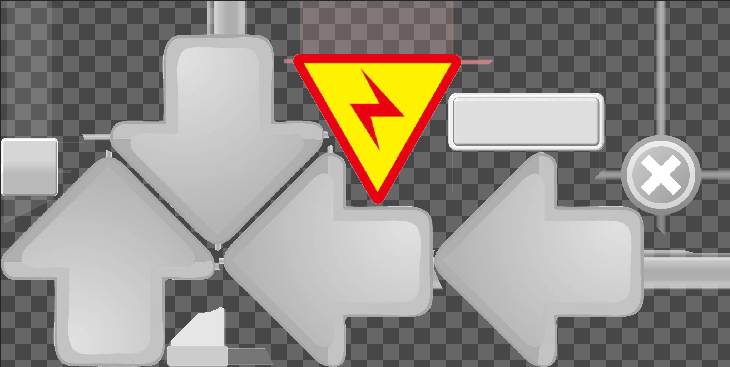


图4.10 UI图集预览

为进一步体现立体感，按钮在按下和弹起时应有明显的缩放效果，利用Leap Motion提供的UIInput资源包中的CompressibleUI脚本可实现该效果。

新建一个Button并配上合适的贴图作为按钮，调整Color属性的Alpha值为30（其他按钮均使用这个配置），在该Button下添加一个Image，使用和Button相同的贴图，但其Alpha值为255，在该Image下再添加一个Text。最顶层的Button需要挂载CompressibleUI脚本，然后将Image和Text分别拖到Floating UI Layers内，设置Image的Shadow为Button。为触发缩放效果，在Button上添加EventTrigger组件，分别配置按下和弹起的事件为CompressibleUI的Retract和Expand方法。

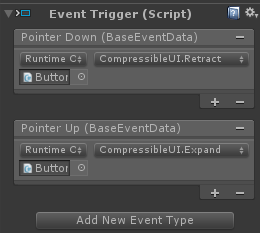
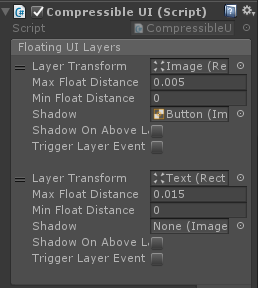


图4.11 可按压UI参数配置

### 画布属性

UGUI的UI元素都是放在一个名为Canvas（画布）的物体之下。由于Leap Motion的UI系统是接触式UI，为了能触摸到UI，需要调整各UI在摄像机视野内的景深位置。通过直接操作画布属性，将同时影响在该画布下面的所有UI元素。

在画布的Canvas组件上选择摄像机模式，该模式下的UI将一直呈现在摄像机正前方的Plane Distance距离位置，距离够近时能够有3D触碰效果，且不会因摄像机位置或角度而偏离中央视野。当Plane Distance属性的值不大于摄像机Clipping Planes（裁剪面）的近距离时，UI界面将会看不到。为了使用Leap Motion的操作空间，Plane Distance的值设为0.45~0.55范围内。

为了多个不同分辨率的显示器都能正确显示UI，画布需要根据开发长宽比和实际长宽比做出拉伸或收缩。在画布下的CanvasScaler组件选择“Scale With Screen Size”模式，匹配模式选择“Match Width Or Height”，其值拉到Width端（0值），即按照屏幕分辨率宽度进行缩放。参考分辨率填如开发时屏幕所用的分辨率，此处是1400\*900。

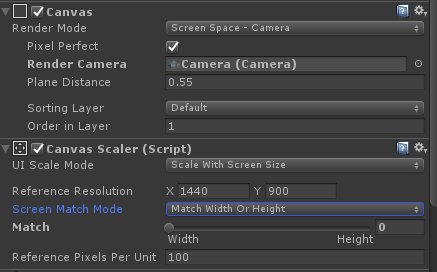


图4.12 画布参数配置

### 菜单界面

菜单界面应符合本应用的主题，采用房屋图片作为菜单的背景图片。为吸引用户进行体验，着重强调鬼屋模式，将界面的主色调定为绿色调。为进一步烘托一种阴森恐怖的气氛，在屏幕范围内增加不断浮动的粒子特效。

菜单界面上所使用的UI元素严格遵循规定，每个功能面板都需要有返回按钮。面板为半透明，防止遮挡了背景图片的中央主要部分。

菜单的每个面板上的文字在颜色上应有所区分，且本应用只能使用同一种字体。



图4.13 菜单一级界面设计

在“操作指引”功能面板上，采用图文结合的生动方式，以及拖拽滚动的形式呈现。



图4.14 操作指引二级界面设计

### 场景界面

应用内各场景使用UI按钮进行镜头切换，按钮需根据分场景实际位置进行布局。每个按钮都要适应人手以及Leap Motion的可操作范围，既不能太拥挤也不能太难触碰到。在切换主要场景的按钮上还应有文本显示。

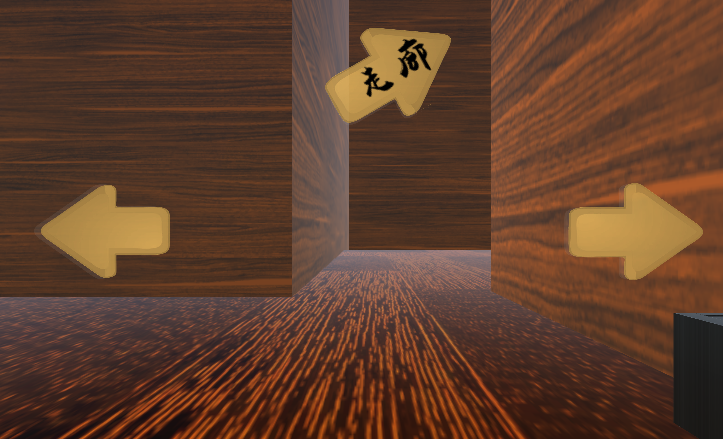


图4.15 主场景UI设计

鬼屋模式下，当触发场景事件时，UI的主体颜色切换为和场景环境相符合的绿色调，UI文字改为红色。当场景恢复正常时，UI的颜色也一同恢复。

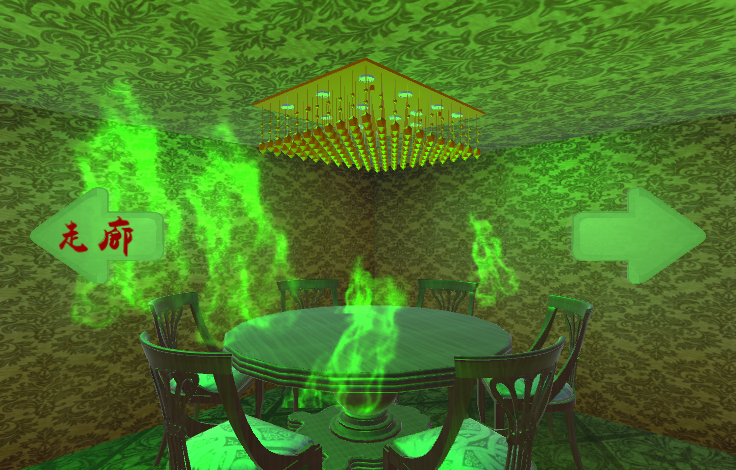


图4.16 主场景鬼屋模式UI设计

# 详细设计与编码

## 集成Leap Motion

Leap Motion提供了适用于Unity应用开发的资源包，其中包括了核心代码脚本和通用模型，通过导入并使用插件资源，可以快速构建出Leap Motion应用。

### Leap Motion插件资源

Leap Motion插件资源包括了以下五个部分：

Core Assets：核心功能包，是使用Unity开发Leap Motion应用的必要插件。

Interaction Engine：交互引擎模块，是一套支持与场景中物体进行交互的更简便的方案，例如进行抓取、抛掷等动作，要求手部与物体直接接触。

Attachments Module：用于控制场景中物体或UI各项属性变换的功能模块，如旋转、缩放、颜色渐变或等，不要求手部与物体直接接触，符合控制条件即可进行变换。

Hands Module：手型资源包，内含Leap Motion提供的一些手部模型，包括可见的骨骼驱动的人手、程序生成的多边形手、组件构成的机械手以及不可见的物理手等。

Detection Examples：使用Detection Utilities（核心包中监测具体动作行为的脚本集）开发的Demo。

本应用开发过程中将会使用到核心包、交互引擎及手型资源包，其余资源可作为参考下载。

下载地址：https://developer.leapmotion.com/unity

### 主要API

LeapServiceProvider：LeapProvider唯一的具体类，其中保留着由传感器传输到应用的帧数据。

表5.1 LeapServiceProvider类主要接口

|  |  |
| --- | --- |
| 方法或属性 | 解释 |
| Controller GetLeapController () | 返回Leap Controller实例 |
| bool IsConnected () | 是否已正确连接Leap Motion设备 |
| bool \_isHeadMounted | 是否头戴式 |
| Frame CurrentFrame | 当前帧数据 |

HandPool：手部模型的对象池，包含一个ModelGroup列表，可配置多种不同的手部模型，它们将会被同时激活或禁用。

ModelGroup：一组手部模型，包括左手和右手。

表5.2 ModelGroup类主要接口

|  |  |
| --- | --- |
| 方法或属性 | 解释 |
| string GroupName | 组别名称 |
| bool CanDuplicate | 是否可以复制新的实例并使用 |
| bool IsEnabled | 是否启用这组手模型 |
| IHandModel LeftModel | 左手模型 |
| IHandModel RightModel | 右手模型 |

InteractionManager：交互引擎管理器，用于设置接触式交互的各项参数。

表5.3 InteractionManager类主要接口

|  |  |
| --- | --- |
| 方法或属性 | 解释 |
| LeapProvider \_leapProvider | 所使用的LeapProvider实例 |
| HandPool \_handPool | 所使用的HandPool实例 |
| string \_ldatPath | 位于StreamingAssets文件夹下的Ldat文件路径 |
| string \_brushGroupName | HandPool下BrushHand手模型所在组别名 |
| bool \_contactEnabled | 是否启用接触，禁用将无法对添加了InteractionBehaviour组件的物体产生作用 |
| bool \_autoGenerateLayers | 是否自动创建交互层 |

InteractionBehaviour：可交互物体，由交互引擎驱动。

表5.4 InteractionBehaviour类主要接口

|  |  |
| --- | --- |
| InteractionManager \_manager | 所使用的InteractionManager实例 |
| InteractionMaterial material | 所使用的InteractionMaterial 文件 |
| bool IsBeingGrasped | 是否正被抓取中。 |
| bool isKinematic | 是否启用运动学 |
| Rigidbody rigidbody | 所使用的刚体组件 |
| bool useGravity | 是否启用重力 |
| Action<Hand > OnHandGraspedEvent | 抓取物体事件 |
| Action<Hand > OnHandReleasedEvent | 释放物体事件 |
| InteractionManager \_manager | 所使用的InteractionManager实例 |

Detector：手势监测器基类，在其派生类内实现监测某一具体的简单手势的方法。

表5.5 Detector类主要接口

|  |  |
| --- | --- |
| 方法或属性 | 解释 |
| void Activate () | 手势匹配成功后调用 |
| void Deactivate () | 手势匹配失败后调用 |
| UnityEvent OnActivate | 手势匹配成功后一次性触发的事件 |
| UnityEvent OnDeactivate | 手势匹配失败后一次性触发的事件 |
| bool IsActive | 手势是否匹配成功，成功则处于激活状态 |

DetectorLogicGate：手势监测逻辑门，通过组合多个简单手势，实现对复杂手势的监测，只有组合逻辑的结果最终为真时才会激活。自身继承自Detector类，可以进一步组合，形成激活条件高度复杂的手势。

表5.6 DetectorLogicGate类主要接口

|  |  |
| --- | --- |
| 方法或属性 | 解释 |
| void AddAllSiblingDetectors () | 添加同一游戏对象上的其他监测器 |
| void AddDetector (Detector detector) | 添加指定监测器 |
| void RemoveDetector (Detector detector) | 移除指定监测器 |
| bool AddAllSiblingDetectorsOnAwake | 是否默认添加同一游戏对象上的其他监测器 |
| LogicType GateType | 逻辑门类型，包括与门和或门 |
| bool Negate | 逻辑门结果输出类型，是否取反 |

## 代码整体框架搭建

### MVC框架简介

MVC是一种软件的架构模式。它把软件系统划分成模型（Model）、视图（View）和控制器（Controller）三个部分，从而将应用程序的输入、处理和输出分离开来。

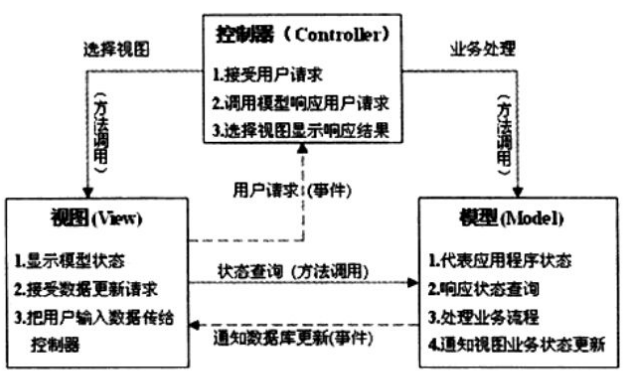


图5.1 MVC流程示意图

Model：存储应用程序的核心数据。

View：用户界面，显示Model数据。

Controller：处理用户输入。

Model、View和Controller，再加上用户，则成为了程序流程的一个闭环。用户通过View进行输入操作，向Controller发出请求事件，Controller根据用户请求调用相应方法更改Model数据和状态，Model发生变更后通过事件去通知相关的View更新界面。这样，便完成了从用户输入开始到数据处理与存储再到返回请求结果的整个过程。

在本程序开发中使用MVC模式，旨在提高代码的复用性和可维护性，同时，会进行一定程度的优化以减少使用该模式所带来的负面作用。

### 观察者模式

模型存储着系统数据，当模型中的数据发生变更时，需要通知视图进行更新。因此，模型宜采用观察者模式实现通知。

观察者模式，又称为订阅-发布模式或模型-视图模式，是常用的软件设计模式之一。当被观察对象的状态发生变更时，会去通知它的所有观察者进行更新。

观察者模式常用“注册——通知——撤销注册”的形式实现。

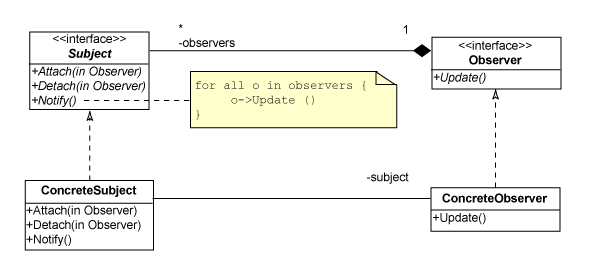
IMG_256

图5.2 观察者模式UML类图

被观察对象保留一份观察者列表，一个被观察对象可以有多个观察者，观察者将自身注册到被观察对象当中。当被观察对象发生了某种变化后，遍历列表内所有已注册的观察者，通知它们更新。

### 框架实现思路

实现MVC框架时结合观察者模式，用户界面作为观察者，而业务数据则是被观察对象。利用C#委托事件的特性，能够简化观察者模式的设计。通过实现接口方法，完成对事件的注册、调用和注销。由于C#类实例方法只归属于类的具体实例，指定了实例方法便指定了某个实例，一个委托事件中可注册多个具体实例的多个实例方法，利用这个特性，被观察对象可只保留一个事件清单，不必保留观察者列表，便能根据完成对多个观察者（即实例方法的归属对象）进行特定事件的通知。

在视图层，为了方便，设定成一个视图必须而且只能直接监视一个模型的数据和变化，若需要监视额外的模型，通过一个全局事件管理器到模型中注册事件以实现间接监视。视图需要记录自身注册到模型的事件，用于后续的注销操作。结合Unity组件式开发方式，视图内的参数需要出现在对象面板上，将其继承至MonoBehaviour类，并由Unity控制视图的生命周期。



图5.3 观察者模式Unity实现类图

## 全局管理器

系统通过一些全局性的管理器去统一控制MVC架构未考虑或未涵盖的操作，如系统设置、资源加载等。使用全局管理器，使得Model、View和Controller均能专注于自身的功能，而一些泛用性强的方法则由管理器完成，解耦的同时提高代码复用性，减少冗余。全局管理器是静态单例类。

### 单例模式

单例模式是一种最简单也是使用最广泛的设计模式。它要求所涉及的类负责创建自身的对象，并且有且仅有单个实例被创建。 单例类对外则提供了一种访问其实例对象的静态方法。

单例类的实现方式有好几种，根据线程安全性和初始化方式划分有：饿汉模式、懒汉模式、双重校验锁、静态内部类、枚举，等等。结合Unity单一线程的特点，绝大多数的开发中，单例的线程安全性可不必纳入考虑，因此可以采用“懒汉模式”的不加锁版本，该实现方式无需同步，免去了传统加锁版本的额外性能开销。

系统开发过程中需要使用到多个单例类，为统一结构，提高复用性，将单例类设计成泛型版本。

public abstract class Singleton<T> where T : class, new()

{

private static T instance;

public static T Instance

{

get

{

if (instance == null) { instance = new T(); }

return instance;

}

}

protected Singleton()

{

Init();

}

protected virtual void Init()

{

}

}

部分管理器需要结合Unity的方法进行使用，可以再添加一个继承自MonoBehaviour的泛型版本。

public abstract class MonoSingleton<T> : MonoBehaviour where T : MonoBehaviour

{

private static T instance;

public static T Instance

{

get

{

if (instance == null)

{

GameObject go = new GameObject(typeof(T).Name.ToString());

instance = go.AddComponent<T>();

DontDestroyOnLoad(go);

}

return instance;

}

}

protected virtual void Awake()

{

Init();

}

protected virtual void Init()

{

}

}

### 单例管理器

通常情况下，Unity开发过程中必要的管理器有：配置管理器、资源管理器、声音管理器、UI管理器、场景管理器和事件管理器。结合本系统的需要，可另外增添一些管理器，如摄像机管理器、Leap Motion管理器、协程管理器，等等。这些管理器提供统一的接口，编写代码时应尽可能使用这些接口。



图5.4 单例与事件管理器继承关系图

表5.7 各单例管理器说明表

|  |  |  |
| --- | --- | --- |
| 父类 | 子类 | 解释 |
| MonoSingleton<T> | CameraManager | 摄像机管理器，控制摄像机位置转换。 |
| CoroutineManager | 协程管理器，提供调用协程的统一接口。 |
| FrameManager | 帧管理器，提供逐帧事件的注册和定期执行。 |
| GlobalManager | 全局配置管理器，配置系统参数和方法。 |
| HandManager | 手管理器，提供当前手模型的访问和更新推送。 |
| LeapMotionManager | Leap Motion管理器，提供Leap Motion各部分组件的访问方式。 |
| SoundManager | 声音管理器，提供多种控制音频播放的方式。 |
| UIManager | UI管理器，管理用户界面的加载和关闭。 |
| Singleton<T> | EventManager | 事件管理器，提供通用事件的注册、发布和注销方法。 |
| ResourceManager | 资源管理器，缓存资源，提供资源获取接口和资源加载的多种方法。 |
| SceneManager | 场景管理器，管理场景切换和方法回调。 |

## 应用场景模块实现

### 资源管理

应用中出现的所有资源包括UI、场景文件、场景物体和音频文件。这些资源文件由资源管理器ResourceManager统一加载和管理。

Unity加载资源有多种途径，常见的有使用Resource.Load等方法直接由本地文件夹加载现成资源，该文件夹指的是相对路径为Assets/Resources的文件夹，需要手动创建。还有一种方式是通过加载AssetBundle资源包体后进行解压缩来获取资源，AssetBundle是以某种方式将一系列的资源文件或者场景文件紧密保存的一种文件，采取此方法可以减小最终发布出来的应用体积，可进行资源的更新替换，并做到按需加载以减少内存占用。

本应用将同时采用以上两类加载方式，在开发过程中采用本地加载，而在发布版本则采用AssetBundle包体加载。除此之外，资源加载还有同步和异步方式，在这里，编辑器模式下使用同步加载，发布版本中采用异步加载。在ResourceManager脚本下添加判断：

#if UNITY\_EDITOR

public bool IsDefaultAsync = false;

public bool IsDefaultFromServer = false;

#else

public bool IsDefaultAsync = true;

public bool IsDefaultFromServer = true;

#endif

资源管理器对外提供GetResource和LoadAsset两个接口，负责获取资源和加载资源，LoadAsset内部通过GetResource判断资源是否已存在内存，不存在时才根据加载方式去加载资源文件，加载所用的路径由PathHelper路径工具单例拼接。



图5.5 资源加载流程图

### UI界面

控制每个场景UI界面（包括菜单界面）的视图类均继承自UIView类，UIView从基类View派生而来。UIView将会自动处理每个UI元素的初始化、子界面切换、按键特效和音效等任务。

当没有Leap Motion设备连接时，需让鼠标也能够操作UI。为了支持鼠标，需提供鼠标辅助控件，并负责处理它们的显隐。在脚本中添加操作模式变更事件的代码：

private void OnConnectedStateChanged(params object[] arg1)

{

for (int i = 0; i < testBtn.Length; i++)

{

testBtn[i].gameObject.SetActive(!(bool)arg1[0]);

}

}

并在事件管理器进行注册：

EventManager.Instance.AddEventHandler(Define.EventType.OpModeChanged, OnConnectedStateChanged);

每个派生自UIView类的具体UI界面还需提供该界面上按钮所需要的操作方法，并在相应的UI预制体按钮上逐一绑定。



图5.6 UI视图继承关系图

### 镜头控制

1. 摄像机镜头转换

在主场景中通过点击界面方位按钮进行镜头切换，按钮触发事件后将获取场景视图的目标位置并调用摄像机管理类CameraManager提供的方法。该流程严格按照MVC架构进行执行，UIView必须请求Controller去通知相应的SceneEntityView，场景视图则根据自身的镜头预设点去发起镜头切换，当镜头切换完毕后，还会去执行UI视图传入的委托，最终完成场景与界面的同步切换。



图5.7 镜头切换控制流程图

当启用镜头切换时，摄像机管理器在每帧对摄像机位置和角度进行平滑处理，在Update方法内添加代码：

if (isStart)

{

Camera.transform.position = Vector3.SmoothDamp(Camera.transform.position, toPos, ref velocity, smoothTime);

Camera.transform.rotation = Quaternion.Slerp(Camera.transform.rotation, toRot, 1 / smoothTime \* Time.deltaTime);

if (IsArriveTargetPos(Camera.transform.position,toPos) && IsArriveTargetRot(Camera.transform.rotation, toRot))

{

isStart = false;

if (func != null)

func();

}

}

脚本需判断是否完成了镜头切换，使用以下两个方法：

bool IsArriveTargetPos(Vector3 current, Vector3 target)；判断是否到达坐标位置。

bool IsArriveTargetRot(Quaternion current, Quaternion target)；判断是否旋转到目标角度。

其中IsArriveTargetRot方法将参数的四元数转换为欧拉角，再当成三维向量去调用IsArriveTargetPos方法进行判断。

1. 场景切换画面渐变过渡

在两个场景间进行切换时，除了摄像机需要对镜头进行转换，还需添加过渡画面，以屏蔽场景加载的卡顿和镜头切换动作。本应用的过渡画面采用黑屏淡入淡出的渐变方式。

在主摄像机下添加一个画布Canvas，并挂载Image组件，选择一张方形的贴图作为背景，颜色初始化为全黑全透明值（0，0，0，0），渐变过程通过改变Alpha值对黑背景进行渐变显隐。为了使黑背景遮住所有UI，将Canvas的渲染层级Order In Layer提高到1；同时，为遮住场景物体，调整黑背景的Z轴位置，使其称为镜头前最近的物体。

在项目中引入DOTween插件，DOTween是当前一种执行效率和开发效率最高的Unity补间动画插件，常用于制作各种UI效果。DOTween通过采用C#方法链的写法使得复杂动画可以一目了然。处于同一方法链下的方法优先级相同，按照动画的生命周期执行。方法链可通过委托层层嵌套，让每个动作节点可以处理更复杂的任务，具有很强的拓展性。

在CameraManager脚本中引用DG.Tweening命名空间，然后编写场景切换的背景动画接口：

public void ChangeScene(float darkDuration, float pauseDuration, float showDuration, Action onComplete)

{

blackBG.DOKill();

ViewToDark(darkDuration)

.OnPlay(() => IsChanging = true)

.OnComplete(() =>

{

ViewToDark(pauseDuration)

.OnPlay(() =>

{

if (onComplete != null) onComplete();

})

.OnComplete(() =>

{

ShowView(showDuration)

.OnComplete(() => IsChanging = false);

});

});

}

此处代码完成了画面渐变的整个流程：黑屏淡入→保持黑屏→执行场景切换→黑屏淡出。其中，ViewToDark和ShowView方法内使用DOTween提供的DOFade方法负责处理黑背景的透明度渐变，其参数为目标Alpha值和动作的执行时长。

### 主场景

1、场景实体视图基类

每个场景都由设置成单独的预制体，并挂载相应的视图脚本，每个脚本都是继承自SceneEntityView类，该类也是View的派生。SceneEntityView类主要负责处理场景镜头以及场景声效的切换。

为了方便在Unity中配置场景声效，在SceneEntityView类中添加一个声效辅助类SceneSoundGroup类的实例列表。SceneSoundGroup类能够对多个音频剪辑进行分组，并设置它们的播放属性。在Unity中，当自定义类包含集合元素，且该自定义类型又是另一个类中集合的类型时，需要对它设置[Serializable]标签，才能使其在Inspector面板上多层嵌套地显示出来。



图5.8 场景声效嵌套配置

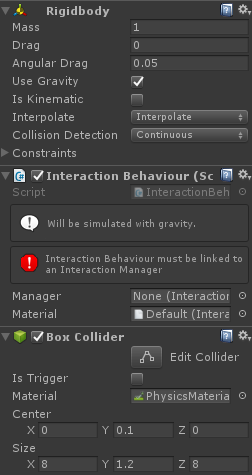
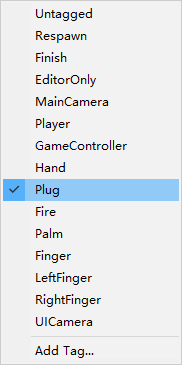


图5.9 场景实体视图与场景声效继承关系图

1. 客厅电视

客厅的主要交互对象为电视，通过模拟接通电源操作来打开电视并播放节目。

在已布置好的电视电源线添加一个Box Collider碰撞体和刚体组件，并挂载InteractionBehaviour脚本，让它成为Leap Motion交互引擎可识别并交互的对象。在插头处也添加一个Box Collider组件，插头需勾选Is Trigger选项以便成为触发器，更改插头的Tag为“Plug”，用于触发标识。

(a) 组件参数配置 (b) 标签设置

图5.10 交互引擎对象配置图

在插座上挂载触发器，并新建SocketView脚本，编写插座触发器逻辑：

void OnTriggerEnter(Collider other)

{

if (other.tag == "Plug")

LivingRoomCtrl.Instance.InsertPlug();

}

void OnTriggerExit(Collider other)

{

if (other.tag == "Plug")

LivingRoomCtrl.Instance.PutPlugOut();

}

当电源线插头部分接触插座时，便会向LivingRoomCtrl发出插入电源请求，并经过LivingRoomModel通知LivingRoomView去操作电线和电视。

private void InsertPlug(params object[] arg1)

{

if (plugView.IsDrag == false)

plugInteraction.OnHandReleasedEvent += insertFunc;

else

insertFunc.Invoke(null);

}

InsertFunc是负责将镜头切换到电视机前并播放视频的委托方法。若采用Leap Motion交互引擎去抓取电源线插入插座时，需为电源线上InteractionBehaviour的回调方法OnHandReleasedEvent添加该事件，当检测到手释放了电源线后便会去执行。

PlugView是挂载在电源线物体上的脚本，当采取鼠标操作时，通过左键点击电源线将触发Unity内置回调方法OnMouseDown，拖动电线平移。

IEnumerator OnMouseDown()

{

Vector3 ScreenSpace = Camera.main.WorldToScreenPoint(transform.position);

Vector3 offset = transform.position - Camera.main.ScreenToWorldPoint(new Vector3(Input.mousePosition.x, Input.mousePosition.y, ScreenSpace.z));

isDrag = true;

while (Input.GetMouseButton(0) && isDrag)

{

Vector3 curScreenSpace = new Vector3(Input.mousePosition.x, Input.mousePosition.y, ScreenSpace.z);

Vector3 CurPosition = Camera.main.ScreenToWorldPoint(curScreenSpace) + offset;

transform.position = CurPosition;

yield return null;

}

}

Camera.main.WorldToScreenPoint：将电源线所在位置转换为屏幕空间，由于鼠标的移动范围是一个二维的平面空间，而物体需在三维立体空间内进行移动，这里需要得到电源线在屏幕空间的z轴位置，才能使平移方向变为二维。

Camera.main.ScreenToWorldPoint：通过鼠标在屏幕空间的位置和以求得的z轴位置，转换成鼠标在世界空间的位置。

代码通过转换鼠标位置到世界空间，加上电源线与鼠标的初始偏移量offset，得到电源线基于鼠标位置的实时三维坐标。

1. 浴室浴缸

浴室的主要交互对象是浴缸，通过进水和排水开关来控制浴缸水位。

在浴缸上的两个开关上分别挂载触发器，然后添加触发器脚本WaterInToggleView和WaterOutToggleView，触发后改变浴缸放水状态。

浴缸在注水过程中自动控制水位，以防水体高过浴缸。为浴室场景视图脚本BathroomView中的Update方法添加以下代码：

if(isInDown && wt.localPosition.y<waterMaxHeight)

{

wt.Translate(Vector3.up\* waterInSpeed);

}

if(isOutDown && wt.localPosition.y > waterMinHeight)

{

wt.Translate(Vector3.down\* waterOutSpeed);

}

当水位达到预设最高值时，注水与排水同时进行，且排水速度略大于注水速度，保证水位平稳。

当浴缸开关被触碰后，开关会自动下压或者抬起。为避免开关下压或抬起过程中再次接触手部而二次触发了触发器，需要设置一个缓冲期，禁用触发器。在脚本中添加一个标志位isAnyStateChanged，代表是否处于开关动作阶段，并用计时器复位。

if(isAnyStateChanged)

{

if(cdTimer<interval)

{

cdTimer += Time.deltaTime;

}

else

{

cdTimer = 0;

isAnyStateChanged = false;

}

}

1. 厨房

厨房的交互对象包括了炉灶、顶灯和冰箱，分别代表了接触式非手势操作、非接触式手势操作、和接触式手势操作三种不同情况。

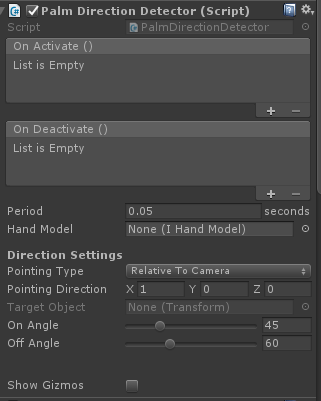
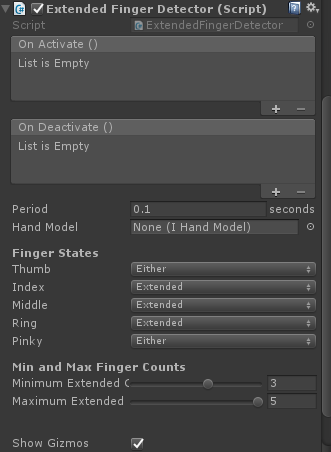
1. 炉灶

厨房的炉灶采用了与浴室浴缸开关相同的操作设计，即通过手部直接触发开关上的触发器，点燃或熄灭炉火。

1. 顶灯

顶灯的控制使用了Leap Motion自带的手势识别脚本Detector及其派生类。并通过Detector Logic Gate组件组合成一套复合手势，只有复合手势的最终输出为真，才可打开或关闭顶灯。

在厨房对象下新建一个空物体，添加2个手掌方向监测脚本PalmDirectionDetector和2个手指伸展监测脚本ExtendedFingerDetector，分别监测左右双手的手掌方向和手指的曲直状态。以左手为例，在PalmDirectionDetector组件中选择指向类型“Pointing Type”为“Relative To Camera”，设定Pointing Direction的x值为1，y和z为0，代表着朝向摄像机正右方为激活，右手配置反之。接着在每个ExtendedFingerDetector组件中都分别设置每个手指的激活条件Finger States，此处仅要求食指、中指即无名指必须为Extended，其余两指不作要求，为Either。

(a) 手掌方向设置 (b) 手指状态设置

图5.11 顶灯监测器配置

为了满足通过拍掌控制灯光，需要再添加一个监测手掌接触的监测器。由于Leap Motion自带的几类监测器都不满足此类需求，需要编写自己的监测器脚本。编写继承自Detector的ObjectProximityDetector类，实现存在多组目标对象情况下，组与组的物体的接触监测，Detector采取协程定时监测，其协程部分代码如下：

IEnumerator proximityWatcher()

{

bool proximityState = false;

float onSquared, offSquared;

while (true)

{

onSquared = OnDistance \* OnDistance;

offSquared = OffDistance \* OffDistance;

if (\_currentLhsObj != null && \_currentRhsObj != null)

{

if (distanceSquared(\_currentLhsObj, \_currentRhsObj) > offSquared)

{

\_currentLhsObj = null;

\_currentRhsObj = null;

proximityState = false;

}

}

else

{

if (TargetObjects.Length <= 1)

{

for (int i = 0; i < TargetObjects.Length; i++)

{

if (TargetObjects[i].Length <= 1)

{

\_currentLhsObj = \_currentRhsObj = null;

proximityState = false;

break;

}

for (int j = 0; j < TargetObjects[i].Length - 1 && !proximityState; j++)

{

GameObject lhs = TargetObjects[i][j];

for (int k = j + 1; j < TargetObjects[i].Length; k++)

{

GameObject rhs = TargetObjects[i][k];

if (distanceSquared(lhs, rhs) < onSquared)

{

\_currentLhsObj = lhs;

\_currentRhsObj = rhs;

proximityState = true;

OnProximity.Invoke(\_currentLhsObj, \_currentRhsObj);

break;

}

}

}

}

}

else

for (int i = 0; i < TargetObjects.Length - 1 && !proximityState; i++)

for (int j = i + 1; j < TargetObjects.Length && !proximityState; j++)

for (int k = 0; k < TargetObjects[i].Length && !proximityState; k++)

{

GameObject lhs = TargetObjects[i][k];

for (int l = 0; l < TargetObjects[j].Length; l++)

{

GameObject rhs = TargetObjects[j][l];

if (distanceSquared(lhs, rhs) < onSquared)

{

\_currentLhsObj = lhs;

\_currentRhsObj = rhs;

proximityState = true;

OnProximity.Invoke(\_currentLhsObj, \_currentRhsObj);

break;

}

}

}

}

if (proximityState)

Activate();

else

Deactivate();

yield return new WaitForSeconds(Period);

}

}

其中，TargetObjects为二维数组，代表着若干个目标对象组，每一组内又包含多个子对象。代码段中的对象组接触监测部分采用了多个循环嵌套，在各对象组之间两两匹配，直到有两组对象成功匹配时停止。若有n个目标对象组，每组m个对象，根据握手定理，最坏匹配次数为m2n(n-1)/2次。当对象组只有一组时，改为组内匹配，最坏匹配次数为m(m-1)/2次。

各对象之间的匹配算法由distanceSquared方法提供，该方法求得了两个带有碰撞体的物体分别与其目标对象的最近点间的距离：

private float distanceSquared(GameObject lhs, GameObject rhs)

{

Collider lhsCollider = lhs.GetComponent<Collider>();

Collider rhsCollider = rhs.GetComponent<Collider>();

Vector3 lhsClosestPoint, rhsClosestPoint;

if (lhsCollider != null)

lhsClosestPoint = lhsCollider.ClosestPointOnBounds(rhs.transform.position);

else

lhsClosestPoint = lhs.transform.position;

if (rhsCollider != null)

rhsClosestPoint = rhsCollider.ClosestPointOnBounds(rhs.transform.position);

else

rhsClosestPoint = rhs.transform.position;

return (lhsClosestPoint - rhsClosestPoint).sqrMagnitude;

}

为所有的监测器添加配置，最后挂载Detector Logic Gate组件，设置门类型为与门，当所有关联的监测器输出全为真时，即双手伸直相对拍掌时，逻辑门输出为真，触发回调方法。

1. 冰箱

冰箱的主要交互在于冰箱门把手，通过握住把手拉开或合上冰箱门，实现带手势的接触式操作。

新建一个空物体，挂载2个ExtendedFingerDetector组件，分别代表左右手。根据握拳习惯，将中指、无名指和小拇指配置为弯曲状态。

在每个冰箱门把手上添加FridgeDoorView冰箱门视图脚本，以控制冰箱门的协同旋转。为了让冰箱门紧跟着手一起旋转，门的旋转角度需要根据手绕门轴的水平偏移角度来计算。具体做法是：将前后两帧手的所在位置向把手所在的水平面做一个投影，然后计算两个投影向量的角度，即求得所需要的门的旋转角，如图5.12所示。



图5.12 手部绕门轴旋转角求算图解

根据上图图解，可编写出求算旋转角的代码：

lastPos = currPos;

currPos = hand.position;

Vector3 lastVec = lastPos - fridgePos.position;

float lastHeight = Vector3.Dot(lastVec, nVec);

Vector3 lastProjVec = new Vector3(lastPos.x, lastPos.y - lastHeight, lastPos.z);

Vector3 currVec = currPos - fridgePos.position;

float currHeight = Vector3.Dot(currVec, nVec);

Vector3 currProjVec = new Vector3(currPos.x, currPos.y - currHeight, currPos.z);

Vector2 from = new Vector2((fridgePos.position - lastProjVec).x, (fridgePos.position - lastProjVec).z);

Vector2 to = new Vector2((fridgePos.position - currProjVec).x, (fridgePos.position - currProjVec).z);

rotateAngle = VectorAngle(from, to);

其中求算投影向量时使用了一种简单的技巧：保持原向量的xz坐标不变，y坐标减去或加上原向量到投影平面的距离，即得到目标投影向量。

当采取鼠标操作时，由于鼠标只能在二维平面移动，缺失了z坐标的信息，因而不能够使用原来的算法。通过忽略y轴的影响，设把手到门轴的距离为Length，鼠标点击位置通过Camera.main.ScreenToWorldPoint函数转换为三维坐标，再根据勾股定理求得未知的z轴坐标，得到了三角形三边，便能够利用反三角函数求算出角度。

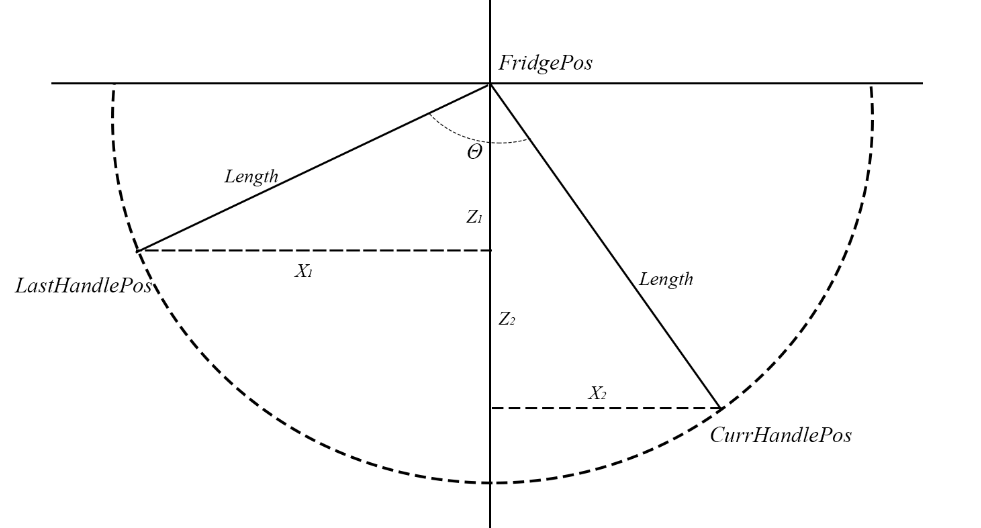


图5.13 鼠标拉动门把旋转角图解

根据图5.13的图解，在OnMouseDown回调方法中编写代码求算出鼠标操控把手的旋转角：

IEnumerator OnMouseDown()

{

isMouseDown = true;

Vector3 friPos = fridgePos.position;

lastPos = Camera.main.ScreenToWorldPoint(

new Vector3(Input.mousePosition.x, 0, Camera.main.WorldToScreenPoint(handle.position).z));

lastPos.y = friPos.y;

while (Input.GetMouseButton(0))

{

currPos = new Vector3(0, friPos.y, handle.position.z);

currPos.x = Camera.main.ScreenToWorldPoint(

new Vector3(Input.mousePosition.x, 0, Camera.main.WorldToScreenPoint(handle.position).z)).x;

float z2 = Mathf.Pow(length, 2) - Mathf.Pow((currPos.x - friPos.x), 2);

if (z2 < 0) z2 = 0;

currPos.z = friPos.z - Mathf.Sqrt(z2);

float sin1 = Mathf.Abs(lastPos.z - friPos.z) / length;

if (sin1 > 1f) sin1 = 1;

float lastAngle = Mathf.Rad2Deg \* Mathf.Asin(sin1);

if (lastPos.x > friPos.x) lastAngle = 180 - lastAngle;

float sin2 = Mathf.Abs(currPos.z - friPos.z) / length;

if (sin2 > 1f) sin2 = 1;

float currAngle = Mathf.Rad2Deg \* Mathf.Asin(sin2);

if (currPos.x > friPos.x) currAngle = 180 - currAngle;

rotateAngle = currAngle - lastAngle;

lastPos = currPos;

yield return null;

}

isMouseDown = false;

}

脚本编写完毕后，为每个门把手添加触发器以响应来自手的触碰事件或鼠标的点击事件。

## 资源打包与发布

# 手势交互操作与测试

## Leap Motion驱动参数调整

## 交互测试和结果分析

参考文献

[1] 王珊，萨师煊. 数据库系统概论（第4版）[M]. 高等教育出版社. 2006-05

[2] 谭云杰. 大象：Thinking in UML（第2版）[M]. 中国水利水电出版社. 2012-03

[3] Jeffrey Richter. CLR via C# 4th Edition [M]. Microsoft Press. 2012-12

[4] Bruce Johnson. Visual Studio 2015高级编程（第6版）[M]. 清华大学出版社. 2016-05

[5] M.Minasi, Kevin, Greene, Christian. 精通Windows Server 2012 R2（第5版）[M]. 清华大学出版社. 2015-04