基于 Unity3D 的体感手柄游戏的设计

专业：电子信息工程　学生：赖星翰　　指导老师：林梅燕

摘要

随着实时计算机三维绘制技术、视景立体仿真技术、对观察者眼球和动作的跟踪技术以及语音识别、触觉反馈、无线传输等技术的发展，虚拟现实技术应势而生，这是一种综合利用各种智能化控制接口设备，并借助计算机图形技术以及现实仿真技术生成的虚拟环境技术，是一种崭新的人机交互状态的技术。

Unity3D 游戏引擎作为当今世界上主流的引擎之一，已在各种游戏平台的制作中得到广泛运用，同时随着虚拟现实技术的发展，VR 应用市场中以 Unity3D 引擎制作的游戏占据了半壁江山。因此，在 Unity3D 游戏引擎强大的技术支持下，基于 Unity3D 的 VR 游戏开发将是未来游戏市场中重要的组成部分。由于目前键盘鼠标以及传统手柄这类输入设备已不能满足 VR 游戏的需求，为了使 VR 玩家得到更完美的虚拟现实体验，一款能把真实世界的环境数据映射到虚拟世界的体感游戏手柄能使游戏中的视角以及射击、劈砍等动作得到不错的反馈，该输入设备将成为 VR 玩家体验第一人称射击类、角色扮演类等游戏最好的选择。

综上所述，以 Unity3D 引擎作为上位机实现对游戏中人物的体感操控的手柄可以更好地为 Unity3D 游戏玩家服务，因此本文提出了一种基于 Unity3D 的体感手柄游戏的设计。该设计由 Unity3D 游戏以及体感手柄这两部分组成，在 Unity3D 游戏引擎中进行人物建模以及游戏场景搭载，并通过体感游戏手柄实现对游戏中角色的操作。游戏手柄以 STC89C52RC 单片机为控制核心，通过带有三轴加速器和三轴陀螺仪的 MPU6050 模块、BLK-MD-HC-06 蓝牙模块、按键电路、MC7805CT 三端稳压输出电路以及最小系统电路等来实现无线体感游戏手柄的功能。本设计极大提升了人机交互地体验感，将给予玩家更真实、更震撼的沉浸感，让开发者和玩家们尽情享受游戏中的另一个“真实”的世界。

关键词：Unity3D 游戏引擎　　体感手柄　　陀螺仪　　蓝牙　　沉浸感

目　录

1 绪论

1.1 研究的背景及意义

1965 年，人机交互的图形显示系统、语音交流方式以及视与力反馈设备的虚拟现实基本思想已被提出，从此，人们对虚拟现实技术的探索有了新的研究方向。到了 80 年代，VPL 公司提出了“Virtual Reality”这一个虚拟现实的专用名词[1]，并逐渐被人们熟知和使用。进入 90 年代，虚拟现实建模语言的出现，为图形数据的传输和交互奠定了基础，并已出现了一些较为典型的虚拟现实应用系统。之后，随着计算机硬件技术和图形技术的飞速发展，使得基于庞大数据集合的实时画面制作成为可能，随之带来的是人机交互设备的不断创新，实用配套的游戏设备开始不断地涌入人们的生活中。这些技术的不断进步都为虚拟现实游戏的发展打下了坚实的基础。

随着游戏领域技术的发展，无论是从一开始的 MUD 文字游戏演变至二维游戏，还是发展至今的三维游戏，带给游戏玩家代入感以及沉浸感才是重中之重[2]。但是，目前仍存在众多现实技术难题难以解决，无法让玩家在游戏中拥有仿佛置身于另一个世界的感受。而虚拟现实技术的出现为广大游戏玩家以及开发者带来了曙光，同时也给竞争激烈的游戏市场带来了翻天覆地的变化，催生了专为游戏而生的虚拟现实设备，它能使游戏玩家在保持实时性和交互性的同时，获得极高的真实度与沉浸感。许多游戏引擎公司也将虚拟现实技术结合到游戏引擎中，Unity3D 游戏引擎更是 VR 游戏市场中的领头人[3]。

虚拟现实技术以及 Unity3D 游戏引擎的发展趋势，带来的则是笨重的传统鼠标和键盘变得难以满足游戏玩家的体验需求，任何超过设备线缆长度范围的移动以及行为都会被束缚[4]。输入设备是能够带给游戏玩家最重要的体验设备，为了使 VR 玩家得到更好的虚拟现实游戏沉浸感，一款基于 Unity3D 引擎开发的游戏配置一台能够控制游戏中人物的无线体感手柄将更好地为 Unity3D 游戏玩家服务。

1.2 国内外研究的发展现状

游戏手柄是主机游戏以及 PC 游戏中的一种常见输入设备，游戏玩家能够通过手柄上的按钮对虚拟角色进行操控。

在国外，传统游戏手柄的标准是由日本任天堂公司确立并延续至今，分为控制方向的十字键，控制动作的 A、B、X、Y 键以及控制菜单的选择、暂停键。

在 1984 年之后，随着游戏设备的日将月就，游戏手柄加入了能控制方向和视角摇杆键，并增加了震动和力回馈等功能。为了配合 PC 游戏的即插即玩，游戏手柄上也开始带有 USB 数据线。除去十字方向键以及摇杆键这样的经典设计外，日本任天堂公司也是最早实现无线手柄的厂商。1989 年，任天堂公司便在 NES 游戏主机上尝试开发带有无线功能的游戏手柄。当时的主机容许多个游戏手柄连接到同一个发射器上，发射器再将信号发射给游戏主机上的接收器[5]，以此实现通过无线来控制游戏角色的目的。到了 2006 年，日本任天堂公司发布的游戏主机 Wii 获得了巨大成功，玩家们首次在游戏主机 Wii 上体验到可以直接以体感的方式控制游戏中的角色，而非传统的手柄按键形式。

在国内，中国游戏手柄生产厂商由于游戏主机的禁令，始终未能在游戏主机以及手柄外设的领域大有作为。但随着改革开放和禁令解除，带来的是中国硬件领域技术的不断提升。凭借成本的优势，中国厂商开始在游戏外设的领域上得到突破性的进展。从这个阶段开始，世界上 90%以上的游戏主机的手柄，均由中国的游戏手柄生产厂商设计和量产。与此同时，Doom、Unreal、Unity3D 等游戏引擎引起了游戏市场上的巨大变化，为中国的游戏手柄品牌走向世界带来了全新的篇章。以新型化、微型化、智能化和高性能化的游戏主机等设备为载体的游戏形态，已经演变为全世界最为主流的游戏发展方向[6]。在这一方向上，中国的游戏手柄品牌也在全球市场上获得了进一步的突破。目前中国十大游戏手柄排行榜前几位分别是北通、罗技、雷柏、任天堂、蓝正等。在 2015 年初，小米、新游、小旗等无线手柄的发布，也让业内人士纷纷预测，手柄市场的春天就要来临。

无论是国内还是国外，为了在激烈的游戏外设市场中脱颖而出，大大小小的游戏手柄厂商纷纷开始支持 Unity3D 游戏平台，为 Unity3D 游戏玩家提供了更加完善的游戏手柄驱动的支持以及更好的售后服务[7]。因此，本文提出一种基于 Unity3D 的体感手柄游戏的设计。本设计作为一款轻巧并带有体感、无线、实时性以及能给予游戏玩家更加真实的沉浸感，且能同 Unity3D 游戏引擎制作出的游戏相匹配的现代游戏手柄，将在游戏市场上占有一席之地并且拥有很好的发展前景。

1.3 研究主要内容

本设计主要是关于体感手柄的制作以及 Unity3D 游戏的开发这两个部分。

体感手柄是由 STC89C52RC 单片机、带有三轴加速度器及三轴陀螺仪的 MPU6050 模块、BLK-MD-HC-06 蓝牙无线传输模块、按键开关电路以及 MC7805CT 三端稳压输出等电路组成。STC89C52RC 单片机通过 I^2C 串行总线与 MPU6050 模块进行通信，接收以及存储从 MPU6050 模块传入的三轴加速度

和三轴角速度的原始数据，再将数据通过蓝牙模块发送给 Unity3D 游戏。

在 Unity3D 游戏引擎所搭建的游戏场景中，本设计利用脚本内编写的 Serial Port 类以及线程接收体感手柄传入的 12 个 8 位 AD 数据，再通过误差计算和互补滤波算法获得欧拉角，赋值给游戏场景中人物对象的 Rotation 这三个分量以控制角色 Camera 的转向。单片机的 P1 串口获取 6 个按键，分别控制人物的前进、后退、左移、右移、攻击及跳跃等功能，并通过角色脚本中的 Animation 代码和人物模型动画实现动作化。敌人拥有自动寻找玩家并朝玩家移动和自动攻击等功能。游戏内带有刚体碰撞、血槽 UI 等。

2 基于 Unity3D 的体感手柄游戏总体设计

2.1 本设计相关技术

2.1.1 Unity3D 游戏引擎

Unity3D 游戏引擎是由美国 Unity Technologies 公司推出的一款大型游戏制作工具，它能让开发者轻松开发例如三维立体游戏、实时三维动画和可视化建筑等类型的互动场景内容。Unity3D 是一个专业的游戏制作引擎，具有强大的跨平台性，业内领先的多平台支持使开发者能在几乎所有的主流游戏平台上发布游戏。Unity3D 游戏引擎本身为开发者提供了音频设置、逻辑架构、图形渲染以及物理碰撞等组件，这些组件实现了游戏引擎所需的基础功能，并在脚本的作用下组合成更为高级的功能。模块功能是以脚本的形式提供的相对独立的通用功能组件，通过与高层的粘合代码脚本的共同作用，将游戏引擎的模块功能组件和基础组件相结合，以实现最终的游戏对象逻辑[8]。

2.1.2 STC89C52RC 控制模块

STC89C52RC 是 STC 公司开发的一款具有高速度、低功耗、高性能、超强抗干扰的微控制器，完全兼容传统 8051 的指令代码。STC89C52 带有经典的 MCS-51 内核，同 AT89C52 单片机芯片相比，内部集成了 4KB 容量的 E^2 PROM 功能。STC89C52RC 单片机采用 Flash 存贮器技术，大大降低了成本，能为绝大多数的嵌入式控制应用设计提供价廉且灵活的解决方案。片内有 4K 字节的可重复编程快擦写的程序存储器，用户的程序可用空间达到 8K 字节，有 32 个双向输入输出，共 3 个十六位定时器及计数器[9]。STC89C52RC 可以通过 Keilμ Vision5 软件和最小系统电路烧录程序。

2.1.3 MPU6050 模块

MPU6050 是 InvenSense 公司生产的一款 6 轴传感器模块，该模块使陀螺仪与加速度器之间的时间差问题得以解决。MPU6050 模块体积小巧、性能稳定、用途广泛，是制作自平衡小车、四旋翼飞行器和体感游戏手柄等设备必不可少的传感器解决方案。MPU6050 的角速度可感测范围为±250、±500、±1000 与±2000°/sec(dps)，加速器可感测范围为±2g、±4g±8g 与±16g，能够准确追踪使用者的快速与慢速动作[10]。MPU6050 通过加速度器和陀螺仪分别测出三维空

间中的 X、Y、Z 三个方向上的加速度有效数字量以及角速度有效数字量，这些以数字形式输出的 6 轴原始数据，经过旋转矩阵、四元数换算、误差计算、滤波以及欧拉角的融合演算等可以得到开发者所需的角度。该模块可以应用于虚拟现实增强、游戏控制器、可穿戴传感器以及电子稳像等途径。

2.1.4 BLK-MD-HC-06 蓝牙模块

BLK-MD-HC-06 是深圳博陆科电子科技有限公司生产的一款无线数据传输的蓝牙模块，其采用英国 Cambridge Silicon Radio 公司推出的 BlueCore4-Ext 芯片，支持 Bluetooth V2.0+EDR 蓝牙规格。该模块内置 2.4GHz 天线，具有高性能、高灵敏度、小体积、低功耗和低成本等优点，只需简单的外围电路就能实现蓝牙传输。该模块避免了复杂的线缆连接，可以方便地与带有蓝牙设备的电脑相连接，能直接代替串口数据线使用。BLK-MD-HC-06 蓝牙模块适用于各种 3.3V 的单片机，能够对名称、密码和波特率等信息通过输入 AT 指令进行设置，默认波特率是 9600，默认配对密码为 1234。

2.1.5 互补滤波技术

对 MPU6050 模块来说，加速度器对游戏手柄的加速度器的取值较为敏感，取瞬时值计算角度的误差较大；而陀螺仪通过积分运算得到的角度会有累积误差，累积误差在短时间内产生的积分漂移和零点漂移都不大，但会随着时间的增加而产生较大的误差。所以加速度器和陀螺仪可以通过各自的优势特性进行互补，获得准确的姿态角度。滤波则是根据这两个传感器不同的特性，通过滤除加速度器的高频噪声信号，并滤掉陀螺仪的低频噪声信号，以不同的滤波算法，相加得到整个频带的信号。互补滤波技术，即在很短时间内获取陀螺仪的最佳角度同时滤除低频信号，并定时从加速度器中采样角度和取平均值，再滤除高频的信号，以此对陀螺仪得到的角度进行校正[11]。

2.2 总体设计介绍

本设计主要是以 STC89C52RC 单片机作为控制的核心，通过 MC7805CT 三端稳压输出电路、带有三轴加速度器和三轴陀螺仪的 MPU6050 模块、BLK-MD-HC-06 蓝牙模块以及按键等电路来实现体感游戏手柄的功能，并将 Unity3D 游戏引擎作为上位机，通过人物的建模、游戏场景的搭载、编写 C#脚本来实现对游戏角色的操作。

总体设计框图如 2-1 所示：

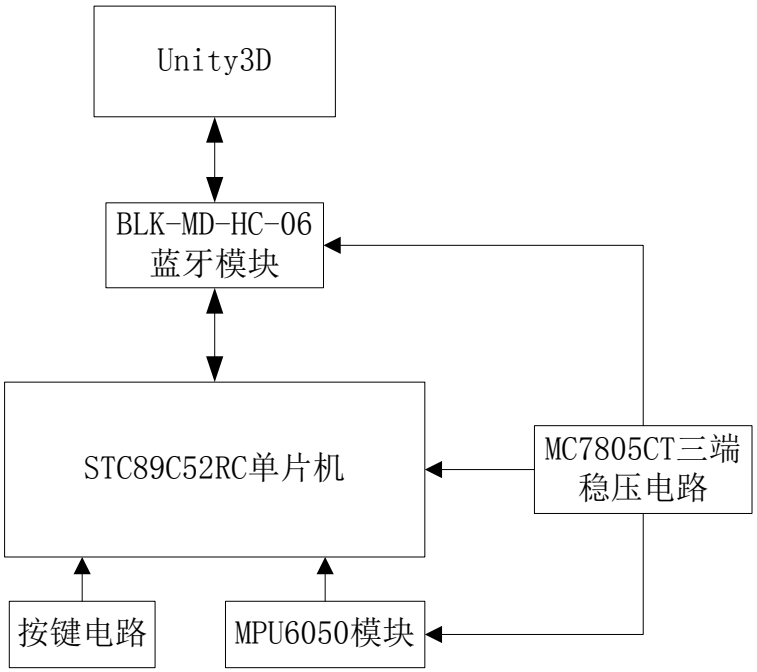

图 2-1 总体设计框图

3 手柄硬件设计

3.1 手柄硬件总体概述

体感游戏手柄的设计基于 STC89C52RC 单片机为主控制器，读取 MPU6050 模块传入的三轴加速度和三轴角速度的原始数据，该原始数据为 6 个低 8 位 AD 值和 6 个高 8 位 AD 值。STC89C52RC 单片机将这 12 个 AD 值数据通过蓝牙模块传给上位机的串口，并在上位机脚本中对高 8 位的 AD 值进行左移 8 位，再同低 8 位的 AD 值相与，可得 3 个加速度的 16 位 AD 值和 3 个角速度的 16 位 AD 值。本设计利用 GP1604S-6F22-9V 电池输出 9V 的电压，经过 MC7805CT 三端稳压电路输出 5V 的电压供给单片机、MPU6050 和蓝牙等模块。

3.2 手柄硬件电路设计

手柄硬件电路如图3-1所示，该电路主要由STC89C52RC单片机、MC7805CT三端稳压芯片、MPU6050 模块、蓝牙模块、按键电路、晶振电路和复位电路等构成。

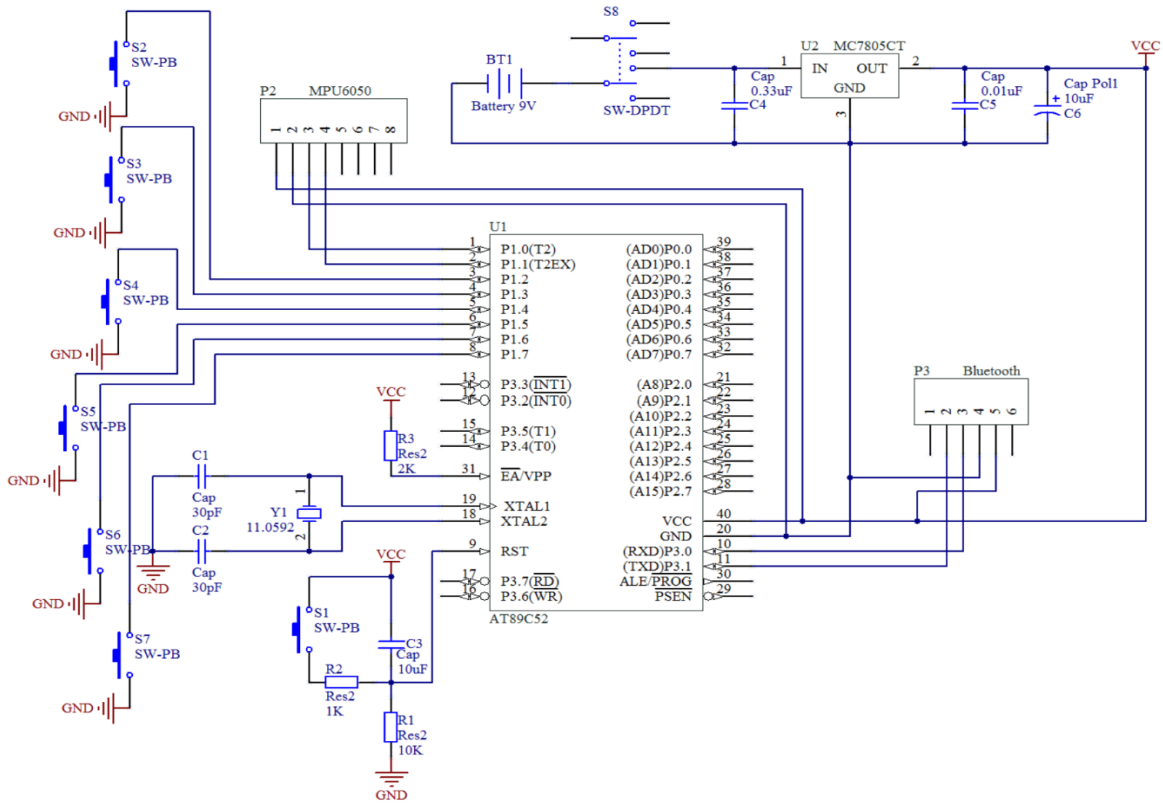

图 3-1 手柄硬件电路图

3.1.1 单片机与 MPU6050 通信电路设计

STC89C52RC 单片机中的 P1.0 引脚和 P1.1 引脚分别与 MPU6050 模块的串行时钟 SCL 的引脚以及串行数据的 SDA 引脚相连，并将 MPU6050 模块上的+5V 电源引脚连接到 MC7805CT 稳压电路的输出端，用以接收 MPU6050 模块传入的三轴加速度和三轴角速度的原始数据。单片机与 MPU6050 通信电路如图 3-2 所示：

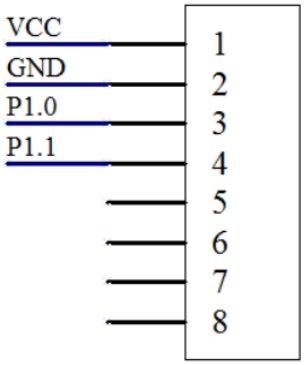

图 3-2 单片机与 MPU6050 通信电路图

3.1.2 无线蓝牙电路设计

STC89C52RC 单片机上的 P3.0 引脚为 RXD，P3.1 引脚为 TXD,将这两个引脚分别同 BLK-MD-HC-06 蓝牙模块上的 TXD 引脚与 RXD 引脚相连接，用以发送单片机中存储的数据。无线蓝牙电路如图 3-3 所示：

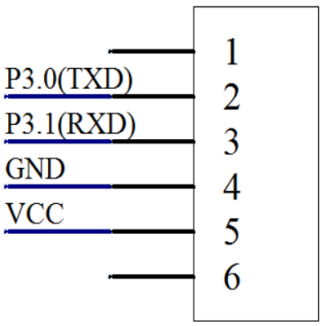

图 3-3 无线蓝牙电路图

3.1.3 MC7805CT 三端稳压电路设计

GP1604S-6F22-9V 电池输出 9V 的电压，经过 MC7805CT 芯片内部电路的稳压以及各级电容滤波后，输出 5V 的稳定电压供给单片机、MPU6050 和蓝牙等模块。因 MC7805CT 芯片在实际测试时会因发热而产生过热保护，需加上一块散热铝片。MC7805CT 三端稳压电路如图 3-4 所示：

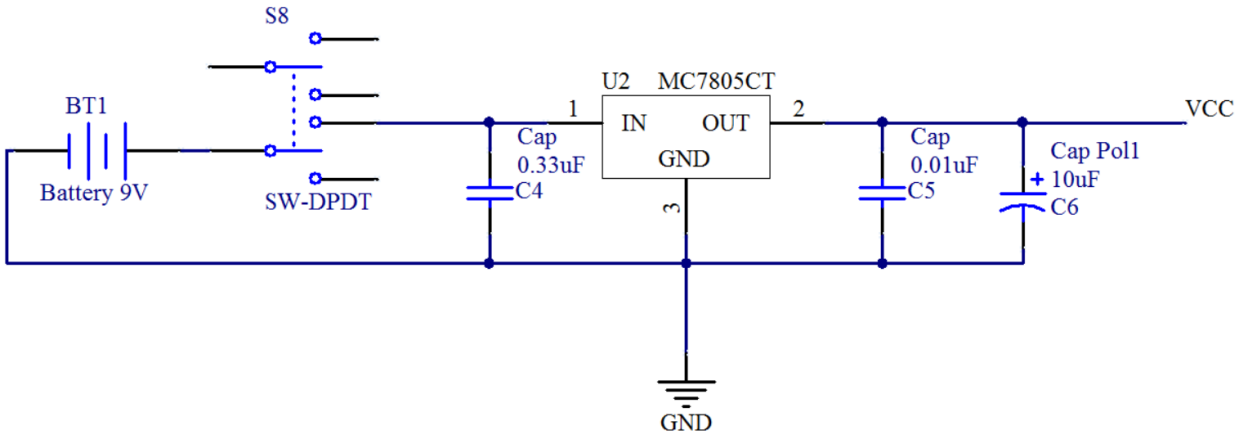

图 3-4 MC7805CT 三端稳压电路图

3.1.4 按键电路设计

将 6 个按键开关分别连接单片机的 P1.2 引脚、P1.3 引脚、P1.4 引脚、P1.5 引脚、P1.6 引脚和 P1.7 引脚，分别发送 0x20、0x10、0x08、0x04、0x02 和 0x01 这 6 个十六进制数据，再由 Unity3D 游戏中的串口类接收和判断。按键电路如图 3-5 所示：

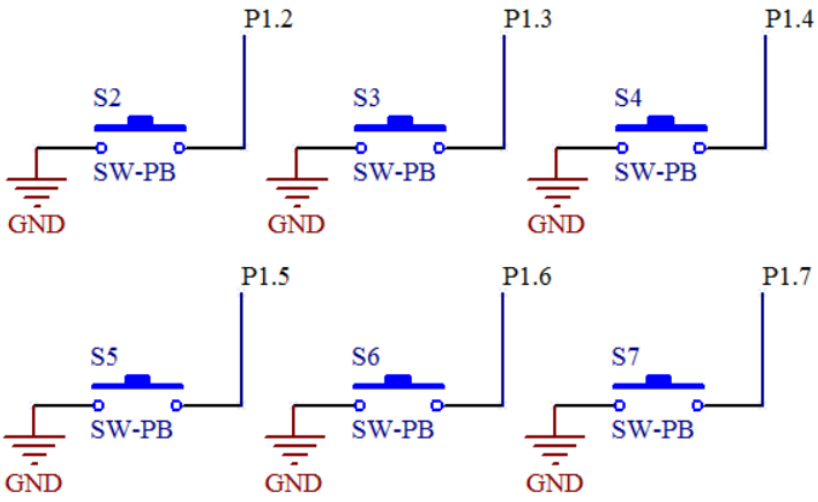

图 3-5 按键电路图

4 手柄软件设计

4.1 手柄程序设计语言及开发环境

STC 系列单片机支持汇编语言和 C 语言，本设计的下位机软件是通过编写 C 语言程序来实现。用 C 语言来开发单片机软件的优点有程序执行效率高、可读性强和软件调试方便等，因此 C 语言编程在单片机控制设计中已成为主流的编程语言。

本设计的手柄软件部分是在 Keilμ Vision5 的环境下编译和烧录。Keilμ Vision5 是美国 Keil Software 公司推出的 C 语言软件开发解决方案，兼容所有 8051 系列单片机，集开发、仿真和编译于一体。Keil μ Vision5 为 8051 系列单片机的软件开发提供了丰富的库函数和强大方便的调试工具，全 Windows 界面，易学易用[12]。

4.2 手柄软件设计

手柄软件主程序流程如图 4-1 所示，先将单片机、蓝牙模块的串口和 MPU6050 模块初始化，再执行扫描按键输入程序，接着单片机获取和存储 MPU6050 传入的 6 轴原始数据和按键数据，经蓝牙模块发送给上位机，数据发送完成后返回扫描按键输入程序，继续循环发送。

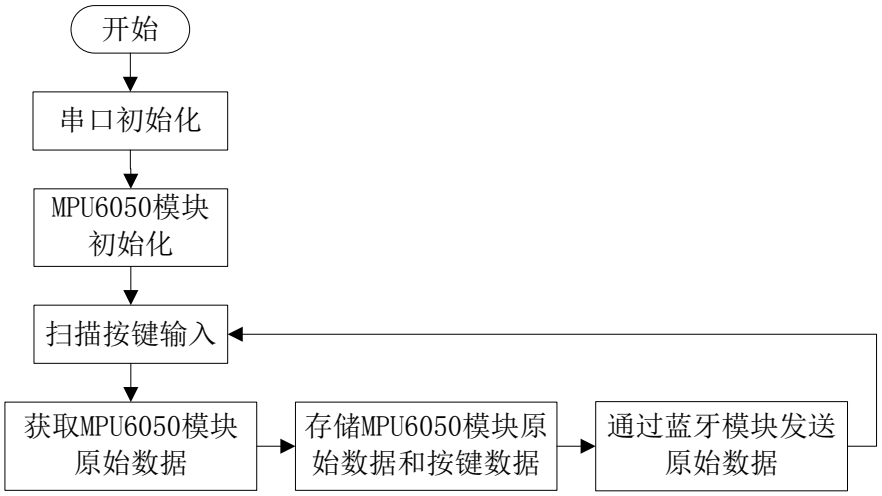

图 4-1 手柄软件主程序流程图

4.2.1 I² C 通信设计

I²C 通信设计是通过串行时钟 SCL 线和串行数据 SDA 线在单片机 STC89C52RC 和 MPU6050 模块间传递数据。MPU6050 检测到串行时钟 SCL 线和串行数据 SDA 线均为高电平，发送给单片机一个起始信号，单片机判断是否接收到该起始信号，若收到则向 MPU6050 回馈应答信号。MPU6050 收到单片机的应答信号后开始发送第一个字节的数据，单片机接着判断是否收到该数据，若收到则返回一个应答信号，MPU6050 收到该应答信号后继续发送下一个数据字节。当 MPU6050 发送完最后一个数据字节，单片机判断数据是否接收完整，若完整则再回馈一个应答信号。MPU6050 收到后，通过向单片机发送一个停止信号，停止本次通讯并返回主程序。I²C 通信流程如图 4-2 所示：

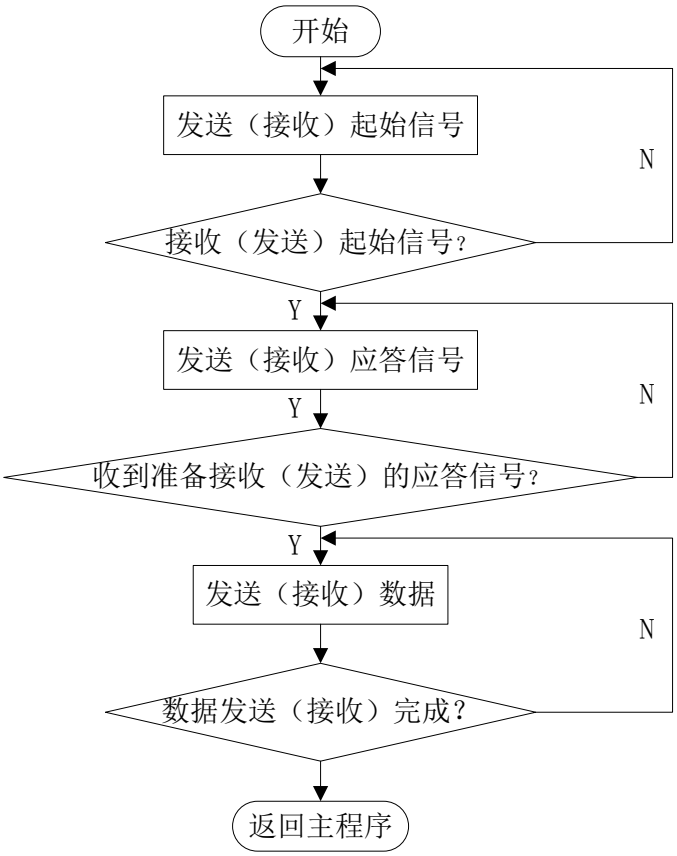

图 4-2 I²C 通信流程图

4.2.2 手柄按键判断设计

单片机的 P1.0 引脚和 P1.1 引脚作为 I²C 通信的时钟引脚 SCL 和数据引脚 SDA 的定义，P1.2 引脚到 P1.7 引脚则作为手柄上的 6 个按键即分别控制上位机中人物的前进、后退、左移、右移、攻击和跳跃。0111 1111、1011 1111、1101 1111、

1110 1111、1111 0111、1111 1011 分别对应为这 6 个按键的 8 位字节，未按下时为 1111 1111，按下时则将每个按键对应的数据传入数组的第十四个字节定义的 Key 中并存储该按键数据。按键数据存储结束后，返回主程序。

手柄按键判断流程如图 4-3 所示：

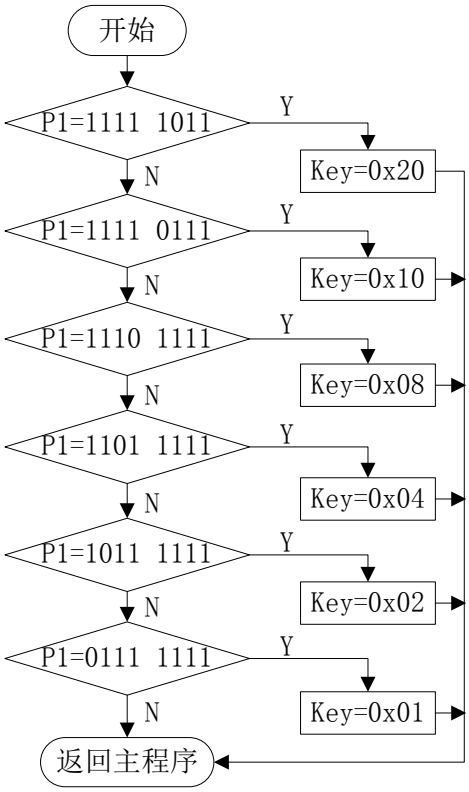

图 4-3 按键判断流程图

4.2.3 原始数据和按键数据的存储设计

在下位机程序中定义一个无符号的数组，一共 15 个字节，每个字节 8 位。分别定义第一个字节为标头和最后一个字节为标尾，通过这两个字节判断数据是否接收完整。第二个字节到第十三个字节作为这 12 个三轴加速度和三轴角速度的原始数据的存储，第十四个字节则作为手柄按键是否按下的判断的数据存储。数据存储结束后，返回主程序。

原始数据和按键数据的存储流程如图 4-4 所示：

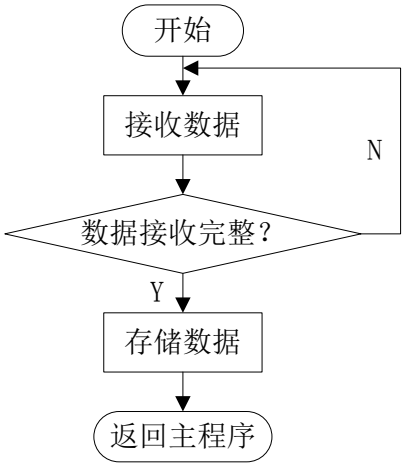

图 4-4 原始数据和按键数据的存储流程图

4.2.4 无线蓝牙串口通信设计

单片机的第 10 个引脚 RXD 和第 11 个引脚 TXD 分别与 BLK-MD-HC-06 蓝牙模块上的 TXD 和 RXD 相连，将单片机中的存储数据通过蓝牙模块传送给 Unity3D 游戏。本通信设计先对单片机和蓝牙模块进行初始化，接着检测与上位机的连接，若与上位机配对成功，则发送或接受数据。当数据发送或者接收结束，返回主程序。无线蓝牙串口通信流程如图 4-5 所示：

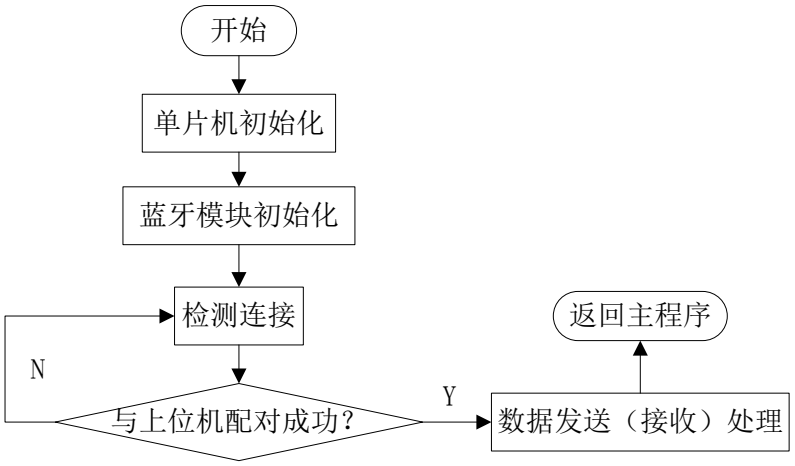

图 4-5 无线蓝牙串口通信流程图

5 Unity3D 游戏设计

5.1 Unity3D 游戏设计语言及开发环境

在本设计的上位机中，运用的是 C#语言来编写 Unity3D 游戏引擎中各种脚本。C#是一种为.net 程序框架而创造的语言，它是由 C 和 C++语言衍生出的编程语言，它在继承 C 和 C++语言功能优点的同时也去掉了一部分复杂特征[13]。C#语言以其简洁的语法、精心的面向对象设计、灵活性、兼容性和完整的安全性与错误处理成为 Unity3D 游戏引擎脚本开发的首选语言。在已经发布的 Unity5 引擎版本中，官方加强了对 C#的支持。市场上大概有超过 70%的 unity3D 游戏是用 C#开发的，而且 C#比起 JavaScript 更加适合搭建大型的游戏项目[14]。

Unity3D 的默认的编辑器是 MonoDevelop，但同 Visual Studio 编辑器相比，无论是响应的速度，各种工具的查找，舒适的颜色界面，快捷键的方便使用，还是调试工具的完善等，后者的功能都更加强大。

本设计是在 Visual Studio 2015 的编辑环境下运用 C#语言编写 Unity3D 游戏中所需的脚本。

5.2 Unity3D 游戏软件设计

在 Unity3D 游戏中，游戏开始时，通过编写的线程脚本接收手柄传入的 12 个 8 位 AD 值数据，再将数据进行互补滤波和误差计算获得欧拉角，赋值给游戏场景中人物对象，并将按键数据赋值给角色。当欧拉角和按键数据接收完成，处理该数据，即通过欧拉角数据控制玩家角色的镜头转向，按键数据控制人物的前进、后退、左移、右移、攻击及跳跃。如果数据未接收完整，则返回线程继续接收该数据。游戏中的敌人拥有自动寻找玩家并朝玩家移动和自动攻击等功能。游戏结束则关闭串口，数据接收中止。

Unity3D 游戏软件总体流程如图 5-1 所示：

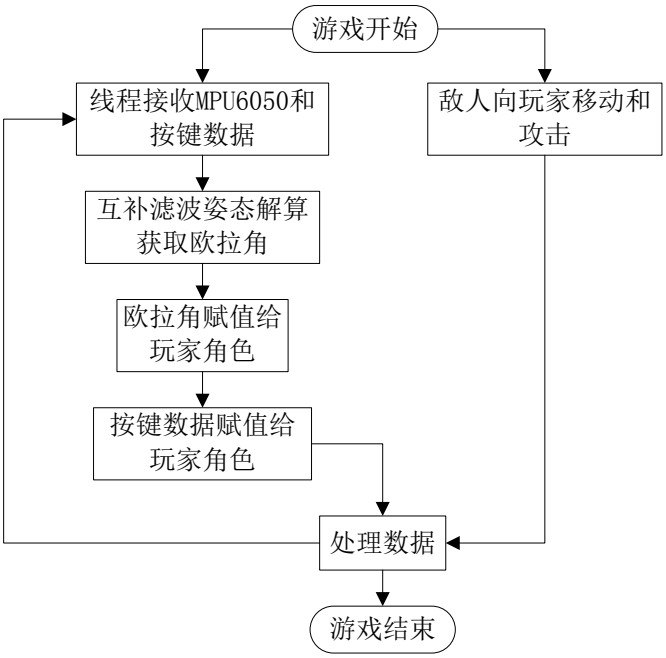

图 5-1 Unity3D 游戏软件总体流程图

5.2.1 串口线程接收设计

通过 C#语言的 Serial Port 类来实现 Unity3D 与单片机的串口数据通信，并利用线程接收单片机 STC89C52RC 发送给 Unity3D 游戏的数据。该线程判断数据是否接收完整，若完整则存储该数据，数据处理完成后返回主程序。

串口线程接收流程如图 5-2 所示：

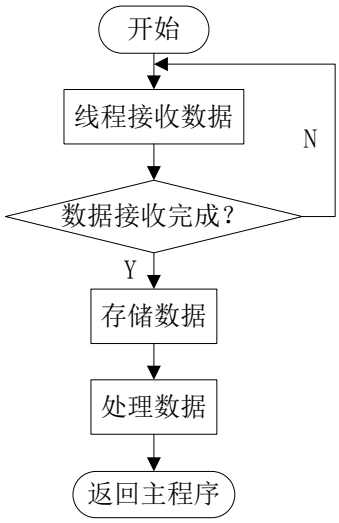

图 5-2 串口线程接收流程图

串口线程接收设计的部分代码如下：

```
//串口线程接收
listStr = new List<byte>();//从 ListByte 中读取数据，用做数据处理
ListByte = new List<byte>();//存放读取的串口数据
spStart = new SerialPort("COM8", 9600, Parity.None, 8, StopBits.One);//串口实例化
spStart.Open();//打开串口
tPort = new Thread(DealData);//该线程处理数据
tPortDeal = new Thread(ReceiveData);//该线程接收数据
```

5.2.2 互补滤波的姿态解算设计

本设计中的互补滤波的姿态解算，是通过互补滤波算法在短时间内采用陀螺仪得到的高通滤波后的角度为最优，同时对加速度计采样来的低通滤波后的角度进行取平均值来校正陀螺仪得到的角度，并修正随时间增加的温度漂移和积分漂移带来的较大误差。

互补滤波的姿态解算流程如图 5-3 所示：

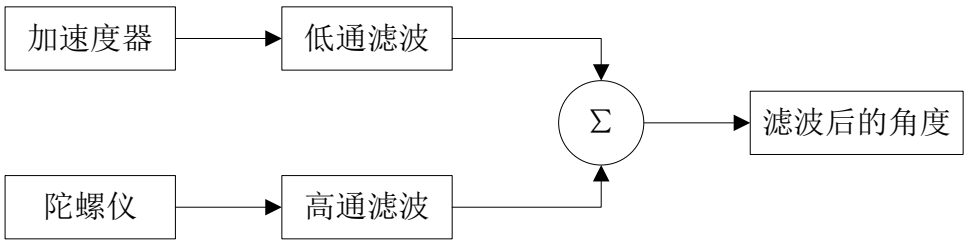

图 5-3 互补滤波姿态解算流程图

在三维空间中的旋转可以用单位四元数来描述。如公式 5-1 所示，定义一个四元数：

$$q = a + \vec{u} = q_0 + q_1 i + q_2 j + q_3 k \qquad \text{公式(5-1)}$$

如公式 5-2 所示，四元数通过欧拉角的方向余弦矩阵描述为：

$$C_n^b = \begin{bmatrix} q_1^2 + q_0^2 - q_3^2 - q_2^2 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & q_2^2 - q_3^2 + q_0^2 - q_1^2 & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & q_3^2 - q_2^2 - q_1^2 + q_0^2 \end{bmatrix} \qquad \text{公式(5-2)}$$

接着根据欧拉角和四元数方向余弦阵的转换关系，再把四元数转换成游戏所

需的欧拉角。如公式 5-3 所示：

$$C_n^b = C_2^b C_1^2 C_n^1 = \begin{bmatrix} \cos\gamma & 0 & -\sin\gamma \\ 0 & 1 & 0 \\ \sin\gamma & 0 & \cos\gamma \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\psi & -\sin\phi & 0 \\ \sin\phi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\gamma\cos\psi + \sin\gamma\sin\psi\sin\theta & -\cos\gamma\sin\psi + \sin\gamma\cos\psi\sin\theta & -\sin\gamma\cos\theta \\ \sin\psi\cos\theta & \cos\psi\cos\theta & \sin\theta \\ \sin\gamma\cos\psi - \cos\gamma\sin\psi\sin\theta & -\sin\gamma\sin\psi - \cos\gamma\cos\psi\sin\theta & \cos\gamma\cos\theta \end{bmatrix} \text{公式(5-3)}$$

设计初期，在脚本中编写滤波代码获取 Unity3D 游戏所需的视角时，采用的是四元数解算姿态获取欧拉角的方法。部分程序如下所示：

```
//gx、gy、gz 对应三个轴的角速度，ax、ay、az 对应三个轴的加速度
norm = (float)Math.Sqrt(ax * ax + ay * ay + az * az); //正常化
ax = ax / norm; //将加速度单位化为向量
vx = 2 * (q1 * q3 - q0 * q2); //四元数方向余弦阵换算
ex = (ay * vz - az * vy); //向量间的叉乘，用于纠正陀螺仪的误差
exInt = exInt + ex * Ki; //对误差进行积分增益
gx = gx + Kp * ex + exInt; //调节 Kp 的参数修正零漂
norm = (float)Math.Sqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
q0 = (q0 + (-q1 * gx - q2 * gy - q3 * gz) * halfT) / norm;//获取四元数
angleRoll = (float)(Math.Atan2(2 * q2 * q3 + 2 * q0 * q1, -2 * q1 * q1 - 2 * q2 * q2 + 1) * 57.3); //获得 Roll 角
angleYaw = (float)(Math.Atan2(2 * q1 * q2 + 2 * q0 * q3, -2 * q2 * q2 - 2 * q3 * q3 + 1) * 57.3); //获得 Yaw 角
anglePitch = (float)(Math.Asin(-2 * q1 * q3 + 2 * q0 * q2) * 57.3);//获得 Pitch 角
```

但是在调试 Unity3D 游戏时发现，将四元数解算姿态获取欧拉角的这一方法通过脚本添加到控制的游戏对象后，游戏的视角不正确且一直在晃动，角色不能正常获取所需要的欧拉角，不能正确对角色进行操控。如左图 5-4 所示：

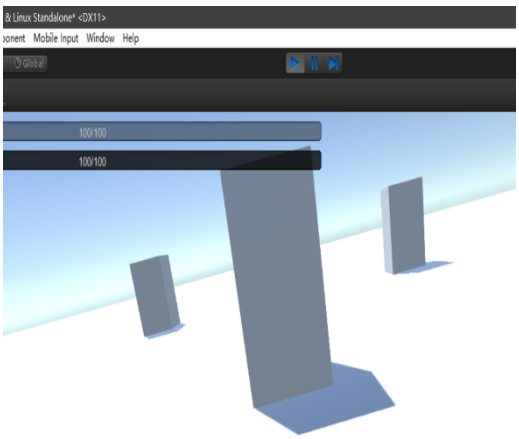

图 5-4 四元数解算姿态视角

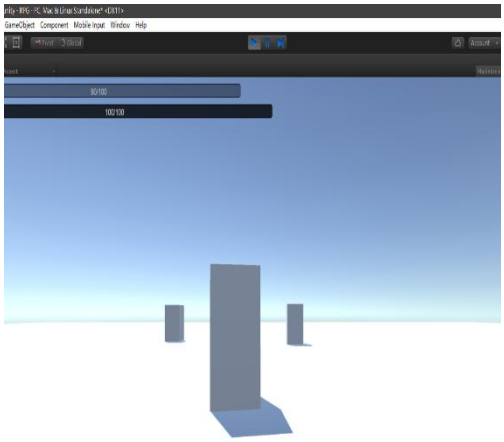

图 5-5 互补滤波算法视角

后决定更换滤波方式，通过误差计算以及互补滤波的方式，去除零点漂移并计算角速度值和加速度值，如右图 5-5 所示，最终成功得到正确的欧拉角以及所需的游戏视角。

误差计算以及互补滤波的部分程序如下所示：

```
float Acc_x = ax; //获取 X 轴的加速度
float Angle_ax = (Acc_x - 1100) / 16384; //消除零点漂移，获取角度
Angle_ax = (float)(Angle_ax * 1.2 * 180 / 3.14); //转化为度
float Gyro_x = gx; //静止时 X 方向的角速度输出
bias_gx = (float)(bias_gx * 0.998 + gy_sped * 0.002); //消除零点误差
gy_temp = gy_sped - bias_gx; //减去零点误差
p = tau / (tau + dtc); //tau 为计算比例，dtc 为抽样时间
//对陀螺仪角速度进行高通滤波处理,并对加速度器得到的角度进行低通滤波处理
angleRoll = p * (angleRoll + gy_temp * dtc) + (1 - p) * angle * (float)1.5;
```

通过图 5-4 和图 5-5 游戏视角的对比，可以得出在 Unity3D 游戏的实际调试中，互补滤波算法比起四元数姿态解算更符合本设计的应用。

5.2.3 游戏按键控制设计

通过编写 C#脚本，添加到玩家和敌人模型的组件上。通过判断 P1.2 引脚到 P1.7 引脚作为手柄上的 6 个按键传给上位机 Unity3D 的数据，从而控制玩家人物

的前进、后退、左移、右移、攻击和跳跃。移动、攻击和跳跃这几个动作都拥有各自的动画来实现。当数据处理结束，返回主程序。游戏按键控制流程如图 5-6 所示：

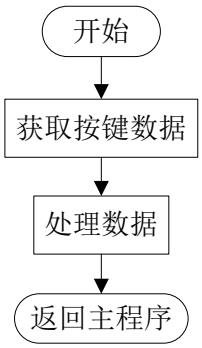

图 5-6 按键控制流程图

部分程序代码如下：

```
public const ushort keyW = 0x20; //对应手柄发送的 0x20 的数据
if (HaveKeyDown(MyKeyCode.keyW) == true) //按下手柄上前进的按钮
  {
      GetComponent<Animation>().Play("Run"); //角色前进的动画
      transform.Translate(Vector3.forward * Time.deltaTime * 6); //向前移动
  }
else if (HaveKeyDown(MyKeyCode.keyS) == true) //按下手柄上后退的按钮
  {
      GetComponent<Animation>().Play("Walk"); //角色后退的动画
  }
```

5.2.4 玩家视角转向设计

通过误差计算和互补滤波得到的欧拉角对需要控制的人物的 Rotation 三个分量进行赋值，以控制 Camera 随手柄的转向而进行的镜头转向。当欧拉角数据处理结束，返回主程序。玩家视角转向流程如图 5-7 所示：

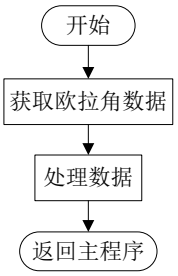

图 5-7 玩家视角转向流程图

部分程序代码如下：

```
void Update()
  {//每帧更新角色状态
      Debug.Log(angleRoll + ","+ angleYaw + "," + anglePitch); //控制台显示获得的欧拉角数据
      transform.eulerAngles = new Vector3(-angleRoll, angleYaw, 0); //对游戏对象进行欧拉角的赋值
  }
```

5.2.5 敌人 AI 脚本设计

通过编写 C#脚本，添加到敌人模型的组件上，让敌人能够自动寻找及朝向玩家，并以一定速度向玩家位置移动。当敌人与玩家的距离小于游戏中定义的矢量长度 2 时，能自动攻击玩家。每次攻击结束，返回主程序。

敌人 AI 脚本流程如图 5-8 所示：

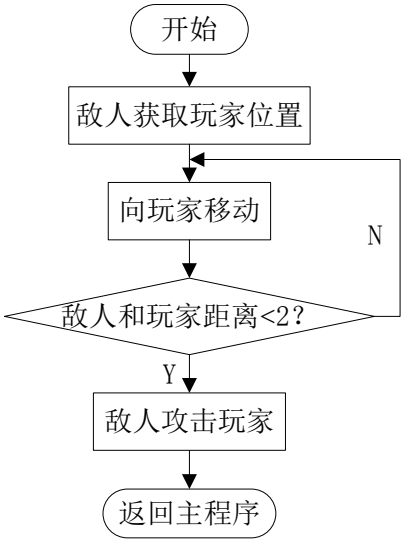

图 5-8 敌人 AI 脚本流程图

部分程序代码如下：

```
GameObject go = GameObject.FindGameObjectWithTag("Player"); //摄像机跟随标签为 Player 的游戏对象
Debug.DrawLine(target.position,myTransform.position,Color.blue); //画一条目标和对象直接蓝色的线
myTransform.rotation = Quaternion.Slerp(myTransform.rotation, Quaternion.LookRotation(target.position - myTransform.position),rotationSpeed*Time.deltaTime); //敌人看向玩家，即正面以一定速度转向玩家的位置
if (Vector3.Distance(target.transform.position, myTransform.position)> 2)
    {//当玩家和敌人的位置大于 2
        myTransform.position += myTransform.forward * Time.deltaTime; //敌人向玩家的位置移动
    }
```

5.2.6 攻击模块设计

通过编写 C#脚本，添加到玩家和敌人模型的组件上。当玩家和敌人处于游戏中定义的矢量长度 2 以内，敌人会自动以多种不同攻击的动作攻击玩家，玩家也可以通过手柄按键来攻击敌人。每次的攻击都设定有攻击时间间隔。每次攻击结束，返回主程序。

攻击模块流程如图 5-9 所示：

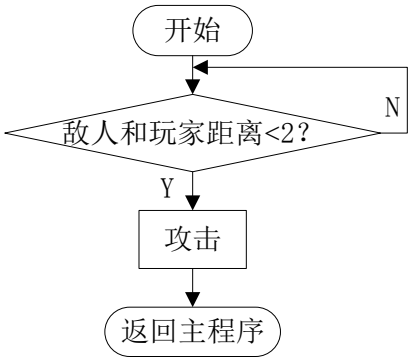

图 5-9 攻击模块流程图

部分程序代码如下：

```
Attack(); //攻击
attackTimer = coolDown; //攻击时间间隔等于冷却时间
float distance = Vector3.Distance(target.transform.position, transform.position); //玩家和敌人的直接距离
Vector3 dir = (target.transform.position - transform.position).normalized; //将玩家和敌人的直接距离变为向量 1.0 形式
float direction = Vector3.Dot(dir, transform.forward); //点乘方式计算敌人和玩家间的夹角
if (distance < 2.5f && direction>0)
    {//如果夹角小于 2.5 并大于 0，获取敌人的最大生命值和当前生命值
        EnemyHealth eh = (EnemyHealth)target.GetComponent("EnemyHealth");
        eh.AddjustCurHealth(-10); //每次攻击减少 10 点生命
    }
```

5.2.7 生命和血槽设计

通过编写 C#脚本，添加到玩家和敌人模型的组件上。玩家和敌人各有 100 的血量，当相距游戏中定义的矢量长度 2 以内，其中的一方每次受到对方的一次攻击，屏幕中的血槽 UI 长度以及血量会随之减少 10。

生命和血槽设计如图 5-10 所示：

图 5-10 生命和血槽设计

部分程序代码如下：

```
healthBarLength = Screen.width / 2; //生命槽长度为屏幕宽度的一半
AddjustCurHealth(0); //调整当前生命（治疗或收到伤害值）
GUI.Box(new Rect(10, 40, healthBarLength , 20), curHealth + "/" + maxHealth); //屏幕左上角显示矩形生命槽
```

curHealth += adj; //当前生命加上被攻击时受到的伤害

healthBarLength = (Screen.width / 2) * (curHealth / (float)maxHealth); //生命槽长度随当前生命大小改变长度

5.2.8 人物建模及游戏场景搭建设计

在 Unity3D 引擎中创建玩家、敌人以及游戏场景，游戏场景包括地形绘制、草地绘制、添加房屋、添加天空盒子、添加树木和添加光源等。同时在场景中对人物和地形添加碰撞检测与触发检测，以防止人物直接穿过地面或者墙壁等。

游戏场景搭建如图 5-11 所示：

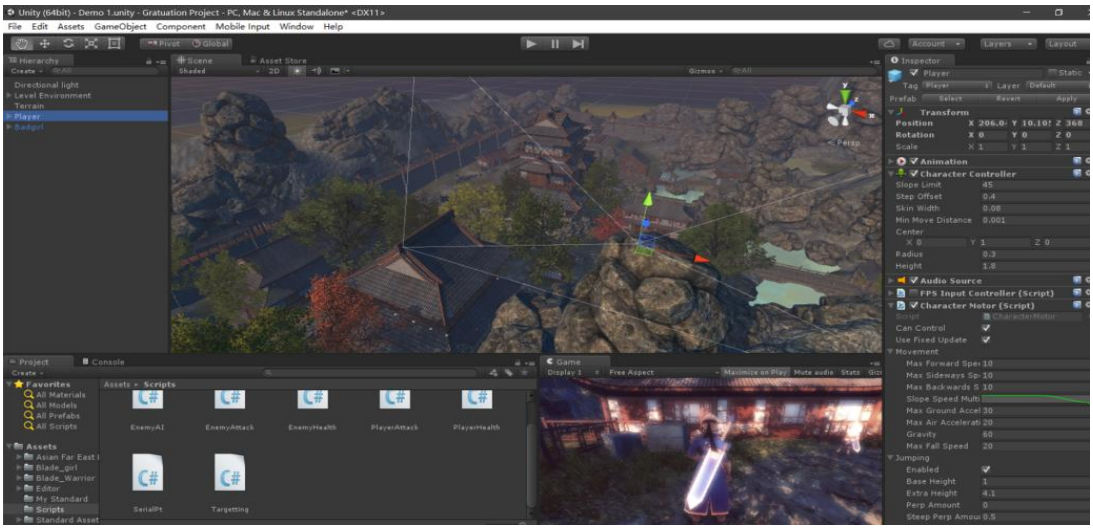

图 5-11 游戏场景搭建

人物搭建如图 5-12 所示：

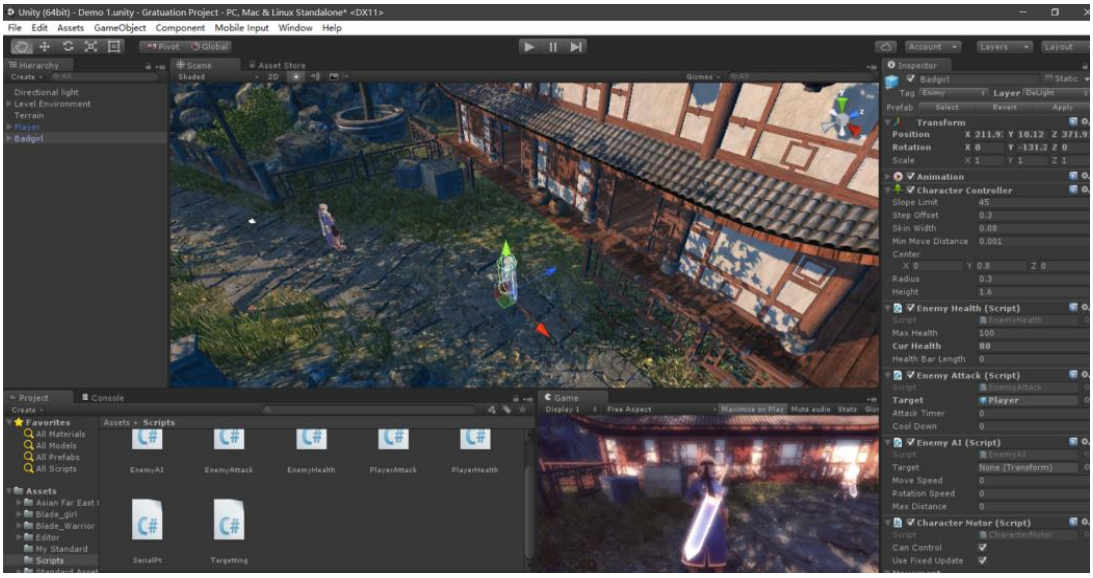
Unity (64bit) - Demo 1.unity - Gratuation Project - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Mobile Input Window Help
Hierarchy
Directional light
Level Environment
Terrain
Player
Badgirl
Scene
Asset Store
Inspector
Transform
Animation
Character Controller
Enemy Health (Script)
Enemy Attack (Script)
Enemy AI (Script)
Character Motor (Script)
Project
Console
Game
Favorites
Assets
Scripts
EnemyAI
EnemyAttack
EnemyHealth
PlayerAttack
PlayerHealth
SerialPt
Targetting

图 5-12 人物搭建

6 设计调试

6.1 体感手柄调试

如图 6-1 所示，该图为体感手柄的 PCB 制板的正面，图 6-2 是 PCB 制板背面的敷铜电路。

在调试过程中，因 MC7805CT 芯片在上电后会因发热而产生过热保护，使供给 STC89C52RC 单片机、MPU6050 模块和蓝牙模块的 5V 电压快速下降，造成手柄停止工作。因此在 MC7805CT 芯片处加上一块散热铝片，防止其产生过热保护。

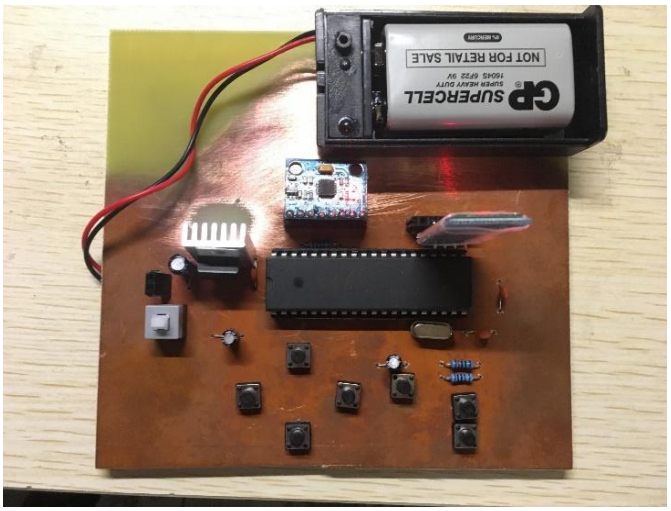

图 6-1 无线体感手柄的 PCB 制板的正面

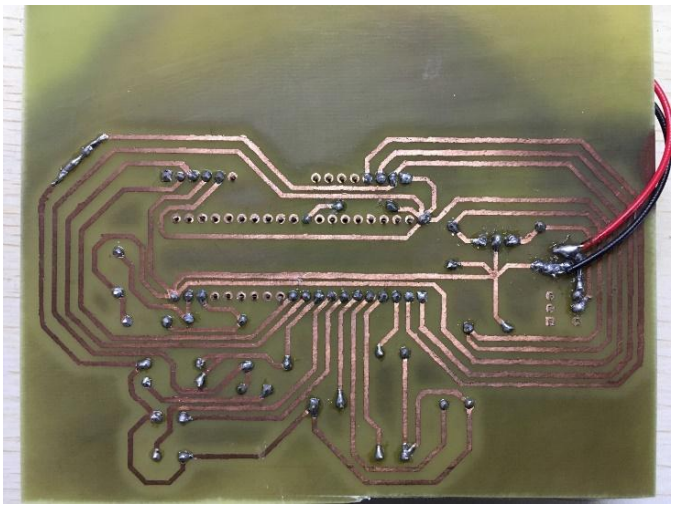

图 6-2 无线体感手柄的 PCB 制板的背面

如图 6-3 所示，该图为体感手柄通过蓝牙模块每一次发送数据给 Unity3D 游戏的信号波形图。从图中可看出，蓝牙模块每发送一次数据的时间为 15.8 毫秒。

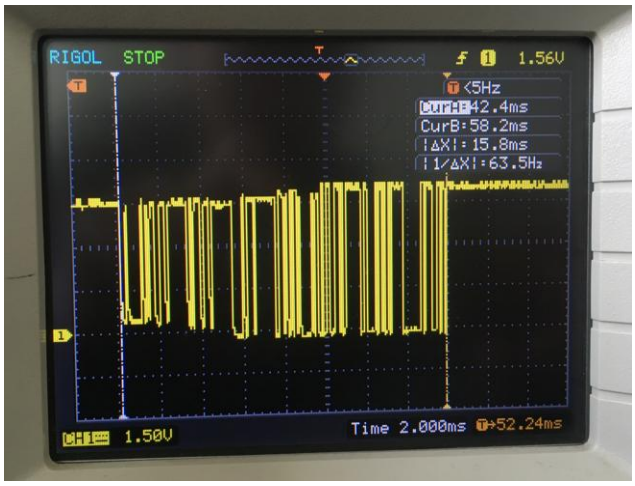

图 6-3 数据发送信号波形图

如图 6-4 所示，该图中的串口调试软件显示的是从游戏手柄中的 MPU6050 模块经过蓝牙模块传入的 16 个十六进制的数据。02 为数据的标头，67 和 FF 为标尾，FF 这个标尾是在下位机的 main 函数中加上 send(0xff)这一句代码，能够更快速地判断所需接收的数据。第 2 个到第 13 个的十六进制数为 MPU6050 模块发送的 12 个三轴加速度和三轴角速度的原始数据。第 14 个的 00 则为按键未输入状态下的十六进制数据。

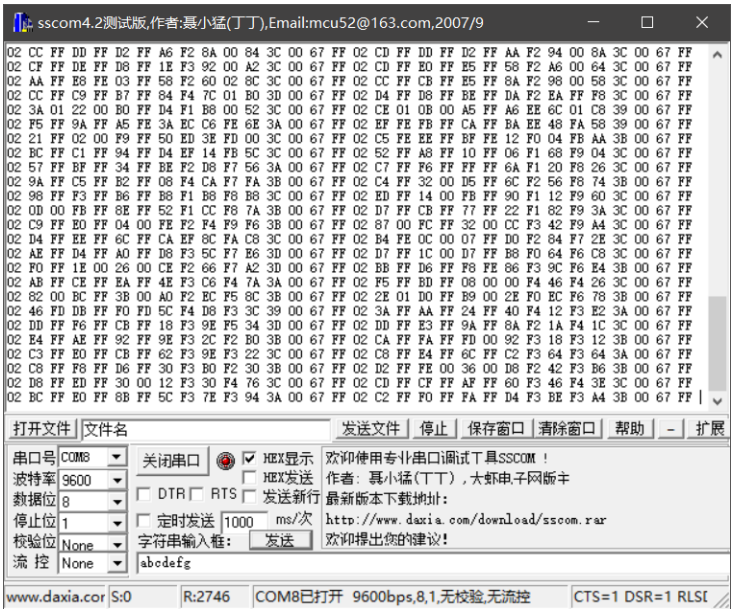

图 6-4 串口数据接收图

6.2 Unity3D 游戏调试

Unity3D 游戏调试界面如图 6-5 所示。该界面为 Unity3D 游戏引擎的场景界面，包括了 Scene、Project、Game、Inspector、Hierarchy、Toolbar 工具栏等界面。

Scene 界面是搭建游戏场景及人物的界面。Project 界面中的 Scripts 是本设计中所编写的 C#代码脚本，包括了串口线程接收数据、互补滤波的姿态解算、敌人 AI、攻击、生命和血槽等脚本。Game 界面是实时显示玩家控制角色的视角以及所搭建的游戏场景。Inspector 界面是显示当前选中的游戏对象的所有组件脚本以及其属性的相关信息。Hierarchy 界面包含了当前场景的所有游戏对象，本设计中拥有玩家、敌人、光源、地面、建筑、植物等游戏场景对象。Toolbar 工具栏分别为播放、暂停和步进，按下播放按钮可进入游戏界面。

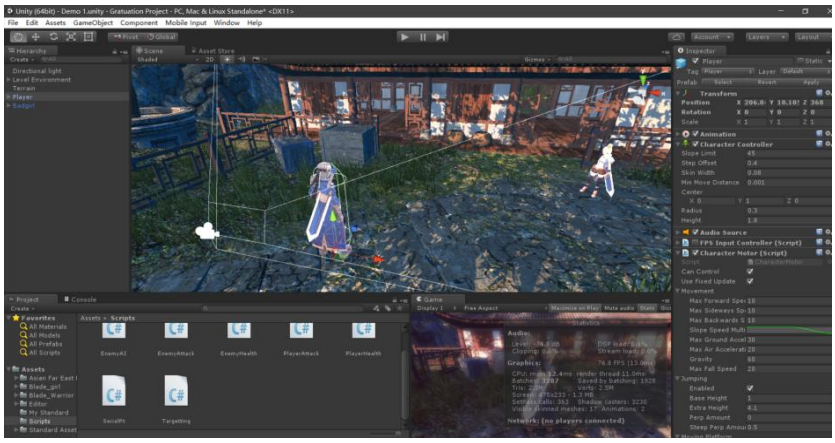

图 6-5 Unity3D 游戏调试界面

游戏界面如图 6-6 所示：

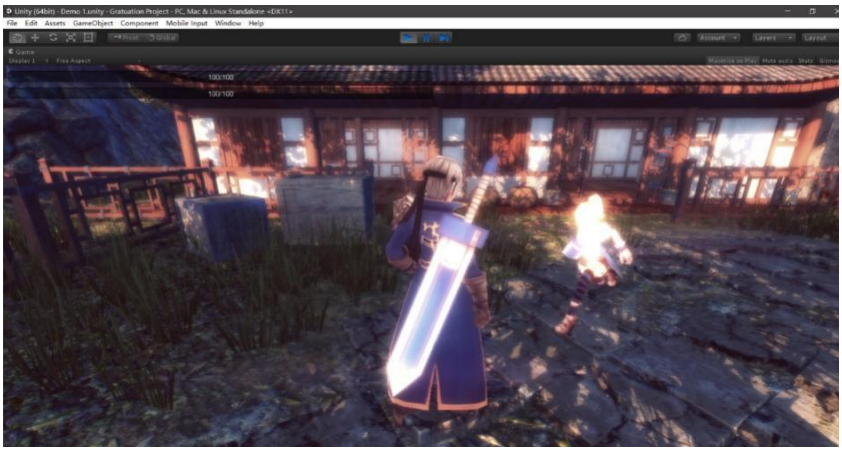

图 6-6 游戏界面

6.3 整体设计调试

体感游戏手柄通过蓝牙控制 Unity3D 游戏中的角色，做出站立、移动、攻击、跳跃等动作。角色的站立如图 6-7 所示：

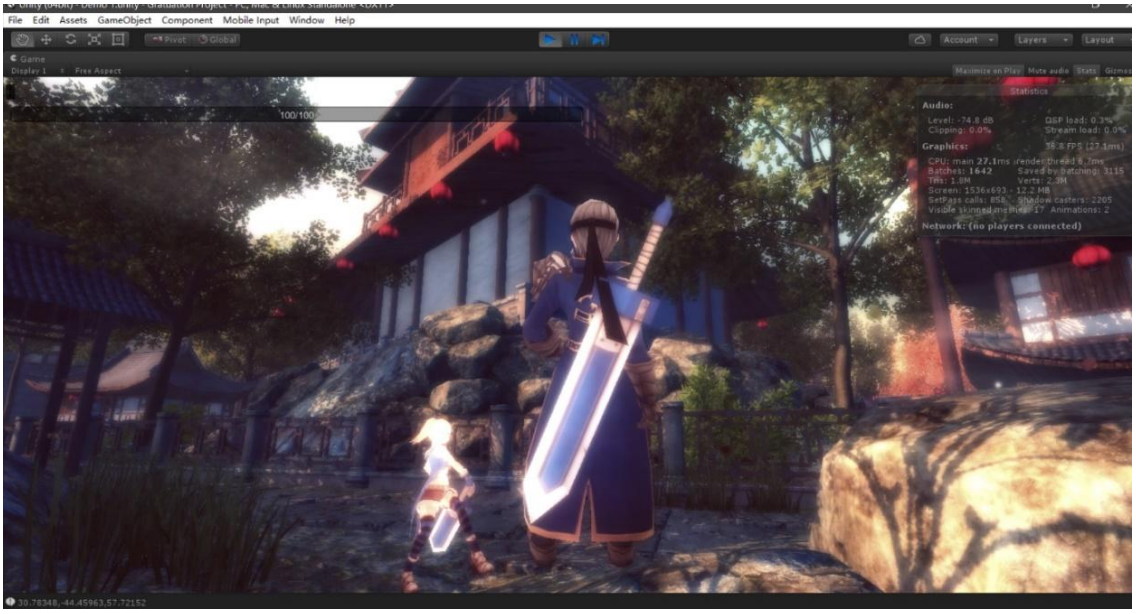

图 6-7 角色的站立图

角色的移动如图 6-8 所示：

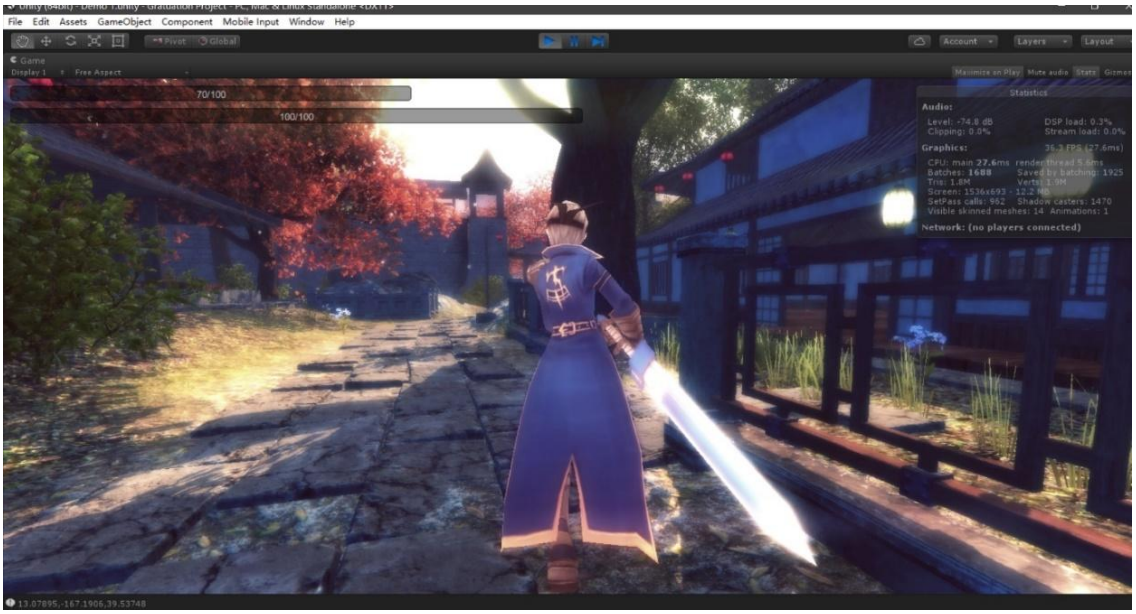

图 6-8 角色的移动图

角色的攻击如图 6-9 所示：

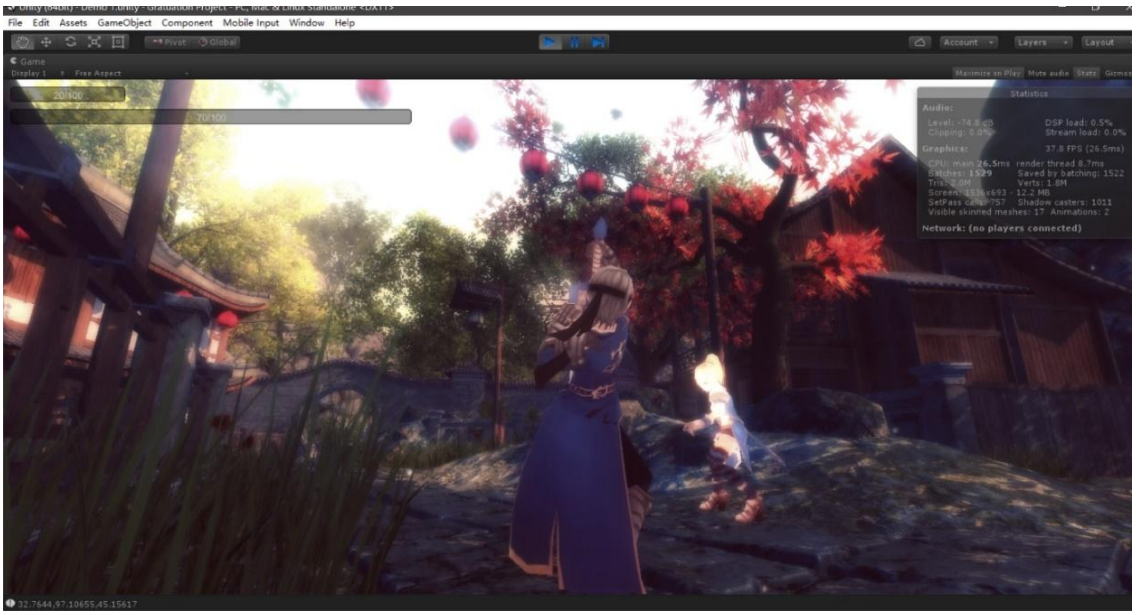

图 6-9 角色的攻击图

角色的跳跃如图 6-10 所示：

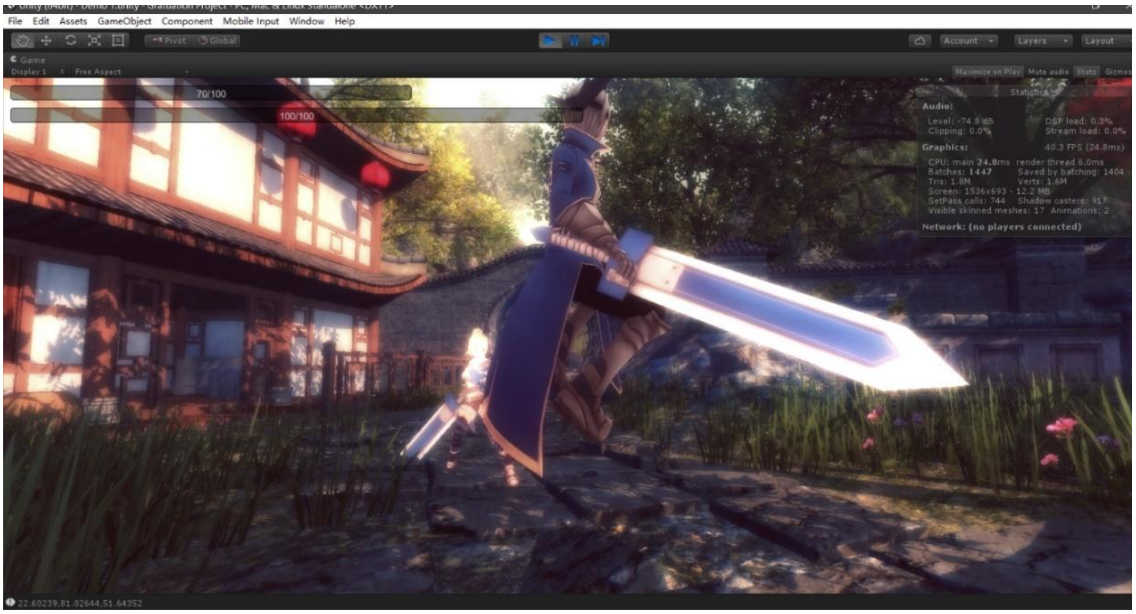
File Edit Assets GameObject Component Mobile Input Window Help
Pivot Global
Account Layers Layout
Game
Display 1 Free Aspect
Maximize on Play Mute audio Stats Gizmos
70/100
100/100
Audio:
Level: -74.8 dB
DSP load: 0.3%
Clipping: 0.0%
Stream load: 0.0%
Graphics:
40.3 FPS (24.8ms)
CPU: main 24.8ms render thread 6.0ms
Batches: 1447
Saved by batching: 1404
Tris: 1.8M
Verts: 1.6M
Screen: 1536x693 - 12.2 MB
SetPass calls: 744
Shadow casters: 917
Visible skinned meshes: 17
Animations: 2
Network: (no players connected)

图 6-10 角色的跳跃图

结论

本设计主要是以 STC89C52RC 单片机作为控制的核心，通过 MC7805CT 三端稳压电路、MPU6050 模块、BLK-MD-HC-06 蓝牙模块以及按键等电路来实现体感游戏手柄的功能。STC89C52RC 单片机通过接收 MPU6050 模块传入的三轴加速度和三轴角速度的原始数据，再将按键数据和原始数据经蓝牙模块发送给 Unity3D 游戏。

基于 Unity3D 游戏引擎搭建的游戏场景中，本设计利用脚本内编写的 Serial Port 类以及线程接收手柄传入的 12 个 8 位 AD 值数据，再通过误差计算和互补滤波算法获得欧拉角，赋值给游戏场景中人物对象的 Rotation 这三个分量以控制角色 Camera 的转向。单片机的 P1 口获取 6 个按键，分别控制人物的前进、后退、左移、右移、攻击及跳跃等功能，并通过角色脚本中的 Animation 代码和动画实现动作化。敌人拥有自动寻找玩家并朝玩家移动和自动攻击等功能。游戏中同时带有刚体碰撞、血槽 UI 等。

本设计中的难点主要在于如何通过 Unity3D 游戏引擎获取体感手柄的数据和对互补滤波算法的理解及应用。虽然完成了所有预设功能，但本设计还存在些许不足，如只是简单实现游戏角色的功能，未开发出一个具有剧情逻辑的完整游戏等。未来，本设计经过进一步优化后，将在 Unity3D 游戏市场中更好地为虚拟现实游戏玩家服务。

参考文献

[1] 陈浩磊,邹湘军,陈燕,等.虚拟现实技术的最新发展与展望[J].中国科技论文,2011,06(1):1-5.

[2] 陈洪,任科.游戏专业概论[M].北京:兵器工业出版社,2010:12-19.

[3] Christer Erison.Real-Time Collision Detection[M].Boca Raton:CRC Press,2004:7-11.

[4] 黄辉,陆利忠,闫镔,等.三维可视化技术研究[J].信息工程大学学报,2010,11(2):218-222.

[5] 汤剑鸣.游戏手柄也在升级[J].电子世界,2001,(1):28-29.

[6] 赵明.基于 DirectX 的三维游戏特效技术的研究与实现[D].哈尔滨:哈尔滨工程大学信息与通讯工程学院,2012.

[7] Unity Technologies.Unity5.X 从入门到精通[M].北京:中国铁道出版社,2016:367-481.

[8] 宣雨松.Unity3D 游戏开发[M].北京:人民邮电出版社,2012:165-196.

[9] 陈海宴.51 单片机原理及应用[M].北京:航空航天大学出版社,2010:3-25.

[10] 王胜超.两轮体感车关键技术研究与实现[D].荆州:长江大学电子信息学院,2016.

[11] 林伟财.四旋翼飞行器控制器的设计及实现[D].江西:南昌大学信息工程学院,2014.

[12] 徐爱钧,徐阳.Keil C51 单片机高级语言应用编程与实践[M].北京:电子工业出社,2013:8-13.

[13] Karli Watson,Christian Nagel,齐立波,等.C#入门经典(第 5 版)[M].北京:清华大学出版社,2010:17-203.

[14] Ramey L,Panter R.Collaborative Storytelling in Unity3D[M].New York:Springer International Publishing,2015:27-31.

附录一

程序源代码

下位机代码：

```
#include<reg52.h>
#include<intrins.h>
#include<stdio.h>                    //Keil library
#include<math.h>                     //Keil library
#define  SMPLRT_DIV                  0x19     //陀螺仪采样率，典型值：0x07(125Hz)
#define  CONFIG                      0x1A     //低通滤波频率，典型值：0x06(5Hz)
#define  GYRO_CONFIG                 0x1B     //陀螺仪自检及测量范围
#define  ACCEL_CONFIG                0x1C   //加速计自检、测量范围及高通滤波频率，
典型值：0x01(不自检，2G，5Hz)
#define ACCEL_XOUT_H       0x3B
#define ACCEL_XOUT_L       0x3C
#define ACCEL_YOUT_H       0x3D
#define ACCEL_YOUT_L       0x3E
#define ACCEL_ZOUT_H       0x3F
#define ACCEL_ZOUT_L       0x40
#define GYRO_XOUT_H        0x43
#define GYRO_XOUT_L        0x44
#define GYRO_YOUT_H        0x45
#define GYRO_YOUT_L        0x46
#define GYRO_ZOUT_H        0x47
#define GYRO_ZOUT_L        0x48
#define PWR_MGMT_1         0x6B       //电源管理，典型值：0x00(正常启用)
#define SlaveAddress       0xD0       //IIC 写入时的地址字节数据，+1 为读取
#define FOSC 11059200L //晶振设置，默认使用 11.0592M Hz
#define BAUD 9600        //波特率

sbit SCL=P1^0;                //IIC 时钟引脚定义
sbit SDA=P1^1;                //IIC 数据引脚定义
//sbit keyW=P1^2;
```

```
//sbit keyS=P1^3;
//sbit keyA=P1^4;
//sbit keyD=P1^5;
//sbit keyF=P1^6;
//sbit keyBlank=P1^7;
unsigned char key=0x00;//moveRate[15],按键判断
unsigned char moveRate[]={0xaa,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff};
//unsigned char flag=0x00;
bit I2C_RecvACK();  //I2C 接收应答信号
/*******************************
延迟一段时间，毫秒
*******************************/
void delay(unsigned int period)
{
    unsigned int i,j;
    for(i=period;i>0;i--)
    for(j=110;j>0;j--);
}
/*******************************
延迟 5ms
*******************************/
void Delay5us()
{
    _nop_();_nop_();_nop_();_nop_();
    _nop_();_nop_();_nop_();_nop_();
    _nop_();_nop_();_nop_();_nop_();
    _nop_();_nop_();_nop_();_nop_();
    _nop_();_nop_();_nop_();_nop_();
    _nop_();_nop_();_nop_();_nop_();
}
void UsartConfiguration()
{
```

```
    SCON = 0X50;   //设置为工作方式 1  10 位异步收发器
    TMOD |= 0x20; //设置计数器工作方式 2    8 位自动重装计数器
    PCON = 0X80;//波特率加倍  SMOD = 1
    TH1 = 256 -(FOSC/12/32/(BAUD/2)); //计算溢出率
    TL1 = 256 -(FOSC/12/32/(BAUD/2));
    TR1 = 1; //打开定时器
    ES=1;//打开串口
    EA = 1;//打开总中断
}
/*******************************
从 I2C 总线接收一个字节数据
*******************************/
unsigned char I2C_RecvByte()
{
    unsigned char i;
    unsigned dat=0;
    SDA = 1;                        //使能内部上拉,准备读取数据,
    for (i=0; i<8; i++)             //8 位计数器
    {
        dat <<= 1;
        SCL = 1;                    //拉高时钟线
        Delay5us();                 //延时
        dat |= SDA;                 //读数据
        SCL = 0;                    //拉低时钟线
        Delay5us();                 //延时
    }
    return dat;
}
/*******************************
I2C 起始信号
*******************************/
void I2C_Start()
{
    SDA = 1;                        //拉高数据线
```

```
    SCL = 1;                    //拉高时钟线
    Delay5us();                 //延时
    SDA = 0;                    //产生下降沿
    Delay5us();                 //延时
    SCL = 0;                    //拉低时钟线
}
/*****************************
I2C 停止信号
*****************************/
void I2C_Stop()
{
    SDA = 0;                    //拉低数据线
    SCL = 1;                    //拉高时钟线
    Delay5us();                 //延时
    SDA = 1;                    //产生上升沿
    Delay5us();                 //延时
}
/*****************************
I2C 发送应答信号，入口参数:ack (0:ACK 1:NAK)
*****************************/
void I2C_SendACK(bit ack)
{
    SDA=ack;
    SCL=1;
    Delay5us();
    SCL=0;
    Delay5us();
}
/*****************************
向 I2C 总线发送一个字节数据
*****************************/
void I2C_SendByte(unsigned char dat)
{
    unsigned char i;
```

```
    for (i=0; i<8; i++)          //8 位计数器
    {
        dat <<= 1;               //移出数据的最高位
        SDA = CY;                  //送数据口
        SCL = 1;                   //拉高时钟线
        Delay5us();                //延时
        SCL = 0;                   //拉低时钟线
        Delay5us();                //延时
    }
    I2C_RecvACK();
}
/*******************************
从 I2C 读取一个字节数据
*******************************/
unsigned char Single_ReadI2C(unsigned char REG_Address)
{
    unsigned char REG_data;
    I2C_Start();                        //起始信号
      I2C_SendByte(SlaveAddress);       //发送设备地址+写信号
      I2C_SendByte(REG_Address);        //发送存储单元地址，从 0 开始
      I2C_Start();                      //起始信号
      I2C_SendByte(SlaveAddress+1);     //发送设备地址+读信号
      REG_data=I2C_RecvByte();          //读出寄存器数据
      I2C_SendACK(1);                   //接收应答信号
      I2C_Stop();                       //停止信号
    return REG_data;
}
/*******************************
I2C 接收应答信号
*******************************/
bit I2C_RecvACK()
{
    SCL = 1;                        //拉高时钟线
    Delay5us();                     //延时
```

```
    CY = SDA;                   //读应答信号
    SCL = 0;                    //拉低时钟线
    Delay5us();                 //延时
    return CY;
}
/**********************************
向 I2C 设备写入一个字节数据
**********************************/
void Single_WriteI2C(unsigned char REG_Address,unsigned char REG_data)
{
    I2C_Start();                //起始信号
    I2C_SendByte(SlaveAddress);   //发送设备地址+写信号
    I2C_SendByte(REG_Address);    //内部寄存器地址，
    I2C_SendByte(REG_data);       //内部寄存器数据，
    I2C_Stop();                 //发送停止信号
}
/**********************************
初始化 MPU6050
**********************************/
void InitMPU6050()
{
    Single_WriteI2C(PWR_MGMT_1, 0x00);    //解除休眠状态
    Single_WriteI2C(SMPLRT_DIV, 0x07);
    Single_WriteI2C(CONFIG, 0x06);
    Single_WriteI2C(GYRO_CONFIG, 0x18);
    Single_WriteI2C(ACCEL_CONFIG, 0x01);
}
/**********************************
串口传送一个字符
**********************************/
void send(unsigned char send)
{
        SBUF = send;
        //delay(60);
```

```
        while(!TI);     //等特数据传送
        TI = 0;          //清除数据传送标志
}
/*******************************
获取 mpu6050 的角速度和加速度，[0]为标头，[1-12]为角速度和加速度的低高 8 位，[13]为 8 位按键判断
*******************************/
    void formString()
{
        moveRate[1]=Single_ReadI2C(GYRO_YOUT_L);
        moveRate[2]=Single_ReadI2C(GYRO_YOUT_H);
        moveRate[3]=Single_ReadI2C(GYRO_ZOUT_L);
        moveRate[4]=Single_ReadI2C(GYRO_ZOUT_H);
        moveRate[5]=Single_ReadI2C(GYRO_XOUT_L);
        moveRate[6]=Single_ReadI2C(GYRO_XOUT_H);
        moveRate[7]=Single_ReadI2C(ACCEL_XOUT_L);   //X 轴低 8 位
        moveRate[8]=Single_ReadI2C(ACCEL_XOUT_H);   //X 轴高 8 位
        moveRate[9]=Single_ReadI2C(ACCEL_YOUT_L);
        moveRate[10]=Single_ReadI2C(ACCEL_YOUT_H);
        moveRate[11]=Single_ReadI2C(ACCEL_ZOUT_L);
        moveRate[12]=Single_ReadI2C(ACCEL_ZOUT_H);
        moveRate[13]=key;

}
/*******************************
发送一段 moveRate 数组的字符串，共 32bit，一比特接一比特发送
*******************************/
    void sendString()
{
        unsigned int i;
        for(i=0;i<15;i++)
        {
            send(moveRate[i]);
        }
```

```
}
/*************************
判断有无按键按下
*****************************/
      void keyscan()
{
          switch(P1)
              {
                  case 251:key=(key|0x20);break;
                  case 247:key=(key|0x10);break;
                  case 239:key=(key|0x08);break;
                  case 223:key=(key|0x04);break;
                  case 191:key=(key|0x02);break;
                  case 127:key=(key|0x01);break;
                  default:key=(key|0x00);
              }
}
void main()
{

          UsartConfiguration();
          InitMPU6050();
          delay(50);
      while(1){

            keyscan();
            formString();
            sendString();
            send(0xff);
            key=0;
            delay(50);

      }
}
```

上位机代码：

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System;
using System.Threading;
using System.Collections.Generic;
using System.ComponentModel;
using System.IO.Ports;
using System.Text.RegularExpressions;
using System.Text;

public class SerialPt : MonoBehaviour
{
    //public PlayerAttack attack;
    public class MyKeyCode
    {
        public const ushort keyW = 0x20;
        public const ushort keyS = 0x10;
        public const ushort keyA = 0x08;
        public const ushort keyD = 0x04;
        public const ushort keyF = 0x02;
        public const ushort keyBlank = 0x01;
    }

    /// <summary>
    ///  串口线程定义
    /// </summary>
    List<byte> listStr;//在 ListByte 中读取数据，用于做数据处理
    List<byte> ListByte;//存放读取的串口数据
    private Thread tPort;//读取串口数据的线程
    private Thread tPortDeal;//处理数据的线程
    bool isStartThread;//控制 FixedUpdate 里面的两个线程是否调用（当准备调用串口的 Close 方法时设置为 false）
```

```
SerialPort spStart;//定义串口

public ushort keyResult;//接收的第 14 个 8 位 AD 值，作为按键判断
private bool flag = false;//判断按键是否按下
private int Count = 0;//攻击动画判断

/// <summary>
///  测试参数
/// </summary>
public long test = 0;
private int tickCount;
public int spanTime;

[HideInInspector]//Inspector 面板中隐藏
public float yaw, pitch, roll, accX, accY, accZ;//原始六轴数据
public float angleRoll,angleYaw, anglePitch;//存放欧拉角

/// <summary>
///  互补滤波参数
/// </summary>
private float tau = 0.065f, p = 0.0f;              //  用于计算比例
private float dtc = 0.01f;                          //  抽样时间 10ms
public static float bias_gx = 0.0f, bias_gy = 0.0f, bias_gz = 0.0f, gy_temp; //  零点漂移误差消除

private Rigidbody rigidBody;//刚体实例化
private new GameObject gameObject;//实例化游戏对象
void Start()
{

    Debug.Log("start");
    isStartThread = true;
    listStr = new List<byte>();//从 ListByte 中读取数据，用于做数据处理队列
```

```
        ListByte = new List<byte>();//存放读取的串口数据
        isStartThread = true;
        spStart = new SerialPort("COM8", 9600, Parity.None, 8, StopBits.One);//串口实例化

        gameObject = GameObject.Find("Player");

        try
        {
            spStart.Open();//打开串口
        }
        catch (Exception e)
        {
            Debug.Log(e.ToString());
        }
        tPort = new Thread(DealData);//该线程处理数据
        tPort.Start();
        tPortDeal = new Thread(ReceiveData);//该线程接收数据
        tPortDeal.Start();
    }
    void Update()
    {
      //一开始按键未按下
        if (flag)
        {
            flag = false;
        }

        KeyMove();//按键控制移动
        Debug.Log(angleRoll + ","+ angleYaw + "," + anglePitch);
        transform.eulerAngles = new Vector3(-angleRoll, angleYaw, 0);//游戏对象的欧拉角赋值

    }
```

```
    void FixedUpdate()
    {
        if (isStartThread)
        {
            if (!tPortDeal.IsAlive)
            {
                tPortDeal = new Thread(ReceiveData);
                tPortDeal.Start();
            }
            if (!tPort.IsAlive)
            {
                tPort = new Thread(DealData);
                tPort.Start();

            }
        }

    }
#region  接收数据和处理数据线程
    private void ReceiveData()
    {
        try
        {
            Byte[] buf = new Byte[1];

            //线程启动并且串口开启
            while (isStartThread && spStart.IsOpen)
            {
                if (spStart.IsOpen)
                {
                    //为了获取最新的角度
                    spStart.Read(buf, 0, 1);
                    tickCount++;
                }
```

```
                if (buf.Length == 0)
                {
                    return;
                }
                if (buf != null)
                {
                    for (int i = 0; i < 1; i++)
                    {
                        ListByte.Add(buf[i]);//向队列中添加数据
                    }
                }
                test++;
            }
        }
        catch (Exception e)
        {
            Debug.Log(e.ToString());
            spanTime = 0;
        }
    }
    private void DealData()
    {
        while (ListByte.Count > 0)
        {
            listStr.Add(ListByte[0]);
            ListByte.Remove(ListByte[0]);
            if (!(listStr[0] == 0x02))//如果帧头错误，清缓存
            {
                listStr.Clear();
            }
            if (listStr.Count == 16)//串口发过来的数据的位数
            {
                //Debug.Log("|" + liststr[14] + "|");
                if (listStr[14] == 0x67 && listStr[15] == 0xFF)//判断帧尾，正确
```

则解析姿态角

```
                    {
                        yaw = (short)(listStr[1] | (listStr[2] << 8));//Y 轴
                        pitch = (short)(listStr[3] | (listStr[4] << 8));//Z 轴
                        roll = (short)(listStr[5] | (listStr[6] << 8));//X 轴
                        accX = (short)(listStr[7] | (listStr[8] << 8));
                        accY = (short)(listStr[9] | (listStr[10] << 8));
                        accZ = (short)(listStr[11] | (listStr[12] << 8));
                        keyResult = (ushort)listStr[13];
                        CorrectResult(roll,yaw,pitch,accX,accY,accZ);       //获取最
终正确结果

                        flag = true;

                    }
                    listStr.Clear();
                }
            }
        }
#endregion

#region  误差计算及互补滤波
    public void CorrectResult(float gx,float gy,float gz,float ax,float ay,float az)
        {
            float Accel_x = ax;       //读取 X 轴加速度
            float Angle_ax = (Accel_x - 1100) / 16384;     //去除零点偏移,计算
得到角度（弧度）
            Angle_ax = (float)(Angle_ax * 1.2 * 180 / 3.14);        //弧度转换为
度,
            float Gyro_x = gx;                //静止时角速度 X 轴输出
            Gyro_x = (float)(-(Gyro_x + 30) / 16.4);              //去除零点偏移，
计算角速度值,负号为方向处理
            float Accel_y = ay;       //读取 Y 轴加速度
            float Angle_ay = (Accel_y - 1100) / 16384;     //去除零点偏移,计算
```

```
得到角度（弧度）
            Angle_ay = (float)(Angle_ay * 1.2 * 180 / 3.14)  ;       //弧度转换为度,
            float Gyro_y = gy;              //静止时角速度 Y 轴输出
            Gyro_y = (float)(-(Gyro_y + 30) / 16.4);            //去除零点偏移，计算角速度值,负号为方向处理

            float Accel_z = az;      //读取 Z 轴加速度
            float Angle_az = (Accel_z - 1100) / 16384;    //去除零点偏移,计算得到角度（弧度）
            Angle_az = (float)(Angle_az * 1.2 * 180 / 3.14);       //弧度转换为度,
            float Gyro_z = gz;              //静止时角速度 Z 轴输出
            Gyro_z = (float)(-(Gyro_z + 30) / 16.4);            //去除零点偏移，计算角速度值,负号为方向处理

            ComplementFilterX(Angle_ax, Gyro_x);

            ComplementFilterY(Angle_ay, Gyro_y);

            ComplementFilterZ(Angle_az, Gyro_z);
        /*

            const float Kp = 50f;
            const float Ki = 0.1f;

            const float halfT = 0.001f;

            float q0 = 1.0f, q1 = 0.0f, q2 = 0.0f, q3 = 0.0f;              // 四元数的元素，代表估计方向
            float exInt = 0, eyInt = 0, ezInt = 0;            // 按比例缩小积分误差
            float norm;
            float vx, vy, vz;          // v-陀螺仪积分后推算出的重力向量
            float ex, ey, ez;          // e-两重力向量的叉积（向量间的误差 a 与 v 之
```

间）

```
            norm = (float)Math.Sqrt(ax * ax + ay * ay + az * az);// 测量正常化
            ax = ax / norm;                         //单位化
            ay = ay / norm;
            az = az / norm;
            // 估计方向的重力
            vx = 2 * (q1 * q3 - q0 * q2);
            vy = 2 * (q0 * q1 + q2 * q3);
            vz = q0 * q0 - q1 * q1 - q2 * q2 + q3 * q3;
            // 错误的领域和方向传感器测量参考方向之间的交叉乘积的总和
            ex = (ay * vz - az * vy);
            ey = (az * vx - ax * vz);
            ez = (ax * vy - ay * vx);
            // 积分误差比例积分增益
            exInt = exInt + ex * Ki;
            eyInt = eyInt + ey * Ki;
            ezInt = ezInt + ez * Ki;
            // 调整后的陀螺仪测量
            gx = gx + Kp * ex + exInt;
            gy = gy + Kp * ey + eyInt;
gz = gz + Kp * ez + ezInt;
            // 整合四元数率和正常化
            q0 = q0 + (-q1 * gx - q2 * gy - q3 * gz) * halfT;
            q1 = q1 + (q0 * gx + q2 * gz - q3 * gy) * halfT;
            q2 = q2 + (q0 * gy - q1 * gz + q3 * gx) * halfT;
            q3 = q3 + (q0 * gz + q1 * gy - q2 * gx) * halfT;
            // 正常化四元
            norm = (float)Math.Sqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
            q0 = q0 / norm;
            q1 = q1 / norm;
            q2 = q2 / norm;
            q3 = q3 / norm;
            angleRoll = (float)(Math.Atan2(2 * q2 * q3 + 2 * q0 * q1, -2 * q1 * q1 - 2 *
q2 * q2 + 1) * 57.3); // roll
```

```
        angleYaw = (float)(Math.Atan2(2 * q1 * q2 + 2 * q0 * q3, -2 * q2 * q2 - 2 *
q3 * q3 + 1) * 57.3); // yaw
anglePitch = (float)(Math.Asin(-2 * q1 * q3 + 2 * q0 * q2) * 57.3); // pitch
        */
    }

    public float ComplementFilterX(float angle, float gy_sped)
        {
            bias_gx = (float)(bias_gx * 0.998 + gy_sped * 0.002);   // 消除零点
误差
            gy_temp = gy_sped - bias_gx;                    // 减去零点误差
            p = tau / (tau + dtc);

            // 对角速度 gy_temp 进行高通滤波,对重力感应得到的角度 angle
进行低通滤波
            angleRoll = p * (angleRoll + gy_temp * dtc) + (1 - p) * angle *
(float)1.5;
            return angleRoll;
        }
    public float ComplementFilterY(float angle, float gy_sped)
        {
            bias_gy = (float)(bias_gy * 0.998 + gy_sped * 0.002);   // 消除零点
误差
            gy_temp = gy_sped - bias_gy;                    // 减去零点误差
            p = tau / (tau + dtc);
            // 对角速度 gy_temp 进行高通滤波,对重力感应得到的角度 angle
进行低通滤波
            angleYaw = p * (angleYaw + gy_temp * dtc) + (1 - p) * angle
*(float)3;
            return angleYaw;
        }
    public float ComplementFilterZ(float angle, float gy_sped)
        {
            bias_gz = (float)(bias_gz * 0.998 + gy_sped * 0.002);   // 消除零点误
```

```
差
                gy_temp = gy_sped - bias_gz;                    // 减去零点误差
                p = tau / (tau + dtc);
                // 对角速度 gy_temp 进行高通滤波,对重力感应得到的角度 angle 进行低通滤波
                anglePitch = p * (anglePitch + gy_temp * dtc) + (1 - p) * angle;
                return anglePitch;
            }
        #endregion

        /// <summary>
        /// 按键判断
        /// </summary>
        /// <param name="keyCode"></param>
        /// <returns></returns>
        public bool HaveKeyDown(ushort keyCode)
        {
            if ((keyResult & keyCode) != 0)
            {
                //keyResult &= (ushort)(~keyCode);
                return true;
            }
            return false;
        }

        /// <summary>
        /// 按键控制移动
        /// </summary>
        public void KeyMove()
        {
            if (HaveKeyDown(MyKeyCode.keyW) == true)
            {
                GetComponent<Animation>().Play("Run");
                transform.Translate(Vector3.forward * Time.deltaTime * 6);
```

```
            //GetComponent<Rigidbody>().transform.Translate(Vector3.forward *
Time.deltaTime * 5);
            //this.gameObject.transform.Translate(new Vector3(0 ,0 ,5 *
Time.deltaTime));//W
            //GetComponent<Rigidbody>().MovePosition(transform.position +
Vector3.forward * 5* Time.deltaTime);
            //GetComponent<Rigidbody>().MovePosition(transform.position +
new Vector3(0, 0, 5 * Time.deltaTime));
        }
        else if (HaveKeyDown(MyKeyCode.keyS) == true)
        {
            GetComponent<Animation>().Play("Walk");

            transform.Translate(Vector3.forward * Time.deltaTime * (-3));
            //GetComponent<Rigidbody>().MovePosition(transform.position +
Vector3.forward * (-3) * Time.deltaTime);
        }
        else if (HaveKeyDown(MyKeyCode.keyA) == true)
        {
            GetComponent<Animation>().Play("Walk");
            transform.Translate(new Vector3(-4 * Time.deltaTime, 0, 0));//A
        }
        else if (HaveKeyDown(MyKeyCode.keyD) == true)
        {
            GetComponent<Animation>().Play("Walk");
            transform.Translate(new Vector3(4 * Time.deltaTime, 0, 0));//D
        }
        else if (HaveKeyDown(MyKeyCode.keyF) == true)
        {
            //GetComponent<Animation>().Play("Attack01");
            StartCoroutine(AttackAnimation());//协程函数
        }
        else if (HaveKeyDown(MyKeyCode.keyBlank) == true)
        {
```

```
            GetComponent<Animation>().Play("Jump");
        }
        else
        {
            GetComponent<Animation>().Play("Idle");
        }
}

/// <summary>
///  主角 3 种攻击动画
/// </summary>
/// <returns></returns>
private IEnumerator AttackAnimation()
{
        if(Count == 0)
        {
            GetComponent<Animation>().Play("Attack01");
            yield return new WaitForSeconds(2);//延时 2 秒，同攻击冷却时间
            Count = 1;
        }
        else if(Count == 1)
        {
            GetComponent<Animation>().Play("Attack");
            yield return new WaitForSeconds(2);
            Count = 2;
        }
        else if(Count == 2)
        {
            GetComponent<Animation>().Play("Attack02");
            yield return new WaitForSeconds(2);
            Count = 0;
        }
}
private void SendSerialPortByte(byte[] data)
```

```
    {
        if (spStart.IsOpen)
        {
            //sp.WriteLine(data);
            spStart.Write(data, 0, data.Length);
        }
    }
    IEnumerator ClosePort()//该方法为关闭串口的方法，当程序退出或是离开该页面或是想停止串口时调用。
    {
        {
            isStartThread = false;//停止掉 FixedUpdate 里面的两个线程的调用
            yield return new WaitForSeconds(1);//等一秒钟，让两个线程确实停止之后在执行 Close 方法
            spStart.Close();
        }
    }
    void OnApplicationQuit()
    {
        spStart.DiscardInBuffer();
        StartCoroutine(ClosePort());
        isStartThread = false;//停止掉 FixedUpdate 里面的两个线程的调用
        spStart.Close();
    }

using UnityEngine;
using System.Collections;

public class PlayerAttack : MonoBehaviour {
    public GameObject target;//
    public float attackTimer;//攻击时间间隔
    public float coolDown;//冷却时间

   // Use this for initialization
```

```
    void Start () {
        attackTimer = 0;
        coolDown = 2.0f;
    }

    // Update is called once per frame
    void Update () {
        if (attackTimer > 0)
        {
            attackTimer -= Time.deltaTime;//间隔时间为其减去当前帧数所用的
时间
        }
        if (attackTimer < 0)
        {
            attackTimer = 0;
        }
        //按下按键F
       // if (Input.GetKeyUp(KeyCode.F))
        if(GetComponent<SerialPt>().HaveKeyDown(2)==true)
         {
            if (attackTimer == 0)
            {
                Attack();
                attackTimer = coolDown;//攻击时间间隔等于冷却时间
            }
        }
    }

     private void Attack()
     {
        float distance = Vector3.Distance(target.transform.position,
transform.position);//玩家和敌人直接距离

        Vector3 dir = (target.transform.position - transform.position).normalized;//
```

规范化后，向量保持同样的方法，但是长度变为1.0

float direction = Vector3.Dot(dir, transform.forward);//点乘：返回1个-1.0到1.0之间的一个值（返回进行Dot计算的两个向量之间的夹角的余弦值Cos）；完全相反返回-1.0，相同返回1.0

```
            if (distance < 2.5f && direction>0)
            {
                EnemyHealth eh = (EnemyHealth)target.GetComponent("EnemyHealth");
                eh.AddjustCurHealth(-10);
            }
        }
}
}

using UnityEngine;
using System.Collections;

public class EnemyAttack : MonoBehaviour {

    public GameObject target;
    public float attackTimer;
    public float coolDown;
    private int Count = 0;
    // Use this for initialization
    void Start () {
        attackTimer = 0;
        coolDown = 3.0f;
    }

    // Update is called once per frame
    void Update () {
        if (attackTimer > 0)
        {
            attackTimer -= Time.deltaTime;
```

```
        }
        if (attackTimer < 0)
        {
            attackTimer = 0;
        }

            if (attackTimer == 0)
            {
                if (Count == 0)
                 {
                    Attack();
                    GetComponent<Animation>().Play("Attack");
                    attackTimer = coolDown;
                    Count = 1;
                 }

                else if (Count == 1)
                {
                    Attack();
                    GetComponent<Animation>().Play("Attack00");
                    attackTimer = coolDown;
                    Count = 2;
                 }

                else if (Count == 2)
                 {
                    Attack();
                    GetComponent<Animation>().Play("Attack01");
                    attackTimer = coolDown;
                    Count = 0;
                  }
        }
}
```

```
    private void Attack()
    {
        float distance = Vector3.Distance(target.transform.position,
transform.position);
        Vector3 dir = (target.transform.position - transform.position).normalized;
        float direction = Vector3.Dot(dir, transform.forward);

        if (distance < 2.5f && direction>0)
        {
            PlayerHealth eh =
(PlayerHealth)target.GetComponent("PlayerHealth");
            eh.AddjustCurHealth(-10);
        }
    }
}

using UnityEngine;
using System.Collections;

public class PlayerHealth : MonoBehaviour {

    public int maxHealth = 100;
    public int curHealth = 100;
    public float healthBarLength;
    // Use this for initialization
    void Start () {
        healthBarLength = Screen.width / 2;
    }
    // Update is called once per frame
    void Update () {
        AddjustCurHealth(0);
    }
    void OnGUI()
    {
```

```
        GUI.Box(new Rect(10, 10, healthBarLength , 20), curHealth + "/" + maxHealth);
    }
    public void AddjustCurHealth(int adj)
    {
        curHealth += adj;
        if (curHealth < 0)
        {
            curHealth = 0;
        }
        if (curHealth > maxHealth)
        {
            curHealth = maxHealth;
        }
        if (maxHealth < 1)
        {
            maxHealth = 1;
        }

        healthBarLength = (Screen.width / 2) * (curHealth / (float)maxHealth);
    }
}
using UnityEngine;
using System.Collections;

public class EnemyHealth : MonoBehaviour {

    public int maxHealth = 100;//最大生命
    public int curHealth = 100;//当前生命

    public float healthBarLength; //生命槽长度

   // Use this for initialization
   void Start () {
```

```
        healthBarLength = Screen.width / 2;//生命槽长度为屏幕宽度的一半
    }
    // Update is called once per frame
    void Update () {
        AddjustCurHealth(0);//调整当前生命（治疗或收到伤害值）
    }
     void OnGUI()
     {
        GUI.Box(new Rect(10, 40, healthBarLength , 20), curHealth + "/" + maxHealth);//屏幕左上角显示矩形生命槽

     }

     /// <summary>
     ///  生命槽显示的调整
     /// </summary>
     /// <param name="adj">治疗或伤害参数</param>
     public void AddjustCurHealth(int adj) {

        curHealth += adj;

        if (curHealth < 0)
        {
            curHealth = 0;
        }
        if (curHealth > maxHealth)
        {
            curHealth = maxHealth;
        }
        if (maxHealth < 1)
        {
            maxHealth = 1;
        }
        healthBarLength = (Screen.width / 2) * (curHealth / (float)maxHealth);//随
```

```
当前生命大小改变长度
        }
}
using UnityEngine;
using System.Collections;

public class EnemyAI : MonoBehaviour {

        public Transform target;//定义目标
        public int moveSpeed;//移动速度
        public int rotationSpeed;//旋转速度
        public int maxDistance;
        private Transform myTransform;//当前形态
        void Awake()
        {
                myTransform = transform;//优化
        }
       // Use this for initialization
       void Start () {
                GameObject go = GameObject.FindGameObjectWithTag("Player");//摄像机跟随标签为Player的游戏对象
                target = go.transform;
                rotationSpeed = 2;
                maxDistance = 2;
       }
       // Update is called once per frame
       void Update () {
                Debug.DrawLine(target.position,myTransform.position,Color.blue);// 画 一条目标和对象直接蓝色的线

                //看向玩家，即敌人以一定速度转向玩家
                myTransform.rotation = Quaternion.Slerp(myTransform.rotation, Quaternion.LookRotation(target.position - myTransform.position),rotationSpeed*Time.deltaTime);
```

```
            if (Vector3.Distance(target.transform.position, myTransform.position) > 2)
            {
                GetComponent<Animation>().Play("Run");
                //向玩家移动
                myTransform.position += myTransform.forward * Time.deltaTime;
            }
        }
    }
```