机器人技术期末文档

成员：

2052844 曹晓慈

2054321 陈亮奇

1. 开发及模拟环境搭建

1.1 实验环境和平台

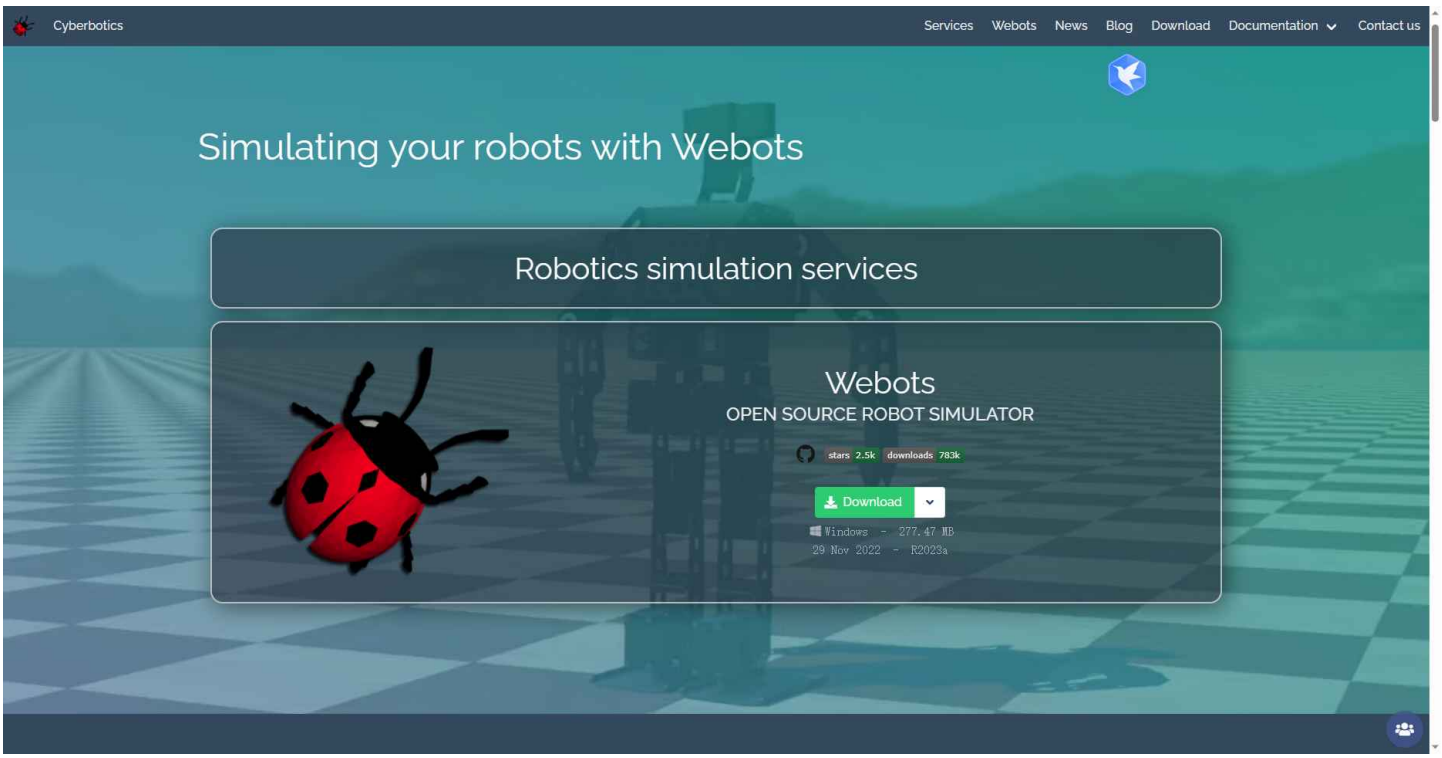
Win10+Webots

1.2 代码语言

C语言

1.3 模拟环境搭建

首先进入Webots官网，因为考虑到最新版的相关技术文档和教学资源不是很多，而且这之后的版本不再进行一些仿真材质的预装，当要使用对应材质时需要访问github进行下载但经常无法正常进行下载，因此我们选择了2020a的版本。



下载完成后进行安装即可正常打开，之后进入创建项目。

我们的项目是一个基于Webots开发的一个无人快递小车。目前我们的进展是基本掌握Webots平台多种工具的使用，重新明确项目需求，并完成了预期中重点功能的开发，基本完成了硬件的选型与设计。

2. 应用场景

本系统主要应用于同济大学嘉定校区递易快递站，原本应用思路为使用快递小车帮助快递站分发部分快递给教学楼，经实地考察以及与快递站工作人员交流之后，重新明确需求与开发迭代思路。

1.基于嘉定路况的不确定性，机器人的应用较为困难，且快递站不便分配人手管理机器人（如挑选适合派发的快递，以及机器人突发状况的回收问题等），因此对功能开发的重点做部分调整：着重开发快递站区域的功能。

2.考虑快递站区域，如果能有机器人协助工作人员进行快递装柜的工作，机器人会更加实用，且该功能的开放仍可以应用于后续迭代开发，如对嘉定路况有进一步的分析研究之后得出成熟的运营策略之后，可以在宿舍楼附近区域增设快递柜，将该功能应用于这些柜子。

综上所述，我们认为一个更加实用的校园快递小车需要实现的功能应该是：从快递站取出对应规格及相应数量的快递包裹之后，前往宿舍楼附近的快递柜，将包裹平稳置于快递柜中，随后再次扫描识别空柜，重新回到快递站取对应包裹，以此循环。

3. 基本方案

对于快递站放货小车，需要保证其在快递站范围内进行安全、精确的驾驶操作，并且具有高效的取货卸货功能。对于快递站的具体场景及特点，我们制定以下需求：

1.基于Webots提供的摄像头api，能够精准的识别物品，并将其传入货架数组中。

2.能够识别货架中对应位置所缺失的快递，并将根据货架左右快递盒的大小来判断所缺失快递盒的型号。

3.得到所需快递盒型号后能进行环绕寻找所需的快递盒。

4.对于不同的快递盒，机器人应采用不同的抓取策略对快递盒进行抓取，并确保快递盒不会掉落。

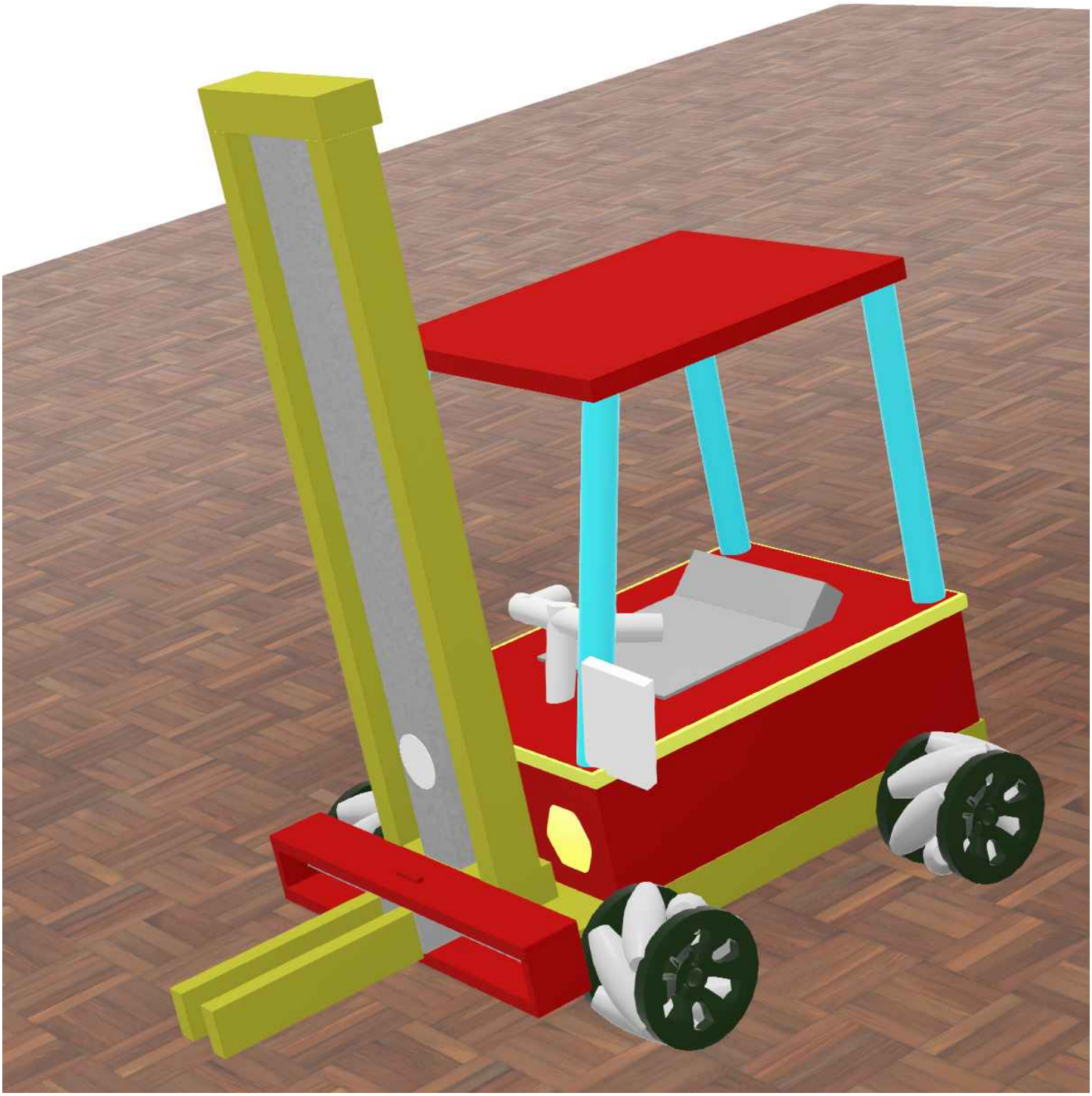
5.抓取快递盒后返回到原先缺失对应快递盒的货架前进行快递盒的装载。

6.不断重复上述流程直至所有快递柜均已装载满，以保证快递的快速流通。

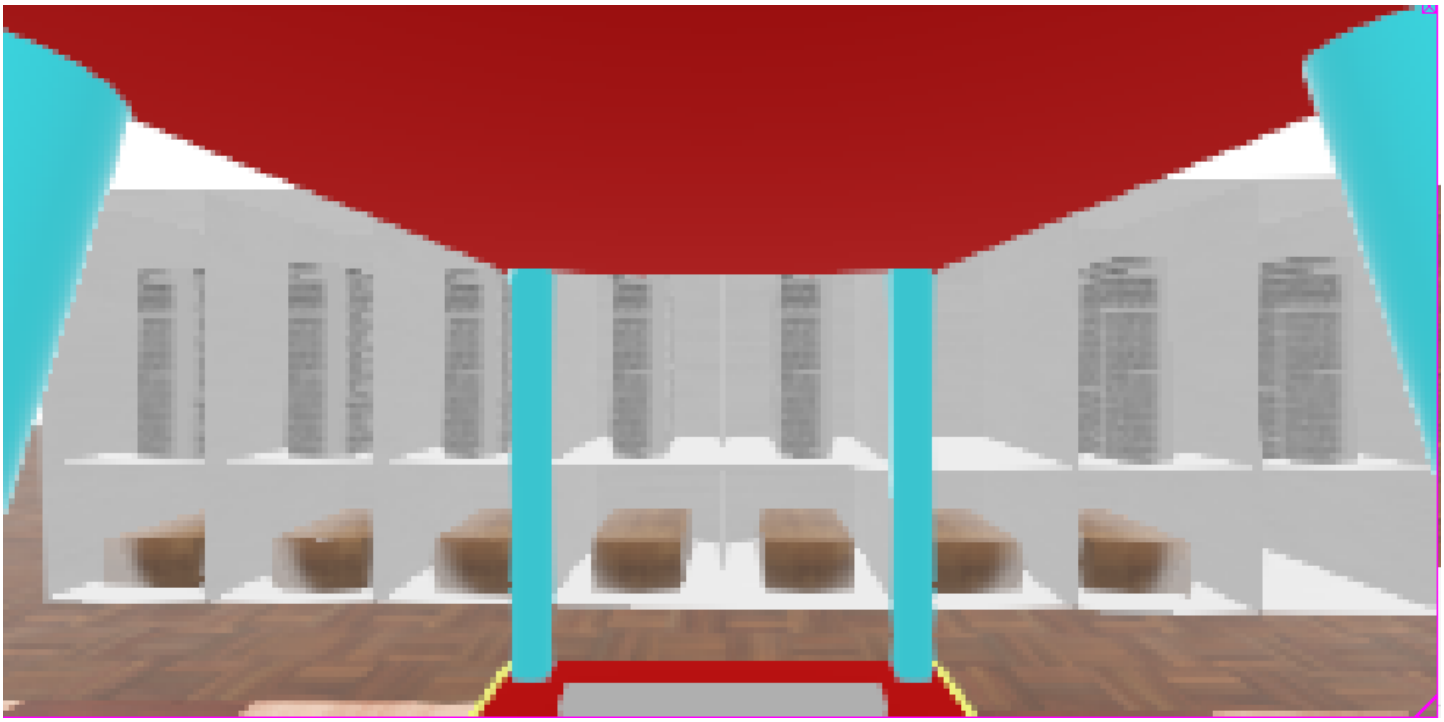
4. 机器人建模

- 遵照软件工程复用思想，将例程中“KUKA youBot”的麦克纳姆轮打包成PROTO，从而实现麦轮节点的复用。四个麦轮直径0.10m，通过HingeJoint安装在车身主体结构上；

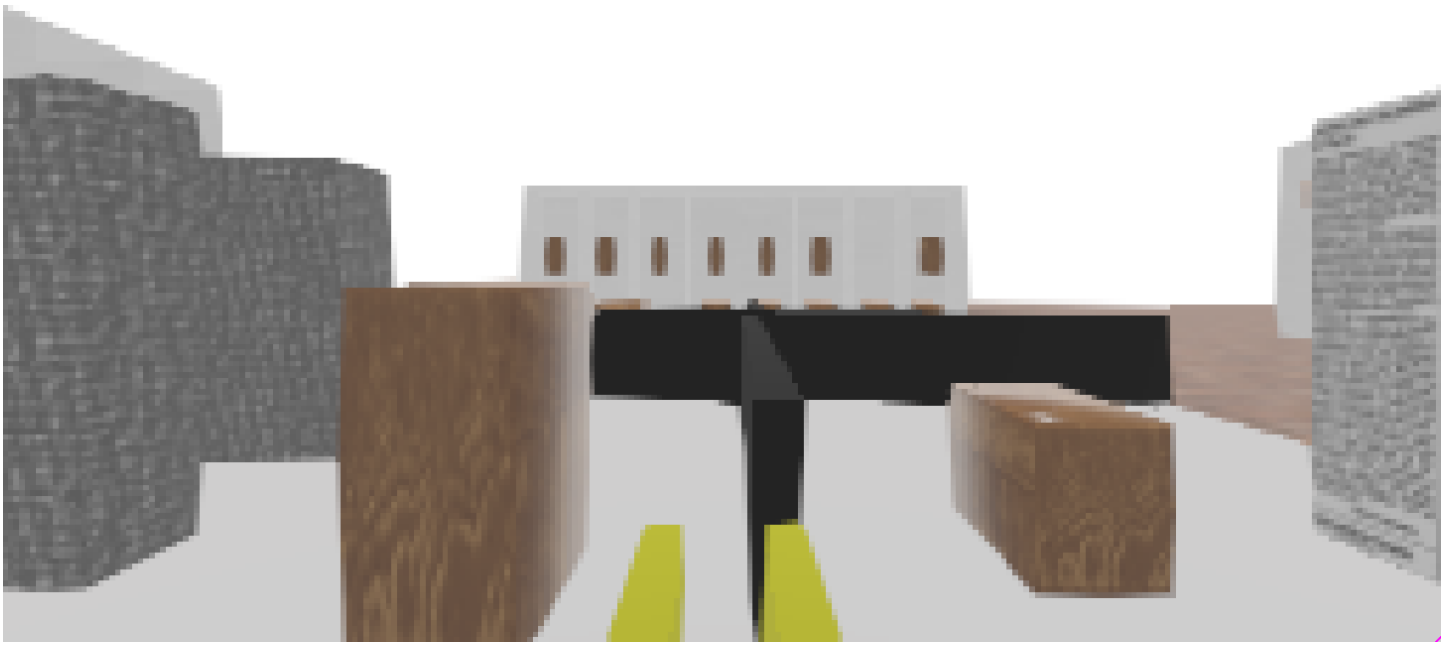
- 车身整体尺寸为0.59m×0.40m×0.64m（长宽高），搭建车身时采用模块化设计，主要分为车身、机械臂两部分。车身主要由Transform、Shape嵌套而成，主要利用到的几何体包括Cylinder、Box等；机械臂主要是在Pioneer3所配置的机械臂的基础上进行适当改造，包括尺寸、电机扭矩及直线电机伸展长度等，同时在机械臂的抓手增加了一对力传感器，以实现对物品抓取力度的精准控制；



- 机器人配置的摄像头分辨率为256*128，视角1.8，配备Webots自带的Recognition API，一定程度上减少了物品识别的难度，使仿真专注于整体软件功能的开发，体现了离线编程的优势；



后置摄像头画面



前置摄像头画面

- 为了实现更精确的物理仿真，我们还对机械臂抓手、麦轮的摩擦力参数进行了设置，以保证机器人能够在当前环境保持正常的运动；此外，还设置了合适的车身重量，以保证车辆的运动性能更加优越、抓取稳定性更好。

5. 机器人的初始化

5.1 初始化函数：`wb_robot_init();`

在调用任何其他C API函数之前，需要先调用初始化函数。此函数用于初始化控制器和Webots之间的通信。

5.2 base_init函数

调用了获取设备函数 `wb_robot_get_device("设备名称")` 。使用电机、传感器等设备前，需使用 `wb_robot_get_device` 函数获取设备的标签(WbDeviceTag)。传入该函数的参数(字符串)是你在Webots中定义的设备名称。如果传入无效的设备名称，则此函数返回0。

该函数主要为四个轮胎的名称进行初始化。

5.3 passive_wait（）函数

函数主要实现一个软件的仿真延时，其中调用了step（）函数，该函数负责实现仿真前进1step。

step（）函数中调用了wb_robot_step(TIME_STEP)通常作为主函数中for或while循环的条件使用。功能：同步控制器数据和仿真器(simulator)之间的数据，例如传感器、电机等数据。所以每个控制器都需要使用 `wb_robot_step` 函数，并且必须定期调用，以更新仿真器的数据。数值 TIME_STEP 表示控制步骤(control step)的持续时间，即 `wb_robot_step` 函数应计算 TIME_STEP 毫秒仿真后返回。此持续时间是指仿真时间，而不是现实世界的时间，因此实际上可能需要花费更少或更多的时间，取决于仿真世界的复杂性。TIME_STEP 必须是 `WorldInfo.basicTimeStep` 的倍数。

5.4 wb_camera_enable

传感器、惯性单元IMU、键盘等在使用前必须使能(启用)，电机使用前不需要使能。如果传感器未使能，则返回未定义(undefined values)的值。可以通过 `wb_*_enable` 函数使能传感器，* 表示传感器类型，例如：

- 距离传感器：`wb_position_sensor_enable(sensor, TIME_STEP);`
- 接触传感器：`wb_touch_sensor_enable(sensor, TIME_STEP);`
- 惯性单元：`wb_inertial_unit_enable(sensor, TIME_STEP);`
- 键盘：`wb_keyboard_enable(TIME_STEP);`

> 函数参数TIME_STEP为更新延迟时间(update delay)，每TIME_STEP毫秒更新一次传感器、IMU、键盘的值，即传感器等数据两次更新之间的间隔为TIME_STEP。一般将更新延迟时间设为与控制周期(control step)相同，使在每次调用 `wb_robot_step(TIME_STEP)` 函数时更新传感器等的值。选择小于控制周期的更新延迟时间是无意义的，因为控制器不可能以高于控制周期的频率处理设备的数据。但是可以选择大于控制周期的更新延迟时间。例如选择更新延迟时间为控制周期的两倍，则调用每两次 `wb_robot_step(TIME_STEP)` 函数，传感器数据才更新一次，可用于仿真慢速设备。设置较大的更新延迟时间可以加快仿真的速度，特别是相机等占用较大CPU资源的设备。

5.5 wb_camera_recognition_enable

`Camera` 节点通过 `Recognition` 节点实现了对象识别功能。通过调用函数 wb_camera_recognition_enable（）启动识别功能。

5.6 base_goto_init

进行一个底盘全方位移动初始化

5.7 set_posture

该函数首次调用的是设置初始位姿，先调用get_gps_values(gps_values)和get_compass_angle(&compass_angle)函数来对全局变量gps的值和罗盘的角度进行初始化，再通过set_posture(initial_posture, gps_values[0], gps_values[1], compass_angle)函数来对初始位姿进行初始化。

第二次调用是设置第一个定点位姿，CurrentShelf初始值为0，故会先针对0号货架的空货物进行查找，调整位姿。

5.8 base_goto_set_target

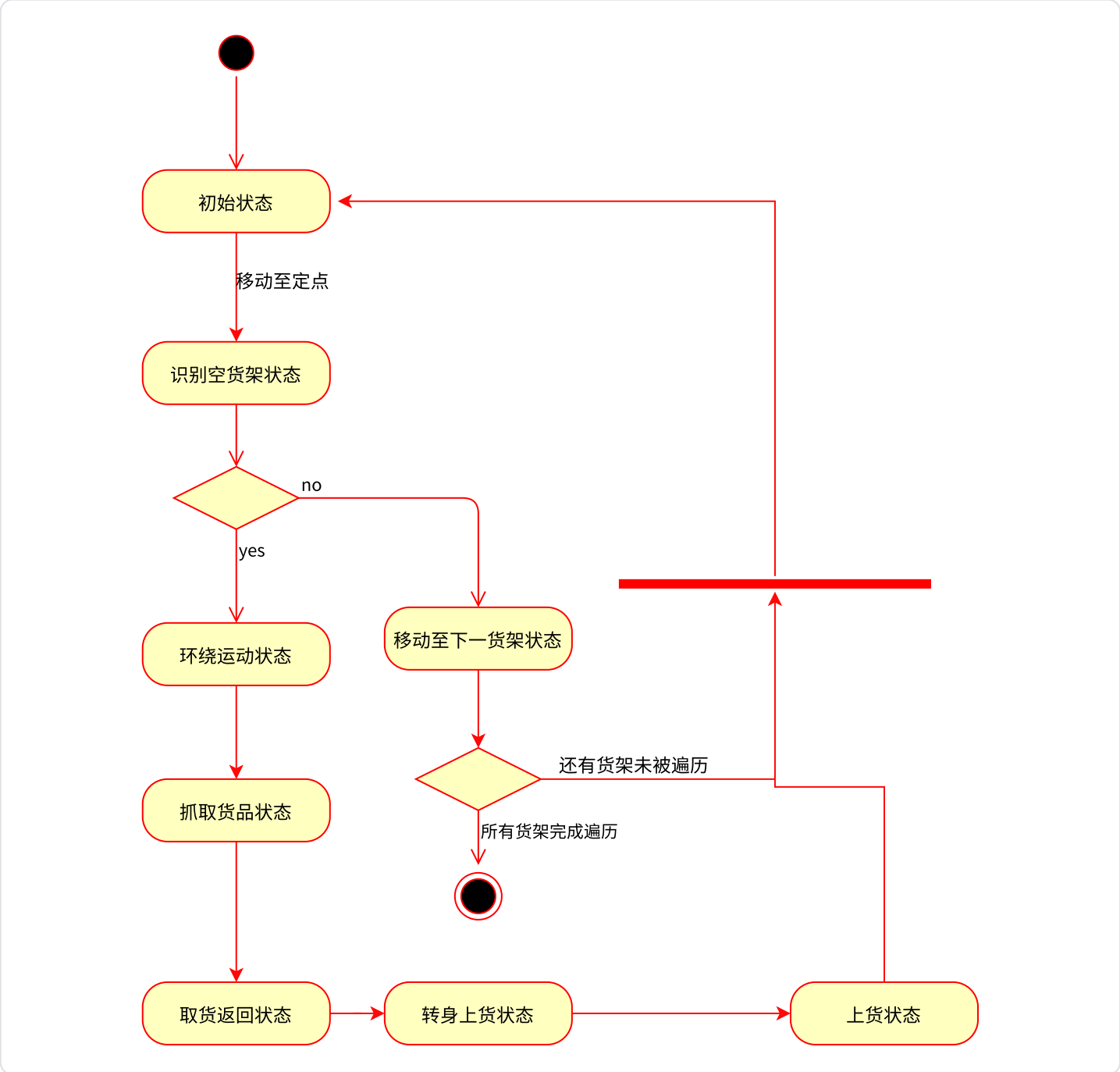
该函数主要是在上一步对临时目标的位姿设定后进行一个目标的设定。

5.9 wb_keyboard_enable

主要是对键盘进行启动。

6. 状态机设计

为方便进行机器人各运动状态之间的复杂控制，设计状态机如下：



- Init_Pose 为初始默认进入工作区状态，到达指定位置后跳转到 Recognize_Empty，对货架进行识别。
- Recognize_Empty 为识别空货架状态，这个状态主要调用了识别货架的函数，识别完成后将货架上的货物状态更新到数组，若有缺货则进行补货判断，计算出需要的货物名称以及需要放置的位置，然后跳转到 Arround_Moving 寻找货品；若没有缺货则跳转到RunTo_NextShelf 状态，前往下一个货架进行检测。
- Arround_Moving 为环游巡检状态，运动过程中一直保持之前暂存的货物名称，以对货仓进行实时检测判断，直到找到了合适可抓取的货物，跳转至 Grab_Item。
- Grab_Item 为抓取仓库货物状态，根据暂存的货物名称行进到指定货物的抓取位置，控制机械臂进行抓取，提升后跳转至 Back_Moving 返程。

- Back_Moving为回程存货环游状态，机器人会携带着需要的货物返回之前的货架，此时需要进行路径规划选择一条最短路线，到达货架前后跳转至 TurnBack_To_LoadItem。
- TurnBack_To_LoadItem 为转身面向货架状态，紧接着跳转至Item_Loading状态。
  Item_Loading 为上货状态，根据货物需要放置的位置计算出机器人的目标位置，之后完成上货操作，跳转至 Init_Pose。
- RunTo_NextShelf 为前往下一货架状态，根据存储的坐标进行路径规划，当到达下一货架定点时，跳转至 Recognize_Empty。
- default为错误报警状态。

利用上述九个状态构成的状态机，能够较地完成对机器人装卸货物的任务要求，各状态内部操作的封装也使得我们的控制器看起来更加简洁干练。

6.1 Recognize_Empty状态的实现

首先调用Find_Empty()函数，通过前置的camera来对当前的货架进行识别，这里调用camera自带的一些识别函数可以得到当前货架的已有物品数量和货架中每个位置的商品类型，并将已有的货品逐个存入到GoodShelf的二维数组中去，没放东西的地方用-1进行标记，通过初始化检测完毕后，再对GoodShelf进行扫描，如果有位置的值为-1，就要寻找该位置的邻近货物，进而判断应该取的货物类型，并存入TargetGood中，该情况下状态机将切换到Arround_Moving状态，如果货架上均有物品存放则直接跳出Find_Empty()函数，并且切换到RunTo_NextShelf状态。

```
if (GoodsonShelf[CurrentShelf][j] == -1)
{
  Empty_Flag = 1;
  TargetIndex = j;
  //寻找邻近货物 判断应该取的货物类型
  //直接覆盖 假装已经放上去了
  int TargetFloor = 0;
  if (j > 7)
    TargetFloor += 8; //层数无关
  if (j % 8 < 4)
    for (int k = 0; k < 8; k++)//从左往右
    {
      if (GoodsonShelf[CurrentShelf][TargetFloor + k] != -1)
      {
        // strcpy(TargetGood, GoodsonShelf[CurrentShelf][TargetFloor + k]);
        TargetGood = GoodsonShelf[CurrentShelf][TargetFloor + k];
        break;
      }
    }
  else
    for (int k = 7; k >= 0; k--)//从右往左
    {
      if (GoodsonShelf[CurrentShelf][TargetFloor + k] != -1)
      {
        // strcpy(TargetGood, GoodsonShelf[CurrentShelf][TargetFloor + k]);
        TargetGood = GoodsonShelf[CurrentShelf][TargetFloor + k];
        break;
      }
    }
  //如果整排都没有可能会出错
  printf("GoodsonShelf[%d][%d] need %s\n", CurrentShelf, j, index2name(TargetGood));
  break;
}
```

6.2 Arround_Moving状态的实现

上一步得到我们需要的TargetGood中后，该状态首先也是调用一个Find_Goods(WbDeviceTag camera,char *good_name,int *item_grasped_id)的函数，该函数主要是调用摄像头的API 识别并判断视野中货物信息（包括 ID、尺寸等），返回是否找到待存放货物的 bool 类型变量。若找到则将该物品的ID存入到item_grasped_id中。考虑到机械臂的抓取能力，寻找的过程中滤除了距离机器人过远的货物，只考虑当前面对的货仓区域。找到需要抓取的物品后状态机将切换为Grab_Item的状态。

倘若此时视野内并没有我们所需要的货物，此时机器人就要实现一个定点巡航的功能，通过环绕中间货物点，我们在周围设置了十二个定点（其中四个顶角因为考虑到机器人的旋转因此各有两个点），机器人在未找到货物的情况下进行定点间的依次移动，这样通过环绕中间的货物一圈以保证能找到所需要的货物。

```
if(strcmp(objects[i].model,good_name) == 0)
{
  if (objects[i].position[2] > 1.3*grasp_dis_threshold)
    // printf("距离 %s 有 %.3f m \n", good_name, -objects[i].position[2]);
  //距离近、左右位置对、且是侧面
  if (objects[i].position[2] > grasp_dis_threshold && fabs(objects[i].position[0]) < 0.1 && objects[i].size[0]
  {
    printf("找到了离我%.3f m 的 %s\n", -objects[i].position[2], good_name);
    *item_grasped_id = objects[i].id;
    return true;
  }
```

6.3 Grab_Item

该状态抓取物品主要是利用Aim_and_Grasp(int *grasp_state, WbDeviceTag camera, int objectID)来实现的，首先选定车体前侧摄像头，传入给定的货物 ID 号来锁定摄像头目标。将抓取货物分为 “调整位置、抓、举” 三个步骤。首先针对不同货物的大小计算一个最合适的抓取位置，然后根据摄像头返回的物体大小粗略地计算手抓的位置，此时激活力传感器，不断收紧抓手直到力传感器达到阈值，表示已经抓紧物体，最后抬升物体达到指定高度，完成抓取功能。

其中对于抓取的过程中，我们的处理是，首先根据目标物的size值估算一个抓手所需张开的宽度值，之后通过wb_motor_get_force_feedback() 函数来获得当前电机的一个力反馈值，如果该力反馈值小于我们预先设定的力反馈值，我们就再将抓手张开的宽度减小0.0003，如果依然小于则继续循环，直到抓手的宽度小到力反馈值达到我们预期的值后，再继续进行后续的抬升操作。

```
      moveFingers(width = objects[i].size[0] / 2);
      wb_robot_step(30000 / TIME_STEP);
    }
  }
  else if (*grasp_state == 1) //抓
  {
    double grasp_force_threshold = 50.0;
    if (wb_motor_get_force_feedback(gripper_motors[1]) > -grasp_force_threshold)
      moveFingers(width -= 0.0003); //步进
```

这里还要注意的是，根据所要抓取的物品不同，机械臂上升直线电机运动到的位置也不同，对水杯我们设置的高度为0.12，对大盒子设置的高度则是0.50，这里情况的特判都是用if-else实现的。举起后抓手的状态则切换为举的状态，同时车的状态机也切换为Back_Moving状态

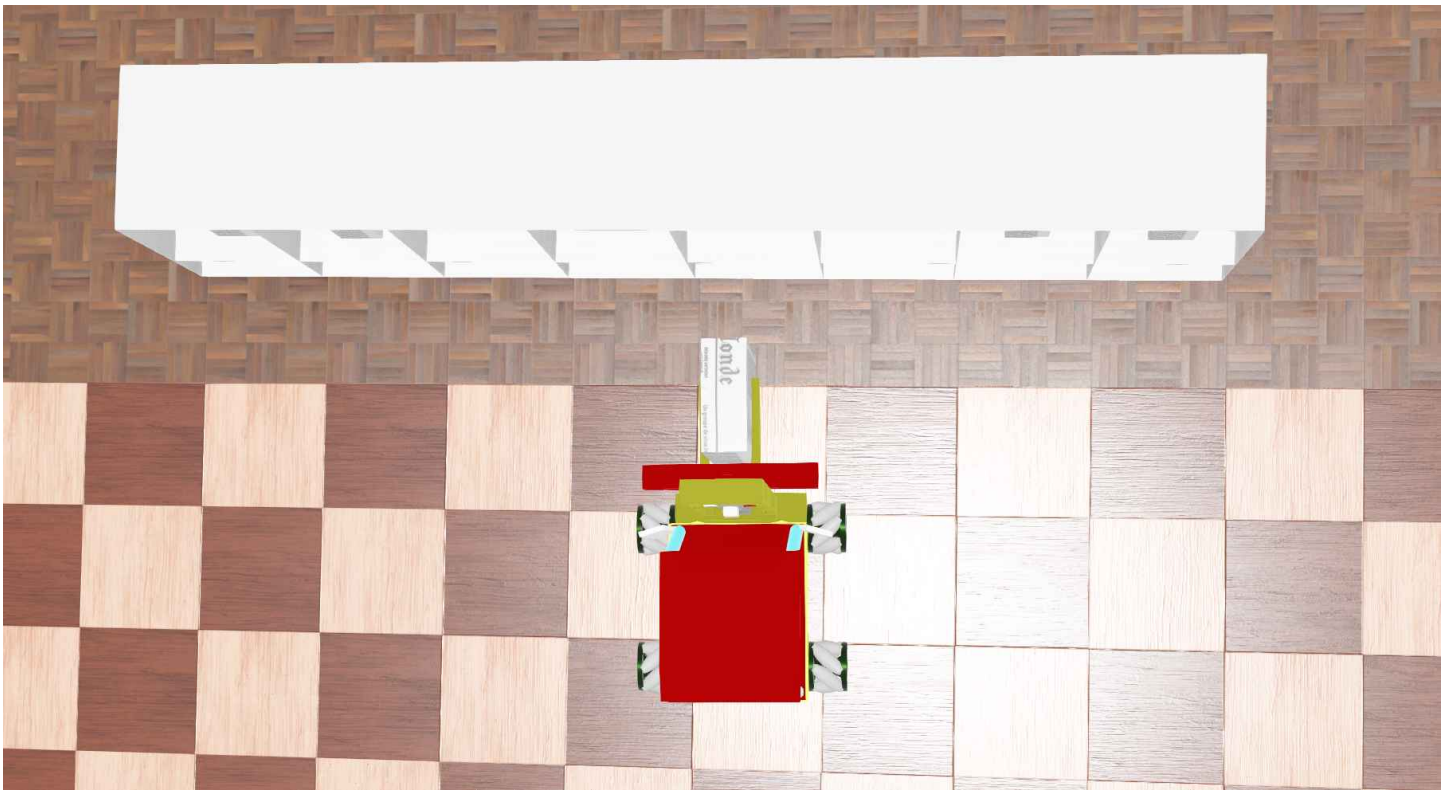
6.4 Back_Moving

Back_Moving为回程存货环游状态，机器人会携带着需要的货物返回之前的货架，此时需要进行路径规划选择一条最短路线，对于路线的选择，这里主要是对当前的位置进行判断来决定顺时针旋转到指定位置还是逆时针回到指定位置，到达上货的指定位置后还要对车进行转向，这里用的就是fixed_posture_loadItem中存储的值，这里相较原本四个定点的区别是角度调转了180度，使货物面向货架，到达货架前后跳转至 TurnBack_To_LoadItem。

```
//处理往哪边近绕圈的问题
if (Moveto_CertainPoint(fin_target_posture, 0.05))
{
  if(fin_target_posture[0] == fixed_posture_findempty[CurrentShelf][0] && fin_target_posture[1] == fixed_posture_findempty[CurrentShelf][1])
  {
    *main_state = TurnBack_To_LoadItem;
    printf("main_state changes from Back_Moving to TurnBack_To_LoadItem!\n");
    set_posture(initial_posture, gps_values[0], gps_values[1], compass_angle);
    //设置下一目标点为反向180度
    set_posture(fin_target_posture, fixed_posture_loadItem[CurrentShelf][0], fixed_posture_loadItem[CurrentShelf][1], fixed_posture_loadItem[CurrentShelf][2]);
    printf("到上货的地方了！\n");
    travel_points_sum = 0;
  }
  else
  {
    set_posture(initial_posture, gps_values[0], gps_values[1], compass_angle);
    if(travel_points_sum >= 6) Travel_Point_Index += 1;//顺时针转
    else
    {
      if(Travel_Point_Index == 0) Travel_Point_Index = 12;
      Travel_Point_Index -= 1;//逆时针转
    }
    Travel_Point_Index %= 12;
    Travel_Point_Index = max(0,Travel_Point_Index);
    set_posture(fin_target_posture, fixed_posture_travelaround[Travel_Point_Index][0], fixed_posture_travelaround[Travel_Point_Index][1], fixed_posture_travelarou
  }
```

6.5 TurnBack_To_LoadItem

这部分算法直接包含在状态机内，根据货物的目标位置，计算机器人水平方向相应的偏移距离，给定一个固定的前后偏移，将上货路径分为两步，由此确定机器人的行驶路径。



6.6 Item_Loading

该部分主要是实现上货的过程，前一步之后我们已经到达了要放货物的指定位置，接下来就是向前前进一些再进行货物的放下，完成放货后进行机器人的初始化，再回到Init_Pose的状态。

7. 机器人移动的主要实现

因为在对于机器人的建模设计中参考了官方示例中“KUKA youBot”，将其麦克纳姆轮的模型封装成PROTO节点并以其为基础搭建了机器人的底盘，因此在处理机器人的移动中我们也基本沿用了KUKA

youBot的移动实现。在开源的代码中，因考虑到机器人的运动除了旋转外还有平移，针对该机器人采用了齐次变换矩阵将旋转和平移进行统一。针对麦克纳姆轮，其实现转向主要是通过改变每个轮子的转速和方向来实现。例如，如果机器人向前移动，那么需要使所有麦克纳姆轮以相同的速度和方向旋转。如果想让机器人向左移动，那么您需要使右侧的麦克纳姆轮向前旋转，而左侧的麦克纳姆轮向后旋转。如果您想让机器人既能够向前移动又能够向左移动，那么需要使右侧的麦克纳姆轮向前旋转，而左侧的麦克纳姆轮向后旋转，并且两侧的麦克纳姆轮以不同的速度旋转。

8. 键盘的基本控制

对于键盘的控制，这里用到了wb_keyboard_get_key()函数来读取键盘值，之后再在实现的keyboard_control（）函数中对其进行处理。进入实现的函数中，主要是利用switch-case语句进行处理，根据传入的不同值，只需要通过调用对应的移动函数即可实现。

具体的操作方式为：

- 四个箭头键用于控制小车的上下左右
- Page_up控制小车向左转，Page_down控制小车向右转
- Shift+上箭头使爪子向上滑动，Shift+下箭头使爪子向下滑动
- Shift+左箭头使夹子张开，Shift+右箭头使夹子闭合

```
case 'C':
{
  printf("Manually stopped! \n");
  *main_state = -1;
  break;
}
case WB_KEYBOARD_UP:
  printf("Go forwards\n");
  base_forwards();
  break;
case WB_KEYBOARD_DOWN:
  printf("Go backwards\n");
  base_backwards();
  break;
case WB_KEYBOARD_LEFT:
  printf("Strafe left\n");
  base_strafe_left();
  break;
case WB_KEYBOARD_RIGHT:
  printf("Strafe right\n");
  base_strafe_right();
  break;
case WB_KEYBOARD_PAGEUP:
  printf("Turn left\n");
  base_turn_left();
  break;
case WB_KEYBOARD_PAGEDOWN:
  printf("Turn right\n");
  base_turn_right();
```

9. 总结和期望

总结

- 模块化和状态机设计

  在整个补货的复杂任务的前提下，将其拆分为“查货架”、“寻货”、“取货”、“回库”及“上货”五个易解决的小任务，在此基础上设计了状态机，实现了任务的模块化。

- 及时选择了合适的编程语言

最早我们选用了Python语言，但后来发现在C语言下状态机的实现更加直白明了，同时官方示例给出的C语言源码的可读性更强，更改语言也让我们后续的开发过程更加快速顺利。

- 复用

在本项目中，我们对官方示例提供的一些模型和源码进行了大量的复用，通过这些给我们大大减小了工作量。

- 最初的需求确认欠缺考虑

最早小组确立的需求为实现一个校内的无人快递小车，但后来考虑到嘉定路面以及快递站的实际情况，以及在虚拟仿真环境下比较难以模拟路面的实际情况，项目的可实现性不强，因此在沟通后决定将项目需求改为实现一个快递站内的智能机器人，更符合实际的需求，但也因此浪费了许多时间。因此应在着手实现项目前应细心考虑需求的可行性，以实现更有意义的课程项目。

期望

- 由于开发时间有限，因此我们的开发场景并不完整，缺少了对于小车抓取货物后到快递柜这中间的场景建模。而在实际的应用场景中对该部分的要求同样比较重要，因为该部分涉及了在运输的过程中点到点的路径规划问题以及智能避障问题，尽管在本项目中这两方面都有简单的体现，但当实际的环境场景变得复杂时，选择更高效的路径规划算法和精准的避障算法尤为重要。
- 目前实现的项目部分对于场景的限制性比较强，一些地方存在特别判断的地方，不具有普遍性，对于场景进行变更容易产生bug，后续应该增加场景的普适性，避免机器人在应用过程中出现问题，保证机器人在出现一定范围的突发情况时仍能正常的运行。
- 应继续对项目进行迭代开发，丰富场景，复杂环境建模和物品种类，使项目更加符合实际场景中的外卖柜样式及取货区场景。