RCS COPPELIASIM WORKSHOP 2025
DAY -3 “THE SURPRISE”

OBSTACLE AVOIDANCE BOT CODE

```
local left_joint
local right_joint
local front_sensor
local state = "forward"
local avoidStartTime = 0
local avoidDuration = 0.6
local forwardSpeed = 10
local turnSpeed = 3

function sysCall_init()
    left_joint = sim.getObject('/robot_base/left_joint')       -- !! REPLACE THIS
PATH !!
    right_joint = sim.getObject('/robot_base/right_joint')     -- !! REPLACE THIS
PATH !!
    front_sensor = sim.getObject('/robot_base/front_sensor')

    if left_joint == -1 or right_joint == -1 or front_sensor == -1 then
        sim.addStatusbarMessage("? Object handle error!")
    else
        sim.addStatusbarMessage("? Robot initialized")
    end
end

function sysCall_actuation()
    local result, distance = sim.readProximitySensor(front_sensor)
    local timeNow = sim.getSimulationTime()

    if state == "forward" then
        if result > 0 then
            state = "avoid"
            avoidStartTime = timeNow
            sim.addStatusbarMessage("?? Obstacle! Turning...")
        else
            sim.setJointTargetVelocity(left_joint, forwardSpeed)
            sim.setJointTargetVelocity(right_joint, forwardSpeed)
        end

    elseif state == "avoid" then
```

```
        sim.setJointTargetVelocity(left_joint, -turnSpeed)
        sim.setJointTargetVelocity(right_joint, turnSpeed)

        if timeNow - avoidStartTime >= avoidDuration then
            state = "forward"
            sim.addStatusbarMessage("? Avoided. Moving again.")
        end
    end
end

function sysCall_cleanup()
    sim.setJointTargetVelocity(left_joint, 0)
    sim.setJointTargetVelocity(right_joint, 0)
end
```

RIGHT WALL FOLLOWING BOT

```
-- ===============================================

-- CoppeliaSim Lua Script: Right Wall Following Bot
-- Attached as a Non-threaded child script to the robot's base.
-- ===============================================

-- Global Handles (accessible by all functions in this script)
local left_joint
local right_joint
local front_sensor
local right_sensor
local robot_dummy_handle -- Dummy object representing the robot's position (e.g.,
at the center of the base)
local goal_dummy         -- Dummy object representing the target goal position

-- Motion settings
local forward_speed_wf = 0.35 -- Base speed for wall following (meters/second)
local turn_kp_wf = 5.0      -- Proportional gain for wall-following distance
correction.
                            -- Higher value = more aggressive correction. Tune
carefully!
local wall_dist_setpoint = 0.30 -- Desired distance to the wall (in meters).
                                -- IMPORTANT: Adjust this based on your
RIGHT_SENSOR's
                                -- effective detection range and your robot's
size.

-- Emergency avoidance parameters (for very close front obstacles)
```

```
local emergency_turn_speed_ratio = 2.0  -- Multiplier for turning speed during
emergency turn
local emergency_reverse_speed_ratio = 0.5 -- Multiplier for reverse speed during
emergency backup

-- ===============================================
-- sysCall_init(): Called once when the simulation starts
-- ===============================================
function sysCall_init()
    -- --- GET OBJECT HANDLES ---
    -- !! IMPORTANT: REPLACE THESE PLACEHOLDER PATHS with the ACTUAL paths from
your scene hierarchy. !!
    -- !! To get the exact path for an object:
    -- !! 1. In CoppeliaSim, navigate to the object in the Scene Hierarchy (left
panel).
    -- !! 2. Right-click on the object.
    -- !! 3. Select "Copy/paste" -> "Selected object name".
    -- !! 4. Paste that copied path here, replacing the placeholder string.
    -- !! Ensure exact case-sensitivity.

    left_joint = sim.getObject('/robot_base/left_joint')           -- Path to
your robot's left wheel joint
    right_joint = sim.getObject('/robot_base/right_joint')         -- Path to
your robot's right wheel joint
    front_sensor = sim.getObject('/robot_base/front_sensor')       -- Path to
your robot's front proximity sensor
    right_sensor = sim.getObject('/robot_base/right_sensor')       -- Path to
your robot's right proximity sensor
    robot_dummy_handle = sim.getObject('/robot_base/robot_dummy') -- Path to a
dummy representing your robot's central position (e.g., child of robot_base)
    goal_dummy = sim.getObject('/goal')                            -- Path to
your target goal dummy object

    -- --- ERROR CHECKING FOR HANDLES ---
    -- This block is crucial for debugging. It will print an error message in the
CoppeliaSim console
    -- if any required object is not found, and will indicate if the script might
not function correctly.
    local all_handles_valid = true
    if left_joint == -1 then sim.addLog(sim.verbosity_scripterr, 'Init Error:
"left_joint" handle invalid. Check path!'); all_handles_valid = false end
    if right_joint == -1 then sim.addLog(sim.verbosity_scripterr, 'Init Error:
"right_joint" handle invalid. Check path!'); all_handles_valid = false end
    if front_sensor == -1 then sim.addLog(sim.verbosity_scripterr, 'Init Error:
"front_sensor" handle invalid. Check path!'); all_handles_valid = false end
```

```
    if right_sensor == -1 then sim.addLog(sim.verbosity_scripterr, 'Init Error: 
"right_sensor" handle invalid. Check path. Wall-following will be affected!'); 
all_handles_valid = false end
    if goal_dummy == -1 then sim.addLog(sim.verbosity_scripterr, 'Init Error: 
"goal_dummy" handle invalid. Check path!'); all_handles_valid = false end
    if robot_dummy_handle == -1 then sim.addLog(sim.verbosity_scripterr, 'Init 
Error: "robot_dummy_handle" handle invalid. Check path: 
/robot_base/robot_dummy'); all_handles_valid = false end

    if not all_handles_valid then
        sim.addLog(sim.verbosity_scripterr, 'CRITICAL: Robot not fully 
initialized due to missing objects. See previous errors. Script will not function 
correctly.')
        -- Consider adding: sim.stopSimulation() here if you want the simulation 
to stop immediately on critical errors.
    else
        sim.addStatusbarMessage("Right Wall Follower initialized. Starting wall 
following behavior.")
    end
end

-- ===============================================
-- sysCall_actuation(): Called at each simulation step for continuous actions
-- ===============================================
function sysCall_actuation()
    -- --- PRE-CHECKS: Ensure all critical handles are valid before proceeding --
-
    -- If any object handle is invalid, stop the motors and exit the function to 
prevent errors.
    if left_joint == -1 or right_joint == -1 or front_sensor == -1 or 
right_sensor == -1
        or robot_dummy_handle == -1 or goal_dummy == -1 then
        if left_joint ~= -1 then sim.setJointTargetVelocity(left_joint, 0) end
        if right_joint ~= -1 then sim.setJointTargetVelocity(right_joint, 0) end
        sim.addStatusbarMessage("Critical: Missing robot objects. Robot 
stopped.")
        return
    end

    -- --- Read the proximity sensors ---
    -- 'front_detected' and 'right_detected' will be > 0 if an object is 
detected, 0 otherwise.
    -- 'front_dist' and 'right_dist' will contain the measured distance to the 
detected object.
```

```
    local front_detected, front_dist, _, _, _ =
sim.readProximitySensor(front_sensor)
    local right_detected, right_dist, _, _, _ =
sim.readProximitySensor(right_sensor)

    -- --- Get Robot and Goal Positions for Goal Check ---
    local pos_robot = sim.getObjectPosition(robot_dummy_handle, -1) -- Robot's
current position
    local pos_goal = sim.getObjectPosition(goal_dummy, -1)           -- Goal's
position

    -- Defensive checks: ensure position data is not nil (unlikely with valid
handles, but safe)
    if not pos_robot or not pos_goal then
        sim.addLog(sim.verbosity_scripterr, 'Runtime Error: Robot or Goal
position is nil. Check model integrity.');
        sim.setJointTargetVelocity(left_joint, 0);
sim.setJointTargetVelocity(right_joint, 0);
        return;
    end

    -- Calculate 2D distance to goal (assuming movement in XY plane)
    local dx = pos_goal[1] - pos_robot[1]
    local dy = pos_goal[2] - pos_robot[2]
    local dist_to_goal = math.sqrt(dx*dx + dy*dy)

    -- === GOAL REACHED CHECK (Highest Priority) ===
    -- If the robot is close enough to the goal, stop the simulation.
    if dist_to_goal < 0.15 then -- Adjust this threshold based on your robot's
size and desired precision
        sim.setJointTargetVelocity(left_joint, 0)
        sim.setJointTargetVelocity(right_joint, 0)
        sim.addStatusbarMessage("?? Goal Reached! Stopping simulation.")
        sim.stopSimulation() -- Stops the CoppeliaSim simulation
        return -- Exit the function as goal is reached
    end

    -- === RIGHT WALL FOLLOWING LOGIC ===
    local lv_wf = forward_speed_wf -- Left wheel velocity (default to forward
speed)
    local rv_wf = forward_speed_wf -- Right wheel velocity (default to forward
speed)

    -- Rule 1: Emergency Collision Avoidance (Highest priority for immediate
safety)
```

```
    -- If the front sensor detects an obstacle very close, execute an evasive
turn.
    if front_detected > 0 and front_dist < 0.20 then -- Adjust threshold if your
sensor is different
        lv_wf = -forward_speed_wf * emergency_reverse_speed_ratio -- Reverse left
wheel (helps turn sharper)
        rv_wf = forward_speed_wf * emergency_turn_speed_ratio   -- Forward right
wheel (causes aggressive left turn)
        sim.addStatusbarMessage("WF: Front obstructed! Turning sharply left to
avoid.")
    -- Rule 2: Maintain Distance to Wall on Right (Primary Wall Following
behavior)
    -- This applies if the right sensor detects a wall.
    elseif right_detected > 0 then
        -- Calculate the error: how far off we are from the desired wall
distance.
        local dist_error = right_dist - wall_dist_setpoint
        -- Adjust wheel velocities proportionally to the error.
        -- If dist_error is positive (too far), lv increases, rv decreases ->
turn right to get closer.
        -- If dist_error is negative (too close), lv decreases, rv increases ->
turn left to move away.
        lv_wf = forward_speed_wf - turn_kp_wf * dist_error
        rv_wf = forward_speed_wf + turn_kp_wf * dist_error

        -- Clamp velocities to prevent them from becoming excessively high or
negative (spinning)
        local max_wheel_speed = forward_speed_wf * 2.0 -- Max allowed wheel speed
        lv_wf = math.max(-max_wheel_speed, math.min(max_wheel_speed, lv_wf))
        rv_wf = math.max(-max_wheel_speed, math.min(max_wheel_speed, rv_wf))

        -- sim.addStatusbarMessage(string.format("WF: Maintaining wall. Dist
Error: %.2f (Actual: %.2f)", dist_error, right_dist)) -- Uncomment for detailed
debug
    -- Rule 3: No Right Wall Detected (e.g., at a corner or end of wall, or
sensor out of range)
    -- If the right sensor sees nothing, turn right to find the wall again.
    else
        lv_wf = forward_speed_wf * 1.0 -- Left wheel moves at base speed
        rv_wf = forward_speed_wf * 0.3 -- Right wheel moves slower (causes a
gentle turn to the right)
        sim.addStatusbarMessage("WF: No wall on right, turning right to search.")
    end

    -- Apply the calculated velocities to the robot's joints
    sim.setJointTargetVelocity(left_joint, lv_wf)
    sim.setJointTargetVelocity(right_joint, rv_wf)

    -- Optional: Display overall status in the status bar for quick debugging
    -- This can be useful, but avoid updating it too frequentlsy (e.g., every
step) as it can affect performance.
    -- For continuous detailed debugging, use
`sim.addLog(sim.verbosity_scriptinfo, "Your message")` which goes to the console.
    sim.addStatusbarMessage(string.format("Dist to Goal: %.2f | Front: %s (%.2f)
| Right: %s (%.2f)",
        dist_to_goal, (front_detected > 0 and "Yes" or "No"), front_dist,
(right_detected > 0 and "Yes" or "No"), right_dist))
end

-- ===============================================
-- sysCall_cleanup(): Called once when the simulation ends
-- ===============================================
function sysCall_cleanup()
    sim.addStatusbarMessage("?? Script cleanup complete.")
    -- Ensure motors are stopped gracefully when the simulation ends
    if left_joint ~= -1 then sim.setJointTargetVelocity(left_joint, 0) end
    if right_joint ~= -1 then sim.setJointTargetVelocity(right_joint, 0) end
end
```

BUG 2 ALGORITHM

```
-- ===============================================
-- CoppeliaSim Lua Script: Bug 2 Path Planning Algorithm
-- Attached as a Non-threaded child script to the robot's base object.
-- ===============================================

-- --- GLOBAL HANDLES ---
-- These variables will store the handles (IDs) of your robot's components and
goal.
-- They are accessible by all functions in this script.
local left_joint
local right_joint
local front_sensor
local right_sensor
local robot_dummy_handle -- A dummy object attached to the robot's base,
representing its central point.
local goal_dummy         -- A dummy object in your scene representing the target
goal position.

-- --- MOTION SETTINGS ---
-- Tune these values based on your robot's dynamics, wheel size, and desired
behavior.
local base_speed = 0.5          -- Base forward velocity for the robot
(meters/second)
local turn_kp = 2.0             -- Proportional gain for turning towards the goal
(in MOVE_TO_GOAL state).
                                -- Higher value = more aggressive turning.

local wall_follow_speed = 0.35 -- Base speed when following a wall. Often
slightly slower for control.
local wall_turn_kp = 5.0      -- Proportional gain for maintaining distance to
the wall.
                                -- Higher value = more aggressive wall
correction.
local wall_dist_setpoint = 0.30 -- Desired distance to the wall (in meters) for
right wall following.
                                -- IMPORTANT: This must be within your
right_sensor's effective range.

-- Emergency avoidance parameters (for very close front obstacles during wall-
follow or initial hit)
local emergency_turn_speed_ratio = 2.0  -- How much faster to turn during an
emergency turn (e.g., when front is blocked)
local emergency_reverse_speed_ratio = 0.5 -- How fast to reverse during a very
close front obstacle detection

-- --- BUG 2 STATES ---
-- Define the possible states of the robot's navigation algorithm.
local STATE_MOVE_TO_GOAL = "MOVE_TO_GOAL"
local STATE_WALL_FOLLOW = "WALL_FOLLOW"
local STATE_BACKING_UP = "BACKING_UP" -- An intermediate state for emergency
obstacle clearance

local state = STATE_MOVE_TO_GOAL -- The robot starts by trying to move directly
to the goal.

-- --- BUG 2 ALGORITHM SPECIFIC VARIABLES ---
local hit_point_pos = {0.0, 0.0, 0.0}       -- Stores the robot's (x,y,z)
position when it first hits an obstacle.
local initial_dist_to_goal_at_hit = 0.0     -- Stores the distance from
hit_point_pos to the goal.
local last_recheck_time = 0.0               -- Timestamp for when the M-line re-
entry condition was last checked.
local recheck_interval = 0.5                -- How often (in seconds) to check
the M-line condition in WALL_FOLLOW state.
```

```
local backup_duration = 0.5                  -- How long (in seconds) the robot
backs up during an emergency.
local backup_start_time = 0.0                -- Timestamp for when the backup
maneuver started.


-- ==============================================
-- sysCall_init(): Called once when the simulation starts.
-- This function initializes handles and sets the initial state.
-- ==============================================
function sysCall_init()
    -- --- GET OBJECT HANDLES ---
    -- !! IMPORTANT: CONFIGURE THESE PATHS/ALIASES ACCURATELY IN COPPELIASIM !!
    --   +- robot_base
    --      +- left_joint
    --      +- right_joint
    --      +- front_sensor
    --      +- right_sensor
    --      +- robot_dummy (a dummy object at the robot's center)
    -- Your goal object would be a separate object at the root, named 'goal'.

    left_joint = sim.getObject('/robot_base/left_joint')
    right_joint = sim.getObject('/robot_base/right_joint')
    front_sensor = sim.getObject('/robot_base/front_sensor')
    right_sensor = sim.getObject('/robot_base/right_sensor')
    robot_dummy_handle = sim.getObject('/robot_base/robot_dummy')
    goal_dummy = sim.getObject('/goal')

    -- --- ERROR CHECKING FOR HANDLES ---
    local all_handles_valid = true
    if not left_joint or left_joint == -1 then
sim.addLog(sim.verbosity_scripterr, 'Bug 2 Init Error: "left_joint" handle
invalid. Check path/alias!'); all_handles_valid = false end
    if not right_joint or right_joint == -1 then
sim.addLog(sim.verbosity_scripterr, 'Bug 2 Init Error: "right_joint" handle
invalid. Check path/alias!'); all_handles_valid = false end
    if not front_sensor or front_sensor == -1 then
sim.addLog(sim.verbosity_scripterr, 'Bug 2 Init Error: "front_sensor" handle
invalid. Check path/alias!'); all_handles_valid = false end
    if not right_sensor or right_sensor == -1 then
sim.addLog(sim.verbosity_scripterr, 'Bug 2 Init Error: "right_sensor" handle
invalid. Check path/alias. Wall-following will be affected!'); all_handles_valid
= false end
```

```
    if not goal_dummy or goal_dummy == -1 then
sim.addLog(sim.verbosity_scripterr, 'Bug 2 Init Error: "goal_dummy" handle
invalid. Check path/alias!'); all_handles_valid = false end
    if not robot_dummy_handle or robot_dummy_handle == -1 then
sim.addLog(sim.verbosity_scripterr, 'Bug 2 Init Error: "robot_dummy_handle"
handle invalid. Check path/alias!'); all_handles_valid = false end

    if not all_handles_valid then
        sim.addLog(sim.verbosity_scripterr, 'CRITICAL: Bug 2 robot not fully
initialized due to missing objects. See previous errors. Script will not function
correctly.')
        -- Consider adding 'sim.stopSimulation()' here if you want the simulation
to stop immediately on critical setup errors.
    else
        sim.addStatusbarMessage("Bug 2 robot initialized. Starting " ..
STATE_MOVE_TO_GOAL .. " state.")
    end

    -- Initialize state machine variables
    state = STATE_MOVE_TO_GOAL
    hit_point_pos = {0.0, 0.0, 0.0} -- Reset to default values
    initial_dist_to_goal_at_hit = 0.0
    last_recheck_time = 0.0
    backup_start_time = 0.0
end

-- ===============================================
-- sysCall_actuation(): Called at each simulation step for continuous actions.
-- This function contains the main Bug 2 logic and state machine.
-- ===============================================
function sysCall_actuation()
    -- --- PRE-CHECKS: Ensure all critical object handles are valid ---
    -- If any required object is missing, stop the robot and exit this function
to prevent errors.
    if (not left_joint or left_joint == -1) or (not right_joint or right_joint ==
-1) or
       (not front_sensor or front_sensor == -1) or (not right_sensor or
right_sensor == -1) or
       (not robot_dummy_handle or robot_dummy_handle == -1) or (not goal_dummy or
goal_dummy == -1) then
        -- Stop motors only if their handles are valid
        if left_joint ~= nil and left_joint ~= -1 then
sim.setJointTargetVelocity(left_joint, 0) end
        if right_joint ~= nil and right_joint ~= -1 then
sim.setJointTargetVelocity(right_joint, 0) end
```

```
        sim.addStatusbarMessage("Critical: Missing robot objects. Robot
stopped.")
        return -- Exit the function early
    end

    local now = sim.getSimulationTime() -- Get current simulation time

    -- --- Read the proximity sensors ---
    -- 'result' will be > 0 if an object is detected, 0 otherwise.
    -- 'distance' will contain the measured distance to the detected object.
    local front_detected, front_dist_raw, _, _, _ =
sim.readProximitySensor(front_sensor)
    local right_detected, right_dist_raw, _, _, _ =
sim.readProximitySensor(right_sensor)

    -- Ensure distances are numbers, even if no object is detected.
    -- Default to a large number (e.g., 999.0) if not detected to prevent
nil/type errors in string.format.
    local front_dist = (front_detected > 0 and front_dist_raw or 999.0)
    local right_dist = (right_detected > 0 and right_dist_raw or 999.0)

    -- --- Get Robot and Goal Positions and Robot Orientation ---
    local pos_robot = sim.getObjectPosition(robot_dummy_handle, -1) -- Robot's
current absolute position
    local orient_robot = sim.getObjectOrientation(robot_dummy_handle, -1) --
Robot's current absolute orientation
    local heading = orient_robot[3] -- Robot's yaw/heading (rotation around Z-
axis)

    local pos_goal = sim.getObjectPosition(goal_dummy, -1)             -- Goal's
absolute position

    -- Defensive checks: ensure position data is not nil (unlikely with valid
handles, but safe)
    if not pos_robot or not pos_goal then
        sim.addLog(sim.verbosity_scripterr, 'Bug 2 Runtime Error: Robot or Goal
position is nil. Check model integrity.');
        if left_joint ~= nil and left_joint ~= -1 then
sim.setJointTargetVelocity(left_joint, 0) end
        if right_joint ~= nil and right_joint ~= -1 then
sim.setJointTargetVelocity(right_joint, 0) end
        return;
    end

    -- --- Calculate Distance and Angle to Goal ---
    local dx = pos_goal[1] - pos_robot[1]
    local dy = pos_goal[2] - pos_robot[2]
    local dist_to_goal = math.sqrt(dx*dx + dy*dy) -- Euclidean distance
    local angle_to_goal = math.atan2(dy, dx)      -- Angle from robot to goal

    -- Calculate heading error (difference between current heading and angle to
goal)
    local heading_error = angle_to_goal - heading
    -- Wrap heading error to [-pi, pi] range for shortest turn
    while heading_error > math.pi do heading_error = heading_error - 2 * math.pi
end
    while heading_error < -math.pi do heading_error = heading_error + 2 * math.pi
end

    -- === GOAL REACHED CHECK (Highest Priority) ===
    -- If the robot is very close to the goal, stop the simulation.
    if dist_to_goal < 0.15 then -- Adjust this threshold based on your robot's
size and desired precision
        sim.setJointTargetVelocity(left_joint, 0)
        sim.setJointTargetVelocity(right_joint, 0)
        sim.addStatusbarMessage("?? Goal Reached! Stopping simulation.")
        sim.stopSimulation() -- Stops the CoppeliaSim simulation
        return -- Exit the function as goal is reached
    end

    -- === BACKUP RECOVERY STATE (Highest immediate priority for collision
avoidance) ===
    -- If the front sensor detects an obstacle *very* close, switch to the backup
state.
    -- This ensures the robot doesn't get stuck or push the obstacle.
    if front_detected > 0 and front_dist < 0.15 and state ~= STATE_BACKING_UP
then
        state = STATE_BACKING_UP
        backup_start_time = now
        sim.addStatusbarMessage("? Obstacle very close! Initiating brief
reverse.")
    end

    -- Handle the `BACKING_UP` state's behavior
    if state == STATE_BACKING_UP then
        sim.setJointTargetVelocity(left_joint, -base_speed *
emergency_reverse_speed_ratio) -- Reverse slowly
        sim.setJointTargetVelocity(right_joint, -base_speed *
emergency_reverse_speed_ratio)
```

```
        -- Check if the backup duration has passed
        if now - backup_start_time >= backup_duration then
            -- After backing up, transition immediately to WALL_FOLLOW to
navigate around the obstacle.
            state = STATE_WALL_FOLLOW
            sim.addStatusbarMessage("Backup complete. Now attempting to navigate
around obstacle.")
        end
        return -- Important: Exit `sysCall_actuation` to let backup complete
before other states interfere.
    end

    -- ==============================================
    -- === MAIN BUG 2 STATE MACHINE LOGIC ===
    -- ==============================================

    if state == STATE_MOVE_TO_GOAL then
        -- Action: Move towards the goal
        -- Calculate turn based on proportional control of heading error.
        local turn = turn_kp * heading_error
        -- Clamp turn amount to prevent excessive spinning
        turn = math.max(-base_speed, math.min(base_speed, turn))

        -- Calculate individual wheel velocities for differential drive
        local lv = base_speed - turn
        local rv = base_speed + turn
        sim.setJointTargetVelocity(left_joint, lv)
        sim.setJointTargetVelocity(right_joint, rv)

        -- Transition Condition: Obstacle detected in front
        if front_detected > 0 then
            state = STATE_WALL_FOLLOW -- Switch to wall-following state
            -- Store 2D hit point for M-line calculation (z-coordinate is often
ignored in 2D path planning)
            hit_point_pos = {pos_robot[1], pos_robot[2], pos_robot[3]}
            initial_dist_to_goal_at_hit = dist_to_goal                -- Record
distance to goal at hit point
            last_recheck_time = now                                   -- Reset
M-line recheck timer
            sim.addStatusbarMessage("?? Obstacle detected! Switching to
WALL_FOLLOW.")
        end

    elseif state == STATE_WALL_FOLLOW then
        -- Action: Follow the wall on the robot's right side.
        local lv_wf = wall_follow_speed -- Left wheel velocity
        local rv_wf = wall_follow_speed -- Right wheel velocity

        -- Rule 1: Emergency Front Collision Avoidance within Wall-Follow
        -- If an obstacle is still directly in front while wall-following, turn
more sharply left.
        if front_detected > 0 and front_dist < 0.25 then -- Use a slightly larger
threshold than backup
            lv_wf = -wall_follow_speed * emergency_reverse_speed_ratio -- Reverse
left wheel
            rv_wf = wall_follow_speed * emergency_turn_speed_ratio  -- Forward
right wheel (aggressive left turn)
            sim.addStatusbarMessage("WF: Front obstructed while following!
Turning sharply left.")
        -- Rule 2: Maintain Distance to Wall on Right (Primary Wall Following
behavior)
        -- This applies if the right sensor detects a wall.
        elseif right_detected > 0 then
            local dist_error = right_dist - wall_dist_setpoint
            lv_wf = wall_follow_speed - wall_turn_kp * dist_error
            rv_wf = wall_follow_speed + wall_turn_kp * dist_error

            -- Clamp wheel velocities to prevent excessive values
            local max_allowed_speed = wall_follow_speed * 1.5
            lv_wf = math.max(-max_allowed_speed, math.min(max_allowed_speed,
lv_wf))
            rv_wf = math.max(-max_allowed_speed, math.min(max_allowed_speed,
rv_wf))

        -- Rule 3: No Right Wall Detected (e.g., at a corner, end of wall, or
sensor out of range)
        -- If the right sensor sees nothing, turn gently right to find the wall
again.
        else
            lv_wf = wall_follow_speed * 1.0 -- Left wheel normal speed
            rv_wf = wall_follow_speed * 0.3 -- Right wheel slower (causes a
gentle turn to the right)
            sim.addStatusbarMessage("WF: No wall on right, turning right to
search.")
        end

        sim.setJointTargetVelocity(left_joint, lv_wf)
        sim.setJointTargetVelocity(right_joint, rv_wf)

        -- === BUG 2 M-LINE RE-ENTRY CONDITION ===
        -- Check periodically if the conditions for returning to MOVE_TO_GOAL are
met.
        if now - last_recheck_time >= recheck_interval then
            last_recheck_time = now -- Reset the recheck timer

            -- Condition 1: Path directly in front of the robot is clear.
            -- Using a more generous distance for "clear" when considering re-
entry.
            local front_clear = (front_detected == 0 or front_dist > 0.6) --
Increased threshold for "clear" path
            -- if not front_clear then sim.addStatusbarMessage("M-Line Check:
Front NOT clear (dist: " .. string.format("%.2f", front_dist) .. ")") end

            -- Condition 2: Robot is closer to the goal than the initial hit
point.
            -- Add a small buffer to avoid oscillating exactly at the hit
distance.
            local closer_to_goal = (dist_to_goal < initial_dist_to_goal_at_hit -
0.05) -- Subtract small margin for stable re-entry
            -- if not closer_to_goal then sim.addStatusbarMessage("M-Line Check:
NOT closer to goal (curr: " .. string.format("%.2f", dist_to_goal) .. ", hit: "
.. string.format("%.2f", initial_dist_to_goal_at_hit) .. ")") end

            -- Condition 3: Robot has crossed (or is very close to) the M-line.
            -- The M-line is the straight line from the 'hit_point_pos' to the
'goal_dummy'.
            -- We use a 2D cross product to determine if the robot's current
position
            -- is "on" this line segment relative to the hit point.
            local vec_h_g_x = pos_goal[1] - hit_point_pos[1]
            local vec_h_g_y = pos_goal[2] - hit_point_pos[2]
            local vec_h_r_x = pos_robot[1] - hit_point_pos[1]
            local vec_h_r_y = pos_robot[2] - hit_point_pos[2]

            local cross_product = vec_h_g_x * vec_h_r_y - vec_h_g_y * vec_h_r_x
            local epsilon_m_line = 0.35 -- Increased tolerance for being "on" the
M-line (adjust based on robot size/path curvature)

            local on_m_line = (math.abs(cross_product) < epsilon_m_line)
            -- if not on_m_line then sim.addStatusbarMessage("M-Line Check: NOT
on M-line (cross_product: " .. string.format("%.2f", cross_product) .. ")") end

            -- Condition 4: Robot is roughly facing the goal.
            local angle_tolerance = math.rad(35) -- Slightly increased tolerance
for facing goal
```

```
            local facing_goal = (math.abs(heading_error) < angle_tolerance)
            -- if not facing_goal then sim.addStatusbarMessage("M-Line Check: NOT
facing goal (heading_error: " .. string.format("%.2f", math.deg(heading_error))
.. " deg)") end

            -- ALL CONDITIONS MUST BE MET TO RE-ENTER MOVE_TO_GOAL
            if front_clear and closer_to_goal and on_m_line and facing_goal then
                sim.addStatusbarMessage("?? M-line re-entry conditions MET!
Switching to MOVE_TO_GOAL.")
                state = STATE_MOVE_TO_GOAL
            else
                -- More detailed debugging message when re-entry is NOT met
                -- This will help you identify which specific condition(s) are
failing.
                local debug_msg = "M-Line Not Met: "
                debug_msg = debug_msg .. "F:" .. (front_clear and "Y" or "N") ..
(front_clear and "" or string.format("(%.2f)", front_dist)) .. " | "
                debug_msg = debug_msg .. "C:" .. (closer_to_goal and "Y" or "N")
.. string.format("(%.2f/%.2f)", dist_to_goal, initial_dist_to_goal_at_hit) .. " |
"
                debug_msg = debug_msg .. "M:" .. (on_m_line and "Y" or "N") ..
(on_m_line and "" or string.format("(%.2f)", cross_product)) .. " | "
                debug_msg = debug_msg .. "A:" .. (facing_goal and "Y" or "N") ..
(facing_goal and "" or string.format("(%.1fdeg)", math.deg(heading_error)))
                sim.addStatusbarMessage(debug_msg)
            end
        end
    end

    -- Optional: Display overall status in the status bar for quick debugging
    -- This message is always active and provides current state at a glance.
    sim.addStatusbarMessage(string.format("[%s] Dist: %.2f | F: %s (%.2f) | R: %s
(%.2f)",
        state, dist_to_goal, (front_detected > 0 and "Yes" or "No"), front_dist,
        (right_detected > 0 and "Yes" or "No"), right_dist))
end

-- ==============================================
-- sysCall_cleanup(): Called once when the simulation ends.
-- This function ensures a clean shutdown (e.g., stopping motors).
-- ==============================================
function sysCall_cleanup()
    sim.addStatusbarMessage("?? Bug 2 Script cleanup complete.")
    -- Ensure motors are stopped gracefully when the simulation ends if handles
are valid
```

```
    -- Check if joint handles exist (not nil) AND if they are valid handles (-1
means not found)
    if left_joint ~= nil and left_joint ~= -1 then
        sim.setJointTargetVelocity(left_joint, 0)
    end
    if right_joint ~= nil and right_joint ~= -1 then
        sim.setJointTargetVelocity(right_joint, 0)
    end
end
```

VISION SENSOR DEMO

```
-- Global variables for object handles
local red_cube_handle
local green_cube_handle
local blue_cube_handle

-- Movement parameters
local speed = -0.05 -- Keep as negative to move in the negative Y-direction
local reset_y = 1.0 -- Y-position to reset to when a cube goes past min_y
local min_y = -0.5  -- Minimum Y-position before reset

-- Called once when the simulation starts
function sysCall_init()
    -- Get handles for the red, green, and blue cubes
    -- These paths are based on your scene hierarchy in image_c38785.png,
    -- assuming 'rgb_dummy' is at the root and 'red', 'green', 'blue' are its
direct children.
    red_cube_handle = sim.getObject('/rgb_dummy/red')
    green_cube_handle = sim.getObject('/rgb_dummy/green')
    blue_cube_handle = sim.getObject('/rgb_dummy/blue')

    -- Error checking for handles (important for debugging)
    local all_handles_valid = true
    if red_cube_handle == -1 then sim.addLog(sim.verbosity_scripterr, 'Error:
"red" cube handle invalid. Check path: /rgb_dummy/red'); all_handles_valid =
false end
    if green_cube_handle == -1 then sim.addLog(sim.verbosity_scripterr, 'Error:
"green" cube handle invalid. Check path: /rgb_dummy/green'); all_handles_valid =
false end
    if blue_cube_handle == -1 then sim.addLog(sim.verbosity_scripterr, 'Error:
"blue" cube handle invalid. Check path: /rgb_dummy/blue'); all_handles_valid =
false end
```

```
    if not all_handles_valid then
        sim.addLog(sim.verbosity_scripterr, 'CRITICAL: Not all cube handles
found. Check console for details. Cubes will not move.');
    else
        sim.addStatusbarMessage("RGB cubes script initialized.")
    end
end

-- Function to handle movement and reset for a single cube
local function move_cube(cube_handle)
    if cube_handle == -1 then return end -- Do nothing if handle is invalid

    local pos = sim.getObjectPosition(cube_handle, -1) -- Get current absolute
position
    pos[2] = pos[2] + speed * sim.getSimulationTimeStep() -- Update Y-position
(pos[2] for Y-axis)

    if pos[2] < min_y then -- Check against min_y for Y-axis
        pos[2] = reset_y -- Reset Y-position if it goes past the minimum
    end

    sim.setObjectPosition(cube_handle, -1, pos) -- Set the new absolute position
end

-- Called at each simulation step
function sysCall_actuation()
    -- Call the move_cube function for each cube
    move_cube(red_cube_handle)
    move_cube(green_cube_handle)
    move_cube(blue_cube_handle)
end

-- Called once when the simulation ends
function sysCall_cleanup()
    sim.addStatusbarMessage("RGB cubes script cleanup complete.")
    -- Optionally, reset cube positions or stop them in some way if needed
end
```