**iCompetition Developer Manual**



**Mor Baruch**

**Gal Lerman**

**Avi Sahar**

**Doron Shaharabani**

Table of Contents

[1 The Robot 3](#_Toc334387095)

[1.1 Connecting to the robot 3](#_Toc334387096)

[1.2 Controlling the robot 3](#_Toc334387097)

[1.3 Distance Sensor 4](#_Toc334387098)

[2 Building the solution 4](#_Toc334387099)

[3 Solution Overview 5](#_Toc334387100)

[4 RobotDriver Project 5](#_Toc334387101)

[4.1 RobotInterface Class 5](#_Toc334387102)

[4.2 SensorData Class 6](#_Toc334387103)

[4.3 PhysicalRobot Class 6](#_Toc334387104)

[4.4 SimulatorRobot Class 7](#_Toc334387105)

[5 Scene Project 7](#_Toc334387106)

[5.1 Scene Class 7](#_Toc334387107)

[5.2 RobotMoverInterface Class 8](#_Toc334387108)

[6 SceneSolver Project 8](#_Toc334387109)

[6.1 SolverPRM Class 9](#_Toc334387110)

[6.2 LocationRobot Class 10](#_Toc334387111)

[7 RobotGui Project 10](#_Toc334387112)

[7.1 RobotGui Class 11](#_Toc334387113)

[7.2 Network and the MultiPlayerHandler Class 11](#_Toc334387114)

[7.3 Adding a New RobotMover 12](#_Toc334387115)

# The Robot

The iRobot Create is a complete robot development kit that allows you to control the robot’s behavior programmatically.

For the iRobot Create manual see the following link: <http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Manual_Final.pdf>

## Connecting to the robot

The iRobot Create can’t transmit or receive information wirelessly on its own, for this reason we used a Bluetooth communication device called BAM (the complete BAM manual can be found at <http://www.elementdirect.com/files/10542B.pdf>).

The BAM is connected to the matching slot in the robot’s cargo bay, uses only the serial input and output pins of the robot and exposes the rest.

The computer that runs the application controlling the robot should have a Bluetooth device. Use the Bluetooth management tool to discover the BAM, your operating system will dedicate a virtual COM port with an index (for example COM7) for this connection. **The BAM’s pairing code is 0000.**

The application connects to the robot using that COM port index.

## Controlling the robot

Controlling the robot is done using the iRobot Create Open Interface, which provides commands and sensor reading methods.

The robot has sensors such as bumper sensor, wheel encoders, battery charge and capacity. It keeps the data from the sensors internally and updates it every 15ms. This data can be accessed using sensor packet retrieval commands.

Our main usage of Open Interface:

1. The drive command provided us the ability to control robot translation, rotation and stopping the robot.
2. The stream command opens a stream of specified sensor packets which the application can read. After calling this command the robot will send the specified packets every 15ms.
3. The select song and play song commands are used to define and play the winner song and loser song (in competition mode when one of the robots reaches its goal it plays the winner song and the other one plays the loser song).

For the complete Open Interface documentation see the following link:

<http://www.irobot.com/filelibrary/pdfs/hrd/create/create%20open%20interface_v2.pdf>

## Distance Sensor

The iRobot create has a wall sensor on its right side. This sensor is intended to detect the distance of the robot from the wall so it will be able to drive along-side it. We found that this sensor detects only flat objects from a short distance and only from the right side, which is not the drive direction – making it not useful for unknown/dynamic obstacle detection.

Due to this, we have decided to make use of an independent distance sensor called MaxBotix EZ0. The EZ0 specification can be found at <http://www.maxbotix.com/documents/MB1000_Datasheet.pdf>.

The EZ0 has several output methods (serial, pulse width and analog). We decided to use the analog method because the BAM already uses the serial pins of the robot and because the robot has a special pin for connecting analog voltage signal.

The EZ0 encodes its distance measurement to an analog value between 0 and Vdd (in volts), where each inch of distance is encoded as an additional Vdd/512 volt. We used 4x 1.5 volt batteries resulting in Vdd of 6 volts.

The iRobot Create analog input discretizes the analog signal to 10 bits when the value 0 means 0 volts and the value 1023 means 5 volts. This implies that the robot’s maximal distance measurement decreased a bit (we use only 5 of the 6 volts) but it is completely acceptable to avoid unknown/dynamic obstacles.

# Building the solution

The following libraries and software should be installed prior to compiling the solution:

1. Visual Studio 2010.
2. CGAL 3.9.
3. Boost 1.47.0.
4. Qt 4.8.1.
5. Qt Visual Studio Add-in 1.1.11.

The following link includes instructions on how to install the libraries: <http://acg.cs.tau.ac.il/cgal-at-tau/installing-cgal-and-related-programs-on-windows>.

Please note that only the x86 version of the libraries should be used.

The following steps should be followed in order to build the solution:

1. Set the environment variable CGAL\_DIR to be CGAL root dir (for example, D:\Code \CGAL-3.9).
2. Set the environment variable BOOST\_PATH to be Boost root dir (D:\Code \boost\boost\_1\_47).
3. Open the solution file RobotWorkshop.sln.
4. Click build in the Visual Studio window.
5. The products will be located in RobotWorkshop\bin\Win32\Release for the release configuration or RobotWorkshop\bin\Win32\Debug for the debug configuration.

The following DLLs are required in order to run the application (either in bin or at the PATH):

1. libgmp-10.dll (Included with CGAL).
2. QtCore4.dll (Included with Qt).
3. QtGui4.dll (Included with Qt).
4. QtOpenGL4.dll (Included with Qt).

# Solution Overview

The solution is composed of the following projects:

1. RobotDriver – defines the robot interface and provides implementation for controlling the iRobot Create.
2. Scene – defines the properties of a scene, provides implementation for reading scene files and provides an interface for robot movers.
3. SceneSolver – includes our implementation of the robot mover interface based on the PRM algorithm.
4. RobotGui – a GUI project used for executing the game.

All four projects are written in C++. The RobotDriver, Scene and SceneSolver projects are compiled as static libraries. The RobotGui produces an executable file.

# RobotDriver Project

The RobotDriver project defines an interface that is used for controlling the robot. In addition, the project also contains two implementations of that interface – an implementation for controlling the iRobot Create and an implementation for the robot simulator.

## RobotInterface Class

The RobotInterface class defines an interface that should be used in order to control the robot.

The interface defines the following functions:

1. driveForward – Makes the robot drive forward at the provided speed.
2. turnClockwise – Makes the robot turn clockwise at the provided speed. The provided speed is actually at millimeters per second. In order to convert it to radians per second the following formula can be used: speed/wheel-base-radius.
3. turnCounterClockwise – Makes the robot turn counter clockwise at the provided speed.
4. stop – Makes the robot stop moving.
5. getSensorData – Returns a SensorData object containing the most current sensor read.
6. beHappy – Makes the robot celebrate (used after winning the competition).
7. beSad – Indicator that the robot has lost the competition.

## SensorData Class

The SensorData class contains information retrieved from the robot sensors.

The sensor information includes:

1. Battery charge of the robot.
2. Battery capacity of the robot.
3. Total distance travelled as retrieved using the robot’s wheel encoders.
4. Current orientation of the robot as retrieved using the robot’s wheel encoders.
5. An indicator for the robot’s bumper.
6. The distance to the closest obstacle as retrieved from the robot’s EZ0 sensor.
7. The iRobot Create’s internal wall sensor (The wall sensor is extremely limited and should not be used).

## PhysicalRobot Class

The physical robot class provides the actual implementation for the RobotInterface class and can be used in order to control the iRobot Create. The protocol for communicating with the iRobot Create is described in the iRobot Create Open Interface document.

The communication with the robot is performed using binary data transferred with the Win32 file methods – CreateFile, ReadFile and WriteFile. The connection is opened by using the CreateFile function along with the COM port of the iRobot Create BAM. The communication with the robot is done in an overlapped method – the reason for this is that the ReadFile and WriteFile would block each other unless this method is used, which will cause high latency when communicating with the robot.

The control methods of the robot driver (drive, turn, etc.) are implemented by sending the appropriate binary command (as described in the Open Interface protocol).

In order to retrieve the sensor data a special method called Sensor Stream is used. The sensor stream is activating by sending the robot the Activate Sensor Stream command which makes the robot send the sensor data to the computer repeatedly every 15ms. The sensor data must then be read fast enough so it does not overflow the internal buffer of the Win32 file implementation.

Since the sensor data is an infinite stream of data some special logic must be performed in order to sync to that stream. In order to sync to the stream we make use of two important features – the first is that every packet in the stream starts with a constant op code and the second is that the last byte in each packet is a checksum of the entire packet. Once we are synced to the stream we read the sensor data and store the most up to date values in the PhysicalRobot class. The actual sensor stream reading is performed in a dedicated thread.

## SimulatorRobot Class

The simulator robot is used for running the motion planning algorithm without moving an actual robot. The difficulty with creating a simulator robot is that the sensor readings of the simulator must imitate the sensors of an actual moving robot. For example, the total distance moved sensor must successfully simulate the total distance that a real robot would move according to the driveForward and stop functions. We achieve that by holding the internal status of the robot (translating, rotating etc.) and update the sensor data whenever it is requested.

# Scene Project

The Scene project contains two important classes – the Scene class that is in charge of reading the scene files and the RobotMoverInterface that is used as an interface to the concrete implementations of the motion planning algorithms.

## Scene Class

Every motion planning problem that is solved in our project is defined by a Scene.

A scene includes the following information:

1. The bounding box of the workspace.
2. The start point (including orientation) and end point (not including orientation) of the first robot.
3. The start point (including orientation) and end point (not including orientation) of the second robot (Only used in multi-player mode).
4. A list of obstacles, each of them defined by a list of points.

The Scene class is in charge of containing that information. In addition, it also contains the logic for parsing the scene files. For the exact format of the scene files please see the user manual.

## RobotMoverInterface Class

The RobotMoverInterface class defines an interface that any motion planning algorithm in the game must implement. Thus, if you would like to provide a new motion planning implementation you must implement this interface. Implementing this interface allows the algorithm to be used in both multi-player mode and single-player mode.

The RobotMoverInterface class defines the following abstract functions:

1. initialize – provides the robot mover with a robot interface (either a physical robot or a simulator) and the scene object. During initialize the robot mover should only assign those two parameters to its member variables.
2. preprocess – the preprocess phase is performed before the robot starts moving. During this phase, the robot mover should perform long-running operations that will ease the computation later on. A good example for such an operation is creating a PRM roadmap.
3. start – the start function indicates that the robot should start moving. The robot mover must send move commands to the robot interface and update its location accordingly. The purpose of the robot mover is to get the robot to its end position.
4. stop – indicates that the robot should stop moving.
5. getDynamicObstacle – the robot mover should return a polygon indicating the location of a dynamic obstacle. In case no dynamic obstacle is detected, an empty polygon should be returned.
6. getLocation – the robot mover should return the current location of the robot.
7. getOrientation – the robot mover should return the current orientation of the robot.
8. getRobotInterface – returns the internal robot interface object.
9. setRobotInterface – changes the internal robot interface object. Usually the robot mover is instantiated with a simulator robot, which is set to an actual physical robot before start is called.
10. getRoadmap – returns the roadmap that the robot mover uses for motion planning computation.
11. getCurrentPath – returns the current path that the robot mover uses in order to reach the end position.

# SceneSolver Project

The SceneSolver project contains an implementation of the RobotMoverInterface class. The implementation makes use of the PRM algorithm, which was modified in order to handle dynamic obstacles. The SceneSolver project contains two main classes – the SolverPRM class that is in charge of the geometric computations and the LocationRobot class that implements the RobotMoverInterface and is in charge of the dynamic algorithm

## SolverPRM Class

The SolverPRM class is in charge for solving the given scene using the PRM algorithm. We represent the robot as an approximation of a circle by a polygon with 16 edges (this is a constant that can be changed).

The SolverPRM class receives the static obstacles from an instance of the Scene class and calculates the forbidden space using the Minkowski Sum of the obstacles with the robot polygon. The result is a CgalPolygonSet called cforb\_ which represents the forbidden points in the scene.

We use the underling CGAL arrangement of this polygon set to check if points fall inside these forbidden areas using the trapezoid point location strategy.

The main methods in SolverPRM are:

1. isInCforb – checks if a point is in the forbidden area.
2. sampleLegalPoints – samples n random points in the free space (uses isInCforb()).
3. buildGraph – builds the road map from the legal points by trying to connect each legal point to its k closest neighbors with an edge. To determine if two points can be connected we sample the line between the points frequently and check if all these sampled points are in a legal area using isInCforb.
4. query – finds a path from source point to goal point by connecting both points to the road map (using the method used when building the graph) and then performing Dijkstra’s shortest path algorithm on the modified graph.
5. smooth – This method tries to make the path returned by the query smoother. This works by spanning the points in the path and trying to connect points directly to the farthest point possible in the path (to avoid paths with a large number of edges). However paths that are too smooth (i.e. paths containing long lines) result in a larger error when it comes to positioning the robot. Due to this, we restrict the smooth method to a maximal length per edge in the smoothed path.
6. queryWithObstacle – This method is used to avoid unknown/dynamic obstacles in the robot’s path. It adds a polygon representing the obstacle to a local copy of cforb\_, acquires a new point location object and queries the graph using the current robot location as the new source and the new point location object. The new point location object guarantees the new source location will be connected to the graph by edges that won’t intersect the dynamic object polygon. However since the road map was built without this polygon in mind, there might be some edges in the new path that intersect the dynamic obstacle polygon. Due to this, after the query returns a path we have to check if all its edges are free with respect to the new cforb\_. If an edge in the path is found to be intersecting the polygon we temporarily remove this edge from the roadmap and query again. This is repeated until either a legal path is found or we exhaust all possible routes.

## LocationRobot Class

The LocationRobot class implements the RobotMoverInterface class. This class is in charge of controlling the robot movement and updating the GUI regarding the calculated location and orientation of the robot according to the robot’s sensors.

This class keeps track of the robot’s location and orientation based on its start configuration and accumulated sensor reports.

Important fields:

1. robot\_ - An instance of the RobotInterface which is a representation of the robot to be controlled by LocationRobot.
2. solverPRM\_ - An instance of SolverPRM which LocationRobot uses in order to find paths for the robot.

Important Methods in the class:

1. startRotating – computes the angle that the robot should rotate based on its current orientation and its next position’s orientation to determine if the robot should rotate clockwise or counter-clockwise (the robot aims to rotate in the direction that would result in the minimal rotation angle) and sends the command to the robot.
2. startTranslating – sends a drive forward command to the robot.
3. onPathUpdate – uses the newOrientation\_ and newLocation\_ fields as tentative goals for the robot and updates the robot current location and orientation based on sensor data. If the robot is currently rotating it decides whether the orientation that is required to reach its next position was reached (within a certain margin of error). In case it was reached it stops the rotation and starts translating to the newLocation\_, otherwise continues rotating. If the robot is currently translating it decides if the newLocation\_ is reached (within a certain margin of error). If it was reached the robot stops translating and returns, otherwise continues translating.
4. run - the run function is initiated when the start function is called. The run function calculates a path to the end position. It then iterates all points along the path. For each point startRotating will be called so that the robot begins the sequence to move to that point. The algorithm will then enter the onPathUpdate loop which will keep track of its position and will return when the desired position it reached. The robot will perform these operations until the last position is reached.

# RobotGui Project

The RobotGui project wraps all three other projects under a graphical user interface. It enables the user to run games either in single-player mode or multi-player mode. It also contains the network code required for coordination of the multi-player mode between the different players and the referee.

## RobotGui Class

The RobotGui class is the main GUI class in the project. It contains implementations for all GUI elements in the main form. For example, upon clicking the Load Scene button the code in the RobotGui class will use the Scene class in order to load the scene.

In addition to management operations, such as loading a scene or solving it, the RobotGui class is also in charge of drawing the entire scene (including the moving robots). This is performed by a special timer that ticks every 200ms. Upon a tick the local Scene and RobotMover objects are examined, and the entire view is rendered according to the information. The 200ms updates in fact create the required animation.

The RobotGui class supports three modes – single player, competition player (one player in multi-player mode) and competition referee (manages the two players in multi-player mode).

The single-player mode loads a scene file, creates the default robot mover (LocationRobot that is implemented in the SceneSolver project). When the solve button is clicked the robot mover’s preprocess function is called, which generates a PRM roadmap. When the start button is clicked the robot mover’s start function is called, which makes the robot start moving. The RobotGui repeatedly sample the robot mover for the robot’s location and updates the GUI accordingly.

The competition player mode creates a TCP listener, which will be explained later on. The listener waits for a connection from the referee. When a connection is established with the referee, the referee pushes the scene data to the player, calls preprocess for both robots and then starts the competition by issuing a start command for the two robots. In order to display both robots in a single GUI, the referee polls the two robots for their location at the main update loop. It uses the information from the two robots in order to update its GUI. The referee also compares the robots’ locations to their end position. When the referee detects that a robot reached its end position it orders the two robots to stop, and then orders the winning robot to celebrate by calling the beHappy command.

## Network and the MultiPlayerHandler Class

The multi-player mode requires that the two players and the referee communicate. The communication includes the scene data that is transferred from the referee to the players, the preprocess and start commands, the location updates and the win notification. In order to perform these communications, the two players acts as a TCP server that waits for a connection from the referee. The referee is to initiate this connection and send commands to the players. It is important to understand that the two players only answer the referee requests and does not initiate requests to the referee.

The network communication in the project is done using the Apache Thrift library. The Apache Thrift library provides a special compiler that given a thrift interface (using a special thrift syntax) generates the entire network code. Thus, the files in the Thrift directory were automatically generated using the thrift compiler and the RobotWorkshop\RobotGui\Robot.thrift file. The automatically generated files contain both a client and server implementations. The client implementation can be used as-is in order to invoke the code at the server side. The server side implementation requires that we implement the interface that is defined in the thrift file.

The MultiPlayerHandler class implements the thrift interface. Since the competition player (and not the referee) is running the TCP server, the implementation is simply a wrapper around the robot mover.

The MultiPlayerHandler class wraps the following functions:

1. setScene – initializes the robot mover with the scene data.
2. preprocess – calls the robot mover’s preprocess function.
3. start – calls the robot mover’s start function.
4. getLocation – returns the location from the robot mover.
5. stop – initiates stop at the robot mover.
6. beHappy/beSad – calls the robot mover’s beHappy/beSad functions.

In case more data is needed from the competition player, or new commands should be passed to the player, one should edit the Robot.thrift file, generate new .cpp and .h files using the thrift compiler and add the required functionality to the MultiPlayerHandler class.

## Adding a New RobotMover

Currently the default robot mover (LocationRobot) is hardcoded into the GUI. In order to switch the default implementation, the LocationRobot allocation in onStartListeningButtonClick (multi-player initialization) and onLoadButtonClick (single-player initialization) should be changed to an allocation of the new RobotMover deriving class.