Software Project Summer 2005

Robots Battle

Overview

The objective of this project is to write a program that simulates a battle between two robots. The simulation takes place on a rectangular ring. The robots are programmed by the players of the game with the goal to gain a victory upon their enemy. Thus, each robot has a program that controls it (a strategy), this program is written in a language that is described in Section 1.1. Your simulator should run the robot's-strategy programs in parallel, taking care of geometric constraints in the game. The output of the simulator is the description of the battle/s (see below).

A robot is represented as a triangle; it can fire, move and turn. In addition there are a few sensors for the robot that allow it to avoid collisions (e.g., with the ring boundary) and laser beam hits. The sensors are values of variables.

At the beginning of the battle each robot has a predefined number of life-points. When a robot is hit by a laser beam, it looses life-points. The game continues till one of the robots ends with ≤ 0 life-points.

Your program is composed of two major parts (the two parts are of different programming complexity): one part understands (parsing) the robot programs and executes them and the other part simulates the flow of the game (collisions, motion animation).

1 Robot Strategy Programming-Language

1.1 Language Overview

The robot programming language is an event driven language. The program is built from a set of routines; these routines are called by the system in response to events that occur. For example, the OnEastWallCollisionAlert event is called when the robot is too close to the east wall. As a result, the current program of the robot halts and a special function that handles this event is called (perhaps start moving in a different direction in this case).

A robot program has the following properties:

- Every line holds at most one command (or it may be empty, or contain a comment).
- Tabs and spaces are ignored. Many spaces and tabs are the equivalent to one space.
- Any character after a '#' is ignored till the end of the line, i.e. the '#' is used for remarks, similar to // in C++.
- The language is case insensitive, this means that upper and lower case are ignored. The following strings are equivalent in the language: "Hello", "hello", "HELLO" and "HelLo".

Strategy programmers can define variables. A variable holds real values (i.e., int float or double). Variables are always global. A variable is initialized when it is first used and cannot be deleted (the value of variables is 0 by default). In addition, there are several built-in variables, these are sensors of the physical world, for example the variable "$RobotDirection" holds the current direction of the robot. Built-in variables are read-only, they cannot be changed by the robot program.

Strategy programmers can also define routines. Each routine doesn't return a value (equivalent to void in C) and it doesn't get any parameters (note that this is not the case with build-in routines where they may have parameters).
Function names are of the following format: [A–Z]([A–Z]—[0–9])* like Compute.
Variable names are of the following format: $[A–Z]([A–Z]—[0–9])* like $RobotDirection.

One routine can call another routine using the following syntax: "FunctionName()".
Here are sample routines:

```
# Computes the square of the variables $dx , $dy


# and stores the results in $dx2 and $dy2 respectively
#
SQR2
{
    $dx2 = $dx * $dx
    $dy2 = $dy * $dy

}
DistanceFromEnemy
{
       $dx = $CurrentPositionX - $CurrentEnemyPositionX
       $dy = $CurrentPositionY - $CurrentEnemyPositionY
       SQR2()
       $d2 = $dx2 + $dy2
       $distance = sqrt($d2)
}
```

When a function is called or registered (defined below) it should already be defined. This means that you cannot call functions that appear below the current function definition, furthermore, recursion is not allowed.

Every program begins execution with a special routine called main. If the main routine exits, the robot can only react to events that were registered earlier (events and event registration are explained in 1.6).

1.2 Sample program

```
RandomTurn
{
   $direction = random(0,359)               # built-in
   Turn($direction)
}

OnNorthWallCollisionAlertHandler
{

    Turn(180)
}

OnEastWallCollisionAlertHandler
{
    Turn(180)
}

OnWestWallCollisionAlertHandler
{
    Turn(180)
}

OnSouthWallCollisionAlertHandler
{
    Turn(180)
}

#User defined routine
```

```
RotateAndFire
{
    RandomTurn()           #User Defined
    Fire()                 #Built-in routine
}

OnEnemyCollisionAlertHandler
{
     RotateAndFire()
}


OnLaserAlertHandler
{
    Turn(30)
    Move(10)
}

MyIdleStrategy
{
$distance = 10
while($CurrentLifePoints > 2)
    Move($distance)
    RandomTurn()
    Fire()
endwhile

while($true)
   Turn($distance)
   Fire()
endwhile
}

main
{
#Register Events (Built-in Functions)
  RegisterOnNorthWallCollisionAlert(OnNorthWallCollisionAlertHandler,1)
  RegisterOnEastWallCollisionAlert(OnEastWallCollisionAlertHandler,1)
  RegisterOnWestWallCollisionAlert(OnWestWallCollisionAlertHandler,1)
  RegisterOnSouthWallCollisionAlert(OnSouthWallCollisionAlertHandler,1)
  RegisterOnEnemyCollisionAlert(OnEnemyCollisionAlertHandler,2)
  RegisterOnLaserAlert(OnLaserAlertHandler,3)

#Main Strategy --- its priority is the smallest by default
  RegisterOnIdle(MyIdleStrategy)
}
```

1.3 Robot Built-In Variables

Variable	Description
$RobotWidth	The width of the robot. It is a constant defined in the source code.
$RobotHeight	The height of the robot. It is a constant defined in the source code.
$Direction	The angle between $[0, 360)$ where direction with 0 angle overlaps with the Y positive axis.
$CurrentPositionX	The current x coordinate of the robot's center position (the center of mass of the triangle) in world coordinates.
$CurrentPositionY	The current y coordinate of the robot's center position in world coordinates.
$NorthWallDistance	The distance of the robot's center from the north wall. Note that if the center is outside the ring, this distance is negative.
$EastWallDistance	The distance of the robot's center from the east wall. Note that if the center is outside the ring, this distance is negative.
$WestWallDistance	The distance of the robot's center from the west wall. Note that if the center is outside the ring, this distance is negative.
$SouthWallDistance	The distance of the robot's center from the south wall. Note that if the center is outside the ring, this distance is negative.
$CurrentEnemyPositionX	The current x coordinate of the enemy-robot's center position in world coordinates.
$CurrentEnemyPositionY	The current y coordinate of the enemy-robot's center position in world coordinates.
$CurrentEnemyDirection	The current enemy direction.
$CurrentLifePoints	The current life points of the robot.
$CurrentEnemyLifePoints	The current life points of the other robot.
$EnemyLaserBeamPositionX	The current x coordinate of a laser beam in world coordinates. This variable can be used only if there is an enemy laser beam in world coordinate frame, (i.e., only if the robot has been strike by its enemy).
$EnemyLaserBeamPositionY	The current y coordinate of a laser beam in world coordinates. This variable can be used only if there is an enemy laser beam in world coordinate frame, (i.e., only if the robot has been strike by its enemy).
$EnemyLaserBeamDirection	The direction of the laser beam in world coordinates. This variable can be used only if there is an enemy laser beam in world coordinate frame. (i.e., only if the robot has been strike by its enemy). See more details on "laser beam" in D.
$IsEnemyLaserDataValid	It is a predicate. Its value is $true (i.e., 1) if $EnemyLaserBeamPositionX, $EnemyLaserBeamPositionY and $EnemyLaserBeamDirection may be considered valid.
$true	Constant 1.
$false	Constant 0.

Note: Directions are in degrees. Both positive (turn right) and negative (turn left) values are allowed.

1.4 Constant build-in Variables

Variable	Description
$EnemyCollisionDistanceAlert	If the distance between the two robots' centers is $\leq$ $EnemyCollisionDistanceAlert, OnEnemyCollisionAlert event will be triggered. (in the source code: #define ENEMY_DISTANCE_ALERT ...)
$WallDistanceAlert	If the distance of the robot's center from a wall is $\leq$ $WallDistanceAlert, an appropriate event will be triggered (i.e., OnXWallCollisionAlert (in the source code: #define WALL_DISTANCE_ALERT ...)
$LaserDistanceAlert	If the distance of the robot's center from the enemy laser beam $\leq$ $LaserDistanceAlert, the OnLaserAlert event will be triggered. (The laser beam is a segment with two endpoints, if one of this endpoints is within this distance from the robot, the event OnLaserAlert will be triggered). (in the source code: #define LASER_DISTANCE_ALERT ...)

1.5 Built In Commands

Command	Description
Move(distance)	Move in the current direction of the robot by the specified distance and stop. distance may be an integer or a variable of type int (see Section A for more details).
Turn(angle)	Rotate the robot by the desired angle $[0-360)$ around its center. The angle is in degrees. This is a relative angle with relation to the direction of the robot. Positive angle means turn right and Negative angle means turn left. Angle is of type int.
Fire()	Fire laser beam from the center of the robot's current position in current robot's direction (see A for more details).
print("string")	Prints the string to the screen. A string is composed of alphanumeric and space characters inside double quotes. There is not a way to print a double quote. Print is used to debug robot programs.
print(variable)	Prints value of this variable to the screen.
exit()	Stop the simulation and exit immediately, this command is used for debugging.
RegisterOnNorthWallCollisionAlert(function, priority)	When the distance between the center of the robot and the north wall is too close (i.e., $\leq$ $WallDistanceAlert), "function" is called.
RegisterOnEastWallCollisionAlert(function, priority)	When the distance between the center of the robot and the east wall is too close, "function" is called.
RegisterOnWestWallCollisionAlert(function, priority)	When the distance between the center of the robot and the west wall is too close, "function" is called.
RegisterOnSouthWallCollisionAlert(function, priority)	When the distance between the center of the robot and the south wall is too close, "function" is called.
RegisterOnEnemyCollisionAlert(function, priority)	When the distance between the two robots is too close (i.e., $\leq$ $EnemyCollisionDistanceAlert), "function" is called.
RegisterOnLaserHit(function, priority)	When the robot is hit by an enemy laser beam, "function" is called.
RegisterOnLaserAlert(function, priority)	When an enemy laser beam is close to the robot, "function" is called. (i.e., when its distance from the robot is $\leq$ $LaserDistanceAlert).
RegisterOnIdle(function)	Set the idle function (see more details in section 1.6). It doesnt get a priority since its priority is the smallest by default.

if (<value> | <BinOp>)
else
endif
	Control the flow of a program. < value > is a constant or a value of a variable. A 0 value means false, any other values are true. The else is optional. BinOp is defined in 1.7.

while (<value> | <BinOp>) endwhile	Continue to execute the statements between the while and endwhile as long as value / BinOp expression ! = 0. BinOp is defined in 1.7.
<variable>=<exp> i.e. $<name>=<exp>	Assign a value to a variable. For example: $x=$y+5

Notes:

1. Any parameter to Move and Turn command can be a constant value or a value of a variable. Expressions are not supported. This means that the following are legal:
   Turn(30)
   Turn($Angle)
   But Turn($angle+34) is not legal.
2. All angles are in degrees in the range of $[0,360)$. If the user input included a number outside that range, the number should be normalized to the $[0,360)$ range.
3. The Move distance-parameter is of type integer. If not, you should normalize it to integer (e.g., Move(2.6) is equivalent to Move(2)).

1.6 Events

When the robot is too close to the wall, or approaches another robot or going to be hit by a laser beam, an event occurs. As a result, a function that handles the event is called.

Events have priorities. If a high priority event occurs while a lower priority event is running, the lower priority event handling routine is terminated and the high priority event handling routine is executed. When the high priority event finishes, the program turns to idle. If the same event occurs while an event is running, the event is ignored (i.e., the earlier event keeps running). If the opposite happens, i.e., a low or equal priority event occurs while a high priority event is running; the lower priority event is discarded. Priorities are numbered from 1 to infinity, where 1 is the highest priority.

Here is the list of possible events:

Event	Description
OnNorthWallCollisionAlert	Indicates that the distance between the robot's center and the north wall is too close, i.e., the distance between the robot and the north wall is less than $WallDistanceAlert.
OnEastWallCollisionAlert	Indicates that the distance between the robot's center and the east wall is too close, i.e., $\leq$ $WallDistanceAlert.
OnWestWallCollisionAlert	Indicates that the distance between the robot's center and the west wall is too close, i.e., $\leq$ $WallDistanceAlert.
OnSouthWallCollisionAlert	Indicates that the distance between the robot's center and the south wall is too close, i.e., $\leq$ $WallDistanceAlert.
OnEnemyCollisionAlert	Indicates that the enemy is too close to the robot and may collide with it, i.e., the distance between their centers is less than $EnemyCollisionDistanceAlert.
OnIdle	Idle is an event that has priority that is smaller than any other event. If registered it is called when no other event is running.
OnLaserAlert	An event which is called when an enemy laser beam is close to the robot. i.e., the laser beam is within $LaserDistanceAlert from the robot (see Section 1.4).
OnLaserHit	An event which is called when the robot is hit by an enemy laser beam.

Notes:

- When the event occurs, the registered routine will be called with the specified priority.
- If a Move command is interfered by an event, the Move command is terminated, namely, after the execution of this event the simulation will proceed from the next command line.
- Registering an event means calling one of the RegisterOnX(function, priority) commands, which assigns a specific routine to the event. When the event occurs, the registered routine will be called with the specified priority.
- Any event can be registered, but it is not required. If an event is not registered, but the event occurs, the flow of the program does not change, i.e., the event is ignored.
- A robot program starts executing with the function main, typically, the main routine will initialize variables and register events, by calling RegisterOnX (like RegisterOnIdle). The priority of main is higher than any other event; this means that no event will be called until main exits.

1.7 Expressions

Numbers in the language are floating point of the form:
$[-]N[.N]$, that is: an optional '-' sign, followed by an integer, followed by an optional .integer.
All of the following are legal: 1, -2, 3.1, -543.123
Formally, expressions are of the form:

exp : Value | BinOp | Func | Func2	
Value : <number> | <variable>	number, as defined above. <variable> is the value of the variable.
BinOp : <Value> <op> <Value>	<op> is any of the legal operators, i.e. +, -, *, etc.
Func : <Function>(<Value>)	function is any of the functions defined below, such as sin, cos, etc.
Func2 : <Function>(<Value>, <Value>)	A function of two variables, like random and atan2

The legal operators are: +, -, *, /, <, >, <=, >=, ==, !=

Supported functions: sin, cos, acos, sqrt, tan, atan and floor (e.g., floor(5.43) returns 5). In case of an illegal value, the function should return 0.
random(low, high) returns an integer random number in the range [low, high] (low and high should be integers as well, otherwise, you should normalize them).

atan2 is like the function with the same name in the standard library.
Note: All these functions except for random and floor may get and return numbers of type double.
For example the following are legal expressions:

```
3.45
$v2
$v1 + $v2
$v2*4.5
sqrt(3)
sqrt(-3)
```

But these are not legal expressions:

```
.76
1+2+3
```

1.8 Language syntax

Formally, the language syntax is as follows:

Program	$\longrightarrow$	FunctionName { commands }	
commands	$\longrightarrow$	oneline commands	
oneline	$\longrightarrow$	singlecommand NEWLINE |	NEWLINE is a mark of end of line from the lexical analyzer
		NEWLINE	comments are ignored by the lexical analyzer
singlecommand	$\longrightarrow$	exp |	as defined in 1.7
		Move(distance) |	
		Turn(angle) |	
		FunctionName ( )|	Function call to FunctionName
	...		

2 Simulation

The simulation is performed on a two dimensional RING_WIDTH $\times$ RING_HEIGHT ring, the bottom-left corner of the ring is coordinate $(0,0)$. Angles are clockwise, that is Turn(90) means turn to the right. The game input will be two strategy files for each robot. Each robot will try to gain a victory using the strategy defined in its file.

The simulation is working in time steps (also called clock ticks). The project header file defines how many such steps are performed at every second. At every tick, each robot executes one instruction (one command line). A laser beam may also move at a clock tick depends whether one of the robots used the Fire() command. By executing one instruction of each robot (e.g., moving the robots a little), we simulate robots that run in parallel.

2.1 Rules of the game

The initial positions and directions of the robots are picked at random so that the robot must be within the ring (this is already implemented in RobotBattleGr.h).

Each robot starts with predefined life points (see below). When a robot collides with one of the walls of the ring (i.e., one of the triangle edges intersects one of the walls) but its center position is inside the ring, it losses 1 life-point and continues its strategy. However, if the center position of the robot is outside the ring this battle stops and the other robot is declared as the winner regardless its life-points. If they both left the ring in the same clock tick (most unlikely), the game ends with a tie.

When two robots collide, they must be set in a new positions randomly and continue the game. However, in this case they both loose 2 life-points.

If a robot is being hit by a laser beam (namely the laser beam segment intersects one of the robot edges), it losses 1 life-point and the game continues.

The battle continues till one of the robots (or both) has ≤ 0 life-points or one of the robots (or both) has left the ring or (if none of this has happened) after a user-defined clock ticks (set to 2000 cycles as default). If the battle was stopped due to clock ticks constrain then the winner is the one with the higher number of life-points. If they have the same life-points, the battle ends in tie. Also, in the case where they both left with ≤ 0 life points we say that the battle ended in tie.

Please pay attention to Move and Fire built in commands. Move($distance) instruction is last until the robot has moved $distance from its position or until this command was interrupted by an event. The robot is not allowed to pass to the next command line till Move instruction is not finished. **Fire command can be executed only after the last laser beam is vanished.** If there is a Fire() command while the last laser beam has not vanished this command will be ignored. The Fire command terminates when the laser beam has hit the other robot or when it's exceeded its hit range.

Note: The robot will continue its strategy after Fire() command, however, all the following Fire commands will be ignored till the last laser beam is vanished. See more details on how to simulate the Move and Fire commands in A.

2.1.1 Simulation Mode

There are two kinds of simulation you need to support:

- **Regular simulation**, in this mode the simulation is executed regularly as described above. This is the default mode. The program should report the description of the battle/s (see below).
- **Single step simulation**, in this mode every instruction that is executed is reported on screen. The user should press <enter> to continue to the next instruction. A report should include: Robot's name, line number and the line that is executed.

2.1.2 Other Simulation Features

Graphical output using the RobotsBattle library, can be found on the webpage.
The command line for the simulation program is:
"startbattle [-c <cycles>] [-s] [-n <number of battles>] <robot's name > <its strategy file name > <robot's name> <its strategy file name>"

-c <cycles>	(optional) Set the maximal number of clock ticks for one battle. The default is 2000.
-s	(optional) run the simulation in a single step mode.
-n <number of battles>	(optional) sets the number of battles between these two robots. The default is 10. (remember that in each battle the positions and directions of the robots are picked at random).

For example: "startbattle Megatron Megatrone.str BumbaleBee BumbaleBee.str " means run in regular mode. There will be 10 battles, each battle may continue for 2000 clock ticks at most. The strategy files end with the extension '.str'

2.1.3 Output

The output of the simulation is as follows: In case of an error (usually during parsing) output the following line and exit the program. Parse errors should include the line number where the error occurred.

```
printf("Error Robot File Name:%s line:%d, Description:%s\n", robot_file_name, line_number, reason);
```

Each battle simulation output should be appended to BattleScore.res file as follows:

```
Battle Duration: <time duration of the battle in clock ticks>
Battle Winner:   <Robot Name>
Battle Score:    <Robot A Life points> <Robot B Life points >
Battle Strategy: <Robot A Strategy File Name> <Robot B Strategy File Name>
Battle Remark:   <cause of winning [Enemy Destruction | Enemy Out of Ring | Time out]>
```

In case of a tie, the two robots' names will appear in the "Battle Winner" line.
Your simulation should execute #n battles (defined by n <number of battles>) on the strategy files (each time the robot positions are picked at random) and in each time you should append the battle details to the file BattleScore.res as shown above. At the end of this file you should write "The final winner after #n battles is <robot's name>" where the final winner is the robot with the highest number of victories during the #n games (regardless life points). If equality occurs you should write "The game ended in a tie".

3 The characteristics of the game world coordinate

There are some predefined constants for the game which are specified in robotBattleDefs.h (can be found in the project webpage). You should use this configuration in your project. You may not add any values to this file, only change the values to test your project. During the checking of the project, these values may change.

- RING_WIDTH //OpenGL Window Width
- RING_HIGHT //OpenGL Window Height
- ROBOT_WIDTH
- ROBOT_HEIGHT
- WALL_DISTANCE_ALERT
- ENEMY_COLLISION_DISTANCE_ALERT
- LASER_DISTANCE_ALERT
- ROBOT_STARTING_LIFE_POINTS

4 Requirements

The project grade is made of these parts:

1. Initial design document. 5% (not checked)
2. Working makefile 5%
3. A working simulator. 50%
4. Documentation (described below) 30%
5. A non trivial (100 lines of code, not including the comments) program for a robot strategy. 10%

To pass the course, the simulation must work.

5 Programming Guidelines

The coding quality is an important part of this project. Following are some general guidelines for correct coding. Try to follow these guidelines, as well as employing any additional knowledge you have in writing software.

- Modularity. Related functions should reside in the same file, while unrelated functions are to reside in different files.
- Functionality. Each function should perform one *thing*. This *thing* should be the function name. Functions should be short and clear, function length should not exceed one page. Any repeated code parts should be concentrated in functions.
- The maximum line length in your code should not exceed 80 characters.
- Documentation. The code should be clear. Therefore, short comments should appear in various places, such as preceding functions or complicated blocks, when defining variables, etc. If there are assumptions regarding the input / output of the functions, they should appear too.
- Naming conventions. A significant part of the documentation resides in the code itself. Proper name choosing is one way of doing it. The names of the variables and functions should imply their role. Each group of names should have the same convention. For example, function names can have capital letters leading each word in the name, variables can use lowercase characters with underscores between words, and constants can use all capital letters: GetNextLine (line_buffer, LINE_LEN);

- Simple data-structures, algorithms and implementations are preferred.
- It is recommended to handle data structures only via dedicated functions. For example, if there is a linked list used in some places, and some algorithms go serially over the list, it is a good idea to have functions such as GetFirst and GetNext. The lists' elements should be accessed only via these functions.
- No compilation errors or warnings are permitted.
- Obviously, no run time errors are acceptable.
- A working project is more important than a fancy project. Make sure the basic features that are required here are working before adding anything to the project. Also, a not so efficient working project is much better than a very efficient non-working project.

6 Documentation

In addition to the code, you should also write a short description of the project. The documentation file should describe the entire project. It should be clear and built of nicely separated sections. Its length should be about 10 two-sided pages.
The idea is that when someone else (beside the author of the project) reads the documentation, he will be able to understand, modify and extend your project.

The documentation should include:

- Header - including the same information as in the partners (see directory structure section below) file.
- A list of all the files in the project, and a short (one line) description of each one of them.
- The modules of the project and their relationships.
- A general description of the project logic and the application flow.
- A list of all the data structures in use, together with a short explanation why they were chosen. Include a description of the alternatives to the decision, and list the pros and cons of each alternative.
- For each data structure, a short time complexity analysis for each of the implemented actions.
- Comments and suggestions about the project.
- The documentation should be printed in English.

The documentation is not part of the code.

7 Administrative Information and Rules

- You should write the project in "C", or if you wish, "C++". The project should run on UNIX machines in class Schriber-04.
- Due: Sunday 11-September-2005, 10am.
- You **must** work in groups of **two** people.
- On the second class (one week after the semester begins), submit an initial design document of the project (1-2 pages). This document should include the participants' names, main modules and relations between them, and data structures that will be used.
- **You should check the web page once a week for any updates on the project. Some of the definitions of the project may change. Pay special attention to the forum. The questions and answers that are posted in the forum complement this document.**
- After you submit your project, check your email regularly (at least once a week), you should expect a message with your grade.

7.1 Directory Structure

- There should be one directory under your account, named "RobotsBattle05" (capital letters) with no sub-directories.
- The directory should contain a file named "partners" that includes:
  One line for each team member with his/her login name, ID number, and full name separated with white spaces.
  The files of the project should reside in the directory of the team member in the first line. The other member should have the same directory that includes only the partners file.
- All of the files (source, header, makefile) should be under that same directory.
- There should be a working makefile named "Makefile". The makefile should generate the executable of the project.
  In addition, there should be a "clean" target in the makefile that will remove the object files from the directory.

7.2 Before you submit, make sure

- There is no debug output to the project, unless it is for debug purposes, disabled by appropriate #if clauses.
- Your project works on the department's UNIX systems, i.e., UNIX machines in the lab.
- Your documentation includes the general description, as explained in the documentation section.
- Your code is well commented, as explained in the Programming Guidelines section.
- The project does not depend on any specific information in your account. A good idea is to copy the RobotsBattle05 directory temporarily to another account and compile it there.

7.3 What to submit

- A hardcopy of the documentation.
- The electronic version of your work should reside in the proper location, untouched after the due date.

8 Grading Criteria

- Project submitted on time. Projects that are not submitted on time will not be checked.
- Fully functional according to the specifications
- The project compiles flawlessly
- Code quality
- Documentation
- Reasonable efficiency, make a tradeoff between implementation time and efficiency. Write your considerations in the documentation.
- Initial design submitted on time.
- It is okay to share ideas with other groups, but **it is forbidden to share code**.

Some remarks on getting a good grade

- Start working on the first day. The end of the semester is more stressed than the beginning.
- If something is not clear, ask.
- Do not leave things to the last minute. Make sure you have a running project ahead of time and test it.
- The documentation is an important part of the project. Write it when you finished coding and not the night before you submit the project.
- Work with a team member you trust, check that both of you are on the right track every once in a while.

A Simulate Move(distance)/Fire() Using Interpolation

In order to simulate the motion (Move(distance) instruction) of the robots and the laser beam movements you need to use linear interpolation as follows:

Laser-Beam display.

A laser beam is defined by a point and a direction. The canonical laser beam is defined by $(0, LASER\_BEAM\_LENGTH)$ and direction 0.
You need to calculate the final laser beam position according to the current robot's position and direction (i.e., the position and direction it fired from) and update it each clock tick. To do this, you need to use some of the following constants which are defined in RobotsBattleDefs.h file attached to the project.

```
#define LASER_BEAM_LENGTH ...
#define LASER_BEAM_DURATION ...  // Laser Beam Distance To Pass
#define LASER_SPEED_FACTOR ...   // number of pixels each clock tick
#define LASER_TIME_STEPS  1.0/LASER_BEAM_DURATION * LASER_SPEED_FACTOR
// the fractional distance the laser moves in each clock tick
```

You can calculate continues movements of the laser beam using these variables as follows:

```
// center coordinates of the robot
FireStartPositionX = Robot.x
FireStartPositionY = Robot.y
FireStartDirection = Robot.direction
// Laser (X,Y) destination coordinates from the (0,0)
LaserX = LASER_BEAM_DURATION*sin(FireStartDirection *PI/180)
LaserY=  LASER_BEAM_DURATION*cos(FireStartDirection *PI/180)
time  = [0.0,1.0]
```

In each clock tick, the laser beam position is defined by the following lines:

```
While(time < 1){
      time+=LASER_TIME_STEPS
      laserBeam.x =  FireStartPositionX + LaserX * time
      laserBeam.y =  FireStartPositionY + LaserY * time
}
```

When $time >= 1$ the laser beam disappears.

Robot's Motion display (similar to laser beam interpolation).

Same as the laser beam interpolation. Here, you need to use the following constants which are defined in RobotsBattleDefs.h file attached to the project.

```
#define MOVE_SPEED_FACTOR  //number of pixels for each clock tick
#define MOVE_TIME_STEPS(d)  1.0/d * MOVE_SPEED_FACTOR
```

Suppose you execute Move($dist) command line. Then you should use the following code.

```
// center coordinates of the robot
StartPositionX = Robot.x
StartPositionY = Robot.y

// (X,Y) destination coordinates after moving by dist with respect to the (0,0)
dest_X = dist*sin(robot.direction*PI/180)
dest_Y=  dist*cos(robot.direction*PI/180)
```

```
time  = [0.0,1.0]
While(time < 1){
     time += MOVE_TIME_STEPS(dist)
     Robot.x =  StartPositionX + dest_X * time
     Robot.y =  StartPositionY + dest_Y * time
}
```

B Intersection of line segments may help you to find collisions

An efficient way of checking whether two segments intersect each other is using left-turn-tests. Three points in the plane A, B, C are defined to be a left-turn if the counter-clockwise angle from the vector $\vec{v} = B - A$ to the vector $\vec{w} = C - A$ is less than 180°. It can be easily verified that if $A = (A_x, A_y)$, $B = (B_x, B_y)$ and $C = (C_x, C_y)$, then (A, B, C) is a left-turn iff:

$$\boxed{L(A,B,C) := (B_x - A_x)(C_y - A_y) - (B_y - A_y)(C_x - A_x) > 0.} \quad (1)$$

It is easy to verify that:

$$L(A,B,C) = det \begin{pmatrix} 1 & A_x & A_y \\ 1 & B_x & B_y \\ 1 & C_x & C_y \end{pmatrix}.$$

Segment AB intersects segment CD if:

$$\boxed{L(A,B,C) \cdot L(A,B,D) < 0 \text{ and } L(C,D,A) \cdot L(C,D,B) < 0.} \quad (2)$$

Note: To test whether two triangles intersect, you need to check whether one of the edge segments of one triangle intersects one of those of the other.

C Rotating a triangle

To rotate a triangle about its center by clockwise angle θ, we need to multiply the vertex coordinates of the triangle by the following matrix:

$$M_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

To rotate by counter-clockwise angle θ, use $M_{-\theta}$.
Suppose the coordinate of the center of the robot is $C = (c_x, c_y)$. Assume the angle (direction) of the robot is θ, the ROBOT_WIDTH is w and the ROBOT_HEIGHT of the robot is h.
We define a canonic robot, so that $(c_x, c_y) = (0,0)$, with the following coordinates: $P_1 = (-[w/2], -[h/3])$, $P_2 = ([w/2], -[h/3])$, $P_3 = ([0], [h*2/3])$. Then the coordinates of the vertex P_1 of the robot in the world are:

$$Q_1 = M_\theta P_1 + C,$$

which means:

$$\boxed{\begin{aligned} x_1 &= -\frac{w}{2}\cos\theta - \frac{h}{3}\sin\theta + c_x \quad (3) \\ y_1 &= \frac{w}{2}\sin\theta - \frac{h}{3}\cos\theta + c_y \quad (4) \end{aligned}}$$

D How to calculate the end point of the laser beam

You can imagine that the laser beam is a line with an origin (one starting point) (X_1, Y_1) and a direction θ in $2D$ space: The canonical laser beam is defined as $(0, LASER\_BEAM\_LENGTH)$, and a general laser beam in the world coordinate is defined as LaserBeam $< X_1, Y_1, \theta >$ In order to calculate the other endpoint (X_2, Y_2) of the laser beam, use the following equations.

$$X_2 = X_1 + LASER\_BEAM\_LENGTH * sin(\theta) \quad (5)$$
$$Y_2 = Y_1 + LASER\_BEAM\_LENGTH * cos(\theta) \quad (6)$$

E Changes

When	Where	What
7.7.05	1.3	$CurrentEnemyLifePoints The current life points of the other robot.
7.7.05	1.1	(the value of variables is 0 by default).
14.8.05	2.1	When two robots collide, they must be set in a new positions randomly and continue the game. You should regard to the collision as an event that the system handles by generating random positions for the two robots.
17.8.05	1.5	If move gets a negative number then it is equivalent to move(0).