```
#ifndef ARENA_H
#define ARENA_H

#include <qwidget.h>

class QPixmap;

/// A graphical representation of the arena
/// This class does not currently store locations of the bots, that task
/// must be looked after by the data classes
class Arena : public QWidget
{

Q_OBJECT

public:
        /// Constructor
        Arena( QWidget *parent=0, const char *name=0 );
        /// Destructor
        ~Arena();

        /// Places a team A pixmap at (X,Y), with Rotation R
        void putBotA (int X,int Y,int R);
        /// Places a team B pixmap at (X,Y), with Rotation R
        void putBotB (int X,int Y,int R);
        /// Places the ball at (X,Y)
        void putBall (int X,int Y);
        /// Clears a grid square with UL Corner (X,y)
        void clearAt (int X,int Y);
        /// Clears the entire arena
        void clearAll(void);

protected:

        /// QT Repaint event
        void paintEvent( QPaintEvent * );
        /// Pixmap image for Team A
        QPixmap *BotImageA;
        /// Pixmap image for Team B
        QPixmap *BotImageB;

};


#endif
```

```
#ifndef BALL_H
#define BALL_H

#include <qobject.h>
#include "coordinate.h"
#include "direction.h"

/// Ball class
class Ball : public QObject {

Q_OBJECT

   public:
      /// Constructor
      Ball(QObject *parent=0, const char * name = 0);
      /// Destructor
      ~Ball();
      /// return speed
      unsigned int speed();
      /// return / set speed
      unsigned int speed(unsigned int);
      /// return position
      Coordinate position();
      /// return / set speed
      Coordinate position(Coordinate);
      /// return direction
      Direction direction();
      /// return / set direction
      Direction direction(Direction);
      /// return which team has the ball
      unsigned int team();
      /// return / set which team has the ball
      unsigned int team(unsigned int);
      /// return which player has the ball
      unsigned int player();
      /// return / set which player has the ball
      unsigned int player(unsigned int);
      /// return Ticks
      unsigned int ticksUntilNextMove();
      /// return / set Ticks
      unsigned int ticksUntilNextMove(unsigned int);
      /// move the ball
      void moveBall(unsigned int, unsigned int);
      /// reset position of the ball
      void reset(Coordinate);

   private:
      /// whatever maximum speed is (10?) - this
      /// = how many ticks between moves
      /// if 0, then not moving
      unsigned int Speed;
          /// (x,y) position on board
      Coordinate Pos;
          /// direction of movement
      Direction Dir;
          /// most recent team to have ball (0, 1 or 2)
      unsigned int Team;
          /// most recent player on team to have ball
      unsigned int Player;
          /// wait states
      unsigned int TicksUntilNextMove;

   signals:
      /// indicate ball movement
      void ballMove(Coordinate, Coordinate);
```

```
};

#endif
```

```
/*************************************************************************
 * -= C++ =-
 *
 * Bot class
 *
 *************************************************************************/

#ifndef BOT_H
#define BOT_H

#include "coordinate.h"
#include "direction.h"
#include "garule.h"
#include "ball.h"
#include <qlist.h>
#include <qobject.h>

using namespace BotRotation;

/// Bot class
class Bot : public QObject {

Q_OBJECT

   public:
      /// Constructor
      Bot(QObject *parent=0, const char *name=0);
      /// Constructor
      Bot(QObject *parent, unsigned int mass);
      /// Destructor
      ~Bot();

      // methods that affect bot state
      /// Return Mass
      unsigned int mass();
      /// Set Mass
      unsigned int mass(unsigned int);
      /// Return Position
      Coordinate position();
      /// Set Position
      Coordinate position(Coordinate);
      /// Return Direction
      Direction direction();
      /// Set Direction
      Direction direction(Direction);
      /// Return RuleSetSize
      unsigned int ruleSetSize();
      /// Return Ticks
      unsigned int ticksUntilNextMove();
      /// Set Ticks
      unsigned int ticksUntilNextMove(unsigned int);
      /// Execute Rule
      void execRule(GARule*, Ball*, unsigned int, unsigned int);
      /// Return myBall
      bool myBall();
      /// Set myBall
      bool myBall(bool);

      // methods to modify or inspect the bot's rules
      /// Get all Rules
      QList<GARule> rules();
      /// Get / Set all Rules
      QList<GARule> rules(QList<GARule>);
      /// Get Rule with specific number
      GARule* rule(unsigned int);
```

```
    /// Remove specific Rule
    GARule* removeRule(unsigned int);
    /// Remove the given Rule
    GARule* removeRule(GARule*);
    /// Add the given Rule in position number
    GARule* insertRule(GARule*, unsigned int);
    /// Add the given Rule at the end of the list
    GARule* insertRule(GARule*);
    /// Return the best rule given a rule of conditions
    GARule* bestRule(GARule*);
    /// Generate a random Bot with a given number of rules and mass
    void randomBot(unsigned int, unsigned int);
    /// Mutate a rule chosen at random
    void mutateBot();

    // methods to access stats
    /// Get number of goals scored by this bot
    int goals();
    /// Get / Set goals
    int goals(int);
    /// Get Weight to give goals
    static float goalsWeight();
    /// Get / Set weight to give goals
    static float goalsWeight(float);
    /// Get number of interceptions by this bot
    unsigned int interceptions();
    /// Get / Set interceptions
    unsigned int interceptions(unsigned int);
    /// Get Weight to give interceptions
    static float interceptionsWeight();
    /// Get / Set weight to give interceptions
    static float interceptionsWeight(float);
    /// Get time with ball
    unsigned int timeWithBall();
    /// Get / Set time with ball
    unsigned int timeWithBall(unsigned int);
    /// Get Weight to time with ball
    static float timeWithBallWeight();
    /// Get / Set Weight to time with Ball
    static float timeWithBallWeight(float);
    /// Compute this bot's fitness based upon stats
    float fitnessFunction();

private:
    /// Mass of the bot
    unsigned int Mass;
    /// Position of the bot
    Coordinate Pos;
    /// Direction facing of the bot
    Direction Dir;
    /// List of Rules
    QList<GARule> Rules;

    // bot stats
    /// Goasl scored by this bot
    unsigned int Goals;
    /// Weight to give goals in fitness function
    static float GoalsWeight;
    /// Interceptions made by this bot
    unsigned int Interceptions;
    /// Weight to give interceptions
    static float InterceptionsWeight;
    /// Number of moves this bot carried the ball
    unsigned int TimeWithBall;
    /// Weight to give moves
```

```
      static float TimeWithBallWeight;
      /// Wait states
      unsigned int TicksUntilNextMove;
      /// True if this bot has the ball
      bool MyBall;
      /// An identifying integer
      int BotID;

   signals:
      /// Indicate the bot has moved
      void botMove(Coordinate,Coordinate);
      /// Indicate the bot has changed direction
      void botDirection(Coordinate,Direction);
};

#endif
```

```
#ifndef BOTVIEW_H
#define BOTVIEW_H

#include <qwidget.h>
#include "coordinate.h" // req by moc
#include "direction.h"

class Arena;
class QLCDNumber;
class Coordinate;

///  This is not the actual "playing field" widget.  It is simply the main
///  widget that will own all the others
class BotView : public QWidget
{

Q_OBJECT

public:
        /// Constructor
        BotView(QWidget *parent,const char*name=0,int w=800,int h=400);
        /// Destructor
        ~BotView();
        /// Return the playing field's width
        int fieldWidth(void);
        /// Return the playing field's width
        int fieldHeight(void);
        /// Set the arena Width, return true on success
        int fieldWidth(int);
        /// Set the arena Height, return true on success
        int fieldHeight(int);

public slots:

        /// Increases Team A's score by one
        void slotScoreA(void);
        /// Increases Team B's score by one
        void slotScoreB(void);
        /// Resets the LCD scores
        void slotClearScores(void);
        /// Resets the Field
        void slotClearField(void);
        /// Moves a bot
        void slotMoveTeamA(Coordinate, Coordinate);
        /// Moves a bot
        void slotMoveTeamB(Coordinate, Coordinate);
        /// Turns a bot
        void slotTurnTeamA(Coordinate, Direction);
        /// Turns a bot
        void slotTurnTeamB(Coordinate, Direction);
        /// Moves the ball, spawns a ball if needed
        void slotMoveBall(Coordinate,Coordinate);

protected slots:

        /// Relay slot for changing game speed
        void slotValueChanged(int);

signals:
        /// Relay signal for changing game speed
        void valueChanged(int);

protected:

        /// Pointer to the widget that is the "playing field"
```

```
        Arena *Field;
        /// LCD widget for Team A's score
        QLCDNumber *LCDScoreA;
        /// LCD widget for Team B's score
        QLCDNumber *LCDScoreB;
        /// Width of field
        int FieldWidth;
        /// Height of field
        int FieldHeight;
        /// Team A's score
        int ScoreA;
        /// Team B's score
        int ScoreB;

};

#endif
```

```
/*********************************************************************
 * -= C++ =-
 *
 * the class for coordinate (x,y)
 *
 *********************************************************************/

#ifndef COORDINATE_H
#define COORDINATE_H

/// Coordinate object
class Coordinate {
   public:
      /// default constructor
      Coordinate() {};
      /// constructor
      Coordinate(int x1, int y1) { x = x1; y = y1; };
      /// direction in x
      int x;
      /// direction in y
      int y;
};

#endif
```

```
/*********************************************************************
 * -= C++ =-
 *
 * enum type for directions
 * currently with 8 different directions
 *
 *********************************************************************/

#ifndef DIRECTION_H
#define DIRECTION_H

/// Direction definition
enum Direction { N, NE, E, SE, S, SW, W, NW };


#endif
```

```
#ifndef GABOT_H
#define GABOT_H

#include <qobject.h>

#include "coordinate.h" // need full def for moc
#include "direction.h"
#include "simplega.h"

// object prototypes
class QString;
class Team;
class Coordinate;
class Game;
class QTimer;

/// GA Bot data object
class GABot : public QObject
{

Q_OBJECT

public:
        /// Constructor
        GABot(QObject *parent=0, const char * name = 0);
        /// Destructor
        ~GABot();

        /// Load a rule set from Filename into TeamNumber
        int loadTeamFromFile(QString Filename, int TeamNumber);

        /// Save a team to file
        int saveTeamToFile(QString Filename, int TeamNumber);
        /// Generate Random team
        void randomTeam(int TeamNumber);

protected:

        /// Pointer to Team A object
        Team *TeamA;
        /// Pointer to Team B object
        Team *TeamB;
        /// Timer for interval between moves
        QTimer *Tick;
        /// Tick interval in milliseconds
        int TickInterval;
        /// Game instance
        Game *GAGame;

        /// Genetic Algorithm instance
        SimpleGA GenAlg;

        /// Prepares the game object
        void prepareGame(void);

protected slots:

        /// Executed every tick interval, if Timer isActive()
        void slotTurn(void);

        /// Game class has indicated the ball has moved
        void slotBallMoved(Coordinate,Coordinate);
        /// Team class has indicated a Bot from Team A has moved
        void slotBotMoveA(Coordinate,Coordinate);
        /// Team class has indicated a Bot from Team B has moved
```

```
        void slotBotMoveB(Coordinate,Coordinate);
        /// Team class has indicated a Bot from Team A has changed Direction
        void slotBotDirectionA(Coordinate,Direction);
        /// Team class has indicated a Bot from Team B has changed Direction
        void slotBotDirectionB(Coordinate,Direction);

        /// Execute when Team A scores
        void slotTeamAScores(void);
        /// Execute when Team B scores
        void slotTeamBScores(void);
        /// Clear the field in the Arena
        void slotClearField(void);

public slots:

        /// Change the tickInterval
        void slotTickInterval(int);
        /// Activate the timer (Go!)
        void slotStartTimer(void);
        /// Stop the timer (Pause)
        void slotStopTimer(void);
        /// End the game
        void slotGameOver(void);

signals:

        /// Signal that Team A has Scored
        void teamAScores(void);
        /// Signal that Team B has Scored
        void teamBScores(void);
        /// Signal that the scores should be cleared
        void clearScores(void);
        /// Signal that ball should be spawned at Coord
        void putBall(Coordinate);
        /// Signal that ball has moved
        void moveBall(Coordinate,Coordinate);
        /// Signal that a team A bot has moved
        void moveTeamA(Coordinate, Coordinate);
        /// Signal that a team B bot has moved
        void moveTeamB(Coordinate, Coordinate);
        /// Signal that a team A bot has turned Direction
        void turnTeamA(Coordinate, Direction);
        /// Signal that a team B bot has turned Direction
        void turnTeamB(Coordinate, Direction);
        /// Signal that the field should be cleared
        void clearField(void);
        /// Signal that it is safe to start a game
        void gameReady(bool);
        /// Signal that the current game is over
        void gameOver(void);

};

#endif
```

```
#ifndef GAME_H
#define GAME_H

#include <qobject.h>
#include "coordinate.h"

class Team;
class Bot;
class GARule;
class Ball;

/// Game class: perform game playing by assigning bot and ball positions and
/// then using each bot's ruleset to determine its moves
class Game : public QObject {

Q_OBJECT

   public:
      /// two teams, game length, and board size (width, height)
      Game(QObject *parent,Team*, Team*, unsigned int, unsigned int, unsigned int);
      /// destruct 1-1-A-2-B
      ~Game();
      /// perform one turn
      void turn();
      /// true if this game is over
      bool over();
      /// reset ball and bots to start positions (at start, after goal)
      void reset();

   private:
      /// determine a Bot's state; also need its team
      GARule* botState(Bot*, unsigned int teamnum);
      /// determine and resolve bots colliding with things
      void botCollision(Bot*, unsigned int teamnum);
      /// determine an index into a sensor array given two coordinates
      unsigned int posIndex(Coordinate, Coordinate);
      /// determine if the ball is in the net and allocate
          /// goals to the proper team and player
      unsigned int ballInNet();
      /// when bots collide, choose which one keeps the ball
      bool tradeBall(Bot*, Bot*, unsigned int, unsigned int);
      /// Pointer to first team
      Team *T1;
          /// Pointer to second team
          Team *T2;
          /// Length of game in turns
      unsigned int GameLength;
          /// Number of turns so far
      unsigned int Turns;
          /// Board width
      unsigned int Width;
          /// Board height
          unsigned int Height;
          /// Start of net location (calculated from height)
      unsigned int NetStart;
          /// End of net location (calculated from height)
      unsigned int NetEnd;
          /// Pointer to the ball
      Ball *B;

   protected slots:
          /// Activated by the ball emitting a location signal
      void slotBallMoved(Coordinate,Coordinate);
```

```
   signals:
          /// emitted when the ball changes locations
      void ballMoved(Coordinate,Coordinate);
          /// emitted when team A scores
      void teamAScores(void);
          /// emitted when team B scores
      void teamBScores(void);
          /// emitted when the field should be cleared
      void clearField(void);
          /// emitted at the end of a game
      void gameOver(void);

};

#endif
```

```
/**********************************************************************
 * -= C++ =-
 *
 * Genetic Algorithm Rule
 *
 **********************************************************************/

#ifndef GARULE_H
#define GARULE_H

// predirectives
#include "rotation.h"
#include "thing.h"
#include <qobject.h>

/// For bot Rotations
using namespace BotRotation;

/// Main class for the rules in our Genetic Algorithm
class GARule : public QObject{
   public:
      /// Constructor
      GARule(QObject *parent=0,const char *name=0);
      /// Destructor
      ~GARule();
      /// Copy Constructor
      GARule(const GARule&);
      /// Copy Assignment
      GARule& operator=(const GARule&);
      /// Member function to return teamBall
      int teamBall();
      /// Member function to return myBall
      bool myBall();
      /// Member function to return the pointer of the sensor array
      Thing* sensors();
      /// Member function to return fire
      bool fire();
      /// Member function to return move
      bool move();
      /// Member function to return turn
      Rotation turn();

      /// Member function to set teamBall
      int teamBall(int);
      /// Member function to set myBall
      bool myBall(bool);
      /// Member function to set the sensors
      Thing* sensors(Thing[]);
      /// Member function to set fire
      bool fire(bool);
      /// Member function to set move
      bool move(bool);
      /// Member function to set rotation
      Rotation turn(Rotation);
      /// Generate a random rule
      void randomRule();
      /// Mutate the rule
      void mutateRule();
      /// Find the difference between the conditions in this rule and the ones in the given rule
      unsigned int difference(GARule*);

   private:
      // conditions
      /// Ball is ours (+ve), theirs (-ve), or nither (0)
```

```
      int TeamBall;
      /// Ball is in this bot's posessioin
      bool MyBall;
      /// Sensor states
      Thing Sensors[8];

      // actions
      /// Fire the ball (shoot or pass, same thing)
      bool Fire;
      /// Move in directions being faced
      bool Move;
      /// Angle to turn by (Left, Right, None)
      Rotation Turn;

};

#endif
```

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#define VERSION "0.99"

// include files for QT
#include <qmainwindow.h>

// Class prototypes
class QString;
class QPopupMenu;
class QAction;
class BotView;
class GABot;
class QTimer;

/// Main widget for the GABots app
class MainWindow : public QMainWindow
{

Q_OBJECT

public:
        /// Constructor
        MainWindow();
        /// Destructor
        ~MainWindow();

protected:

        /// Create the QT Actions
        void initActions();
        /// Create the menu bar
        void initMenuBar();
        /// Create the tool bar
        void initToolBar();
        /// Create the status bar
        void initStatusBar();
        /// Create the GA Bot object
        void initGABotDoc();
        /// Create the GA Bot view object
        void initView();
        /// Query the user if they wish to quit without saving
        bool queryExit();
        /// Open a team ruleset file
        void teamFileOpen(int);
        /// Save a team ruleset to file
        void teamFileSave(int);
        /// Generate new team randomly
        void teamGenerateRandom(int);

protected slots:

        /// This should be changed
        void slotFileClose();
        /// Runs on quiting the application
        void slotFileQuit();
        /// Toggles the Toolbar
        void slotViewToolBar(bool toggle);
        /// Toggles the Statusbar
        void slotViewStatusBar(bool toggle);
        /// Toggles the Game Display
        void slotViewGame(bool toggle);
        /// Launches the About box
        void slotHelpAbout();
```

```
        /// Runs on loading a file
        void loading(QString);
        /// Load a file for team A
        void slotLoadTeamA();
        /// Load a file for team B
        void slotLoadTeamB();
        /// Save a file for team A
        void slotSaveTeamA();
        /// Save a file for team B
        void slotSaveTeamB();
        /// Generate a new team for team A
        void slotGenerateTeamA();
        /// Generate a new team for team B
        void slotGenerateTeamB();
        /// Start QTimer, begin a game
        void slotGoGame(void);
        /// Stops an active game
        void slotStopGame(void);
        /// Toggles game ready status
        void slotGameReady(bool);
        /// Changed the game speed
        void slotTickInterval(int);
        /// Handle game over
        void slotGameOver(void);

private:

        /// Pointer to the View
        BotView *View;
        /// Pointer to the GABot document
        GABot   *GABotDoc;
        /// Pointer to the file menu pop up
        QPopupMenu *FileMenu;
        /// Pointer to the view menu pop up
        QPopupMenu *ViewMenu;
        /// Pointer to the help menu pop up
        QPopupMenu *HelpMenu;
        /// Pointer to the tool bar pop up
        QToolBar *Toolbar;
        /// Action for opening team A file
        QAction *TeamAFileOpen;
        /// Action for opening team B file
        QAction *TeamBFileOpen;
        /// Action for saving team A file
        QAction *TeamAFileSave;
        /// Action for saving team B file
        QAction *TeamBFileSave;
        /// Action for quitting the application
        QAction *FileQuit;
        /// Action for toggling the toolbar
        QAction *ViewToolBar;
        /// Action for toggoling the status bar
        QAction *ViewStatusBar;
        /// Action for toggling the view game
        QAction *ViewGame;
        /// Action for launching the About box
        QAction *HelpAbout;
        /// Action for starting a game
        QAction *GoGame;
        /// Action for stopping a game
        QAction *StopGame;
        /// Action for generate a random team A
        QAction *TeamAGenerate;
        /// Action for generate a random team B
        QAction *TeamBGenerate;
```

```
        /// Action for reset display
        QAction *ResetScreen;

};
#endif
```

```
/// some useful functions for generating random numbers

#ifndef RANDOM_H
#define RANDOM_H

class Random {
   public:
      /// initialize the seed with the time
      static void initseed();
      /// random integer from start to end
      static int randint(int start, int end);
      /// random double from start to end
      static double randd(double start, double end);
      /// random boolean value
      static bool randbool();
};

#endif
```

```
#ifndef ROTATION_H
#define ROTATION_H

/// Rotation enumerated type
/// The namespace was needed to resolve some naming conflicts.
namespace BotRotation {
    /// Enumeric type for bot rotations
    enum Rotation {Left, None, Right};
}

#endif
```

```
/*************************************************************************
 * -= C++ =-
 *
 * Simple Genetic Algorithm - Perform GA on a given team
 *
 *************************************************************************/

#ifndef SIMPLEGA_H
#define SIMPLEGA_H

#include "team.h"

/// Main class for Simple Genetic Algorithm used in the program
class SimpleGA {
   public:
      /// Constructor
      SimpleGA();
      /// Constructor
      SimpleGA(double);
      /// Destructor
      ~SimpleGA();
      /// Member function for returning mutation rate
      double mutationRate();
      /// Member function for setting mutation rate
      double mutationRate(double rate);
      /// Crossover two bots and produce two new bots
      void crossover(Bot*, Bot*, Bot*, Bot*);
      /// Evolve a team
      void evolve(Team*);

   private:
      /// Mutation Rate
      double MutationRate;
};

/// Use for sorting bots in order of fitness
struct floatbot {
   /// Value of fitness
   float fit;
   /// Bot number
   unsigned int bot;
};

#endif
```

```
#ifndef TEAM_H
#define TEAM_H

class Bot;
#include <qlist.h>
#include <qstring.h>
#include <qobject.h>
#include "coordinate.h"
#include "direction.h"

/// Team class: defines a "team" or population of Bots
/// presumably there are two of these at a time, unless we decide to play
/// more than two teams at once
class Team : public QObject {

Q_OBJECT

   public:
      /// Constructor
      Team(QObject *parent=0, const char *name=0);
      /// Destructor
      ~Team();
          /// return size of team
      unsigned int size();
          /// return a list of bots
      QList<Bot> bots();
          /// set the list of bots
      QList<Bot> bots(QList<Bot>);
          /// send bot X back to the minors
      Bot* removeBot(unsigned int X);
          /// pull a bot up from the minors
      unsigned int insertBot(Bot*);
          /// pull a bot up from the minors, with number
      unsigned int insertBot(Bot*, unsigned int);
          /// returns a particular bot, by number
      Bot* bot(unsigned int);
          /// generate a random team of size X
      void randomTeam(unsigned int X);
          /// return number of team goals
      int goals();
          /// set number of team goals
      int goals(int);
          /// return number of wins
      unsigned int wins();
          /// set number of wins
      unsigned int wins(unsigned int);
          /// return number of losses
      unsigned int losses();
          /// set number of losses
      unsigned int losses(unsigned int);
          /// return number of ties
      unsigned int ties();
          /// set number of ties
      unsigned int ties(unsigned int);
          /// return number of generations
      unsigned int generations();
          /// set number of generations
      unsigned int generations(unsigned int);
          /// return team name
      QString name();
          /// set team name
      QString name(QString);

   private:
      /// ball is ours (+ve), theirs (-ve), or neither (0)
```

```
      int TeamBall;
      /// goals scored by this team in its current game
      int Goals;
      /// number of wins by this team
      unsigned int Wins;
      /// number of losses by this team
      unsigned int Losses;
      /// number of ties by this team
      unsigned int Ties;
      /// number of generations
      unsigned int Generations;
      /// Team name
      QString Name;
      /// List of bots
      QList<Bot> Bots;

   protected slots:
      /// Handle Bot movements
      void slotBotMove(Coordinate,Coordinate);
      /// Handle Bot direction changes
      void slotBotDirection(Coordinate,Direction);

   signals:
      /// emitted when a bot has moved
      void botMove(Coordinate,Coordinate);
      /// emitted when a bot has changed direction
      void botDirection(Coordinate,Direction);
};

#endif
```

```
/// -= C++ =-
///
/// read and write team xml files

#ifndef TEAMDATA_H
#define TEAMDATA_H

// include files for QT
#include <qstring.h>
#include <qfile.h>
#include <qxml.h>
#include <qtextstream.h>

// other predirectives
#include "team.h"
#include "teamparser.h"

/// Read and Write team xml files
class TeamData{

   public:
      /// read team data from xml file
      Team* readTeamData(QString filename);
      /// write team data into xml file
      bool writeTeamData(QString filename, Team*);

};

#endif
```

```
/// -= C++ =-
///
/// Read team data from XML file

#ifndef TEAMPARSER_H
#define TEAMPARSER_H

// include files for QT
#include <qxml.h>

// other predirectives
#include <string>
#include "team.h"
#include "bot.h"
#include "garule.h"
#include "thing.h"

// Class prototypes
class QString;

/// Main class for parsing team XML data files,
/// Derived from QXmlDefaultHandler from QT Library
class TeamParser : public QXmlDefaultHandler
{
   public:
      /// Constructor
      TeamParser();
      /// Member function for the start of document
      bool startDocument();
      /// Member function for the start of a XML element (tag)
      bool startElement(const QString&, const QString&, const QString&,
                        const QXmlAttributes& );
      /// Member function for the end of a XML element (tag)
      bool endElement(const QString&, const QString&, const QString&);
      /// Member function for return the parsed data
      Team* teamData();

   private:
      /// Pointer to the team data
      Team* team;
      /// Pointer to the bot data
      Bot* bot;
      /// Pointer to the rule data
      GARule* rule;
      /// Poniter to the sensor array
      Thing* sens;
      /// Internal counter for bots
      int botcount;
      /// Internal counter for rules
      int rulecount;

};

#endif
```

```
#ifndef THING_H
#define THING_H

/// Thing type - enumerated to store various things we find in the game
typedef enum { Nothing,                 // ABSOLUTELY NOTHING!!!
               MyBot,                   // another bot on the same team
               OtherBot,                // a bot on another team
               TheBall,
               Net,                     // the other team's net
               Wall,
             } Thing;

#endif
```

```
#include <qpainter.h>
#include <qpixmap.h>

#include "arena.h"
#include "bot_a.xpm"
#include "bot_b.xpm"

//---------------------------------------------------------------------

Arena::Arena( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
        QPixmap Image;
        Image.load("rink.png",0);
        setBackgroundPixmap(Image);
        BotImageA = new QPixmap(BotA_XPM);
        BotImageB = new QPixmap(BotB_XPM);
}

//---------------------------------------------------------------------

Arena::~Arena()
{
        delete BotImageA;
        delete BotImageB;
}

//---------------------------------------------------------------------

void Arena::paintEvent( QPaintEvent *e )
{

}

//---------------------------------------------------------------------

void Arena::putBotA(int X, int Y, int A)
{
        if (X == -1 || Y == -1) return;
        QRect Area(X,Y,10,10);
        QPixmap Pix( Area.size() );
        Pix.fill( this, Area.topLeft() );
        QPainter p( &Pix );
        p.translate( Pix.width()/2.0, Pix.height()/2.0);
        p.rotate( A );
        p.translate( -Pix.width()/2.0, -Pix.height()/2.0);
        p.drawPixmap(0,0,*BotImageA);
        p.end();
        p.begin( this );
        p.drawPixmap( Area.topLeft(), Pix );
}

//---------------------------------------------------------------------

void Arena::putBotB(int X, int Y, int A)
{
        if (X == -1 || Y == -1) return;
        QRect Area(X,Y,10,10);
        QPixmap Pix( Area.size() );
        Pix.fill( this, Area.topLeft() );
        QPainter p( &Pix );
        p.translate( Pix.width()/2.0, Pix.height()/2.0);
        p.rotate( A );
        p.translate( -Pix.width()/2.0, -Pix.height()/2.0);
        p.drawPixmap(0,0,*BotImageB);
```

```
        p.end();
        p.begin( this );
        p.drawPixmap( Area.topLeft(), Pix );
}

//---------------------------------------------------------------

void Arena::clearAt(int X, int Y)
{
        QRect Area(X,Y,10,10);
        QPixmap Pix( Area.size() );
        Pix.fill( this, Area.topLeft() );
        QPainter p( this );
        p.drawPixmap ( Area.topLeft(), Pix );
        p.end();
}

//---------------------------------------------------------------
void Arena::putBall(int X, int Y)
{
        QPainter p ( this );
        p.setBrush( blue );
        p.setPen( NoPen );
        p.drawEllipse ( X+5.5, Y+5.5, 5.5, 5.5 );
        p.end();
}
//---------------------------------------------------------------
void Arena::clearAll(void)
{
        repaint();
}
//---------------------------------------------------------------
```

```
/*********************************************************************
 * -= C++ =-
 * Ball class
 *********************************************************************/

#include "ball.h"
#include "coordinate.h"
#include "direction.h"

//-----------------------------------------------------
Ball::Ball(QObject *parent, const char * name)
        :QObject(parent,name)
{
   Coordinate p;
   p.x = 0; p.y = 0;
   reset(p);
}
//-----------------------------------------------------
Ball::~Ball()
{
}
//-----------------------------------------------------
void Ball::reset(Coordinate p)
{
   Speed = 0;
   Pos.x = p.x; Pos.y = p.y;
   Dir = N;
   Team = 0;
   Player = 0;
   TicksUntilNextMove = 0;
}
//-----------------------------------------------------
unsigned int Ball::speed()
{
   return(Speed);
}
//-----------------------------------------------------
unsigned int Ball::speed(unsigned int s)
{
   if (Speed != s) {
      Speed = s;
   }
   return(Speed);
}
//-----------------------------------------------------
Coordinate Ball::position()
{
   return(Pos);
}
//-----------------------------------------------------
Coordinate Ball::position(Coordinate p)
{
   if ( (Pos.x != p.x) || (Pos.y != p.y) ) {
      emit(ballMove(Pos, p));
      Pos.x = p.x; Pos.y = p.y;
   }
   return(Pos);
}
//-----------------------------------------------------
Direction Ball::direction()
{
   return(Dir);
}
//-----------------------------------------------------
Direction Ball::direction(Direction d)
```

```
{
   if (Dir != d) {
      Dir = d;
   }
   return(Dir);
}
//-------------------------------------------------------
unsigned int Ball::team()
{
   return(Team);
}
//-------------------------------------------------------
unsigned int Ball::team(unsigned int t)
{
   if (Team != t) {
      Team = t;
   }
   return(Team);
}
//-------------------------------------------------------
unsigned int Ball::player()
{
   return(Player);
}
//-------------------------------------------------------
unsigned int Ball::player(unsigned int p)
{
   if (Player != p) {
      Player = p;
   }
   return(Player);
}
//-------------------------------------------------------
unsigned int Ball::ticksUntilNextMove()
{
   return(TicksUntilNextMove);
}
//-------------------------------------------------------
unsigned int Ball::ticksUntilNextMove(unsigned int t)
{
   if (TicksUntilNextMove != t) {
      TicksUntilNextMove = t;
   }
   return(TicksUntilNextMove);
}
//-------------------------------------------------------
void Ball::moveBall(unsigned int ArenaWidth, unsigned int ArenaHeight)
{
   Coordinate pos = position();
   Direction dir = direction();
   Direction newdir = dir;

   switch(dir) {
      case N:
         pos.y--;
         break;
      case NE:
         pos.y--; pos.x++;
         break;
      case E:
         pos.x++;
         break;
      case SE:
         pos.y++; pos.x++;
         break;
```

```
      case S:
         pos.y++;
         break;
      case SW:
         pos.y++; pos.x--;
         break;
      case W:
         pos.x--;
         break;
      case NW:
         pos.y--; pos.x--;
         break;
   }
   if ( (pos.y < 0) && (pos.x < 0) && (dir == NW) ) {
      // collided with nw corner
      newdir = SE;
      pos.x = pos.y = 0;
   } else
   if ( (pos.y < 0) && (pos.x >= ArenaWidth) && (dir == NE) ) {
      // collided with ne corner
      newdir = SW;
      pos.y = 0;
      pos.x = ArenaWidth - 1;
   } else
   if ( (pos.y >= ArenaHeight) && (pos.x < 0) && (dir == SW) ) {
      // collided with sw corner
      newdir = NE;
      pos.x = 0;
      pos.y = ArenaHeight - 1;
   } else
   if ( (pos.y >= ArenaHeight) && (pos.x >= ArenaWidth) && (dir == SE) ) {
      // collided with se corner
      newdir = NW;
      pos.x = ArenaWidth - 1;
      pos.y = ArenaHeight - 1;
   } else
   if (pos.y < 0) {
      // collided with north wall
      switch(dir) {
         case N:
            newdir = S;
            break;
         case NE:
            newdir = SE;
            break;
         case NW:
            newdir = SW;
            break;
         // default should not be reached
      }
      pos.y = 0;
   } else
   if (pos.y >= ArenaHeight) {
      // collided with south wall
      switch(dir) {
         case S:
            newdir = N;
            break;
         case SE:
            newdir = NE;
            break;
         case SW:
            newdir = NW;
            break;
         // default should not be reached
```

```
      }
      pos.y = ArenaHeight - 1;
   } else
   if (pos.x < 0) {
      // collided with west wall
      switch(dir) {
         case W:
            newdir = E;
            break;
         case NW:
            newdir = NE;
            break;
         case SW:
            newdir = SE;
            break;
         // default should not be reached
      }
      pos.x = 0;
   } else
   if (pos.x >= ArenaWidth) {
      // collided with east wall
      switch(dir) {
         case E:
            newdir = W;
            break;
         case NE:
            newdir = NW;
            break;
         case SE:
            newdir = SW;
            break;
         // default should not be reached
      }
      pos.x = ArenaWidth - 1;
   }
   Coordinate old = position();
   position(pos);
   direction(newdir);
   emit(ballMove(old,pos));
}
//-------------------------------------------------------
```

```
/*********************************************************************
 * -= C++ =-
 *
 * Bot class
 *
 *********************************************************************/

#include "bot.h"
#include "garule.h"
#include "ball.h"
#include "random.h"
#include <cstdlib>
#include <qlist.h>

Bot::Bot(QObject *parent, const char *name)
        :QObject(parent,name) {
   Mass = 5; Pos.x = 0; Pos.y = 0;
   Dir = N;
   MyBall = false;
   Rules.clear();
   Goals = Interceptions = TimeWithBall = 0;
   TicksUntilNextMove = 1;  // will move anon
   BotID = 0;
}

Bot::Bot(QObject *parent, unsigned int m)
        : QObject(parent,0) {
   Mass = m; Pos.x = 0; Pos.y = 0;
   Dir = N;
   MyBall = false;
   Rules.clear();
   Goals = Interceptions = TimeWithBall = 0;
   TicksUntilNextMove = 1;  // will move anon
   BotID = 0;
}

Bot::~Bot(){
   Rules.clear();
}

// init static vars
float Bot::GoalsWeight = 1.0;
float Bot::InterceptionsWeight = 0.5;
float Bot::TimeWithBallWeight = 0.2;

unsigned int Bot::mass() {
   return(Mass);
}

unsigned int Bot::mass(unsigned int m) {
   Mass = m;
   return(Mass);
}

Coordinate Bot::position() {
   return(Pos);
}

Coordinate Bot::position(Coordinate c) {
   if ( (Pos.x != c.x) || (Pos.y != c.y) ) {
      emit(botMove(Pos,c));
      Pos.x = c.x; Pos.y = c.y;
   }
   return(Pos);
}
```

```
Direction Bot::direction() {
   return(Dir);
}

Direction Bot::direction(Direction d) {
   if (Dir != d) {
      emit(botDirection(Pos,d));
      Dir = d;
   }
   return(Dir);
}

unsigned int Bot::ruleSetSize() {
   return(Rules.count());
}

unsigned int Bot::ticksUntilNextMove() {
   return(TicksUntilNextMove);
}

unsigned int Bot::ticksUntilNextMove(unsigned int t) {
   TicksUntilNextMove = t;
   return(TicksUntilNextMove);
}

bool Bot::myBall() {
   return(MyBall);
}

bool Bot::myBall(bool b) {
   MyBall = b;
   return(MyBall);
}

QList<GARule> Bot::rules() {
   return(Rules);
}

QList<GARule> Bot::rules(QList<GARule> rs) {
   Rules = rs;
   return(Rules);
}

GARule* Bot::rule(unsigned int num) {
   return Rules.at(num);
}

GARule* Bot::removeRule(unsigned int num) {
   GARule *p = Rules.at(num);
   Rules.remove();
   return p;
}

GARule* Bot::removeRule(GARule *r) {
   Rules.remove(r);
   return r;
}

GARule* Bot::insertRule(GARule *r, unsigned int num) {
   insertChild(r);
   Rules.insert(num, r);
   return r;
}
```

```
GARule* Bot::insertRule(GARule *r) {
   insertChild(r);
   Rules.append(r);
   return r;
}

int Bot::goals() {
   return(Goals);
}

int Bot::goals(int g) {
   Goals = g;
   return(Goals);
}

unsigned int Bot::interceptions() {
   return(Interceptions);
}

unsigned int Bot::interceptions(unsigned int i) {
   Interceptions = i;
   return(Interceptions);
}

unsigned int Bot::timeWithBall() {
   return(TimeWithBall);
}

unsigned int Bot::timeWithBall(unsigned int t) {
   TimeWithBall = t;
   return(TimeWithBall);
}

float Bot::goalsWeight() {
   return(GoalsWeight);
}

float Bot::goalsWeight(float w) {
   GoalsWeight = w;
   return(GoalsWeight);
}

float Bot::interceptionsWeight() {
   return(InterceptionsWeight);
}

float Bot::interceptionsWeight(float w) {
   InterceptionsWeight = w;
   return(InterceptionsWeight);
}

float Bot::timeWithBallWeight() {
   return(TimeWithBallWeight);
}

float Bot::timeWithBallWeight(float w) {
   TimeWithBallWeight = w;
   return(TimeWithBallWeight);
}

float Bot::fitnessFunction() {
   float fitness;

   fitness = GoalsWeight*Goals +
             InterceptionsWeight*Interceptions +
```

```
            TimeWithBallWeight*TimeWithBall;

   return(fitness);
}

GARule* Bot::bestRule(GARule *cond) {
   unsigned int *diff;                  // list of differences
   QList<GARule> r;                     // list of rules
   GARule *match;                       // chosen rule
   unsigned int numrules;               // number of rules this bot has
   unsigned int minimum;                // minimum diff between rules
   unsigned int *matches;               // list of indices of matching rules
   unsigned int nummatch;               // number of matches
   unsigned int i;                      // counter

   // initialize vars
   numrules = ruleSetSize();
   r = rules();
   diff = new unsigned int[numrules];
   matches = new unsigned int[numrules];

   // load the differences and find the minimum
   diff[0] = (r.at(0))->difference(cond);    // get first difference and
   minimum = diff[0];                   // call it the minimum
   for (i=1; i<numrules; i++) {
      diff[i] = (r.at(i))->difference(cond);
      if ( diff[i] < minimum ) {        // this one is smaller
         minimum = diff[i];             // so replace the min
      }
   }

   // find how many minima there are and where they live
   nummatch = 0;
   for (i=0; i<numrules; i++) {
      if ( diff[i] == minimum ) {
         matches[nummatch++] = i;
      }
   }

   // return one at random
   // pick an int from 0 to nummatch-1
   i = Random::randint(0, nummatch-1);
   match = r.at(matches[i]);

   // clear some memory
   delete matches;
   delete diff;

   return(match);
}

void Bot::execRule(GARule *r, Ball *B, unsigned int ArenaWidth, unsigned int ArenaHeight) {
   Coordinate pos;
   Direction dir, newdir;
   if (r->fire() && myBall()) {
      // fire the ball in the direction the bot is facing
      myBall(false);
      B->speed(mass());
      B->direction(direction());
      B->moveBall(ArenaWidth, ArenaHeight);    // move ball away from bot
      B->ticksUntilNextMove(1);                // ball will move immediately
   } else if (r->move()) {
      // move in the direction being faced
      pos = position();
      switch (direction()) {
```

```
         case N:
            pos.y--;
            break;
         case NE:
            pos.y--; pos.x++;
            break;
         case E:
            pos.x++;
            break;
         case SE:
            pos.y++; pos.x++;
            break;
         case S:
            pos.y++;
            break;
         case SW:
            pos.y++; pos.x--;
            break;
         case W:
            pos.x--;
            break;
         case NW:
            pos.y--; pos.x--;
            break;
      }
      if (pos.x < 0) pos.x = 0;
      if (pos.y < 0) pos.y = 0;
      if (pos.x >= ArenaWidth) pos.x = ArenaWidth - 1;
      if (pos.y >= ArenaHeight) pos.y = ArenaHeight - 1;
      position(pos);
      if (myBall()) B->position(pos);  // also move the ball if the player's holding it
   } else if (r->turn() != BotRotation::None) {
      if (r->turn() == BotRotation::Right) {
         switch(dir) {
            case N:
               newdir = NE;
               break;
            case NE:
               newdir = E;
               break;
            case E:
               newdir = SE;
               break;
            case SE:
               newdir = S;
               break;
            case S:
               newdir = SW;
               break;
            case SW:
               newdir = W;
               break;
            case W:
               newdir = NW;
               break;
            case NW:
               newdir = N;
               break;
         }
      } else {
         switch(dir) {
            case N:
               newdir = NW;
               break;
            case NE:
```

```
               newdir = N;
               break;
            case E:
               newdir = NE;
               break;
            case SE:
               newdir = E;
               break;
            case S:
               newdir = SE;
               break;
            case SW:
               newdir = S;
               break;
            case W:
               newdir = SW;
               break;
            case NW:
               newdir = W;
               break;
         }
      }
      direction(newdir);
      if (myBall()) B->direction(newdir);  // also rotate the ball if the player's holding it
   }
   if (myBall()) timeWithBall(timeWithBall()+1);  // increment the time with the ball if it's being held
}

void Bot::randomBot(unsigned int numrules, unsigned int m) {

   mass(m);
   for (unsigned int i=0; i<numrules; i++) {
      GARule *NewRule = new GARule;
      NewRule->randomRule();
      insertRule(NewRule);
   }
}

void Bot::mutateBot() {
   unsigned int rulenum = Random::randint(1, ruleSetSize()); rulenum--;
   GARule *r;

   r = rule(rulenum);
   r->mutateRule();
}
```

```
/*****************************************************************
 * -= C++ =-
 *
 *****************************************************************/

#include <qlayout.h>
#include <qlcdnumber.h>
#include <qvbox.h>
#include <qslider.h>

#include "botview.h"
#include "arena.h"
#include "coordinate.h"
#include "direction.h"

//-----------------------------------------------------------------
BotView::BotView(QWidget *parent,const char *name, int w, int h)
        : QWidget(parent,name)
{
        FieldWidth  = w;
        FieldHeight = h;
        ScoreA = ScoreB = 0;

        QVBox *Box = new QVBox( this, 0 );
        Box->setFrameStyle( QFrame::WinPanel | QFrame::Sunken );
        Field = new Arena( Box, "Field" );

        LCDScoreA  = new QLCDNumber( this, 0 );
        LCDScoreA->setMaximumSize(LCDScoreA->size());
        LCDScoreA->setMinimumSize(LCDScoreA->size());
        LCDScoreB  = new QLCDNumber( this, 0 );
        LCDScoreB->setMaximumSize(LCDScoreB->size());
        LCDScoreB->setMinimumSize(LCDScoreB->size());

        QSlider *TimerSlider = new QSlider( 25, 1000, 25, 250, QSlider::Horizontal, this);
        TimerSlider->setTracking(TRUE);
        TimerSlider->setMinimumSize(250,TimerSlider->height());
        TimerSlider->setMaximumSize(250,TimerSlider->height());

        QGridLayout *Grid = new QGridLayout( this, 1, 1, 10 );
        Grid->addWidget( LCDScoreA, 0, 0, Qt::AlignLeft);
        Grid->addWidget( TimerSlider, 0, 0, Qt::AlignCenter);
        Grid->addWidget( LCDScoreB, 0, 0, Qt::AlignRight);
        Grid->addWidget( Box, 1, 0 );

        Field->setMaximumSize(FieldWidth,FieldHeight);
        Field->setMinimumSize(FieldWidth,FieldHeight);
        Box->setMaximumSize(Field->size());
        Box->setMinimumSize(Field->size());

        connect(TimerSlider, SIGNAL(valueChanged(int)), this, SLOT(slotValueChanged(int)));
}
//-----------------------------------------------------------------
BotView::~BotView()
{
}
//-----------------------------------------------------------------
int BotView::fieldWidth(void)
{
        return FieldWidth;
}
//-----------------------------------------------------------------
int BotView::fieldHeight(void)
{
        return FieldHeight;
```

```
}
//---------------------------------------------------------------
int BotView::fieldWidth(int A)
{
        if (A > 0) {
                FieldWidth = A;
                repaint();
                return A;
        } else {
                return 0;
        }
}
//---------------------------------------------------------------
int BotView::fieldHeight(int A)
{
        if (A > 0) {
                FieldHeight = A;
                repaint();
                return A;
        } else {
                return 0;
        }
}
//---------------------------------------------------------------
void BotView::slotScoreA(void)
{
        ScoreA++;
        LCDScoreA->display(ScoreA);
}
//---------------------------------------------------------------
void BotView::slotScoreB(void)
{
        ScoreB++;
        LCDScoreB->display(ScoreB);
}
//---------------------------------------------------------------
void BotView::slotClearScores(void)
{
        ScoreA = ScoreB = 0;
        LCDScoreA->display(0);
        LCDScoreB->display(0);
}
//---------------------------------------------------------------
void BotView::slotClearField(void)
{
        Field->clearAll();
}
//---------------------------------------------------------------
void BotView::slotMoveBall(Coordinate Old, Coordinate New)
{
        if (!(Old.x == New.x && Old.y == New.y)) Field->clearAt(Old.x*10,Old.y*10);
        Field->putBall(New.x*10,New.y*10);
}
//---------------------------------------------------------------
void BotView::slotMoveTeamA(Coordinate P1, Coordinate P2)
{
        if (P1.x == P2.x && P1.y == P2.y) {
                return;
        }
        int Dir;
        if (P1.x == P2.x && P1.y > P2.y)      Dir = 0;
        else if (P1.x < P2.x && P1.y > P2.y)  Dir = 45;
        else if (P1.x < P2.x && P1.y == P2.y) Dir = 90;
        else if (P1.x < P2.x && P1.y < P2.y)  Dir = 135;
        else if (P1.x == P2.x && P1.y < P2.y) Dir = 180;
```

```
        else if (P1.x > P2.x && P1.y < P2.y)  Dir = 225;
        else if (P1.x > P2.x && P1.y == P2.y) Dir = 270;
        else Dir = 315;
        Field->clearAt(10*P1.x,10*P1.y);
        Field->putBotA(10*P2.x,10*P2.y,Dir);
}
//---------------------------------------------------------------------
void BotView::slotMoveTeamB(Coordinate P1, Coordinate P2)
{
        if (P1.x == P2.x && P1.y == P2.y) {
                return;
        }
        int Dir;
        if (P1.x == P2.x && P1.y > P2.y)      Dir = 0;
        else if (P1.x < P2.x && P1.y > P2.y)  Dir = 45;
        else if (P1.x < P2.x && P1.y == P2.y) Dir = 90;
        else if (P1.x < P2.x && P1.y < P2.y)  Dir = 135;
        else if (P1.x == P2.x && P1.y < P2.y) Dir = 180;
        else if (P1.x > P2.x && P1.y < P2.y)  Dir = 225;
        else if (P1.x > P2.x && P1.y == P2.y) Dir = 270;
        else Dir = 315;
        Field->clearAt(10*P1.x,10*P1.y);
        Field->putBotB(10*P2.x,10*P2.y,Dir);
}
//---------------------------------------------------------------------
void BotView::slotValueChanged(int V)
{
        emit(valueChanged(V));
}
//---------------------------------------------------------------------
void BotView::slotTurnTeamA(Coordinate P1, Direction B)
{
        int Dir = 45 * B;
        Field->putBotA(10*P1.x,10*P1.y,Dir);
}
//---------------------------------------------------------------------
void BotView::slotTurnTeamB(Coordinate P1, Direction B)
{
        int Dir = 45 * B;
        Field->putBotB(10*P1.x,10*P1.y,Dir);
}
//---------------------------------------------------------------------
```

```
/*****************************************************************
 *****************************************************************/

// QT includes
#include <qstring.h>
#include <qtimer.h>

// App specific includes
#include "gabot.h"
#include "team.h"
#include "teamdata.h"
#include "game.h"

//---------------------------------------------------------------
GABot::GABot(QObject *parent, const char * name)
        : QObject(parent,name)
{
        // Set our pointers to null
        TeamA = TeamB = 0;
        GAGame = 0;

        // initialize mutation rate (fixed at 5% right now)
        GenAlg.mutationRate(0.05);

        // Set up our game ticker
        TickInterval = 500;
        Tick = new QTimer(this);
        connect(Tick, SIGNAL(timeout()), this, SLOT(slotTurn()) );
}
//---------------------------------------------------------------
GABot::~GABot()
{

}
//---------------------------------------------------------------
int GABot::loadTeamFromFile( QString Filename, int TeamNumber )
{
        int Err=0;

        TeamData data;
        if (TeamNumber == 0) {
                TeamA = data.readTeamData(Filename);
                if (TeamA) {
                        Err = 1;
                        insertChild(TeamA);
                        connect( TeamA, SIGNAL(botMove(Coordinate,Coordinate)),
                                 this,  SLOT(slotBotMoveA(Coordinate,Coordinate)));
                        connect( TeamA, SIGNAL(botDirection(Coordinate,Direction)),
                                 this,  SLOT(slotBotDirectionA(Coordinate,Direction)));
                }
        } else if (TeamNumber == 1) {
                TeamB = data.readTeamData(Filename);
                if (TeamB) {
                        Err = 1;
                        insertChild(TeamB);
                        connect( TeamB, SIGNAL(botMove(Coordinate,Coordinate)),
                                 this,  SLOT(slotBotMoveB(Coordinate,Coordinate)));
                        connect( TeamB, SIGNAL(botDirection(Coordinate,Direction)),
                                 this,  SLOT(slotBotDirectionB(Coordinate,Direction)));
                }
        }

        if (TeamA && TeamB) {
                prepareGame();
                emit(gameReady(TRUE));
```

```
        }

        return Err;

}
//-----------------------------------------------------------------
int GABot::saveTeamToFile( QString Filename, int TeamNumber)
{
        TeamData data;
        bool flag;
        if (TeamNumber == 0) {
                flag = data.writeTeamData(Filename, TeamA);
        } else {
                flag = data.writeTeamData(Filename, TeamB);
        }
        if (flag) return (1);
        else      return (0);
}
//-----------------------------------------------------------------
void GABot::randomTeam(int TeamNumber)
{
   if (TeamNumber == 0){
      if (TeamA) delete TeamA;
      TeamA = new Team(this);
      TeamA->randomTeam(5);
      connect( TeamA, SIGNAL(botMove(Coordinate,Coordinate)),
         this,  SLOT(slotBotMoveA(Coordinate,Coordinate)));
      connect( TeamA, SIGNAL(botDirection(Coordinate,Direction)),
         this,  SLOT(slotBotDirectionA(Coordinate,Direction)));

   }else if (TeamNumber == 1){
      if (TeamB) delete TeamB;
      TeamB = new Team(this);
      TeamB->randomTeam(5);
      connect( TeamB, SIGNAL(botMove(Coordinate,Coordinate)),
         this,  SLOT(slotBotMoveB(Coordinate,Coordinate)));
      connect( TeamB, SIGNAL(botDirection(Coordinate,Direction)),
         this,  SLOT(slotBotDirectionB(Coordinate,Direction)));
   }


   if (TeamA && TeamB){
      prepareGame();
      emit(gameReady(TRUE));
   }
}
//-----------------------------------------------------------------
void GABot::slotTurn(void)
{
        // Execute a tick generated turn here
        GAGame->turn();
}
//-----------------------------------------------------------------
void GABot::slotTickInterval(int A)
{
        if (Tick->isActive()) {
                TickInterval = A;
                Tick->changeInterval(A);
        } else {
                TickInterval = A;
        }
}
//-----------------------------------------------------------------
void GABot::slotStartTimer(void)
{
```

```
        Tick->start(TickInterval);
}
//-----------------------------------------------------------------
void GABot::slotStopTimer(void)
{
        Tick->stop();
}
//-----------------------------------------------------------------
void GABot::slotBallMoved(Coordinate Old, Coordinate New)
{
        emit(moveBall(Old, New));
}
//-----------------------------------------------------------------
void GABot::prepareGame(void)
{
        GAGame = new Game(this, TeamA, TeamB, 1000, 80, 40);
        TeamA->goals(0);
        TeamB->goals(0);
        GAGame->reset();
        connect(GAGame, SIGNAL(ballMoved(Coordinate,Coordinate)),
                this, SLOT(slotBallMoved(Coordinate,Coordinate)));
        connect(GAGame, SIGNAL(teamAScores()), this,SLOT(slotTeamAScores()) );
        connect(GAGame, SIGNAL(teamBScores()), this,SLOT(slotTeamBScores()) );
        connect(GAGame, SIGNAL(clearField()), this, SLOT(slotClearField()) );
        connect(GAGame, SIGNAL(gameOver()), this, SLOT(slotGameOver()) );
}
//-----------------------------------------------------------------
void GABot::slotBotMoveA(Coordinate P1, Coordinate P2)
{
        emit(moveTeamA(P1,P2));
}
//-----------------------------------------------------------------
void GABot::slotBotMoveB(Coordinate P1, Coordinate P2)
{
        emit(moveTeamB(P1,P2));
}
//-----------------------------------------------------------------
void GABot::slotBotDirectionA(Coordinate P, Direction B)
{
        emit(turnTeamA(P,B));
}
//-----------------------------------------------------------------
void GABot::slotBotDirectionB(Coordinate P, Direction B)
{
        emit(turnTeamB(P,B));
}
//-----------------------------------------------------------------
void GABot::slotTeamAScores(void)
{
        emit(teamAScores());
}
//-----------------------------------------------------------------
void GABot::slotTeamBScores(void)
{
        emit(teamBScores());
}
//-----------------------------------------------------------------
void GABot::slotClearField(void)
{
        emit(clearField());
}
//-----------------------------------------------------------------
void GABot::slotGameOver(void)
{
        Tick->stop();
```

```
        emit(gameOver());
        GenAlg.evolve(TeamA);
        GenAlg.evolve(TeamB);
        if (TeamA->goals() > TeamB->goals()) {
                TeamA->wins(TeamA->wins() + 1);
                TeamB->losses(TeamB->losses() + 1);
        } else if (TeamA->goals() < TeamB->goals()) {
                TeamA->losses(TeamA->losses() + 1);
                TeamB->wins(TeamB->wins() + 1);
        } else {
                TeamA->ties(TeamA->ties() + 1);
                TeamB->ties(TeamB->ties() + 1);
        }
        delete GAGame;
        prepareGame();
        Tick->start(TickInterval);
}
//--------------------------------------------------------------------
```

```
/*********************************************************************
 * -= C++ =-
 * Game class: perform game playing
 *********************************************************************/

//predirectives
#include "game.h"
#include "team.h"
#include "ball.h"
#include "bot.h"
#include "coordinate.h"
#include "rotation.h"
#include "garule.h"
#include "thing.h"
#include "random.h"
#include <cstdlib>
#include <cmath>

#ifndef M_PI
#define M_PI            3.14159265358979323846  /* pi, from glibc 2.2.2 */
#endif

Game::Game(QObject *parent,Team *t1, Team *t2, unsigned int l, unsigned int w, unsigned int h)
        : QObject(parent,0) {


   B = new Ball(this);
   connect (B,SIGNAL(ballMove(Coordinate,Coordinate)),this,SLOT(slotBallMoved(Coordinate,Coordinate)));

   T1 = t1; T2 = t2;
   GameLength = l;
   Turns = 0;
   Width = w; Height = h;

   NetStart = Height / 3;
   NetEnd = Height - NetStart;

   reset();
}

Game::~Game() {
}

void Game::reset() {
   Coordinate pos;

   emit(clearField());

   // initialize bot positions
   unsigned int i, teamsize, rank; double spacing;
   Bot *curbot;
   teamsize = T1->size();
   spacing = 40.0 / (teamsize % 40);    // no more than 40 bots in a rank (will teams be bigger than that? :-)
   for (i=0; i<teamsize; i++) {
      curbot = T1->bot(i);
      rank = (int)((double)i / 40.0 + 0.5);
      pos.x = 35 - 5 * rank;
      pos.y = (int)( (spacing / 2.0) + (i % 40) * spacing + 0.5);
      pos = curbot->position(pos);
      curbot->direction(E);
   }
   teamsize = T2->size();
```

```
   spacing = 40.0 / (teamsize % 40);    // no more than 40 bots in a rank (will teams be big↙
ger than that? :-)
   for (i=0; i<teamsize; i++) {
      curbot = T2->bot(i);
      rank = (int)((double)i / 40.0 + 0.5);
      pos.x = 45 + 5 * rank;
      pos.y = (int)( (spacing / 2.0) + (i % 40) * spacing + 0.5);
      pos = curbot->position(pos);
      curbot->direction(W);
   }

   // reset the ball
   pos.x = (Width + 1) / 2;
   pos.y = (Height + 1) / 2;
   B->reset(pos);
}

void Game::turn() {
   Bot *curbot;
   unsigned int teamsize;
   unsigned int ticks;
   GARule *state;
   GARule *best;

   Turns++;
   if (ballInNet() != 0) {  // check and score goals
      reset();              // start over if a goal was scored
   }

   teamsize = T1->size() > T2->size() ? T1->size() : T2->size();
   for (unsigned int i=0; i<teamsize; i++) { // do both teams at once
      if (i < T1->size()) {            // in case team sizes are different
         curbot = T1->bot(i);
         ticks = curbot->ticksUntilNextMove(curbot->ticksUntilNextMove() - 1);
         if (ticks == 0) {             // this bot's time is up!

            // reset ticks
            curbot->ticksUntilNextMove(curbot->mass());

            // here's where the state should be programmed into a rule
            state = botState(curbot, 0);

            // determine best match
            best = curbot->bestRule(state);

            // execute rule
            curbot->execRule(best, B, Width, Height);

            // handle collisions
            botCollision(curbot, 0);

         }
      }
      if (i < T2->size()) {            // now do it again for bot i on team 2
         curbot = T2->bot(i);
         ticks = curbot->ticksUntilNextMove(curbot->ticksUntilNextMove() - 1);
         if (ticks == 0) {             // this bot's time is up!

            // reset ticks
            curbot->ticksUntilNextMove(curbot->mass());

            // here's where the state should be programmed into a rule
            state = botState(curbot, 1);

            // determine best match
```

```
            best = curbot->bestRule(state);

            // execute rule
            curbot->execRule(best, B, Width, Height);

            // handle and resolve collisions
            botCollision(curbot, 1);

         }
      }
   }

   // do stuff to ball if it's moving
   if (B->speed() != 0) {
      ticks = B->ticksUntilNextMove(B->ticksUntilNextMove() - 1);
      if (ticks == 0) {

         // reset ticks
         B->ticksUntilNextMove(11 - B->speed());  // assumes max speed of 10

         // move it
         B->moveBall(Width, Height);

      }
   }
   if (Turns >= GameLength) emit(gameOver());
}

bool Game::over() {
   return (Turns >= GameLength);
}

unsigned int Game::posIndex(Coordinate first, Coordinate second) {
   // find angle, rounded to 45 degrees
   double angle = atan2(first.y - second.y, second.x - first.x);  // yeah, that's right
   angle *= 4.0 / M_PI;  // convert to a value from 0 to 4
   unsigned int index;   // now convert to an index into the sensor array
   if (angle < 0) {
      angle = fabs(angle);
      if (angle < 0.5) index = 2;
      else if (angle < 1.5) index = 3;
      else if (angle < 2.5) index = 4;
      else if (angle < 3.5) index = 5;
      else index = 6;
   } else {
      if (angle < 0.5) index = 2;
      else if (angle < 1.5) index = 1;
      else if (angle < 2.5) index = 0;
      else if (angle < 3.5) index = 7;
      else index = 6;
   }
   return index;
}

void Game::botCollision(Bot *b, unsigned int teamnum) {
   Team *myTeam, *otherTeam;
   Bot *curbot;
   Coordinate myPos, theirPos;
   unsigned int myBotNum;
   bool myBall;
   unsigned int i;

   if (teamnum == 0) {
      myTeam = T1;
      otherTeam = T2;
```

```
   } else {
      myTeam = T2;
      otherTeam = T1;
   }

   myPos = b->position();
   myBall = b->myBall();

   // handle collisions with my team
   for (i=0; i<myTeam->size(); i++) {
      curbot = myTeam->bot(i);
      if (curbot != b) {
         theirPos = curbot->position();
         if ( (theirPos.x == myPos.x) && (theirPos.y == myPos.y) ) {

            // right now, bots on the same team won't steal the ball
            // bots are courteous and let higher-numbered bots move

            // move back in the opposite direction
            switch (b->direction()) {
               case N:
                  myPos.y++;
                  break;
               case NE:
                  myPos.y++; myPos.x--;
                  break;
               case E:
                  myPos.x--;
                  break;
               case SE:
                  myPos.y--; myPos.x--;
                  break;
               case S:
                  myPos.y--;
                  break;
               case SW:
                  myPos.y--; myPos.x++;
                  break;
               case W:
                  myPos.x++;
                  break;
               case NW:
                  myPos.y++; myPos.x++;
                  break;
            }
            if (myPos.x < 0) myPos.x = 0;  // automagically do wall collisions
            if (myPos.y < 0) myPos.y = 0;
            if (myPos.x >= Width) myPos.x = Width - 1;
            if (myPos.y >= Height) myPos.y = Height - 1;
            b->position(myPos);
            if (b->myBall()) B->position(myPos);  // move ball along with player
         }
      } else myBotNum = i;
   }

   // handle collisions with other team
   for (i=0; i<otherTeam->size(); i++) {
      curbot = otherTeam->bot(i);
      theirPos = curbot->position();
      if ( (theirPos.x == myPos.x) && (theirPos.y == myPos.y) ) {
         if (myBall) {
            // see if the ball got traded (stolen)
            if (tradeBall(b, curbot, (teamnum == 0 ? 1 : 0), i)) {
               myBall = false;  // guess I lost it, eh?
            }
```

```
         } else if (curbot->myBall()) {
            if (tradeBall(curbot, b, teamnum, myBotNum)) {
               myBall = true;  // woo!
            }
         }

         // bots are _really_ courteous and let bots on the other team move

         // each bot turn right 45 degree to go around

         switch(b->direction()){
            case N:
                    b->direction(NE);
                    break;
            case NE:
                    b->direction(E);
                    break;
            case E:
                    b->direction(SE);
                    break;
            case SE:
                    b->direction(S);
                    break;
            case S:
                    b->direction(SW);
                    break;
            case SW:
                    b->direction(W);
                    break;
            case W:
                    b->direction(NW);
                    break;
            case NW:
                    b->direction(N);
                    break;
         }

         if (myPos.x < 0) myPos.x = 0;  // automagically do wall collisions
         if (myPos.y < 0) myPos.y = 0;
         if (myPos.x >= Width) myPos.x = Width - 1;
         if (myPos.y >= Height) myPos.y = Height - 1;
         b->position(myPos);
         if (b->myBall()) B->position(myPos);  // move ball along with player
      }
   }

   // handle collisions with the ball (only considered a collision if the player didn't alre⇙
ady have the ball)
   theirPos = B->position();
   if ( (theirPos.x == myPos.x) && (theirPos.y == myPos.y) && (!b->myBall()) ) {
      unsigned int lt = B->team();
      unsigned int lp = B->player();
      if (lt != 0) {  // count an interception if the ball was last touched by another playe⇙
r
         lt--;  // stupid: teamnum is 0 or 1, but lt is 0 to denote a ball that hasn't been ⇙
touched yet
         if (lt != teamnum) b->interceptions(b->interceptions()+1);
      }
      myBall = b->myBall(true);
      b->timeWithBall(b->timeWithBall()+1);  // increment time with ball
      B->team(teamnum+1);
      B->player(myBotNum);
      B->speed(0);  // move the ball along with the player holding it
   }
```

```
}

bool Game::tradeBall(Bot *a, Bot *b, unsigned int bTeam, unsigned int bNum) {
   double prob, rnd;
   bool ret = false;

   // larger bots have a better chance of getting the ball
   prob = 0.5 - ( (b->mass() - a->mass()) / 20.0 );
   rnd = Random::randd(0,1);
   if (rnd > prob) {  // bot b gets the ball
      a->myBall(false);
      b->myBall(true);
      B->team(bTeam);
      B->player(bNum);
      ret = true;
   }
   return ret;
}

GARule* Game::botState(Bot *b, unsigned int teamnum) {
   Team *myTeam, *otherTeam;
   Bot *curbot;
   unsigned int Range = 5;             // all sensor ranges set to 5
   Coordinate myPos, theirPos;
   GARule *state = new GARule(this);
   unsigned int mySensor[8];           // separate sensors that will be compiled at the end
   unsigned int otherSensor[8];        // these are absolute, they'll be converted to
   unsigned int ballSensor[8];         // relative at the end, based on the dir the bot is f↙
acing
   unsigned int netSensor[8];
   unsigned int wallSensor[8];
   unsigned int allSensors[8];
   unsigned int index, distance;       // some useful variables
   unsigned int i, j;                  // useful index variable
   int teamBall = 0;
   bool myBall;

   if (teamnum == 0) {
      myTeam = T1;
      otherTeam = T2;
   } else {
      myTeam = T2;
      otherTeam = T1;
   }

   // clear all the sensor arrays
   for (i=0; i<8; i++) mySensor[i] = 0;
   for (i=0; i<8; i++) otherSensor[i] = 0;
   for (i=0; i<8; i++) wallSensor[i] = 0;
   for (i=0; i<8; i++) netSensor[i] = 0;
   for (i=0; i<8; i++) ballSensor[i] = 0;

   myPos = b->position();
   myBall = b->myBall();
   if (myBall) teamBall = 1;

   // scan for bots on my own team
   for (i=0; i<myTeam->size(); i++) {
      curbot = myTeam->bot(i);
      if (curbot != b) {  // not me
         theirPos = curbot->position();
         distance = (unsigned int)sqrt( pow(myPos.x - theirPos.x, 2) +  // compute distance ↙
to object
                                        pow(myPos.y - theirPos.y, 2) );
         if ( distance <= Range ) {             // within range, so I can see it
```

```
            index = posIndex(myPos, theirPos);  // compute index into sensor array
            mySensor[index] = distance;
         }
      }
      if (curbot->myBall()) teamBall = 1;
   }

   // scan for bots on other teams
   for (i=0; i<otherTeam->size(); i++) {
      curbot = otherTeam->bot(i);
      theirPos = curbot->position();
      distance = (unsigned int)sqrt( pow(myPos.x - theirPos.x, 2) +  // compute distance to object
                                     pow(myPos.y - theirPos.y, 2) );
      if (distance <= Range) {
         index = posIndex(myPos, theirPos);
         otherSensor[index] = distance;
      }
      if (curbot->myBall()) teamBall = -1;
   }

   // determine if the ball is in range
   theirPos = B->position();
   distance = (unsigned int)sqrt( pow(myPos.x - theirPos.x, 2) +  // compute distance to object
                                  pow(myPos.y - theirPos.y, 2) );
   if (distance <= Range) {
      index = posIndex(myPos, theirPos);
      ballSensor[index] = distance;
   }

   // find walls
   if (myPos.y < Range) {  // north wall
      wallSensor[0] = wallSensor[1] = wallSensor[7] = myPos.y + 1;  // since technically the wall's at y=-1
   } else if ( (Height - myPos.y) <= Range) {  // south wall
      wallSensor[3] = wallSensor[4] = wallSensor[5] = Height - myPos.y;
   }
   if (myPos.x < Range) {  // west wall
      wallSensor[5] = wallSensor[6] = wallSensor[7] = myPos.x + 1;
   } else if ( (Width - myPos.x) <= Range) {  // east wall
      wallSensor[1] = wallSensor[2] = wallSensor[3] = Width - myPos.x;
   }

   // find net
   if (teamnum == 0) {
      if ( (Width - myPos.x) <= Range) {
         if (myPos.y == NetStart) netSensor[2] = netSensor[3] = Width - myPos.x;
         else if (myPos.y > NetStart) {
            if (myPos.y == NetEnd) netSensor[1] = netSensor[2] = Width - myPos.x;
            else if (myPos.y < NetEnd) netSensor[1] = netSensor[2] = netSensor[3] = Width - myPos.x;
         }
      }
   } else {
      if (myPos.x < Range) {
         if (myPos.y == NetStart) netSensor[2] = netSensor[3] = myPos.x + 1;
         else if (myPos.y > NetStart) {
            if (myPos.y == NetEnd) netSensor[1] = netSensor[2] = myPos.x + 1;
            else if (myPos.y < NetEnd) netSensor[1] = netSensor[2] = netSensor[3] = myPos.x + 1;
         }
      }
   }
```

```
   // now find the closest object and put that in state
   Thing *sensors = new Thing[8];
   for (i=0; i<8; i++) {
      unsigned int offset;            // offset of index into direction in state
      switch(b->direction()) {
         case N:  offset = 0; break;  // already in right order
         case NE: offset = 7; break;
         case E:  offset = 6; break;
         case SE: offset = 5; break;
         case S:  offset = 4; break;
         case SW: offset = 3; break;
         case W:  offset = 2; break;
         case NW: offset = 1; break;
      }
      index = (i + offset) % 8;
      sensors[index] = Nothing;
      // search for nearest object, and place it in the real sensor array
      allSensors[i] = Range + 1;
      if ( (mySensor[i] > 0) && (mySensor[i] < allSensors[i]) ) {
         allSensors[i] = mySensor[i];
         sensors[index] = MyBot;
      }
      if ( (otherSensor[i] > 0) && (otherSensor[i] < allSensors[i]) ) {
         allSensors[i] = otherSensor[i];
         sensors[index] = OtherBot;
      }
      if ( (ballSensor[i] > 0) && (ballSensor[i] < allSensors[i]) ) {
         allSensors[i] = ballSensor[i];
         sensors[index] = TheBall;
      }
      if ( (netSensor[i] > 0) && (netSensor[i] < allSensors[i]) ) {
         allSensors[i] = netSensor[i];
         sensors[index] = Net;
      }
      if ( (wallSensor[i] > 0) && (wallSensor[i] < allSensors[i]) ) {
         allSensors[i] = wallSensor[i];
         sensors[index] = Wall;
      }
   }
   // at this point, sensors should be ready to stick into the rule
   state->sensors(sensors);

   // other stuff should be set, too
   state->teamBall(teamBall);
   state->myBall(myBall);

   delete [] sensors;
   // should be ready to go
   return state;
}

void Game::slotBallMoved(Coordinate Old, Coordinate New) {
        emit (ballMoved(Old, New));
}

unsigned int Game::ballInNet() {
   Coordinate pos = B->position();
   Bot *b;

   unsigned int ret = 0;

   if ( (pos.y >= NetStart) && (pos.y <= NetEnd) ) {
      if (pos.x == 0) {
         // in team A's net, so team B gets the point
         T2->goals(T2->goals() + 1);
```

```
         emit(teamBScores());
         ret = 2;
         if (B->team() == 1) {
            // Oh, you suck! You scored on your own net!
            b = T1->bot(B->player());
            b->goals(b->goals() - 1);
         } else if (B->team() == 2) {
            // YAY!
            b = T2->bot(B->player());
            b->goals(b->goals() + 1);
         }
      } else if (pos.x == (Width - 1)) {
         // in team B's net, so team A gets the point
         T1->goals(T1->goals() + 1);
         emit(teamAScores());
         ret = 1;
         if (B->team() == 1) {
            // YAY!
            b = T1->bot(B->player());
            b->goals(b->goals() + 1);
         } else if (B->team() == 2) {
            // Oh, you suck! You scored on your own net!
            b = T2->bot(B->player());
            b->goals(b->goals() - 1);
         }
      }
   }
   return ret;
}
```

```
/*************************************************************************
 * -= C++ =-
 *
 * Genetic Algorithm Rule
 *
 *************************************************************************/

#include "garule.h"
#include "thing.h"
#include "random.h"
#include <cstdlib>

GARule::GARule(QObject *parent,const char *name):QObject(parent,name) {
   // makes dumbot rule that just stands there
   TeamBall = 0;
   MyBall = false;
   for (int i = 0; i < 8; i++){
      Sensors[i] = Nothing;
   }
   Fire = false;
   Move = false;
   Turn = None;
}

GARule::~GARule() {
}

GARule::GARule(const GARule& t){

   TeamBall = t.TeamBall;
   MyBall = t.MyBall;
   for (int i = 0; i < 8 ;i++){
      Sensors[i] = t.Sensors[i];
   }
   Fire = t.Fire;
   Move = t.Move;
   Turn = t.Turn;

}

//copy assignment
GARule& GARule::operator=(const GARule& t){

   if(this != &t){
      TeamBall = t.TeamBall;
      MyBall = t.MyBall;
      for (int i = 0; i < 8; i++){
         Sensors[i] = t.Sensors[i];
      }
      Fire = t.Fire;
      Move = t.Move;
      Turn = t.Turn;
   }
   return *this;
}


int GARule::teamBall(){
   return(TeamBall);
}

bool GARule::myBall(){
   return(MyBall);
}
```

```
Thing* GARule::sensors(){
   return(Sensors);
}

bool GARule::fire(){
   return(Fire);
}

bool GARule::move(){
   return(Move);
}

Rotation GARule::turn(){
   return(Turn);
}

int GARule::teamBall(int t){
   TeamBall = t;
   return(TeamBall);
}

bool GARule::myBall(bool b){
   MyBall = b;
   return(MyBall);
}

Thing* GARule::sensors(Thing s[]){
   for (int i = 0; i < 8; i++){
      Sensors[i] = s[i];
   }
   return(Sensors);
}

bool GARule::fire(bool f){
   Fire = f;
   return(Fire);
}

bool GARule::move(bool m){
   Move = m;
   return(Move);
}

Rotation GARule::turn(Rotation t){
   Turn = t;
   return(Turn);
}

unsigned int GARule::difference(GARule *cond) {
   unsigned int diff;
   Thing *condSensors, *mySensors;

   diff = 0;                          // initialize difference

   if ( abs(cond->teamBall() - teamBall()) > 1 ) {  // one is +ve, one is -ve
      // above line only works if valid values for TeamBall are -1, 0, and 1
      diff += 2;                      // opposite conditions, give it a +2
   } else if ( abs(cond->teamBall() - teamBall()) == 1 ) {  // one is zero, other is not
      // same, only works if TeamBall is one of -1, 0, or 1
      diff += 1;                      // close conditions, give it a +1
   }

   if ( cond->myBall() != myBall() ) {  // not the same, this is bad
      diff += 10;                     // weighed more than the rest, since having the
   }                                  // ball is a lot different than not having it
```

```
   condSensors = cond->sensors();
   mySensors = sensors();
   for (unsigned int i=0; i<8; i++) {
      if ( condSensors[i] != mySensors[i] ) {
         diff++;                        // increment for each differing sensor
      }
   }

   return(diff);
}

void GARule::randomRule() {
   int rInt;                            // random integer in case we need it more than once
   int j;                               // variable for scratch use

   j = teamBall(Random::randint(-1,1));
   if (j == 1) myBall(Random::randbool());     // have at least some logic in the random rul⇗
es;
   else myBall(false);                  // if my team doesn't have the ball, then I can't hav⇗
e it

   Thing sens[8];
   for (int i = 0; i < 8; i++){
      rInt = Random::randint(0,5);
      switch(rInt){

         case(0):
              sens[i] = Nothing;
              break;
         case(1):
              sens[i] = MyBot;
              break;
         case(2):
              sens[i] = OtherBot;
              break;
         case(3):
              sens[i] = TheBall;
              break;
         case(4):
              sens[i] = Net;
              break;
         case(5):
              sens[i] = Wall;
              break;

      }//end switch
   }//end for
   sensors(sens);
   fire(Random::randbool());
   move(Random::randbool());
   rInt = Random::randint(0,2);

   if (rInt == 0)      turn(BotRotation::Left);
   else if (rInt == 1) turn(BotRotation::Right);
   else                turn(BotRotation::None);
   // generates rules with more than one action, but should work anyway since
   // only one action will occur
}

void GARule::mutateRule() {
   int rule = Random::randint(0,12);
   int rInt, j;

   if (rule < 8) {  // modify the sensor array
```

```
      rInt = Random::randint(0,5);
      Thing *sensold, sens[8];
      sensold = sensors();
      for (j=0; j<8; j++) sens[j] = sensold[j];
          sensold = 0;
      switch(rInt){
         case(0):
              sens[rule] = Nothing;
              break;
         case(1):
              sens[rule] = MyBot;
              break;
         case(2):
              sens[rule] = OtherBot;
              break;
         case(3):
              sens[rule] = TheBall;
              break;
         case(4):
              sens[rule] = Net;
              break;
         case(5):
              sens[rule] = Wall;
              break;
      }
      sensors(sens);
   } else {
      switch(rule) {
         case 8:  // teamBall
            j = teamBall(Random::randint(-1,1));
         case 9:  // myBall
            j = teamBall();
            if (j == 1) myBall(!myBall());
            break;
         case 10: // fire
            fire(!fire());
            break;
         case 11: // move
            move(!move());
            break;
         case 12: // turn
            rInt = Random::randint(0,2);
            if (rInt == 0)      turn(BotRotation::Left);
            else if (rInt == 1) turn(BotRotation::Right);
            else                turn(BotRotation::None);
            break;
      }
   }
}
```

```
/*****************************************************************
 * -= C++ =-
 *
 * Genetic Algorithm bot program demo
 *
 *****************************************************************/

#include <qapplication.h>
#include <qfont.h>
#include <qstring.h>
#include <qtextcodec.h>
#include <qtranslator.h>
#include <qstyle.h>
#include <qwindowsstyle.h>

#include "mainwindow.h"
#include "random.h"

int main(int argc, char *argv[])
{
        QApplication a(argc, argv);

        Random::initseed();

        a.setStyle(new QWindowsStyle);
        MainWindow *Main=new MainWindow();
        a.setMainWidget(Main);
        Main->show();

        int Err = a.exec();
        delete Main;

        return Err;
}
```

```
// QT Includes
#include <qapp.h>
#include <qaccel.h>
#include <qdir.h>
#include <qcombobox.h>
#include <qtimer.h>
#include <qmainwindow.h>
#include <qaction.h>
#include <qmenubar.h>
#include <qpopupmenu.h>
#include <qtoolbar.h>
#include <qtoolbutton.h>
#include <qstatusbar.h>
#include <qwhatsthis.h>
#include <qstring.h>
#include <qpixmap.h>
#include <qmsgbox.h>
#include <qfiledialog.h>
#include <qprinter.h>
#include <qpainter.h>
#include <qslider.h>

// Application specific includes
#include "mainwindow.h"
#include "botview.h"
#include "gabot.h"
#include "go.xpm"
#include "stop.xpm"
#include "lab.xpm"

//----------------------------------------------------------------------------

MainWindow::MainWindow()
{
        setCaption(tr("GA Bots " VERSION));

        initActions();
        initMenuBar();
        initToolBar();
        initStatusBar();

        initGABotDoc();
        initView();

        ViewToolBar->setOn(true);
        ViewStatusBar->setOn(true);
        //ViewGame->setOn(true);

        connect(GABotDoc, SIGNAL(gameReady(bool)), this, SLOT(slotGameReady(bool)));
        connect(GABotDoc, SIGNAL(teamAScores()), View, SLOT(slotScoreA()));
        connect(GABotDoc, SIGNAL(teamBScores()), View, SLOT(slotScoreB()));
        connect(GABotDoc, SIGNAL(clearScores()), View, SLOT(slotClearScores()));

        connect(GABotDoc, SIGNAL(moveTeamA(Coordinate, Coordinate)), View,
                SLOT(slotMoveTeamA(Coordinate, Coordinate)));
        connect(GABotDoc, SIGNAL(moveTeamB(Coordinate, Coordinate)), View,
                SLOT(slotMoveTeamB(Coordinate, Coordinate)));
        connect(GABotDoc, SIGNAL(turnTeamA(Coordinate, Direction)), View,
                SLOT(slotTurnTeamA(Coordinate, Direction)));
        connect(GABotDoc, SIGNAL(turnTeamB(Coordinate, Direction)), View,
                SLOT(slotTurnTeamB(Coordinate, Direction)));

        connect(GABotDoc, SIGNAL(moveBall(Coordinate,Coordinate)), View,
                SLOT(slotMoveBall(Coordinate,Coordinate)));
```

```
        connect(GoGame, SIGNAL(activated()), this, SLOT(slotGoGame()));
        connect(GoGame, SIGNAL(activated()), GABotDoc, SLOT(slotStartTimer()));
        connect(StopGame, SIGNAL(activated()), GABotDoc, SLOT(slotStopTimer()));
        connect(StopGame, SIGNAL(activated()), this, SLOT(slotStopGame()));

        connect(View, SIGNAL(valueChanged(int)), this, SLOT(slotTickInterval(int)));
        connect(GABotDoc, SIGNAL(clearField()), View, SLOT(slotClearField()));

        connect(ResetScreen, SIGNAL(activated()), View, SLOT(slotClearField()));
//not sure if we should clear the field whenever regenerating the team
//      connect(TeamAGenerate,SIGNAL(activated()), View, SLOT(slotClearField()));
//      connect(TeamBGenerate,SIGNAL(activated()), View, SLOT(slotClearField()));

        connect(GABotDoc, SIGNAL(gameOver()), View, SLOT(slotClearField()));
        connect(GABotDoc, SIGNAL(gameOver()), View, SLOT(slotClearScores()));
        connect(GABotDoc, SIGNAL(gameOver()), this, SLOT(slotGameOver()));

}

//-------------------------------------------------------------------------------

MainWindow::~MainWindow()
{

}

//-------------------------------------------------------------------------------

void MainWindow::initActions(){

        TeamAFileOpen = new QAction(tr("Load Team A"), tr("Load Team &A..."), 0, this);
        TeamAFileOpen->setStatusTip(tr("Load a rule set for team A"));
        connect(TeamAFileOpen, SIGNAL(activated()), this, SLOT(slotLoadTeamA()));

        TeamBFileOpen = new QAction(tr("Load Team B"), tr("Load Team &B..."), 0, this);
        TeamBFileOpen->setStatusTip(tr("Load a rule set for team B"));
        connect(TeamBFileOpen, SIGNAL(activated()), this, SLOT(slotLoadTeamB()));

        TeamAFileSave = new QAction(tr("Save Team A"), tr("Save Team A..."), 0, this);
        TeamAFileSave->setStatusTip(tr("Save team A data"));
        connect(TeamAFileSave, SIGNAL(activated()), this, SLOT(slotSaveTeamA()));

        TeamBFileSave = new QAction(tr("Save Team B"), tr("Save Team B..."), 0, this);
        TeamBFileSave->setStatusTip(tr("Save team B data"));
        connect(TeamBFileSave, SIGNAL(activated()), this, SLOT(slotSaveTeamB()));

        TeamAGenerate = new QAction(tr("Generate Team A"), tr("Generate Team A"), 0, this);
        TeamAGenerate->setStatusTip(tr("Generate Team A"));
        connect(TeamAGenerate, SIGNAL(activated()), this, SLOT(slotGenerateTeamA()));

        TeamBGenerate = new QAction(tr("Generate Team B"), tr("Generate Team B"), 0, this);
        TeamBGenerate->setStatusTip(tr("Generate Team B"));
        connect(TeamBGenerate, SIGNAL(activated()), this, SLOT(slotGenerateTeamB()));

        FileQuit = new QAction(tr("Exit"), tr("E&xit"), 0, this);
        FileQuit->setStatusTip(tr("Quits the application"));
        FileQuit->setWhatsThis(tr("Exit\n\nQuits the application"));
        connect(FileQuit, SIGNAL(activated()), this, SLOT(slotFileQuit()));

        ViewToolBar = new QAction(tr("Toolbar"), tr("Tool&bar"), 0, this, 0, true);
        ViewToolBar->setStatusTip(tr("Enables/disables the toolbar"));
        ViewToolBar->setWhatsThis(tr("Toolbar\n\nEnables/disables the toolbar"));
        connect(ViewToolBar, SIGNAL(toggled(bool)), this, SLOT(slotViewToolBar(bool)));

        ViewStatusBar = new QAction(tr("Statusbar"), tr("&Statusbar"), 0, this, 0, true);
```

```
        ViewStatusBar->setStatusTip(tr("Enables/disables the statusbar"));
        ViewStatusBar->setWhatsThis(tr("Statusbar\n\nEnables/disables the statusbar"));
        connect(ViewStatusBar, SIGNAL(toggled(bool)), this, SLOT(slotViewStatusBar(bool)));

        /*
        ViewGame = new QAction(tr("Game Display"), tr("&Game Display"), 0, this, 0, true);
        ViewGame->setStatusTip(tr("Enables/disables the game play on screen"));
        ViewGame->setWhatsThis(tr("Game Display\n\nEnables/disables the game play on screen"
));
        connect(ViewGame, SIGNAL(toggled(bool)), this, SLOT(slotViewGame(bool)));
        */

        HelpAbout = new QAction(tr("About"), tr("&About..."), 0, this);
        HelpAbout->setStatusTip(tr("About the application"));
        HelpAbout->setWhatsThis(tr("About\n\nAbout the application"));
        connect(HelpAbout, SIGNAL(activated()), this, SLOT(slotHelpAbout()));

        ResetScreen = new QAction(tr("Reset Screen"), tr("Reset Screen"), 0, this);
        ResetScreen->setStatusTip(tr("Reset Screen"));
        ResetScreen->setWhatsThis(tr("Reset Screen\n\nReset Arena Display"));
        //connect(ResetScreen, SIGNAL(activated()), View, SLOT(slotClearField()));

}

//-----------------------------------------------------------------------------

void MainWindow::initMenuBar()
{

        //----
        FileMenu=new QPopupMenu();
        TeamAFileOpen->addTo(FileMenu);
        TeamBFileOpen->addTo(FileMenu);
        FileMenu->insertSeparator();
        TeamAFileSave->addTo(FileMenu);
        TeamBFileSave->addTo(FileMenu);
        FileMenu->insertSeparator();
        TeamAGenerate->addTo(FileMenu);
        TeamBGenerate->addTo(FileMenu);
        FileMenu->insertSeparator();
        ResetScreen->addTo(FileMenu);
        FileMenu->insertSeparator();
        FileQuit->addTo(FileMenu);

        //---- menuBar entry viewMenu
        ViewMenu=new QPopupMenu();
        ViewMenu->setCheckable(true);
        ViewToolBar->addTo(ViewMenu);
        ViewStatusBar->addTo(ViewMenu);
        //ViewGame->addTo(ViewMenu);

        //---- menuBar entry helpMenu
        HelpMenu=new QPopupMenu();
        HelpAbout->addTo(HelpMenu);

        //----
        menuBar()->insertItem(tr("&File"), FileMenu);
        menuBar()->insertItem(tr("&View"), ViewMenu);
        menuBar()->insertSeparator();
        menuBar()->insertItem(tr("&Help"), HelpMenu);

        TeamAFileSave->setEnabled(FALSE);
        TeamBFileSave->setEnabled(FALSE);

}
```

```
//-----------------------------------------------------------------------------

void MainWindow::initToolBar()
{
        Toolbar = new QToolBar(this);
        Toolbar->setHorizontalStretchable(TRUE);
        Toolbar->setVerticalStretchable(TRUE);
        StopGame = new QAction(0, QPixmap(Stop_XPM), 0, 0, this);
        GoGame = new QAction(0, QPixmap(Go_XPM), 0, 0, this);

        StopGame->addTo(Toolbar);
        StopGame->setEnabled(FALSE);

        GoGame->addTo(Toolbar);
        GoGame->setEnabled(FALSE);

        QWidget *MT = new QWidget(Toolbar);
        Toolbar->setStretchableWidget(MT);

}

//-----------------------------------------------------------------------------

void MainWindow::initStatusBar()
{
        statusBar()->message(tr("Ready."), 2000);
}

//-----------------------------------------------------------------------------

void MainWindow::initGABotDoc()
{
        // Create a new doc
        GABotDoc = new GABot(this);
}

//-----------------------------------------------------------------------------

void MainWindow::initView()
{
        // set the main widget here
        View = new BotView(this);
        setCentralWidget(View);
}

//-----------------------------------------------------------------------------

bool MainWindow::queryExit()
{
        int exit=QMessageBox::information(this, tr("Quit..."),
                tr("Do your really want to quit?"),
                QMessageBox::Ok, QMessageBox::Cancel);

        if (exit==1) {
                // do something
        } else {
                // do something else
        };
        return (exit==1);
}

//-----------------------------------------------------------------------------

void MainWindow::teamFileOpen(int TeamNumber)
```

```
{
        statusBar()->message(tr("Opening file..."));
        QFileDialog *TempFileDialog;
        TempFileDialog = new QFileDialog;
        TempFileDialog->setMode(QFileDialog::Directory);
        QString FileName = TempFileDialog->getOpenFileName(0,0,this,0,0);
        if (!FileName.isEmpty()) {
                if (GABotDoc->loadTeamFromFile(FileName,TeamNumber)) {
                        if (TeamNumber) {
                                statusBar()->message(tr("Loaded file "+FileName+" into Team B."), 5000);
                                TeamBFileSave->setEnabled(TRUE);
                        } else {
                                statusBar()->message(tr("Loaded file "+FileName+" into Team A."), 5000);
                                TeamAFileSave->setEnabled(TRUE);
                        }
                } else {
                        QMessageBox::warning(this,tr("Invalid file format"),
                                tr("\nThe selected file is not a GA Bot XML file, you idiot."));
                        statusBar()->message(tr("Opening aborted"), 1000);
                }
        } else {
                statusBar()->message(tr("Opening aborted"), 1000);
        }
        delete TempFileDialog;
}

//---------------------------------------------------------------------------

void MainWindow::teamFileSave(int TeamNumber)
{
        statusBar()->message(tr("Saving team data file..."));
        QFileDialog *TempFileDialog = new QFileDialog;
        TempFileDialog->setMode(QFileDialog::Directory);
        QString FileName = TempFileDialog->getSaveFileName(0,0,this,0,0);
        if(!FileName.isEmpty()){
                if(GABotDoc->saveTeamToFile(FileName,TeamNumber)){
                        if (TeamNumber) {
                                statusBar()->message(tr("Saving team B to "+FileName+"."), 5000);
                        }else {
                                statusBar()->message(tr("Saving team A to "+FileName+"."), 5000);
                        }
                }else{
                        QMessageBox::warning(this,tr("File saving error"),tr("You probably don't have enough space on disk or something."));
                        statusBar()->message(tr("Saving aborted"), 1000);
                }
        }else{
                statusBar()->message(tr("Saving aborted"), 1000);
        }
        delete TempFileDialog;

}

//---------------------------------------------------------------------------

void MainWindow::teamGenerateRandom(int TeamNumber)
{
        GABotDoc->randomTeam(TeamNumber);
}
```

```
//-----------------------------------------------------------------------------

void MainWindow::slotFileClose()
{
        statusBar()->message(tr("Closing file..."));
        // Close the file, I suppose -- may not need this for this app.  Dunno.
        // Presumably it would be a GABotDoc->function() thing
        statusBar()->message(tr("Ready."), 2000);
}

//-----------------------------------------------------------------------------

void MainWindow::slotFileQuit()
{
        statusBar()->message(tr("Exiting application..."));

        // exits the Application
/*  Comment this out until we have a doc.  In fact, I'm not even sure
 *  if this applies to our project or not.  Probably not. I'm hungry.
        if(Doc->isModified()) {
                if(queryExit()) {
                        // Prompt luser to save
                        qApp->quit();
                }
        } else {
                qApp->quit();
        }
*/
        qApp->quit();
}

//-----------------------------------------------------------------------------

void MainWindow::slotViewToolBar(bool toggle)
{
        statusBar()->message(tr("Toggle toolbar..."));

        // Toggle your Toolbar (get your head out of the gutter)

        if (toggle) {
                Toolbar->show();
        } else {
                Toolbar->hide();
        }

        statusBar()->message(tr("Ready."), 2000);
}

//-----------------------------------------------------------------------------

void MainWindow::slotViewStatusBar(bool toggle)
{
        statusBar()->message(tr("Toggle statusbar..."));

        if (toggle) {
                statusBar()->show();
        } else {
                statusBar()->hide();
        }

        statusBar()->message(tr("Ready."), 2000);
}

//-----------------------------------------------------------------------------
void MainWindow::slotViewGame(bool toggle)
```

```
{
        //currently doing nothing
}
//-----------------------------------------------------------------------------

void MainWindow::slotHelpAbout()
{
        QMessageBox About(this,0);
        About.setIconPixmap(QPixmap(Lab_XPM));
        About.setCaption(tr("About..."));
        About.setText(tr("GA Bots\nVersion " VERSION "\n                \n(c) 2002 by Edmond
 Lau, Chris Odorjan and Richard Voino") );
        About.exec();
}

//-----------------------------------------------------------------------------

void MainWindow::loading(QString FileName)
{
        QString Message;
        if (FileName) {
                QString Message=tr("Loading ")+FileName+tr("...");
                statusBar()->message(Message);
        } else {
                Message=tr("Done.");
                statusBar()->message(Message, 2000);
        }
}

//-----------------------------------------------------------------------------

void MainWindow::slotLoadTeamA (void)
{
        teamFileOpen(0);
}

//-----------------------------------------------------------------------------

void MainWindow::slotLoadTeamB (void)
{
        teamFileOpen(1);
}

//-----------------------------------------------------------------------------

void MainWindow::slotSaveTeamA (void)
{
   teamFileSave(0);
}

//-----------------------------------------------------------------------------

void MainWindow::slotSaveTeamB (void)
{
   teamFileSave(1);
}

//-----------------------------------------------------------------------------

void MainWindow::slotGenerateTeamA(void)
{
        teamGenerateRandom(0);
        TeamAFileSave->setEnabled(TRUE);
}
```

```
//-------------------------------------------------------------------------------

void MainWindow::slotGenerateTeamB(void)
{
        teamGenerateRandom(1);
        TeamBFileSave->setEnabled(TRUE);
}

//-------------------------------------------------------------------------------

void MainWindow::slotGoGame(void)
{
        GoGame->setEnabled(FALSE);
        StopGame->setEnabled(TRUE);
}

//-------------------------------------------------------------------------------

void MainWindow::slotStopGame(void)
{
        StopGame->setEnabled(FALSE);
        GoGame->setEnabled(TRUE);
}

//-------------------------------------------------------------------------------

void MainWindow::slotGameReady(bool GameReady)
{
        GoGame->setEnabled(GameReady);
}

//-------------------------------------------------------------------------------

void MainWindow::slotTickInterval(int Inter)
{
        GABotDoc->slotTickInterval(1025 - Inter);
}

//-------------------------------------------------------------------------------

void MainWindow::slotGameOver(void)
{
        statusBar()->message(tr("Game ended. Evolving teams, and preparing new game."), 5000
);
}

//-------------------------------------------------------------------------------
```

```
/*********************************************************************
 * -= C++ =-
 * some useful functions for generating random numbers
 *********************************************************************/

#include "random.h"
#include <cstdlib>
#include <ctime>

void Random::initseed() {
   srand(time(NULL));
}

int Random::randint(int start, int end) {
   int r;

   r = start + (int) ((end - start + 1.0)*rand()/(RAND_MAX+1.0));
   return r;
}

double Random::randd(double start, double end) {
   double r;

   r = start + (end - start + 1.0)*rand()/(RAND_MAX+1.0);
   return r;
}

bool Random::randbool() {
   double rd;
   bool r;

   rd = randd(0,1);
   r = (rd < 0.5 ? false : true);
   return r;
}
```

```
/*****************************************************************
 * -= C++ =-
 *
 * Simple Genetic Algorithm - Perform GA on the given team
 *
 *****************************************************************/

// predirectives
#include "simplega.h"
#include "team.h"
#include "bot.h"
#include "garule.h"
#include "random.h"
#include <qarray.h>

SimpleGA::SimpleGA() {
   MutationRate = 0;
}

SimpleGA::SimpleGA(double mRate) {
   MutationRate = mRate;
}

SimpleGA::~SimpleGA() {
}

double SimpleGA::mutationRate() {
   return(MutationRate);
}

double SimpleGA::mutationRate(double rate) {
   MutationRate = rate;
   return(MutationRate);
}

void SimpleGA::crossover(Bot *aIn, Bot *bIn, Bot *aOut, Bot *bOut) {
   unsigned int aRules, bRules, i;
   unsigned int aMass, bMass;
   double aProp, bProp;
   GARule *r;

   aRules = Random::randint(1, aIn->ruleSetSize()); aRules--;
   bRules = Random::randint(1, bIn->ruleSetSize()); bRules--;

   for (i=0; i<aIn->ruleSetSize(); i++) {
      r = new GARule;
      *r = *(aIn->rule(i));  // I hope this makes a deep copy
      if (i <= aRules) {
         // add to first child bot
         aOut->insertRule(r);
      } else {
         // add to second child bot
         bOut->insertRule(r);
      }
   }
   for (i=0; i<bIn->ruleSetSize(); i++) {
      r = new GARule;
      *r = *(bIn->rule(i));
      if (i <= bRules) {
         aOut->insertRule(r);
      } else {
         bOut->insertRule(r);
      }
   }
```

```
   // new bot masses are determined by the amount of each ruleset they took
   aProp = (double)aRules / (double)(aIn->ruleSetSize());
   bProp = (double)bRules / (double)(bIn->ruleSetSize());
   aMass = (unsigned int) ( aProp * aIn->mass() + bProp * bIn->mass() + 0.5);
   bMass = (unsigned int) ( (1.0 - aProp) * aIn->mass() + (1.0 - bProp) * bIn->mass() + 0.5)
;
   if (aMass < 1) aMass = 1;   if (bMass < 1) bMass = 1;
   if (aMass > 10) aMass = 10; if (bMass > 10) bMass = 10;
   aOut->mass(aMass);
   bOut->mass(bMass);

}

void SimpleGA::evolve(Team *T) {
   Team *oldTeam, *newTeam, *delTeam;
   QArray<floatbot> botFitness;
   unsigned int teamSize, breed;
   int i;
   Bot *curbot;
   Bot *newbot1, *newbot2;

   oldTeam = T;
   newTeam = new Team;
   delTeam = new Team;

   teamSize = oldTeam->size();
   botFitness.resize(teamSize);
   for (i=0; i< (int)teamSize; i++) {
      curbot = oldTeam->bot(i);
      botFitness[i].fit = curbot->fitnessFunction();
      botFitness[i].bot = i;
   }
   botFitness.sort();

   // reverse fitness array
   for (i=0; i<(int)(teamSize/2); i++) {
      floatbot temp;
      temp.fit = botFitness[i].fit;
      temp.bot = botFitness[i].bot;
      botFitness.at(i).fit = botFitness.at(teamSize-i-1).fit;
      botFitness.at(i).bot = botFitness.at(teamSize-i-1).bot;
      botFitness.at(teamSize-i-1).fit = temp.fit;
      botFitness.at(teamSize-i-1).bot = temp.bot;
   }

   // add the good bots to the new team
   breed = teamSize / 2;                 // number of bots to breed
   if ( (breed/2)*2 != breed ) breed--;  // make sure its even
   for (i=0; i<(int)(teamSize - breed); i++) {
      // number of bots to throw out is the number bred, so keep the number not thrown out
      curbot = oldTeam->bot(botFitness[i].bot);
      newTeam->insertBot(curbot);
   }
   for (i=(teamSize - breed); i<(int)teamSize; i++) {
      // put bots to be deleted on a separate team
      curbot = oldTeam->bot(botFitness[i].bot);
      delTeam->insertBot(curbot);
   }
   while (oldTeam->size() > 0) {
      // remove bots from oldTeam
      curbot = oldTeam->removeBot(0);
   }
   while (delTeam->size() > 0) {
      // actually delete bots
      curbot = delTeam->removeBot(0);
```

```
      delete curbot;
   }

   // do the crossover of the best bots
   for (i=0; i<(int)breed; i+=2) {
      newbot1 = new Bot(oldTeam);
      newbot2 = new Bot(oldTeam);
      crossover(newTeam->bot(i), newTeam->bot(i+1), newbot1, newbot2);
      newTeam->insertBot(newbot1);
      newTeam->insertBot(newbot2);
   }

   // mutate a bot given the mutation rate
   for (i=0; i<(int)teamSize; i++) {
      double rnd = Random::randd(0,1);
      if (rnd < MutationRate) {
         curbot = newTeam->bot(i);
         curbot->mutateBot();
      }
   }

   // copy bots back into old team
   while (newTeam->size() > 0) {
      curbot = newTeam->removeBot(0);
      oldTeam->insertBot(curbot);
   }
   oldTeam->generations(oldTeam->generations() + 1);
   delete delTeam;
   delete newTeam;
}
```

```
// Team class: defines a "team" or population of Bots

#include "team.h"
#include "bot.h"
#include "random.h"
#include <qlist.h>

Team::Team(QObject *parent, const char *name)
        : QObject(parent,name) {
   wins(0);
   losses(0);
   ties(0);
   goals(0);
   generations(0);
   Bots.clear();
}

Team::~Team() {
   // bots get destroyed automagically
   Bots.clear();
}

unsigned int Team::size() {
   return(Bots.count());
}

QList<Bot> Team::bots() {
   return Bots;
}

QList<Bot> Team::bots(QList<Bot> bs) {
   Bots = bs;
   return Bots;
}

Bot* Team::removeBot(unsigned int num) {
   Bot *p = Bots.at(num);
   Bots.remove();
   return p;
}

unsigned int Team::insertBot(Bot *b, unsigned int num) {
   connect( b, SIGNAL(botMove(Coordinate,Coordinate)),
            this, SLOT(slotBotMove(Coordinate,Coordinate)));
   connect( b, SIGNAL(botDirection(Coordinate,Direction)),
            this, SLOT(slotBotDirection(Coordinate,Direction)));
   Bots.insert(num, b);
   return Bots.at();
}

unsigned int Team::insertBot(Bot *b) {
   connect( b, SIGNAL(botMove(Coordinate,Coordinate)),
            this, SLOT(slotBotMove(Coordinate,Coordinate)));
   connect( b, SIGNAL(botDirection(Coordinate,Direction)),
            this, SLOT(slotBotDirection(Coordinate,Direction)));
   Bots.append(b);
   return Bots.at();
}

Bot* Team::bot(unsigned int num) {
   return Bots.at(num);
}

unsigned int Team::wins() {
   return(Wins);
```

```
}

unsigned int Team::wins(unsigned int w) {
   Wins = w;
   return(Wins);
}

unsigned int Team::losses() {
   return(Losses);
}

unsigned int Team::losses(unsigned int l) {
   Losses = l;
   return(Losses);
}

unsigned int Team::ties() {
   return(Ties);
}

unsigned int Team::ties(unsigned int t) {
   Ties = t;
   return(Ties);
}

unsigned int Team::generations() {
   return Generations;
}

unsigned int Team::generations(unsigned int g) {
   Generations = g;
   return Generations;
}

QString Team::name(){
   return(Name);
}

QString Team::name(QString n){
   Name = n;
   return(Name);
}

void Team::randomTeam(unsigned int size) {
   unsigned int numrules;
   unsigned int mass;

   wins(0);
   losses(0);
   ties(0);
   goals(0);
   name("Team Stochastic");

   for (unsigned int i=0; i<size; i++) {
      numrules = Random::randint(100,200);       // pick a random number of rules for each b↙
ot
      mass = Random::randint(1,10);            // pick a random number for the bot mass
      Bot *NewBot = new Bot(this);
      NewBot->randomBot(numrules, mass);
      insertBot(NewBot);
   }
}

void Team::slotBotMove(Coordinate P1,Coordinate P2)
{
```

```
   emit(botMove(P1,P2));
}

void Team::slotBotDirection(Coordinate P,Direction B)
{
   emit((botDirection(P,B)));
}

int Team::goals() {
   return Goals;
}

int Team::goals(int g) {
   Goals = g;
   return Goals;
}
```

```
/*****************************************************************
 * -= C++ =-
 *
 * read and write team data to xml file
 *****************************************************************/

#include "teamdata.h"

//read data from xml file
Team* TeamData::readTeamData(QString filename){

   QFile xmlFile(filename);
   QXmlInputSource source(xmlFile);
   QXmlSimpleReader reader;
   TeamParser *handler = new TeamParser();
   reader.setContentHandler(handler);
   reader.parse(source);

   return(handler->teamData());
}

//write team data to file
bool TeamData::writeTeamData(QString filename, Team* team){

   // check if team success
   if (!team){
      return false;
   }

   //open the text file, if already exists overwrite
   QFile xmlFile(filename);
   if (xmlFile.open(IO_WriteOnly)){ //file opened as overwrite
      QTextStream ts(&xmlFile);
      QString str;

      //write team tag
      ts << "<team ";
      ts << "name=\"" << team->name() << "\" ";
      ts << "wins=\"" << QString::number(team->wins()) << "\" ";
      ts << "losses=\"" << QString::number(team->losses()) << "\" ";
      ts << "generations=\"" << QString::number(team->generations()) << "\" ";
      ts << "ties=\"" << QString::number(team->ties()) << "\">\n";

      //write bot tag
      Bot *b;
      for (unsigned int i = 0; i < team->size(); i++){
         b = team->bot(i); //get bot in sequence
         ts << "<bot ";
         ts << "mass=\"" << QString::number(b->mass()) << "\" ";
         ts << "goals=\"" << QString::number(b->goals()) << "\" ";
         ts << "interceptions=\"" << QString::number(b->interceptions()) << "\">\n";
         ts << "<gaRule>\n";

         //write rule tag
         GARule *r;
         for (unsigned int j = 0; j < b->ruleSetSize(); j++){
            r = b->rule(j);
            ts << "<rule ";
            ts << "teamBall=\"" << QString::number(r->teamBall()) << "\" ";
            ts << "myBall=\""; (r->myBall()) ? ts << "T" : ts << "F"; ts << "\" ";
            ts << "fire=\""; (r->fire()) ? ts << "T" : ts << "F";
            ts << "\" ";
            ts << "move=\""; (r->move()) ? ts << "T" : ts << "F";
            ts << "\" ";
            ts << "turn=\""; (r->turn() == Left) ? ts << "left" :
```

```
                        (r->turn() == Right) ? ts << "right" : ts << "none";
            ts << "\">\n";

            //write sensors tag
            Thing* sen = r->sensors();
            ts << "<sensors ";
            for (unsigned int k = 0; k < 8; k++){
               QString t;
               switch(sen[k]) {
                  case Nothing:  t = "nothing"; break;
                  case MyBot:    t = "mybot"; break;
                  case OtherBot: t = "otherbot"; break;
                  case TheBall:  t = "ball"; break;
                  case Wall:     t = "wall"; break;
                  case Net:      t = "net"; break;
               }//end switch

               //put the appropriate end tag
               ts << "s" << QString::number(k) << "=\"" << t;
               if (k < 7){
                  ts << "\" ";
               }else{
                  ts << "\"/>\n";
               }//end if

            }//end putting sensor tag
            ts << "</rule>\n";
         }//end for

         ts << "</gaRule>\n";
         ts << "</bot>\n";
      }//end for

      ts << "</team>\n";

      xmlFile.close(); //close file
   }//end if
   return(true);

}//end writeTeamData
```

```
/*****************************************************************
 * -= C++ =-
 *
 * read team data from xml files
 *****************************************************************/

//predirectives
#include <qstring.h>
#include <string>

#include "teamparser.h"
#include "thing.h"
#include "team.h"

//constructor
TeamParser::TeamParser():QXmlDefaultHandler(){
   team=0;
   botcount=0;
   rulecount=0;
   bot = 0;
}

//start document
bool TeamParser::startDocument()
{
   return true;
}

//this function execute whenver the parser sees a start tag of xml
bool TeamParser::startElement(const QString& namespaceURI, const QString& localName,
                        const QString& qName, const QXmlAttributes& attributes){

   // check tags and build the Team
   if (qName == "team"){
      if (attributes.length() > 0){
         team = new Team;

         for (int i = 0; i < attributes.length();i++){
            if (attributes.qName(i) == "name"){
               team->name(attributes.value(i));
            }else if(attributes.qName(i) == "wins"){
               team->wins((attributes.value(i)).toUInt());
            }else if(attributes.qName(i) == "losses"){
               team->losses((attributes.value(i)).toUInt());
            }else if(attributes.qName(i) == "ties"){
               team->ties((attributes.value(i)).toUInt());
            }else if(attributes.qName(i) == "generations"){
               team->generations((attributes.value(i)).toUInt());
            }else{
               debug("error in team tag");
               return (false);  //error in team tag
            }//end if
         }//end for
      }else{
         debug("error in team tag, missing all attributes");
         return(false);  //error in team tag, missing all attributes
      }//end if
   }else if(qName == "bot"){

      if (attributes.length() > 0){
         //allocate memory for the local bot
         bot = new Bot(team);

         for (int i = 0; i < attributes.length(); i++){
            if(attributes.qName(i) == "mass"){
```

```
            bot->mass((attributes.value(i)).toUInt());
         }else if(attributes.qName(i) == "goals"){
            bot->goals((attributes.value(i)).toInt());
         }else if(attributes.qName(i) == "interceptions"){
            bot->interceptions( (attributes.value(i)).toUInt());
         }else{
            debug("error in bot tag");
            return (false);  //error in bot tag
         }
      }
   }else{
      debug("error in bot tag, missing all attributes");
      return(false);  //error in bot tag, missing all attributes
   }//end if

}else if(qName == "gaRule"){
   rulecount = 0;
}else if(qName == "rule"){

   //allocate memory for the rule
   rule = new GARule;

   if (attributes.length() > 0){
      for (int i = 0; i < attributes.length(); i++){
         if(attributes.qName(i) == "teamBall"){
            rule->teamBall( (attributes.value(i)).toInt());
         }else if(attributes.qName(i) == "myBall"){
            if (attributes.value(i) == "T"){
               rule->myBall(true);
            }else{
               rule->myBall(false);
            }

         }else if(attributes.qName(i) == "fire"){
            if (attributes.value(i) == "T"){
               rule->fire(true);
            }else{
               rule->fire(false);
            }

         }else if(attributes.qName(i) == "move"){
            if(attributes.value(i) == "T"){
               rule->move(true);
            }else{
               rule->move(false);
            }

         }else if(attributes.qName(i) == "turn"){
            if (attributes.value(i) == "left") {
               rule->turn(Left);
            } else if (attributes.value(i) == "right") {
               rule->turn(Right);
            } else {
               rule->turn(None);
            }

         }else{
            debug("error in rule tag");
            return(false); //error in rule tag
         }//end if
      }
   }else{
      debug("error in rule tag, missing all attributes");
      return(false);
   }//end if
```

```
   }else if(qName == "sensors"){

      //allocate memory for the sens array
      sens = new Thing[8];

      if (attributes.length() > 0){
         for (int i = 0; i < attributes.length(); i++){
            Thing t; unsigned int index;

            if (attributes.value(i) == "mybot") t = MyBot;
            else if (attributes.value(i) == "otherbot") t = OtherBot;
            else if (attributes.value(i) == "net") t = Net;
            else if (attributes.value(i) == "ball") t = TheBall;
            else if (attributes.value(i) == "wall") t = Wall;
            else t = Nothing;

            if (attributes.qName(i) == "s0") index = 0;
            else if (attributes.qName(i) == "s1") index = 1;
            else if (attributes.qName(i) == "s2") index = 2;
            else if (attributes.qName(i) == "s3") index = 3;
            else if (attributes.qName(i) == "s4") index = 4;
            else if (attributes.qName(i) == "s5") index = 5;
            else if (attributes.qName(i) == "s6") index = 6;
            else if (attributes.qName(i) == "s7") index = 7;
            else { debug("error in sensor tag"); return false; }

            sens[index] = t;
         }//end for

      }else{
         debug("error in sensor tag, missing all attributes");
         return (false);
      }
      //attach straight to the rule
      rule->sensors(sens);

   }//end if

   return true;
}//end startElement

//function execute whenever the parser sees a end xml tag
bool TeamParser::endElement(const QString&, const QString& qName, const QString&){
   if (qName == "bot"){
      botcount++;
      team->insertBot(bot);  // attach this bot to the team

   }else if (qName == "rule"){
      //attach this rule to bot
      bot->insertRule(rule);
      rulecount++;

   }
   return true;
}//end endElement

//return the team just loaded
Team* TeamParser::teamData(){
   return(team);
}

//end file
```

```
TEMPLATE = app
CONFIG = qt release
INCLUDEPATH = include
HEADERS = \
        include/bot.h \
        include/coordinate.h \
        include/direction.h \
        include/rotation.h \
        include/thing.h \
        include/garule.h \
        include/simplega.h \
        include/mainwindow.h \
        include/botview.h \
        include/team.h \
        include/gabot.h \
        include/arena.h \
        include/ball.h \
        include/teamparser.h \
        include/teamdata.h \
        include/game.h \
        include/random.h
SOURCES = \
        src/main.cpp \
        src/bot.cpp \
        src/garule.cpp \
        src/simplega.cpp \
        src/mainwindow.cpp \
        src/botview.cpp \
        src/team.cpp \
        src/gabot.cpp \
        src/arena.cpp \
        src/ball.cpp \
        src/teamparser.cpp \
        src/teamdata.cpp \
        src/game.cpp \
        src/random.cpp
TARGET = gabot
```