MiS20
the robotic soccer simulator

- Implementation document -

Names:	Hans Dollen, ih4a
	Wim Fikkert, ih4a
Date:	June, 10th 2003
Version:	1.3

Preface

This document is contains the implementation results from the MiS20 robotic soccer simulator project. It has been written for software developers who can use this document to reuse and expand on this simulator program. A few example expansions are an automated referee, more extensive kinematical and dynamical robot and ball models, another interface with the existing Mi20 system components. Java knowledge is recommend when reading this document. Also, we expect the reader to have knowledge of the MiS20 project. We will not explain every bit of the design which was created as well as the created requirements.

Table of Contents

Illustration Index

1. About

This document contains implementation results from the Mi20 simulator project (MiS20). First design changes will be explained. Next the actual implementation of the simulator design is stated, the details of which are explained. The performance issues we came accross are described next. Further, a short software developer manual will be stated which enables other software developers to reuse, expand on and change the MiS20 simulator program. Also future areas of the simulator which can still be optimized are stated. All classes are documented using javadoc[1].

1. The resulting Javadoc can be viewed on the MiS20 homepage, see reference [W6] and click “javadoc”.

2. Design changes

This chapter will describe what alterations have been made to the simulator design as stated in reference [B1]. The design divides the MiS20 simulator program into four seperate components[1]; model, view, control and communication. The exact implementation of these components is described in the following chapter. Sufficient to say, we made no very extensive changes to our design. Changes we did make however concern the match referee, general simulator settings, popup usage, are stated next.

2.1. The referee

In a match has a referee who calls situations like free kick, penalty kick and so on. In the MiS20 design this object was positioned in the model component where it could manipulate match data. However, since a referee untakes actions the better position for this object would be in the control component.

A referee also has a set of standard functions which can be called in stead of just a simple list of actions it called. This list of actions is described by a *Referee* interface which the *CtrlReferee*, of which the manual referee is an instance. As described in the requirements of the MiS20 project, we still only created a manual referee which is operated by a human user. The *Referee* interface enables future software developers to implement a new automated referee by implementing all functions stated in that interface.

2.2. Settings

In our design we had a number of classes which contained static data. For example, the *SimField* class which contains all data concerning the soccer field. Since that data is not likely to change dynamically we removed all get and set functions which were designed and we made all settings concerning the field *public static final* variables. As described in chapter 4, this also enhances performance for the MiS20 simulator program.

Also, we created such *public static final* variables in the *GuiButton*, *GuiMenu*, *GuiPopupGenerator* and *SimSettings* classes, the latter two requiring more explanation. The *GuiPopupGenerator* creates, as its name insinuates, popups at request. We will explain this class futher on in this chapter. The *SimSettings* class contains only settings of the simulator. This class is not included in the design we created. The reason why we created it though is because it holds variables which are used throughout the simulator program and which are not likely to change. We also moved the general system settings such as the *boolean*s which indicate collision detection to be on or off from the *SimAdministrator* class to the *SimSettings* class to have all such important data in one place for easy access.

2.3. Popups

We did not design how to use popups in the simulator. These are needed when asking the user to decide which team has to benefit from a certain referee call for example. Also, popups are needed to adjust settings like the host and port settings in the MiS20 program communication component.

We created one class, the *GuiPopupGenerator* class, which has various methods to create and display popups to the user. The actual implementation of this class can be found in paragraph 3.3.6.

2.4. Messaging

The message panel as seen in the design document[1] contains a list of messages which are displayed to the user. However, to record the messages of a specific match the *SimMatch* class will also have to record the messages displayed to the user in that specific match. Therefore, we added a list of messages to the *SimMatch* class. When a new message is added to that list, the GUI component which displays those messages is informed. See paragraph 3.3.1.4 for a more detailed explanation.

1. The MiS20 design can be found in reference [B1].

3. Simulator implementation

This chapter contains all specific implementation details. It describes the way the Mi20 simulator design has been implemented using Java, Java3D and JNI. A source code template was constructed by which all source code for MiS20 has been constructed[1]. Also, CVS was used to ensure version management.

This chapter will be structered as follows. First general decisions concerning implementation aspects are stated. Second, the model, view and control MiS20 components are described and explained. Finally, the interface component with the Mi20 system components is described in full extend.

All screenshots in this document have been created on a Sun blade workstation on the CDE, or Common Desktop Enviroment. The windows look and feel we would like to use cannot run on such machine so the default java look and feel is presented.

3.1. General

This paragraph will describe all decisions made for the MiS20 program in general. First, the program structure using the MVC model is described. Second, the usage of object sets is described.

3.1.1. Program stucture

The MVC[2] (Model View Control) model was used to divide the program in four different components structured in a star shaped model: Model, View, Control and Communication. Figure 3.1. displays this model.

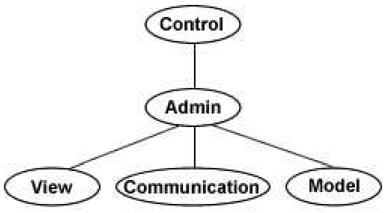


Figure 3.1. The MVC model as implemented in the MiS20 simulator

The components, as described in figure 3.1 are all replaceable by another component implementing the same functionality. The following paragraphs (3.2 through 3.5) will describe this functionality.

For each of these components specific prefixes will be used to name the source code files:

• Communication:	"Comm";
• View:	"Gui";
• Model:	"Sim";
• Control:	"Ctrl".

As seen above, the model is not a “pure” MVC model. The Model, View and Control components implement the MVC model which is administrated by the Admin component. The “extra” communication component is an interface with the already existing Mi20 control system components which determine strategy and which control the robots.

This Mi20 interface was added to communicate with the existing programming of the Mi20 team. This part can also be replaced by a different version, for instance when the Mi20 decides to use a different communication technique instead of the currently used C++ socket connections. Further, various design patterns where used to ensure reuse and expansion possibilities.

1. See appendix A for this source code template.
2. The MVC design pattern is descirbed extensively in reference [B2].

3.1.1.1. Strategy

A strategy design pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. The strategy pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure[3].

In the MiS20 program the strategy design pattern is used in the collision handling model, which will be called by the *SimAdministrator*. The difference with the strategy pattern is that the handling class has been determined before compiling on programming. This can be expand using a run-time choosen handler. The collision handling model was implemented using a default parent class which can be expanded on by any class.

For example, the collision handler includes a function which detects collisions between objects. Basically, this function uses a sphere versus sphere collision detection system which can be expanded or changed by a child class. Figure 3.2 displas this sphere versus sphere collision detection idea. This type of collision detection will result in a less precise collision detection between objects.

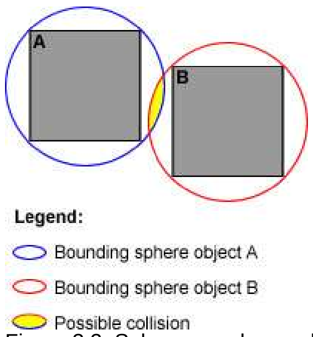
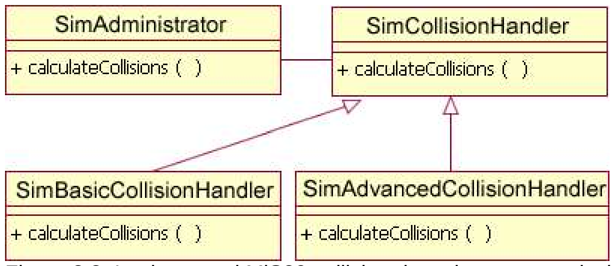


Figure 3.2. Sphere vs sphere collision detection

Figure 3.3 displays the actual implementation of the design pattern in the collision detection algorithm of the MiS20 program. As displayed in figure 3.3, the collision detection of the MiS20 program deferres the *calculateCollisions()* function to a child class. This function determines a more precise collision detection when the basic collision detection (using sphere versus sphere collision detection) indicates a potential collsion between objects.


SimAdministrator
+ calculateCollisions ( )
SimCollisionHandler
+ calculateCollisions ( )
SimBasicCollisionHandler
+ calculateCollisions ( )
SimAdvancedCollisionHandler
+ calculateCollisions ( )
Figure 3.3. Implemented MiS20 collision detection strategy design pattern

3. See reference [B2].

3.1.1.2. Prototyping

The prototype design pattern specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype[4].

The MiS20 program uses the prototype design pattern in its XML file loading class *SimXMLFile*. As is described in paragraph 4.4, the object creation times when using cloning are reduced with 67% compared to the use of constructors. We use a number of prototypes of the *SimVector*, *SimRobot*, *SimBall*, *SimTeam* and *SimMatch* classes for the XML file loading. Also, when updating object vectors we use a prototype of the *SimTimestamp* class. To have these prototypes generally available within the MiS20, we created a new class *SimObjectFactory* which contains these prototypes as public static objects. Another object can simply fetch these prototypes like stated in the example below:

```
SimTimestamp oPrototype = SimObjectFactory.oPrototypeSimTimestamp;
```

3.1.2. Arrays and ArrayLists

The data model, designed using NIAM and UML in the design document[5], has been constructed using mainly arrays and *ArrayList*s where sets of objects where needed. These *ArrayList*s can be found in the java.util package.

Other options are hashsets, hashmaps, *Vector*s and *LinkedList*s to name a few. The choice for using relatively simple and error prone arrays and *ArrayList*s is described extensively in paragraph 4.2. The summary of that story is that these collection types improve performance a great deal.

The *SimMatch* contains two *SimRobot* Arrays, each five places large. These *SimRobot* instances contain an *ArrayList* which grows in size during a soccer match. *SimSnapshot*s also contain a set of 2 Arrays, however, they are of a *SimVector* type since only vectors are needed and no *SimRobot* data is required when commucating with the Mi20 control system.

4. See reference [B2].
5. See reference [B1].

3.2. Model

This paragraph will describe the implementation process of the MiS20 model component. Figure 3.7 displays a class diagram of this program component. The object relations are described here whereas the OO[6] related get and set functions are left out.

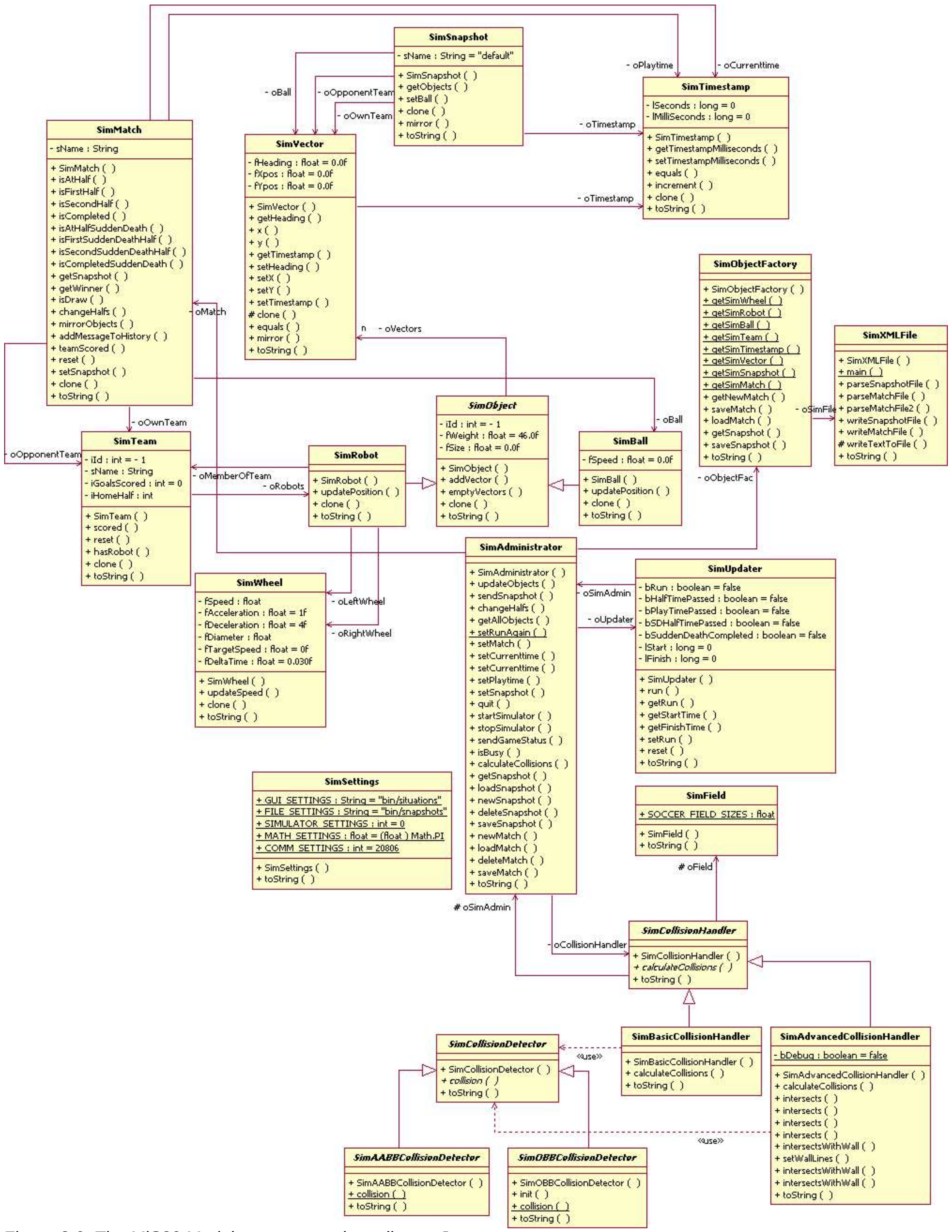


Figure 3.6. The MiS20 Model component class diagram[7]

6. OO, Object Oriented, see glossary.
7. The entire class diagram can be found in reference [B1].

The *SimMatch* class is in essence the base of the MiS20 data model. It contains the two robot soccer teams (*SimTeam* classes) and the match ball (*SimBall* class). The *SimTeam* classes, in turn, contain five robots each, which is implemented by the *SimRobot* class. The *SimRobot* and *SimBall* classes have a common parent class, the *SimObject* class which contains the list of object vectors[8] (*SimVector* class instances).

The *SimMatch* and *SimObject* classes contains a list of property change listeners which are notified when a certain properties are altered. Paragraph 3.2.2 describes exactly how we used property change listeners. The remainder of this chapter will describe how files are used to store the model information.

3.2.1. Files

The MiS20 requirements[9] state that robot soccer matches are to be logged to file. Also, it most be possible to store snapshots[10] to file. These two file types are very different from eachother.

A match contains the entire match (which lasts at least 10 minutes, or more depending on the need for a sudden death scheme when a draw occurs after the default 10 minutes of match time[11]). A match consists of a name, date, referee name, two teams of robots and a ball. Of the latter two, position information (mathematical vectors) of the entire match is logged. All this information (and the vector information) are written to file when a match ends or when a user desires this action. This list of positions can be as large as 18.000 vectors since there are 30 positions for each match second per object[12].

The other group, snapshots, consist of only those vector data for all objects. Snapshots are used to load default positions which are called by the match referee. For instance, a referee calls a free ball situation. A snapshot, consistant with the FIRA regulations is loaded which positions the objects according to the freeball snapshot.

3.2.1.1. XML

To read and write matches and snapshots from and to file a XML file layout will be used. The reason why this option was chosen in stead of the usage of Java Property files is because of the standarization of XML and the possible reuse of the XML files by the remaining MiS20 simulator and Mi20 control system components[13].

There are many different XML parsers available on the internet. However, the UT also has a XML parser of its own which has been used in the MiS20 simulator program. This parser is included by the *parlevink.xml* package which is not publically available.

8. A mathematical vector which contains x and y positions with a heading and timestamp.
9. See reference [B5] for the requirements which apply to the MiS20 project.
10. Snapshots contain mathematical vector information on all objects in the soccer field.
11. All FIRA regulations can be found in reference [B3].
12. 30 snapshots per second results in 30*60*10 = 18.000 snapshots for a basic, 10 minutes long match.
13. A more extensive explanation can be found in reference [B1].

Generally the XML parser will read a tag (and any variables it might contain), process this tag and continu to the next tag. The source code sample below states the general XML file layout which makes up a snapshot file.

```
<?xml version="1.0"?>
<root>
       <snapshot>
              <obj id="0" x="1.8000001" y="0.89920026" h="0.3" />
              <obj id="1" x="0.06500015" y="0.9" h="0.0" />
              <obj id="2" x="0.5799999" y="0.9000001" h="0.010000013" />
              <obj id="3" x="1.5399998" y="0.31500003" h="0.020041827" />
              <obj id="4" x="1.5349998" y="1.425" h="4.1007996E-5" />
              <obj id="5" x="1.7000002" y="0.8949999" h="0.010149889" />
              <obj id="6" x="2.1200001" y="0.90000015" h="0.023782123" />
              <obj id="7" x="2.007" y="0.72729987" h="8.2844496E-4" />
              <obj id="8" x="2.0060005" y="1.0799999" h="0.0030951798" />
              <obj id="9" x="2.0049999" y="0.45" h="-0.0019364022" />
              <obj id="10" x="2.0049999" y="1.325" h="-9.1177225E-4" />
       </snapshot>
</root>
```

Code sample 3.1. XML snapshot file layout

The parlevink XML parser reads a tag and returns its contents (parameters) directly. DOM parsers for example read the entire file before returning any data. Also, they will generate extensive hierarchy trees which provide too much overhead for fast usage in the MiS20 program. The parlevink XML parser will create in effect a stream of data which can be processes by our MiS20 simulator. This also results in better performance.

The match file XML layout is not so very different from the snapshot file as portraited in code sample 3.1. Code sample 3.2 displays a short summary of such a match file. This example match file has been stripped from its snapshot list (only a few snapshots remain). A match file which contains an entire match can hold up to 18.000 snapshots; each containing vectors for 11 objects.

```
<?xml version="1.0"?>
<root>
       <general
              messages_size="9">
              <message value="[01:30] Simulator stopped"/>
              <message value="[01:29] Simulator started/resumed"/>

              ... etcetera ...

              <message value="[00:00] Simulator started/resumed"/>
              <message value="[00:00] program startup complete"/>
       </general>
       <snapshot s="1054132548" m="309852" n="2186">
              <obj id="0" x="0.309355" y="1.20442" h="2.87064" />
              <obj id="1" x="0.0807806" y="0.9941" h="1.15312" />
              <obj id="6" x="0.251935" y="0.120785" h="0.114726" />
              <obj id="2" x="0.365558" y="0.593438" h="3.91293" />
              <obj id="7" x="1.2132" y="1.00693" h="0.232325" />
              <obj id="3" x="0.770626" y="1.14493" h="0.555542" />
              <obj id="8" x="1.1004" y="0.677319" h="2.64171" />
              <obj id="4" x="1.15403" y="1.17788" h="0.164021" />
              <obj id="9" x="0.869656" y="0.913085" h="2.73063" />
              <obj id="5" x="0.993895" y="0.954677" h="1.53673" />
              <obj id="10" x="2.11342" y="0.915818" h="5.90845" />
       </snapshot>
```

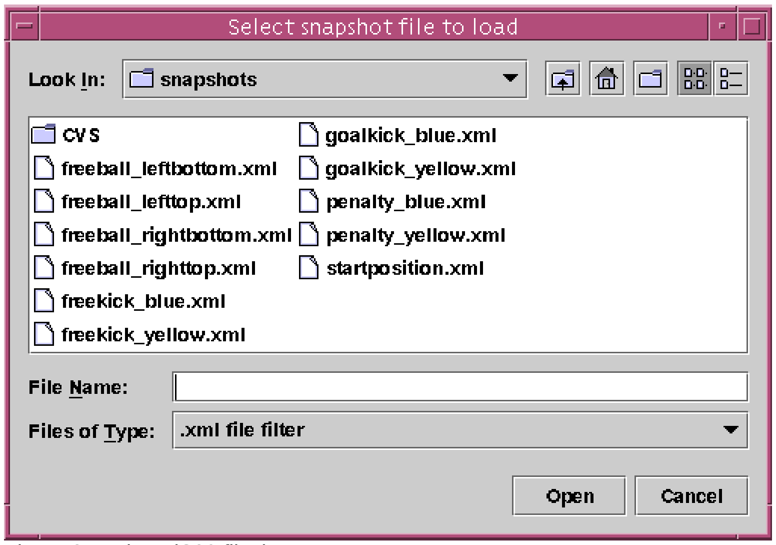
```
        <snapshot s="1054132548" m="343535" n="2187">
                <obj id="0" x="0.301543" y="1.20574" h="2.88677" />
                <obj id="1" x="0.0813798" y="0.994282" h="1.36153" />
                <obj id="6" x="0.252596" y="0.113699" h="0.114726" />
                <obj id="2" x="0.358231" y="0.585673" h="3.93365" />
                <obj id="7" x="1.2145" y="1.00506" h="5.6611" />
                <obj id="3" x="0.771049" y="1.14402" h="0.591658" />
                <obj id="8" x="1.08935" y="0.68298" h="2.65005" />
                <obj id="4" x="1.13823" y="1.177" h="0.18319" />
                <obj id="9" x="0.869499" y="0.91614" h="2.53966" />
                <obj id="5" x="0.994387" y="0.95155" h="1.54145" />
                <obj id="10" x="2.11342" y="0.915599" h="5.90845" />
        </snapshot>

        ....

        <snapshot s="1054132549" m="843088" n="2232">
                <obj id="0" x="0.0214581" y="1.26366" h="2.93535" />
                <obj id="1" x="0.0512417" y="1.15503" h="1.94638" />
                <obj id="6" x="0.178397" y="0.066465" h="2.98457" />
                <obj id="2" x="0.221241" y="0.260055" h="4.39835" />
                <obj id="7" x="1.51809" y="0.167628" h="5.05376" />
                <obj id="3" x="0.611839" y="0.947873" h="2.82644" />
                <obj id="8" x="0.838801" y="0.888784" h="1.96064" />
                <obj id="4" x="1.108" y="1.21699" h="2.51922" />
                <obj id="9" x="0.992884" y="1.20566" h="0.38622" />
                <obj id="5" x="0.953664" y="0.894558" h="2.21929" />
                <obj id="10" x="2.11344" y="0.915771" h="5.90845" />
        </snapshot>
</root>
```
Code sample 3.2. XML match file layout

3.2.1.2. Filechoosers

In order to provide the user with a user friendly way to select files from disk a file chooser has been implemented. Java includes a default filechooser which can be included from the *javax.swing package*. This class (*JFileChooser*) can be set to a specific directory using a specific file filter (for instance, all files having the extension ".xml"). Since the default file used in the MiS20 simulator is of an XML markup a filefilter has been created which only displays files having this extension. Figure 3.7 displays the resulting filechooser the MiS20 simulator uses.



Look In: snapshots

CVS
freeball_leftbottom.xml
freeball_lefttop.xml
freeball_rightbottom.xml
freeball_righttop.xml
freekick_blue.xml
freekick_yellow.xml
goalkick_blue.xml
goalkick_yellow.xml
penalty_blue.xml
penalty_yellow.xml
startposition.xml

File Name:

Files of Type: .xml file filter

Figure 3.7. The MiS20 filechooser

3.2.2. Property change events

For updating the model information in the GUI property change events, included by the *java.beans* package where used. This technique includes a property change listener and a property change fire event. The first listens to property change events. When such an event is caught, it is checked if it is the proper type and, if so, the proper action is taken. The latter fires an property change event when its data is changed.

For example, when an object's x and y position is changed, a property change event is fired by that object (an instance of the *SimObject* class). The GUI components which display that object data then update that data (*GuiShowObjectData* and *Gui3DObject* class instances). When no event is fired, the relative GUI components are not updated.

3.3. Collisions

This paragraph describes how collision detection and handling is implemented in the MiS20.

3.3.1. Detection

Collisions can occur between multiple objects and between objects and one or more walls. These collisions are detected using two levels of collision detection as described in the MiS20 design document. The first level of detection uses AABB'es for each object. A easy check can be performed to test if another object is colliding with a test object. If the first collision detection passes, a second, more precise, is performed. If this test also fails a collision might be possible. The second level of collision detection is then performed.

The first level of detection uses equation 2 as stated in the design document, which is also stated below. The following code sample illustrates the implementation of this equation. The SimSettings class has a number of static class variables which are calculated only once, on program startup. This also improves the performance of this algorithm.

$$\begin{aligned} &x_a - x_b > AABB\,side_{length} \\ &y_a - y_b > AABB\,side_{length} \\ &where\ AABB\,side = \sqrt{2} * CUBE\ side \end{aligned} \quad \text{(Eq. 2)}$$

The implemented version of the equation above (number two) is:

```
float fDeltaX = Math.abs(oA.getVector().x() - oB.getVector().x());
if(fDeltaX  <  SimSettings.fROBOT_COLLISION_SPHERE_RADIUS +
               SimSettings.fBALL_COLLISION_SPHERE_RADIUS    )
{
       float fDeltaY = Math.abs(oA.getVector().y() - oB.getVector().y());
       if(fDeltaY  <  SimSettings.fROBOT_COLLISION_SPHERE_RADIUS +
                      SimSettings.fBALL_COLLISION_SPHERE_RADIUS)
       {
              /* ... a collision might be possible! Go to level 2 detection ... */
       }
}
```

The second level of collision detection uses lines to detect collisions. These lines can be drawn on a *TestFrame* by adding them to the frame that has been made for testing. The robot consists of four lines, one per side (left, front, right and back). An collision between two robots occurs when one or more lines of the first robot intersects one or more lines of the second robot. To detect collisions between objects within the time interval of 30 ms, a line will be drawn from the robot to a point which is the point on which the robot will be on the next update.

Ball collisions will be detected by checking if the distance from the center of the ball to the wall (which is in fact a line) or robots' line is smaller than the ball's radius. Also for the ball, detecting of collisions between updates will be *done by drawing a line to the next position, and check if the ball collides with an object.*

3.3.2. Collision handling

Collision handling has been implemented using the lines described in the paragraph above. Even the equations described in previous chapter have been used and thus implemented. The handling handles collisions for robot versus robot, robot versus wall, ball versus robot and wall versus ball. The latter two will be handled rather precisely.

3.4. Kinematic models

This paragraph described the way of simulation of the robots and the ball in MiS20.

3.4.1. The soccer ball

The ball will be simulated as described in the previous chapter. The formulas described in the previous chapter have been implemented. A deceleration factor has been added to the SimBall with which the ball will decelerate every update.

3.4.2. The robots

The robots will be simulated as described in the previous chapter. The SimRobot contains a maximum deceleration and acceleration. The robots always accelerate or decelerate at max to its target speed. The acceleration and deceleration variables have been retrieved from real world data of matches played by the Mi20 team. The acceleration factor is about 1,0 m/s^2 and the decelleration factor is about 3,0 m/s^2, which includes friction with air and floor. Other formulas which have been used have been described in the previous chapter.

3.3. View

The GUI (Graphical User Interface) was created using Java en Java3D. In general, the GUI consists of five panels which contain information for the user. Figure 3.8. displays those five different panels and explains what information each panel contains.

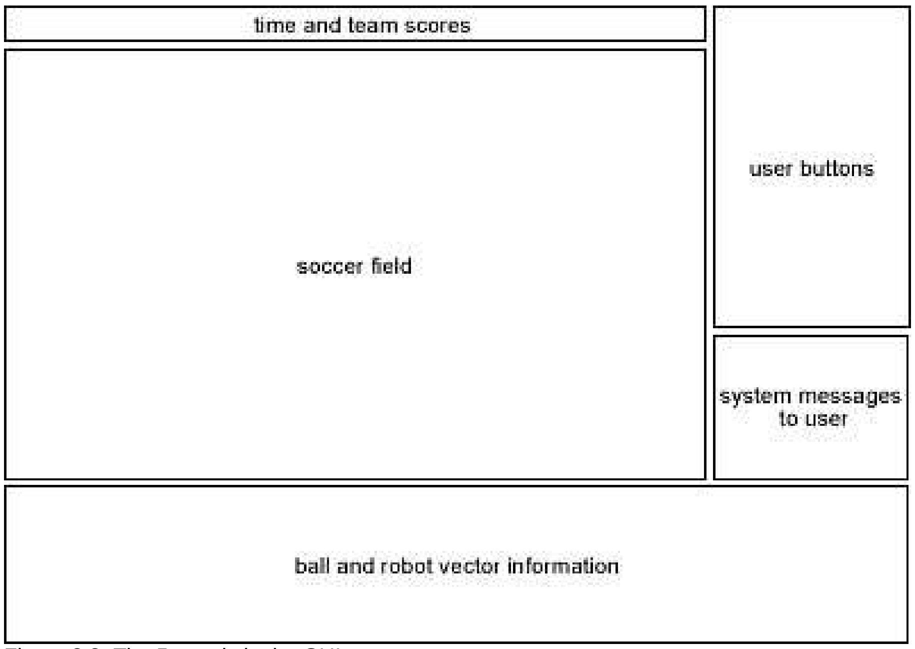
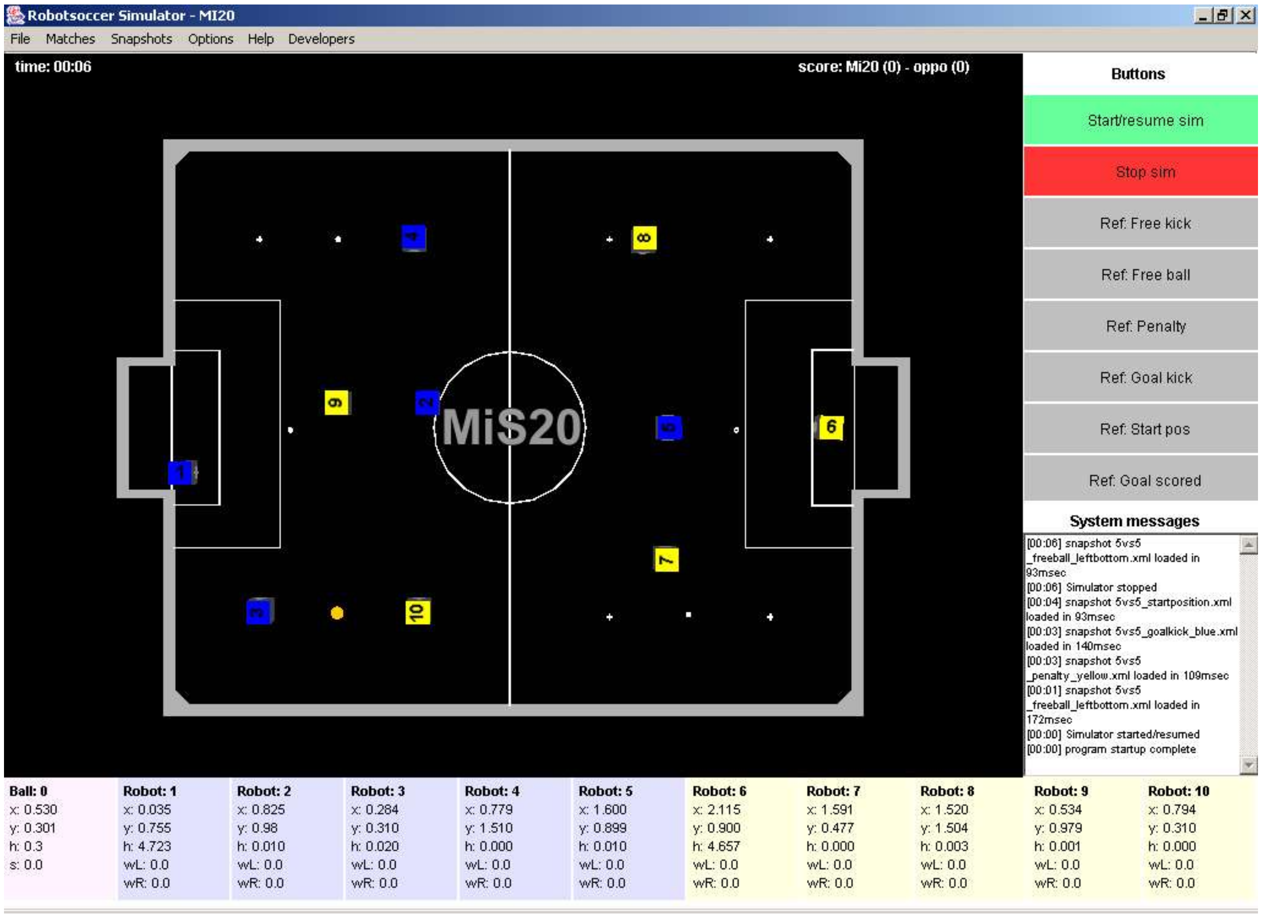


Figure 3.8. The 5 panels in the GUI

Figure 3.9 displays the resulting GUI in a screenshot of the MiS20 program. As you can see, all panels are nicely implemented. On resizing, the button has a set width and height whereas the system messages panel has only a fixed width. The ball and vector information panel, implemented by the *GuiObjectData* class has a fixed width. The same applies for the time and score panel. On a resize, the soccer field will expand whereas the remaining panels will not resize beyond proportions. The following paragraph will explain how the resizing works. The UML diagram of the classes we have created can be found in appendix D: UML diagrams.



```
[00:06] snapshot 5vs5
_freeball_leftbottom.xml loaded in
93msec
[00:06] Simulator stopped
[00:04] snapshot 5vs5_startposition.xml
loaded in 93msec
[00:03] snapshot 5vs5_goalkick_blue.xml
loaded in 140msec
[00:03] snapshot 5vs5
_penalty_yellow.xml loaded in 109msec
[00:01] snapshot 5vs5
_freeball_leftbottom.xml loaded in
172msec
[00:00] Simulator started/resumed
[00:00] program startup complete
```

Ball: 0	**Robot: 1**	**Robot: 2**	**Robot: 3**	**Robot: 4**	**Robot: 5**	**Robot: 6**	**Robot: 7**	**Robot: 8**	**Robot: 9**	**Robot: 10**
x: 0.530	x: 0.035	x: 0.825	x: 0.284	x: 0.779	x: 1.600	x: 2.115	x: 1.591	x: 1.520	x: 0.534	x: 0.794
y: 0.301	y: 0.755	y: 0.98	y: 0.310	y: 1.510	y: 0.899	y: 0.900	y: 0.477	y: 1.504	y: 0.979	y: 0.310
h: 0.3	h: 4.723	h: 0.010	h: 0.020	h: 0.000	h: 0.010	h: 4.657	h: 0.000	h: 0.003	h: 0.001	h: 0.000
s: 0.0	wL: 0.0	wL: 0.0	wL: 0.0	wL: 0.0	wL: 0.0	wL: 0.0	wL: 0.0	wL: 0.0	wL: 0.0	wL: 0.0
	wR: 0.0	wR: 0.0	wR: 0.0	wR: 0.0	wR: 0.0	wR: 0.0	wR: 0.0	wR: 0.0	wR: 0.0	wR: 0.0

Figure 3.9. A screenshot of the finished MiS20 GUI

3.3.1. Panels

Figure 3.8 states the six panels which are included in the GUI. In this paragraph, all panels will be explained in some detail and functionality. Resizing of these panels is not done by using a layout manager such as a borderlayout (included from the java.awt package) but is manually coded in the repaint function of the GUI frame. This is required because the Java3D field panel is not fully compatible with the java.awt and javax.swing packages.

3.3.1.1. Time/score

The time and score panel displays only two things. Left, the match time is displayed. When the simulator is playing this time is being updated (the clock is viewed as ticking). The score is displayed on the right. This data is updated when a goal is scored, which requires the match to be busy. Using the *java.awt* and *javax.swing* packages this panel is visualized.

An optional implementation is to append another variable to the user: timestamp numbers. One of the propable areas of use for the MiS20 simulator program is, of course, to be used in the Mi20 robot soccer project. When the Mi20 team logs a match to file, they will be able to display that match in the MiS20 simulator. The Mi20 program also works with timestamp numbers in stead of just timestamps. These numbers are used throughout the Mi20 match logs so different logs can matched to each other. When displaying these timestamp numbers it will be possible for the Mi20 team to look up the exact logged data for that timestamp.

3.3.1.2. Field

The soccer field panel displays the soccer field in 3D. How this 3D view is generated is explained in paragraph 3.3.2. This panel will display the entire soccer field at default. The correct sizes of the objects and the field are used, the lines are positioned correctly etc. See figure 3.9 for more information on what is displayed. This panel also includes methods to set the camera viewing the field to a different position and to update the 3d objects.

The sizes of the soccer field can be found in the *SimField* class as described in paragraph 2.2. When these values are changed, the soccer field will also be changed accordingly.

3.3.1.3. Buttons

This panel houses the buttons which represent a human referee. Using the FIRA regulations[14] a user referee can call situations which are required. The functionality incorporated by this panel includes:

- Starting and stopping the simulator;
- Free kick situation;
- Free ball situation;
- Penalty kick situation;
- Goal kick situation;
- Start positions.

For more information on the working of these buttons, see the user manual as described in the bachelor thesis[15]. This panel consists of eight buttons which fire action events (included from the *java.awt.event* package). When such a button is pressed, the action event is fired after which the action listener (see paragraph 3.4) will take the appropriate action. The visual representation of the buttons is generated using the *javax.swing* package, which includes borders etcetera.

3.3.1.4. Messages

This panel displays system messages to the user for extra information. For example, it will display messages such as “[xx:xx] snapshot freekick_blue loaded” which ensures the user that the freekick positions for the blue team have been loaded succesfully. The “[xx:xx]” String indicates the current timestamp of the match.

The message panel will display the last 10 messages which where generated by the system. For that, it maintains a linkedlist of messages. Each time a new message is displayed, the new message is added to the end of the list. If the list already consists of 10 messages, it will remove the first message in order to make room for the new one. This list is *not* equal to the list which can be found in the *SimMatch* class. That message list only records the messages displayed to the user when the match was in progress. When a message is displayed to the user it is added to the *SimMatch* messages list. The SimMatch has a list of propertychange listeners. The messages panel is one of those listeners. When a propertychange event is fired and caught the new message is also added to and displayed by this GUI panel.

When a new match is loaded, the list of property change listeners is moved to the new *SimMatch* class instance.

14. See reference [B2].
15. See reference [B5].

3.3.1.5. Object data

The object data (position vectors, teamcolor, id etc.) are represented in the object data panel. This panel will display all mathematical vector information on an object; x and y positions and the heading. Also, the speed of the object is displayed according to the type of object (a robot has two wheel speeds whereas a ball has only one speed).

This panel is divided into a number of subpanels which all represent one object in the match (a ball or a robot). Using the *java.awt* and *javax.swing* packages, the information of that object is displayed to the user. The team color is displayed as a background of each object. The ball has no team which it belongs to of course, so the background color of the ball is orange.

We did not use very contrast rich colors since it would result in a very “busy” GUI which is to be avoided as is described in the design document[16].

3.3.1.6. Resizing

The panels are not laid out using a default *java.awt.LayoutManager* implementing class instance such as *BorderLayout* or *FlowLayout*. The reason for this is that Java3D does not handle very well with *java.swing* classes when resizing a set of panels. This is caused by the fact that Java3D is heavy weight whilst *java.swing* and *java.awt* classes are not. The solution we implemented creates and resizes the panels using set widths and heights for some panels whereas the remaining panels can grow to fill up the window. These set widths and heights are set in the *SimSettings* class.

3.3.2. Java3D

The MiS20 simulator uses Java3D in combination with the *java.awt* and *javax.swing* packages to display a 3D representation of the soccer field to the user. Java3D creates a so called scene graph which contains all objects in the Java3D 3D scene. This scene graph consists of one (or more) object branch group(s), containing all objects in the scene, and of one (or more) view branch group(s), containing all view related components.

A simplified representation of the Scene Graph used in the MiS20 simulator is displayed in figure 3.10.[17] Because of the relative simplicity of the 3D world used in this simulator the Scene Graph will also be not very complex. Figure 3.10 displays the designed scene graph for the MiS20 simulator program. However, this design has not changed on implementation of it in Java3D.

16. See reference [B1].
17. See reference [B1] for the complete Java3D scene graph.

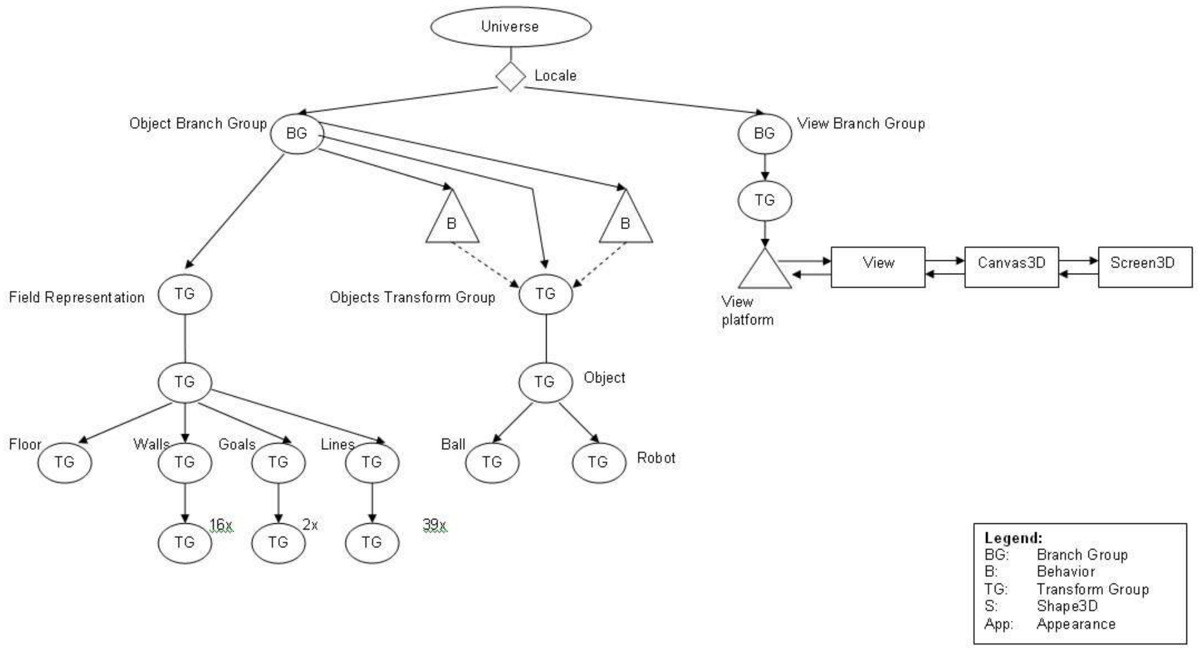


Figure 3.10. The simplified Java3D scene graph in the MiS20 simulator

A few inheretance principles have been used to create this 3D scene. First of all, the soccer field itself contains wall and lines and so forth. These can all be viewed as field components which need to be scaled, translated and rotated. This results in a single parent class which does those transformations whilst the child classes clarify the 3D object represention.

Also, the *Gui3DObject* class is a parent class for the *Gui3DRobot* and *Gui3DBall* classes which represent the robots and the ball respectively. This setting can be compared with the MiS20 model component in which the *SimObject* is a parent class for the *SimRobot* and *SimBall* classes.

3.3.2.1. Axis

Java3D uses the x and y axis by default as the bottom plane on which the scene is to be displayed. We will use the x and z axis for our bottom plane since that is most commoningly used in the 3D graphics industry. We created a transform function which transforms the default co-ordinates to the co-ordinates set we use.

3.3.2.2.Camera positioning

The camera which displays the 3D scene is positioned using the known lookat principle in 3D graphics. OpenGL also implements such a function. Lookat works by telling a camera where to place itself, where to look at, and which way is considered to be the up direction. We implemented a *GuiCamera* class which calculates a new camera position using the *lookat(Point3d, Point3d, Vector3d)* method of the *Transform3D* class in the *javax.media.j3d* package[18].

When given a scene graph object branchgroup to look at, the *GuiCamera* class positions the camera. This class also lends itself for dynamic positioning of the camera which is an optional requirement of the MiS20 simulator project.

For example, a pursuit cameraview of an object can be implemented by altering the camera each time that object moves across the soccer field. The lookat point will simply be the position of the object to follow and the point from which to look is to be set above or behind the object.

Finally, it will also be possible to use user input such as mouse events to reposition the camera across the soccer field. All these camera modi are changed in the GUI menubar, described in paragraph 3.3.4.

18. See reference [W5] for the Java3D API javadoc.

3.3.3. Updating

The GUI is updated by propertychange events alone. The *updateObjects()* method of the *SimAdministrator* class updates the object vectors after which propertychange events are fired. These events are then caught by propertychange events across the GUI classes as described in paragraph 3.2.2.

Java3D includes a default update *Thread* which updates its components when implemented the standard way. However, this *Thread* boast too much overhead for usage in our simulator. Therefore we do not make use of this Java3D feature for updating the 3D components in the GUI.

3.3.4. Menu

The simulator also includes a dropdown menubar. This menubar is also created using a the *java.awt* and *javax.swing* packages. It uses a function which creates a menu of objects. These objects can vary between *null*, *JMenuItem*, *JMenuBar* and *JCheckBoxMenuItem*, the latter three from the *javax.swing* package.

When an object is *null*, a dividing line is displayed, otherwise, the object has a representation defined in the swing package. Figure 3.11 displays a screenshot of the MiS20 menubar.

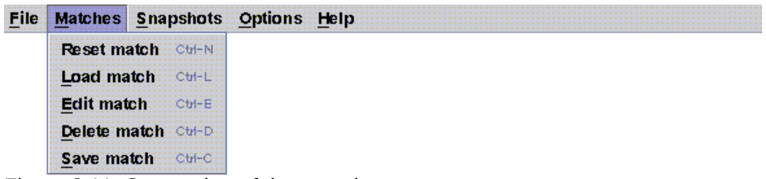


Figure 3.11. Screenshot of the menubar

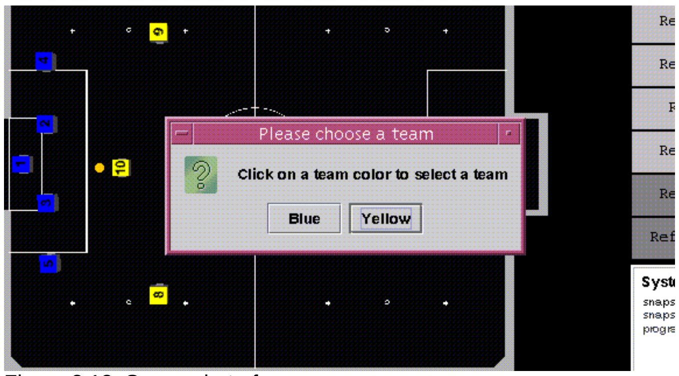
When the user selects one of the menu items, either via keyboard (via shortcut keys) or mouse, an *ActionEvent* is fired. This event is caught by the MiS20 simulator control component and the appropriate action is then undertaken. See paragraph 3.4 for more information on the implemented *ActionListener*.

3.3.5. Splash screen

To add to a professional look and feel of the simulator program, a splash screen was created whilest the program is starting up. This is done by displaying the MiS20 logo in a *Window* from the *java.awt* package. It is shown when the GUI is being loaded and hidden when the simulator has completed loading.

3.3.6. Popups

As described in chapter 2, the MiS20 simulator uses various different popup windows to provide the user with warning messages, ask questions and display help information. Since there are many different popups which need to be displayed, a popupgenerator has been created which creates these popups on command. It uses *JOptionPane* class from the *javax.swing* package to create these popups. Various variables have to be added, such as the message to be displayed, which icon is to be displayed and what user actions may be taken (“ok” and “cancel” or just “ok”?). Figure 3.12 displays an example of a screenshot in the MiS20 simulator.



Click on a team color to select a team

Figure 3.12. Screenshot of a popup

3.4. Control

Our simulator uses the MVC model. This chapter will explain how the control component was implemented. This component consists of two interaction ways. The user has the ability to use the keyboard and mouse.

3.4.1. Keyboard

The keyboard interaction has been implemented using the action listener interface which is included from the *java.awt.event* package. This listener catches action events and, depending on the event origin, takes an appropriate action. For example, when a user presses a shortcut key (Ctrl+Q) an *ActionEvent* is fired in the menu. The event is caught by the *ActionListener* implementation which determines that the user has selected the “quit” menut item. The simulator will then shut down.

The *ActionListener* also listens to *ActionEvent*s fired from the button panel. The name of the button indicates the action to be taken. The user is asked which team should benifit from this referee call when needed and the appropriate snapshot is loaded from file.

The button names are set in the *GuiButton* class. These static variables are used in the *ActionListener* class to determine which button has been pressed. The same approach is used for the *GuiMenu* class and the menu items a user can select from the MiS20 menubar.

3.4.2. Mouse

The mouse interaction is not implemented as a seperate control component using *MouseEvent*s but is, in stead, included in the View component. The reason for this is that Java3D includes default mouse behaviors which will work very inefficient when separated from the 3D scene they are part of. The mouse interaction is needed to select, drag and rotate objects in the 3D field representation.

However, a problem arises when an object in the 3D world is to be selected by selecting it using the mouse pointer, which resides in a 2D plane. To transform a 2D screen position of the mouse pointer to a 3D ray Java3D implements a *PickTool* class which enables such a ray calculation. Figure 3.13 displays the user (the person with the funny hat) which uses the mouse pointer to select an object in the 3D world. There are three objects which can be selected, a plane, boat or truck. Since they are all situated behind eachother on the users monitor the ray originating from the user crosses all three objects. The *PickTool* class in Java3D calculates which object is in crossed first. This object is the one the user has selected.

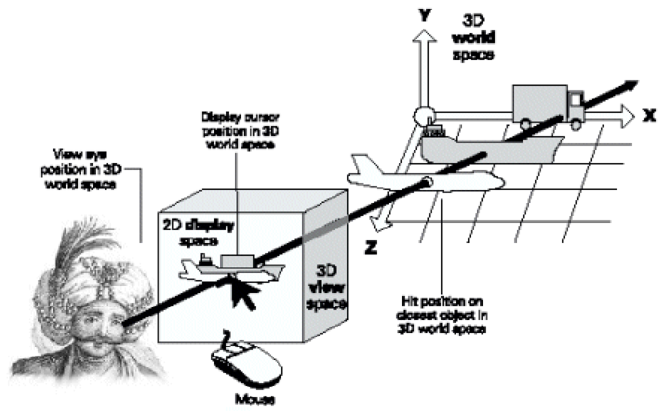

Y
3D world space
X
Display cursor position in 3D world space
View eye position in 3D world space
2D display space
3D view space
Z
Hit position on closest object in 3D world space
Mouse

Figure 3.13. Translation between 2D mouse pointer and 3D scene co-ordinates, source [B4]

The picking tool (*PickTool*) is laocated in the *com.sun.j3d* package. Two different picking behaviors have been implemented, a behavior which translates an object along the x and z axis and a rotation behavior which rotates an behavior along the x and z axis (default is all three axis). The robots and ball can be repositioned and rotated respectively on the soccer field. These picking behaviors are switched on only when the simulator is stopped. When a match is busy no objects can be repositioned.

3.4.2.1. Transforming

PickRotateBehavior and *PickTranslateBehavior* are two behaviors which have been used in order to translate and rotate Java3D objects. If the pick reporting of an Java3D object has been enabled, it can be translated and rotated.

The behaviors reacts on mouse events, and calls the java3D object method *setTransform(Transform3D)* to rotate or translate the object. The contra of the *PickRotateBehavior* is that it rotates the object in all directions, while the robots and ball may only rotate along its y-axis. The contra of the *PickTranslateBehavior* is that it will move objects in an x- and y-direction, while the robots and ball may only move in an x- and z-direction. For these reasons, the *setTransform* has been overwritten. This overwritten method converts the y-movement into a z-movement, and removes rotations around the x-axis and z-axis. The overwritten *setTransform* method calls the *getNewSimVector* method which will return the new *SimVector* according the transformation matrix. This function calculates the delta x and delta y to translate the object in respectively the x and z direction, and the new heading will be calculated according one rotation variable in the matrix.

The behaviors reacts on mouse events, and calls the java3D object method *setTransform(Transform3D)* to rotate or translate the object. The contra of the *PickRotateBehavior* is that it rotates the object in all directions, while the robots and ball may only rotate along its y-axis. The contra of the *PickTranslateBehavior* is that it will move objects in an x- and y-direction, while the robots and ball may only move in an x- and z-direction. For these reasons, the *setTransform* has been overwritten. This overwritten method converts the y-movement into a z-movement, and removes rotations around the x-axis and z-axis.

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & x \\ 0 & 1 & 0 & y \\ -\sin(\theta) & 0 & \cos(\theta) & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(Eq. 17)}$$

The code below illustrates how the information from the matrix will be used in order to create a new *SimVector*. The Transform3D contains the matrix with new positions, which are the old positions with the delta position or heading add. De y rotation will be retrieved from the matrix stated in equation 17 in our design document (also stated above).

```
public SimVector getNewSimVector(Transform3D oTransform3D)
{
        Transform3D oldTransform3D = new Transform3D();
        SimObject oSimObject = getSimObject();
        if(oSimObject !=null)
        {
                float fHalfSize = SimSettings.fHALF_CUBE_SIZE;
                if(oSimObject instanceof SimBall)
                        fHalfSize = SimSettings.fBALL_RADIUS;
                SimVector oSimVector = oSimObject.getVector();
                float[] matrix = new float[16];
                oTransform3D.get(matrix);

                float newHeading = ((float)Math.asin(-matrix[8]);
                        if(newHeading<0)
                                newHeading+=(2*SimSettings.fPI);

                // z movement is the y-movement the object will do
                // which is the new y position - old y position
                float fDeltaZ = matrix[7]-fHalfSize;
                SimVector oNewSimVector = (SimVector)oSimVector.clone();
                oNewSimVector.setHeading(newHeading);
                oNewSimVector.setX(oSimVector.x()+fDeltaX);
                oNewSimVector.setY(oSimVector.y()+fDeltaZ);
                return oNewSimVector;
        }
        return null;
}
```

3.4.3. Referee

As mentioned in paragraph 2.1, we implemented a manual referee in the *CtrlReferee* class. This class implements the *Referee* interface which states all referee functionality. A referee will call match situations like free ball, goal kick and penalty kick for a specific team. The user decides which action by pressing a button in the button panel (which we have seen in paragraph 3.3).

When such a button is pressed, the *ActionEvent* following that button press will be handled by the *CrtlActionListener* class. This action listener class will determine which function to call in the *CtrlReferee* class.

We implemented most of the referee tasks to go manually; for instance, if a goal kick is decided, the user is asked which team should get that goal kick. However, we did implement some automated features. For instance, when the user decides a goal has been scored, the *CtrlReferee* class instance will determine which team has scored automatically using the current ball location. Free ball situations are also determined automatically. When the ball is in the upper left corner, a free ball will also be in that region.

3.5. Communication

This chapter will describe the communication component of the MiS20 simulator program. The communication component will handle all communication with the existing Mi20 system programmed in C. We will first describe the C socket connection interface the Mi20 system has and which we must also comply to. We will then describe how Java can communicate with those C socket connections. Finally we will discuss how we implemented this solution.

3.5.1. Mi20 interface

The Mi20 control system has been implemented entirely in C. It is constructed to be a distributed system which can be run on various machines without having to adjust much in the source code. For this reason, an elaborate set of socket connections has been created which enable the various components of the Mi20 system to communicate with each other.

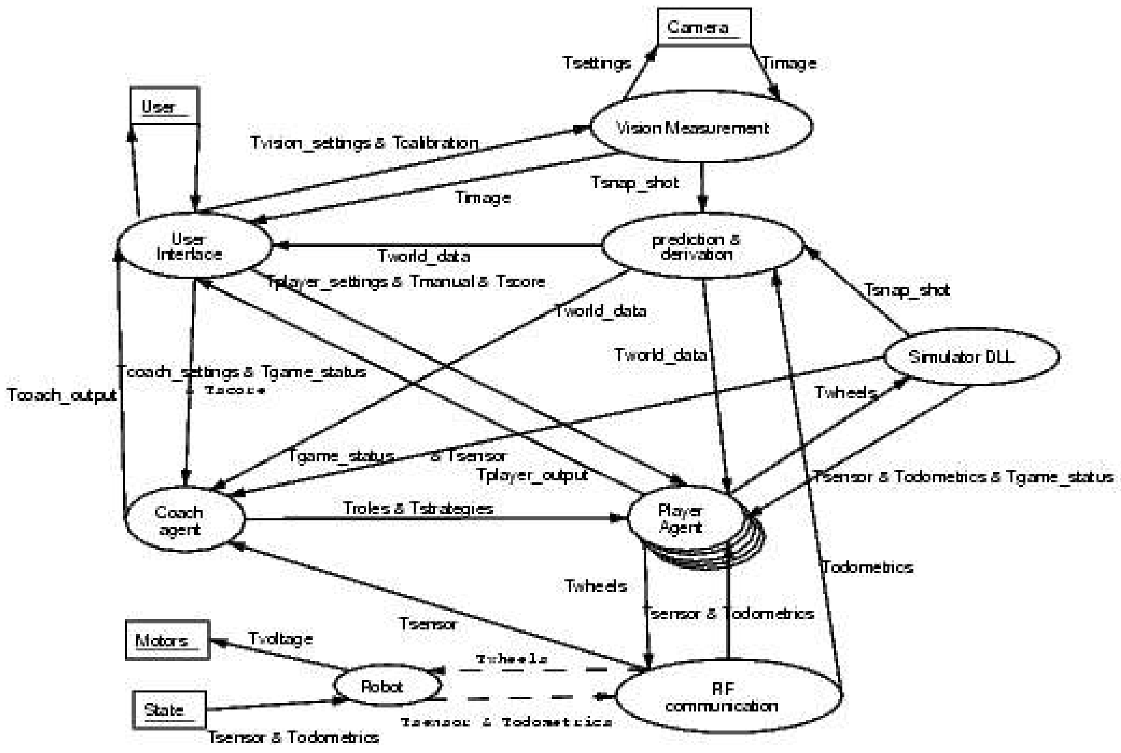


Figure 3.14. Mi20 robot soccer component design

Figure 3.14 displays component design which has been designed by the Mi20 team. Each arrow represents a C socket connection. An extensive description of this design can be found in our design document[19]. Here we only have to consider the Simulator DLL oval in the right of figure 3.14. It receives the C structures *Twheels*, *Tsensor*, *Todometrics*, and *Tgame_status*. If sends the C structures *Tsnap_shot* and *Tgame_status* to the Mi20 control system.

Of the received C structures only the *Twheels* structure needs to be used. It contains wheel speeds for both wheels of a specific robot. These are to be translated into a position change for the robot. Once this is complete for all robots and the ball, a *Tsnap_shot* structure is created which contains all current robot and ball vectors[20].

Summarizing, we will need to be able to receive wheel speeds and send snapshot in order to communicate with the Mi20 control system. This will, however, only be a basic communication so we will also implement the other socket connections which are to be received and send (*Tgame_status* etc.).

3.5.2. JNI

Since we will need to implement the MiS20 simulator using Java we will have to find a way to enable Java to communicate with C programming. The best method to accomplish this is to use JNI[21]. JNI enables Java to interact with other program languages such as C++ and C programs by constructing a layer between them, the so called Native Interface. Figure 3.15 shows an example, letting Java communicate with C using JNI.

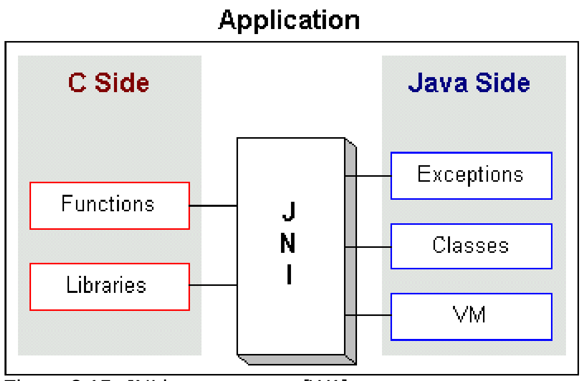


Figure 3.15. JNI layers, source [W1].

For the communication between Java and C++, JNI enables the C++ side to use the JVM[22]. From the Java side native methods can be called, which gives a direct connection with the C++ implementation of the specific function. The C++ side, see figure 3.15, only contains the implementation of the native functions.

The C++ files have to be compiled into a library, which will be dynamically loaded in Java. The Java class which contains the native methods has to load the library with use of a static *System.loadLibrary (libraryName).*

When a Java object then call a native method, the JVM will load the native method in the precompiled library. Then the JVM will call the native method (a pointer to a function) in the library, so the C++ method will be executed.

19. See reference [B1].
20. Snapshots contain mathematical vector information on all objects in the soccer field.
21. JNI, or Java Native Interface, see glossary.
22. Java Virtual Machine, see [B1]

When receiving data, the data will be converted to Java data. This data will be 'sent' to Java by calling a Java method from within C++. This can be done by use of the environment pointer, which is a pointer to the JVM, so the JVM will be used in both ways. This way will also be used to interact with the Mi20 Communication library. In the receive part of the communication of the simulator, a Java thread (1) starts the C++ thread. When the C++ thread receives new data, this data will be converted to a Java object (2), and will be passed to Java by calling a method of the Java thread (1).

4. Performance issues

This chapter will describe all performance related topics concerning the construction of the MiS20 program. First, the completed MiS20 program was profiled to indicate where bottlenecks where. These bottlenecks are to be targeted. Second, the choice for which collection method to use is explained.Third, the application and performance issues of *String*s in the MiS20 program are described. Fourth,

4.1. Profiling

Profiling is a technique which indicates where bottlenecks are in a program. Java has a build in method to profile a program. By using the *-Xrunhprof* parameter when running a program an elaborate profile of that program is created. These files can be distilled by various program which offer a graphical view which indicates the problem areas in the profiled program[1]. The program is run like this:

```
java -Xrunhprof:cpu=samples,depth=6,thread=y  RobotsoccerSimulator
```

The values given along with the -Xrunhprof paramter indicate how to profile the program. The result is a huge file (easily over a few megabytes in size) which contains all method calls down to the user indicated depth level (in this case, six). The table below displays where the bottlenecks in the MiS20 program where. Only the top 10 methods are displayed here. To visualize this a profile analyse tool, PrefAnal[2], was used. It does not generate a graph but offers a structured representation of the program's profile. The percentage column is the percentage of total CPU cycle usage, the calls are the number of calls which were resposible for that percentage:

percentage	**calls**	**function**
67.93%	2726	CommReceiveStub.startReceiverThreadOnSpecificPort
20.41%	819	CommSendStub.startSendThreadOnSpecificPort
5.83%	234	sun.awt.motif.MToolkit.run
0.35%	14	sun.awt.X11Renderer.doFillRect
0.30%	12	javax.media.j3d.Canvas3D.swapBuffers
0.25%	10	javax.media.j3d.Canvas3D.setModelViewMatrix
0.22%	9	javax.media.j3d.Canvas3D.callDisplayList
0.17%	7	javax.media.j3d.Canvas3D.createContext
0.12%	5	javax.media.j3d.RenderingEnvironmentStructure.getInfluencingAppearance
0.12%	5	java.lang.Thread.yield

The main botlleneck functions are to be found in the JNI communication component of the MiS20 program, using almost 90% of the cpu cycles. Both sending and receiving of data for the MiS20 program can be regarded as a bottleneck. The usage of Java3D is not very performance costworthy so we can ignore improving performance there, it uses only 1% of the available cpu cycles.

The sending method used in the communication process converts *SimSnapshot* instances to *CommSnapShotObjects* which are sent to the Mi20 control system. This translation process also requires mirroring of vectors[3] when the Mi20 team is playing from the other half of the soccer field (default is left). This results in overhead which can be reduced to when applying a fast accessable data structure. This is described in paragraph 4.2.

1. Profiling is described on the Java website, see reference [W1].
2. PrefAnal, or Preformance Analys tool can be found on the Java website, reference [W1].
3. Mathematical vectors, not to be mistaken by instances of the java.util.Vector class.

4.1.1. JNI performance enhancements

To boost the JNI performance some enhancements has been made. An important boost option is to use the static library as few as possible, which means only for creating, deleting, or checking a C++ thread. Unfortunately the static library have also been used to send objects, because Java cannot contain a pointer to a C++ function. The receiver threads contains a pointer to the JVM, so when new data has been received a function of the Java thread will be called with the received data as it arguments.

One of the few performance enhancements we were able to make for JNI was the use of cloning for the creation of objects. This reduces the creation time of an object by 67% as is described in paragraph 4.4.

4.2. Collections

The model, as described in chapter 3.1, contains a lot of information. A match has two teams, each containing five robots. These robots, as well as the ball, which is also contained by the match, have a list of positions called vectors[4]. The parent class for the ball and robot, the *SimObject* class contains this vector collection (of the *SimVector* class). However, there are various approaches on how to actually implement such a collection. One thing to take into account is that the list of vectors will be approximately 18.000 units long. There are four good candidates; *LinkedList*'s, *ArrayList*s, *Vector*s and arrays of objects[5], all of which can be found in the *java.util* package.

LinkedLists are very robust in their usage, they provide in size methods, as well as adding, replacing, iteration functions. However, they do not implement the *RandomAccess* interface. This interface ensures a list an be searched very fast using random access to the list. A good argument to use LinkedLists though, is they have very good performance when managing data in the front to middle sections of a list[6].

A second option is the usage of *ArrayLists*. These lists implement the *RandomAccess* interface which ensures very fast list access. However, ArrayLists are slow compared with LinkedLists when managing data in the front of the list since all objects have to be replaced in the list. LinkedLists just change their link to the following object in the list in comparison.

Third, *Vectors* are a good alternative. They also implement the *RandomAccess* interface from the java.util package. However, Vectors have overhead whilst it uses synchronized functions. This means it is threadsafe at extra cost. When viewing the design[5] it becomes clear that there is no need to synchronize the objects since only one instance may alter any lists, the *SimAdministrator* instance. Also, when a *Vector* instance has outgrown its original size it automatically allocates twice its current size for future usage. In comparison, ArrayLists add only half their size which resulst in better performance concerning.

Fourth, the usage of arrays to store objects is not to be recommended. They have a static length which can be extended by creating a new array only. This boasts too much overhead.

Taking all the pro's and con's into account, the best alternative to choose is the usage of ArrayLists for the vectors.

4.3. StringBuffer

A second performance issue is the usage of *String* instances. Strings are very often appended using simple “+” operators. These can be relatively costly when used in great numbers.

An example of the application of Strings in the MiS20 program. Every time a full second has elapsed when a snapshot is created and sent to the Mi20 control system[7], the simulator time (in the GUI) is updated using a propertychange event. The value sent along is a String containing the new time. This String is constructed using the *toString()* method of the current *SimTimestamp* class instance. This function calls five “+” operators to create the *String*. A better and more effective way to accomplish this task is to apply a *StringBuffer*. As its name states, it uses buffering to improved performance aspects.

4. Mathematical vectors, not to be mistaken by instances of the java.util.Vector class.
5. The Java 1.4.1 API describes all Java API classes in great detail, see reference [W2].
6. Perfomance issues for Java programs are described in great detail in reference [W3].
7. See reference [B1] for more information on the individual components of the Mi20 system.

4.4. Object creation

Every time the vectors are updated using the update thread, a new *SimSnapshot* containing new *SimVector*s for all objects is created. Normally constructors would be used to create a new class instance. However, using a constructor to create a new object is relative costly compared with the copying of an object.

Arrays can be copied using the *System.arrayCopy()* method. Objects can override the *clone()* method in the *Object* class. There are then two approaches to implement such a *clone()* function; using a shallow or a deep clone. By calling the clone function of the *Object* class, a bitwise copy is created of the object, in other words, a shallow clone. References unique to a specific class instance are to be cloned as well in the MiS20 program. How cloning works is explained in our implementation document[8]. We created a simple clone versus constructor test to measure performance differences between the two. This program can be found in the appendices of our implementation document. The results are:

```
constructor loop took: 131msec
clone loop took: 51msec
```

The results on the previous page prove that cloning is less costly compared with constructor usage. The reason why cloning is faster and less costly than using a constructor is that the latter requires calculation of the amount of memory to be allocated. This process is passed by when cloning since the needed amount of memory is already known. We therefore use cloning as much as possible in the MiS20 program.

Another point to be taken into account can be found in the XML parser we wrote for the MiS20 program. Here, a file is parsed and each XML tag is returned as a *java.util.String*. Since various match information is to be considered as a float, int or other numeric data we will have to convert these *String*s to their numeric values. For example an number 0 is to converted to int. We can create a new *Integer* class instance of which we call the *intValue()* method. This returns an int value 0 which we wanted. Since constructors are slow compared to static class methods for which no class instance has to be created we are better of using such static methods. The results of a small test program we created:

```
constructor loop took: 128 msec
static function loop took: 41 msec
```

This test, and the cloning test together prove that constructors are to be avoided when able to do so.

4.5. Timestamping

To update the current time in a match and to keep track of all generated snapshots in that match we use timestamps for each such snapshot (a *SimSnapshot* class instance). The timestamps consist of the number of milliseconds since the beginning of a match. When a match is reloaded, the timestamps are reset to 0. To determine the time which has passed between to successive method calls we need to obtain the current time in milliseconds from the system.

The Java SDK offers two approaches to get the current time in milliseconds. The *java.util.Date* class has a method called *getTime()* which returns the current time in milliseconds. However, it requires an instanciation of the *Date* class before that method can be called. As seen in chapter 4.4, this is relatively costly since the *Date* class constructor has to be called.

The other approach to obtain the current time in milliseconds is to call the *System.currentTimeMillis()* method. This is a static method meaning it can be called without having to create a *System* class instance. The second approach is twice as fast as the first one[9] which is why we used the second option.

8. For the MiS20 implementation document see reference [B7].
9. See reference [W4] for more information.

4.6. Java3D

Java3D also has a few perfomance gains to obtain. First of all, the parts of the scene graph which will not change (in our case, the soccer field representation) can be precompiled. That means that no extra effort has to be undertaken at runtime to check for changes in that part of the scene graph.

Also, the level of detail can be altered for the 3D objects. For example; we use two overlapping circles to draw the central circle line on the soccer field. Those circles can be drawn with 50 vertices in height and width but also with 1 in height and 12 in width. This resulst in a dramatic increase in framerate of the program[10]. Another example is the representation of the *Gui3DBall*. For representing the ball we use a *Sphere* class. This class has a method, *setNumVertices(int)*, which sets the number of vertices to be used when drawing that sphere. It speaks for itself that the lower this number is, the more performance is gained.

4.7. Array copying

The java *java.lang.System* class provides a method which can copy an array to a new location, in effect cloning it. It is also possible to set a new array size when doing so. We use this in the*addThread(Thread)* method of the *CommAdmin* class. Normally we would create a new array using *new Thread[size]*. However, the *System.arraycopy(..)* method is much faster[11] when creating a new array. We can also enlarge the array with one unit which we then fill up with the new*Thread* to add.

10. We measured the framerate increase at 24.2 frames per second extra.
11. See appendix C for performance tests.

5. Developers manual

The MiS20 simulator program has been constructed for reuse and expansion by other teams of software developers. This chapter will explain the areas which will, most likely, fall into this category. Software developers can use this information to easily implement those expantions. We will discuss how to create an automated referee and how to realize the various camera vantage point in the 3D soccer field representation.

5.1. Automated referee

Currently, the MiS20 simulator program has been implemented with a manual referee. This referee (implemented in the *CtrlReferee* class) implements the *Referee* interface we created for each referee implementation to comply to. As described in paragraph 3.4.3, the currently implemented referee requires a user to operate it.

An automated referee can be implemented by creating a new class, *CtrlAutoReferee* for example, which extends the *java.lang.Thread* class. This new *Thread* is best started in the *CtrlAdmin* class on creating of this class. The *SimAdministrator* can be asked if it is running using the *isBusy()* method. And if so, the *CtrlAutoReferee* class can determine if a referee call is required. Another approach is to add this functionality to the *SimUpdater Thread*'s *run()* method. This method will then decide, before or after calling the *updateObjects()* method of the *SimAdministrator* class if a referee call is needed.

The big difficulty of an automated referee is that will have to be able to not only detect when a goal has been scored or what to do when a goal kick is required. It will also need to be able to detect various game situations which require referee interferance. For example, a free ball situation is to be called whenever a stalemate occurs which lasts 10 seconds. But when does a match situation comply to such a stalemate?

This problem, accompanied by other simular problems, will have to be solved in order to create an effective automated referee.

5.2. Various camera vantage points

Paragraph 3.3.2.2 describes the method used to implement the camera in the MiS20 simulator. The *GuiCamera* class can be used to position the camera freely in the 3D world which represents the soccer field. A up *Vector3D*, and two *Point3D* classes have to set to move the camera to a new position. We use the lookat method of the *Transform3D* class to calculate the new camera position.

5.2.1. Pursuit camera

A camera which follows a specific object across the soccer field can be implemented by setting the selectObject with a specific object and having the *GuiCamera* class listen to vector property changes of that object. When a property change event has been caught by the GuiCamera the new camera position has to be calculated. The up *Vector3D* will not change, so only the lookAt and lookFrom *Point3D*'s have to be recalculated. The first can be set simply as the current *SimVector* values of the object being pursued. The latter has be calculated each time.

The easiest and best way to accomplish this task is to use a static difference Vector3D which always positiones the pursuit camera a bit behind and above the pursued object.

5.2.2. Free camera

A free cam can be implemented in two ways. The first is to use the *SimpleUniverse* with the Java3D default *OrbitBehavior* behavior. This behavior enables a user to move the camera freely about in the a 3D world. However, the controls are not user friendly in usage at all.

The second option is to use the lookat functionality we mentioned earlier in the *GuiCamera* class. The method used for moving the camera about has to be considered then; keyboard or mouse. If the mouse is used, the *MouseEvent*s can be used to alter the camera position. If the keyboard is used, the already created *CtrlActionListener* class can be expanded to add this functionality. A good way to use the keyboard is to use the arrow keys. Left and right indicate a spherical movement whilst up and down control zooming.

5.3. MiroSot small and large leagues

The MiS20 has been designed and implemented for usage in the FIRA MiroSot middle league in which two teams, consisting of five robots each, will play. However, the MiroSot league also contains a small and a large league competition in which three and seven robots per team respectively play each other. The rules are not so very different in these leagues. It is therefor conceivable that the MiS20 would be altered to be used in these leagues as well.

We have taken steps to ensure this will be possible in the future. The steps a developer will need to take to accomplish this transformation are:

5.3.1. Field sizes

Set the correct field sizes in the *SimField* class. The small league has a smaller field in which the robots will play whereas the large league will have a larger soccer field to play in. The measurements of these soccer fields are stated in the FIRA regulations for the small and large leagues. These can be found on the FIRA website[1].

5.3.2. Snapshots

Create and store new snapshots for each of the situations a match can be in. These can be found, as with the situations in the middle league, in the FIRA regulations[1]. The snapshots are stored in XML files, as was described in paragraph 3.2.1.1. These files are stored with the prequel “5vs5_” indicating the number of robots in this snapshot. The snapshots for the small league are to be named “3vs3_” and the rest of their name. The large league snaphots will logically have “7vs7_” as a prequel.

5.4. Match settings

Currently, the settings for a match are stored in the SimSettings class mainly. The game duration time for example. It is very much conceivable to have these settings set by the user each time he or she starts a match. The settings which can be set here are:

- Team names (actual team names and not simply “MiSteam” and “Opponents”);
- Game duration time. If tests have to be performed which require a shorter game duration, this can be a setting which the user may provide;
- Team colorings. In a real match both teams will be given a color by the referee. However, in the MiS20 it will not matter a great deal what color a team has since the teams are identified by number. The team color is just a indication in the GUI for the user. It might be usefull to change these colorings, but it will not have a great impact to do so.

1. Download these regulations on the FIRA website, reference [W7].

5.5. Improved collisions

As we have stated before, we designed and implemented the MiS20 program to be improved upon by others in the future. Here, we will describe how a new collision detection algorithm can be integrated into the MiS20. Also, we will describe how collision handling can be updated.

5.5.1. Detecting

A list of colliding objects is generated by looping through all objects in the match. This is done by the collision handler since it can reuse the object references which results in better performance. This list is then handled by the collision handler further. Collisions between objects in the MiS20 are detected using two levels of detection. First we use AABB collision detection to detect if a collision might be possible. If so, the second level of detection uses a combination of OBB and bounding sphere detection to detect collisions very precisely. These tests are performed in the *SimAdministrator* class' *calculateCollisions()* method.

This method will call upon an instance of the *SimCollisionHandler* class, currently called *SimHansCollisionHandler3*. This collision handler will first test for possible collisions using a static method of the *SimAABBCollisionDetector* class. This class can be altered or expanded upon so that the collision detection can improve.

The second phase of collision detection uses OBB and bounding sphere collision detection. The method calls for these collision detection techniques can also be found in the *SimHansCollisionHandler3* class. A new (possibly better) detection algorithm can be integrated here as well.

5.5.2. Handling

As described in the previous paragraph, the collision handling class calls the collision detection. This is done to improve performance by reusing object references. This paragraph will describe how a new and better collision handler can be implemented in the MiS20.

Looping through the list of colliding objects we generated using the collision detections we will handle the collisions using a kinematic model of the objects involved. The goal of handling collisions is to simulate an actual collision as best as possible. To accomplish that we need to use various physics aspects such as object weight, momentum, velocity, turning speed and so on.

The list of colliding objects consists of colliding lines for each of these objects. These lines are either front, back, left side or right side of a robot or a point for a ball. The *calculateCollisions(SimObject[])* method in the *SimHansCollisionHandler3* class contains our version of collision handling. This method is to be updated or replaced by future developers.

5.6. Kinematic models

We also use kinematic models to represent the robots and the ball apart from the collisions they can be involved in. These models included decelleration speeds etc. For example; when the ball is kicked by a robot, it will roll across the soccer field and will eventually come to a stop. The *SimBall* and *SimRobot* classes include a method called *updatePosition()* which will, as its name states, update the positions of the ball and robot. This is done differently for the ball and robot as is described in chapter three.

To improve upon these models, a developer will have to update these methods. A physics equation, such as the one we use which described in our design document[2], can be used to update the object vectors to their new values. Currently, we do not use skidding, drifting and noise movement for the ball and robots. These are points to be researched and implemented by future developers.

5.6.1. Skidding

Skidding occurs whenever an object tries to accelerate faster that it possibly can resulting in loss of movement force. This force can not be transferred to the floor fully.

2. For the MiS20 design document, see reference [B1].

5.6.2. Drifting

Drifting occurs whenever an object, especially a robot, tries to turn to fast for its current velocity. It will not drift out of the turn.

5.6.3. Noise movement

The ball is as not perfectly round as we simulate it currently in the MiS20. It is an orange golf ball which means it has a number of small dents across its surface. This also implies that this ball will not move in a straight line always, it will move from side to side sometimes. This is very hard to simulate. The best way to accomplish a simulation of this behavior is to implement a noisy movement in stead of a linear movement as we have done.

References

Documents:

[B1] Robot soccer – Design document
H. Dollen, W. Fikkert

[B2] Design Patterns - Elements of Reusable Object-Oriented Software
E. Gamma, R. Heml, R. Johnson, J. Vlissides - ISBN: 0-203-63361-2

[B3] FIRA Middle League MiroSot Game Rules
Federation of International Robot-soccer Association

[B4] 3D User Interfaces with Java 3D
Jon Barrileaux – ISBN: 1884777902

[B5] MiS20, a robotic soccer simulator – bachelor thesis
H. Dollen, W. Fikkert

Websites:

[W1] Java Native Interface
http://java.sun.com

[W2] Java 1.4.1 API javadoc
http://java.sun.com/j2se/1.4.1/docs/api/

[W3] Thirty ways to improve the performance of your Java programs
ftp://ftp.glenmccl.com/pub/free/jperf.pdf

[W4] Codito ergo sum
http://freeroller.net/page/nil

[W5] Java3D 1.3.1 API javadoc
http://www.informatik.uni-frankfurt.de/java/J3D/J3D_doku/html/

[W6] MiS homepage
http://cs.utwente.nl/~fikkert/index.html

[W7] Federation of International Robot-soccer Association homepage
http://www.fira.net

Glossary

JNI	JNI stands for Java Native Interface. It is a method to communicate between Java and C programs.
Mi20	Mission Impossible, a project group at the University Twente, the Netherlands, which is trying to construct a team of soccer playing robots in the FIRA MiroSot middle league.
MVC	Model View Control, a relatively standard design method used in Object Oriented programs which resulst in a complete seperation of the program model, the program view and the program control for easy re-use, adding, changing and removing of these individual MVC program components.
NIAM	"Natuurlijke taal Informatie Analyse Methode", or Natural language Information Analysis Method. NIAM distilles data structures from a project description. These data structures can then be used to create a database, UML objects etc.
UML	UML stands for **U**niversal **M**odeling **L**anguage. It is a modeling technique designed by Grady Booch, Ivar Jacobson, and James Rumbauch of Rational Rose. It is used for OOAD (**O**bject **O**riented **A**nalysis and **D**esign) It is supported by a broad base of industry-leading companies which, arguably, merges the best of the various notations into one single notation style. It is rapidly being supported by many tool vendors, but the primary drive comes from Rational Rose.
XML	XML stands for eXtensive Markup Language. It is a standard method to communicate and store information using a default layout. This layout can be parsed by a parser for any program.

Appendices

The appendix chapter consists of the following:

Appendix A. Source code template

This appendix describes the source code template which has been used to create the Mi20 simulator source. It incorporates the usage of Javadoc for source code documentation as well as a clear method to indicate the type of variables using a character at the beginning of a variable to indicate its type. For instance, a boolean variable will be called bVariable whilest a String variable will be named sVariable and so on. Finally, since CVS is used to store source files, some CVS utilities will be used in the source files. For example, CVS has the ability to automatically add version numbers, dates and authors. This is done by adding particular text in the source file. An example of this: "$date$" will add the last date and time this file was editted in this file.

```
/**
       $Log: changelog for this file generated by CVS
*/
/**
       Description of what this class does and for it is used.
       @author:       Hans Dollen
       @author:       Wim Fikkert
       @date:         Last modified date
       @version:      current version this class
       @since:        present since version number x
*/

// commentline

/*
       multicomment
       lines
*/

import com.sun.j3d.AclassName;
import java.util.LinkedList;
import java.util.HashSet;
import java.awt.Color;

public class          ClassName
       extends        ParentClassName
       implements     InterfaceClassName1,
                      InterfaceClassName2
{
       /**    desciption of bBooleanVariableName
              @see #getFunctionName() // for further info
              @see #setFunctionName() // for further info */
       public boolean bBooleanVariableName;
       /**    desciption of iIntergerVariableName1
              @see #getFunctionName() // for further info
              @see #setFunctionName() // for further info */
       private int    iIntegerVariableName1 = 1;
       /**    desciption of iIntergerVariableName2
              @see #getFunctionName() // for further info
              @see #setFunctionName() // for further info */
       private int    iIntegerVariableName2 = 2;
       /**    desciption of sStringVariableName
              @see #getFunctionName() // for further info
              @see #setFunctionName() // for further info */
       private String sStringVariableName = "a string";
       /**    desciption of iIntegerArrayName
              @see #getFunctionName() // for further info
              @see #setFunctionName() // for further info */
       private int[] iIntegerArrayName = {1,2,3};
       /**    desciption of oLinkedListVariableName
              @see #getFunctionName() // for further info
              @see #setFunctionName() // for further info */
       private LinkedList oLinkedListVariableName = new LinkedList();
       /**    desciption of oSelfMadeClassVariableName
              @see #getFunctionName() // for further info
              @see #setFunctionName() // for further info */
       private SelfMadeClass oSelfMadeClassVariableName;
```

```
        /**     Constructor description when needed
                @param  bBooleanVariableName  description of this boolean
                @param  iIntegerVariableName1 description of this integer*/
        public ClassName(boolean bBooleanVariableName,
                         int iIntegerVariableName1    )
             {
                super();
                this.bBooleanVariableName     = bBooleanVariableName;
                this.iIntegerVariableName1    = iIntegerVariableName1;
        }


        /**     Description of what the function aFunctionName will do and
                how it will do it.
                @return void
                @since  present since version number x              */
        public void aFunctionName()
        {
                for(int i = 0; i < 10; i++)
                {
                        // do something
                }
        }

        /**     Description of what toString does and how it does it.
                @return String The String description
                @since  present since version number x              */
        public String toString()
        {
                return "description this object"
        }
}
```

Appendix B. XML snapshot sample

Below, a XML file is displayed. This file is the representation of a snapshot which can be loaded into the match currently being played. A snapshot contains mathematical vector information of all objects (x and y positions as well as heading).

```
<?xml version="1.0"?>
<root>
       <snapshot>
              <obj id="0" x="1.8000001" y="0.89920026" h="0.3" />
              <obj id="2" x="0.5799999" y="0.9000001" h="0.010000013" />
              <obj id="3" x="1.5399998" y="0.31500003" h="0.020041827" />
              <obj id="4" x="1.5349998" y="1.425" h="4.1007996E-5" />
              <obj id="5" x="1.7000002" y="0.8949999" h="0.010149889" />
              <obj id="6" x="2.1200001" y="0.90000015" h="0.023782123" />
              <obj id="7" x="2.007" y="0.72729987" h="8.2844496E-4" />
              <obj id="8" x="2.0060005" y="1.0799999" h="0.0030951798" />
              <obj id="9" x="2.0049999" y="0.45" h="-0.0019364022" />
              <obj id="10"x="2.0049999" y="1.325" h="-9.1177225E-4" />
       </snapshot>
</root>
```

Appendix C. MiS20 source code

This appendix contains fragments of source code which are referenced to from the chapters of this document.

C.1. SimSnapshot.clone()

The *SimSnapshot* class implements the *java.lang.Cloneable* interface. The method to implement is the *clone()* method. The *super.clone()* statement creates a bitwise copy, or shallow clone, of the object. Next, all variables are cloned to create a deepclone of the *SimSnapshot* instance.

```
/**
       implements the cloneable interface
       @return Object  cloned version of this snapshot
       @since          1.0
*/
public final Object clone()
{
       SimSnapshot oClone = null;
       try
       {
              oClone = (SimSnapshot)super.clone();
              oClone.setName(this.getName());
              SimVector[]  oOwnTeamClones      = new SimVector[SimSettings.iTEAM_SIZE];
              SimVector[]  oOpponentTeamClones = new SimVector[SimSettings.iTEAM_SIZE];
              SimVector    oBallClone          = (SimVector)this.getBall().clone();
              SimTimestamp oTimestampClone     = (SimTimestamp)this.getTimestamp().clone();
              for(int i = 0; i < SimSettings.iTEAM_SIZE; i++)
              {
                     oOwnTeamClones[i]      = (SimVector)this.oOwnTeam[i].clone();
              }
              for(int i = 0; i < SimSettings.iTEAM_SIZE; i++)
              {
                     oOpponentTeamClones[i] = (SimVector)this.oOpponentTeam[i].clone();
              }
       }
       catch(Exception oExcept)
       {
              System.err.println(   "SimSnapshot.clone(): error whilest " +
                                    "cloning SimSnapshot: " + oExcept           );
              System.exit(1);
       }
       return oClone;
}
```

C.2. CloneVersusConstructorTest

This test program tests the performance differences between clone usage and constructor usage when creating a new object. The first loop tests the usage of constructors and the second loop tests the clone usage. The results prove the better option, regarding performance, is to use cloning:

```
constructor loop took: 131msec
clone loop took: 51msec
```

The source code for this test program:

```
long s = System.currentTimeMillis();

// the first test using the constructors
for(int i = 0; i < 10000; i++)
{
       new CloneTest();
}

System.out.println("constructor loop took: " + (System.currentTimeMillis()-s) + "msec");

CloneTest o = new CloneTest();
CloneTest t = null;

s = System.currentTimeMillis();

// the second test using the static functions of Float
for(int i = 0; i < 10000; i++)
{
       t = (CloneTest)o.clone();
}

System.out.println("clone loop took: " + (System.currentTimeMillis()-s) + "msec");
```

Where the CloneTest class looks like:

```
class CloneTest implements Cloneable
{
       public int i = 0;
       public float f = 0f;
       public String s = "a string";

       public CloneTest()
       {
               i = 1;
               f = 1f;
               s = "a longer string";
       }

       public Object clone()
       {
               try
               {
               CloneTest c = (CloneTest)super.clone();
               c.i = i;
               c.f = f;
               c.s = s;
               return c;
               }
               catch(Exception e)
               {
                       System.err.println("error whilest cloning");
               }
               return null;
       }
}
```

C.3.NumericConvertTest

This test program was written to test the effect of using constructors versus not usnig them to convert a *String* to a *float* value. It first calls 1000 Float constructors after which the floatValue() is called. The second loop calls the static *parseFloat(String)* method of the *Float* class.

The time needed for both loops is measured and written to the command prompt line. The results are:

```
constructor loop took: 128
static function loop took: 41
```

And the code for this simple test program is:

```
long s = System.currentTimeMillis();

// the first test using the constructors
for(int i = 0; i < 10000; i++)
{
       new Float("0.12345").floatValue();
}

System.out.println(   "constructor loop took: "
                      + (System.currentTimeMillis()-s) + "msec" );

s = System.currentTimeMillis();

// the second test using the static functions of Float
for(int i = 0; i < 10000; i++)
{
       Float.parseFloat("0.12345");
}

System.out.println(   "static function loop took: "
                      + (System.currentTimeMillis()-s) + "msec" );
```

Appendix D. UML diagrams

This appendix contains the UML diagrams of the various components of the MiS20; model, view, control and communication. Notice the *SimAdministrator* class is at the root of the program; all components connect to that class.

D.1. Modeling

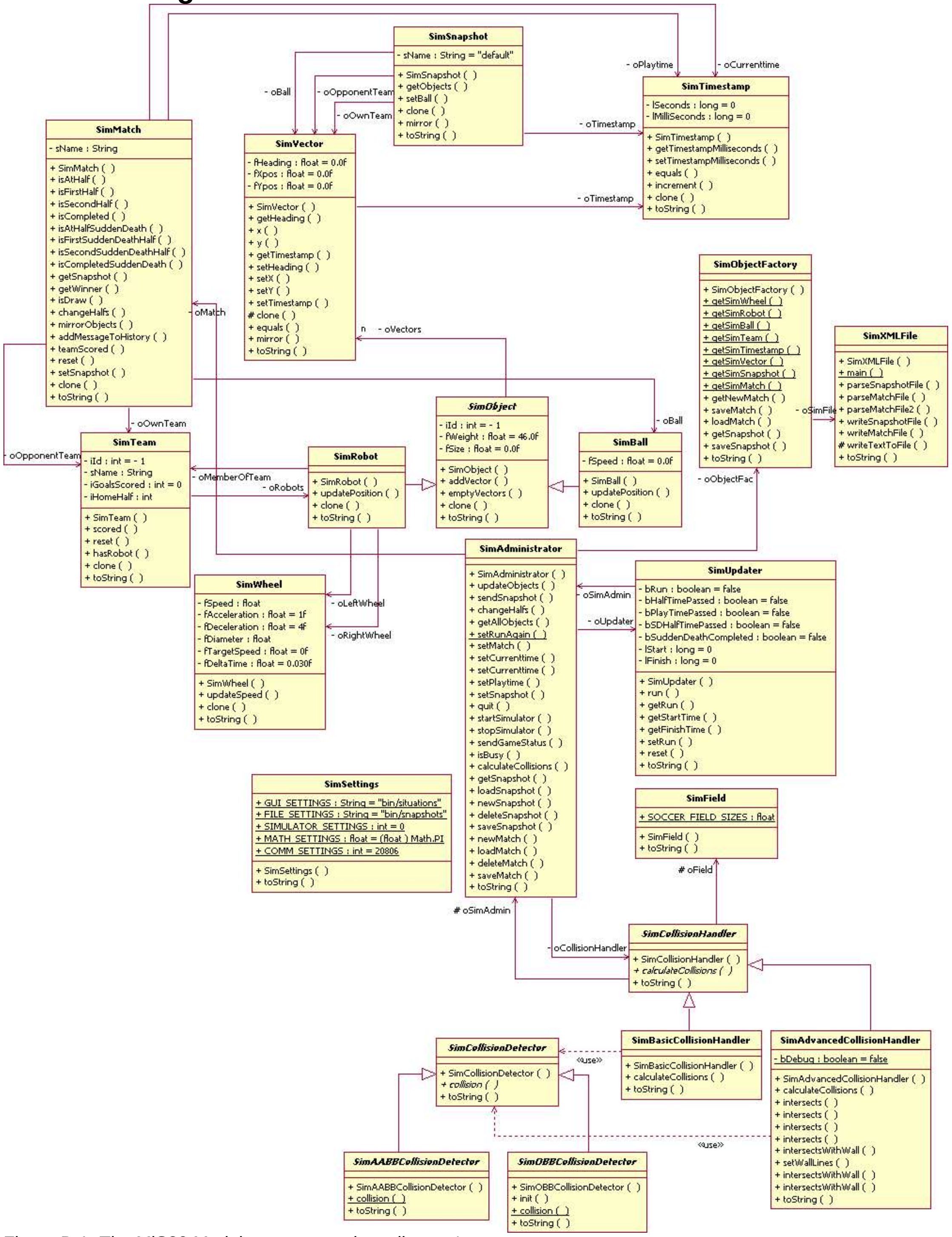


Figure D.1. The MiS20 Model component class diagram[1]

1. The entire class diagram can be found in reference [B1].

D.2. View

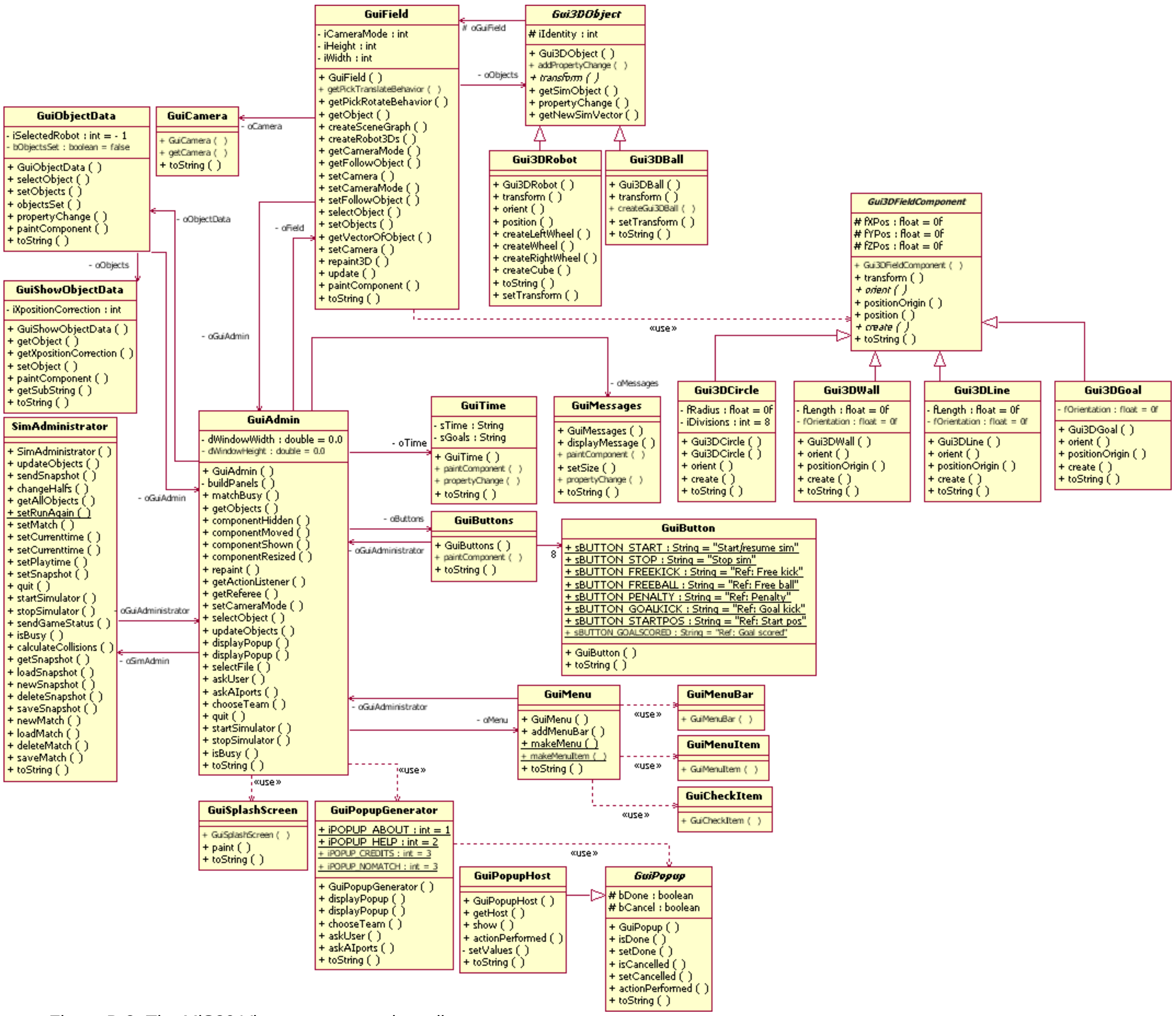


Figure D.2. The MiS20 View component class diagram

D.3. Control

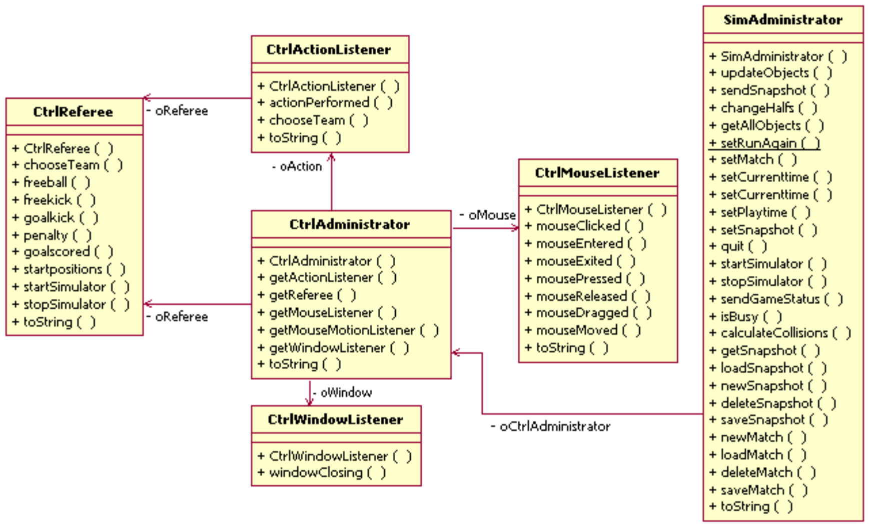


Figure D.3. The MiS20 Control component class diagram

D.4. Communication

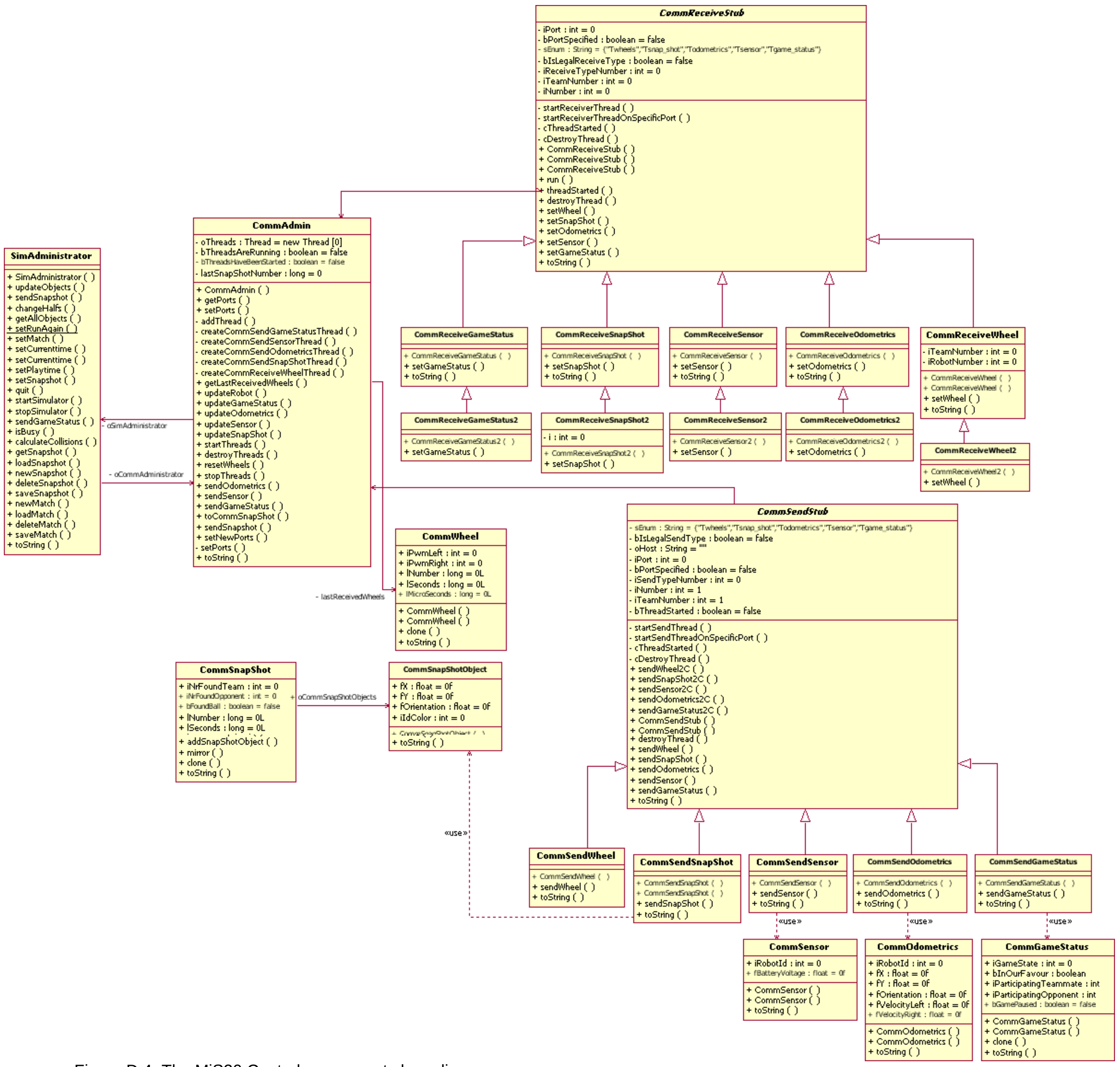


Figure D.4. The MiS20 Control component class diagram